

Operativsystemer

Dette notatet har innhold fra boken *Modern Operating Systems: Global Edition* av Tanenbaum og Bos i tillegg til forklaringer fra løsningsforslag til eksamener i faget TDT4186. Notatet er kun et sammendrag for egen eksamslesing, og det er nok litt skrivefeil her og der da jeg ikke har rettet det i ettertid. Hvis du finner en feil kan du gjerne sende en mail til vildera@stud.ntnu.no. Notatet fokuserer på det generelle og går derfor ikke dypt inn i de spesifikke operativsystemene.

Operativsystem

Et operativsystem er den grunnleggende programvaren i en datamaskin og kobler sammen maskinvaren med all annen programvare. Operativsystemet har ansvar for å tildele de forskjellige ressursene i en datamaskin til programmer. Interne ressurser er blant annet prosessor og minne, mens eksterne ressurser er for eksempel en skriver.

Operativsystemets delprogrammer

- Implementering, synkronisering og tidsstyring av tråder og prosesser.
- Implementering og håndtering av lager/minnet.
- Implementering og håndtering av I/O.
- Sikring av datamaskiner.

Disse delene samvirker gjerne lagdelt og med mikrokjerne:



Hvorfor et operativsystem er viktig

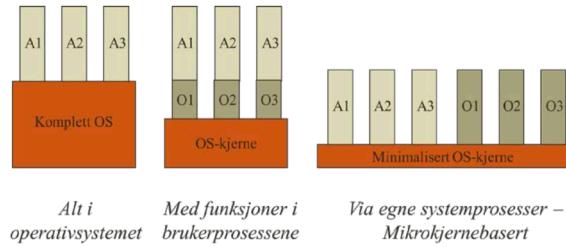
1. Operativsystemet sikrer at datamaskinens **tjenester** for brukere gjøres så **enkelt** som mulig.
2. Operativsystemet sikrer at **forvaltning** av ressurser gjøres så **effektivt** som mulig, slik at:
3. Tjenestetilbud og ressursforvaltning kan **utvikles** over tid på en billig og fleksibel måte.

Operativsystemer kan ofte sorteres inn i størrelse og bruk hvorav angitt applikasjonsområde angir bruk og angitt pris antyder størrelse:

- 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)

Organisering av operativsystemer

Komplett OS-modell er lite effektiv, men ganske trygg. OS-kjerne er mer effektiv men mindre trygg og minimalisert OS-kjerne er mer trygg, men enda mindre effektiv enn komplett OS-modell.



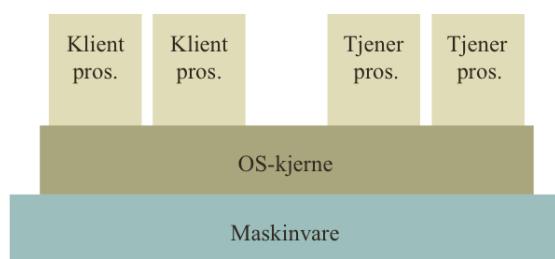
Moderne operativsystemer

Moderne operativsystemer er ofte organisert med **mikrokjerne**, i en **objektorientert** struktur som benytter **tråder**.

Dette sikrer at **kjernemodus**, programvaren som har aksess til hele lageret og instruksjonssettet er av så lite omfang som mulig. I tillegg kan programvaren **lett tilpasses** nye maskinvaretilbud og programvarebehov. Det tillater også en **effektiv strukturering** for hvordan oppgaver kan løses internt i operativsystemet, som samtidig kan håndtere **mangfoldet arkitekturmessig**. Dette inkluderer støtte av blant annet tette eller løse koblinger, en eller flere prosessorer, og en eller flere kjerner per prosessor.

Mikrokjerne

Et OS med mikrokjernearkitektur har en **kjerne med kun essensielle funksjoner/tjenester** som må være til stede. Dette inkluderer adresseområder, interprosess-kommunikasjon (IPC) og enkel tidsstyring (scheduling). Andre tjenester bygges i lag over mikrokjernen, kjøres ofte i brukermodus og behandles som en applikasjon av mikrokjernen. Denne arkitekturen frakobler kjerneutvikling fra tjenesteutvikling og tjenerne kan tilpasses spesifikke applikasjoner eller miljøer. Dette gir en enklere programvare som er lettere å tilpasse, utvide og flytte samt har en distribuert og objektorientert tankegang. Ulempen med mikrokjerne er endring fra eksisterende systemer og at det er unødvendig for mindre systemer.



Objektorientert

I et OS med objektorientert design har prosesser mulighet til å legge til modulære utvidelser på en liten kjerne. På OS-nivå lar en objektorientert struktur programmereren tilpasse OS uten å bryte med systemintegritet. Dette kan også lette utviklingen av distribuerte verktøy og fullverdige distribuerte systemer.

Tråder

Moderne OS benytter seg ofte av *multithreading* som er en teknikk hvor en prosess som utfører en applikasjon deles inn i tråder som kan utføres samtidig. Dette er nyttig når en utfører et antall uavhengige oppgaver som ikke behøver å kjøre serielt. En tråd er en komponent av en prosess og har en egen tråd-kontekst (prosessorkontekst med program counter og stack pointer, samt eget dataområde for en stack).

Utfordringer i moderne operativsystemer

Tilbudsutvikling

Det er økende spenn mellom ulike maskintyper og behov de har (store/små).

Teknologiutviklingen

Utviklingen i teknologi har ført til et større gap mellom effektiviteten til ulike komponenter i operativsystemet, som prosessor vs. Minnet og I/O.

Funksjonalitetsutvikling

Nye systemer har ofte multiprosessorer og/eller multikjerner. Dette gir en mer kompleks programvare og nye hensyn må tas i forhold til tidsstyring, synkronisering etc.

Etterspørselsutvikling

Tidligere systemer var hovedsakelig batch-orienterte, mens mange moderne systemer er interaktive, noe som introduserer nye utfordringer som for eksempel frister på utførelse.

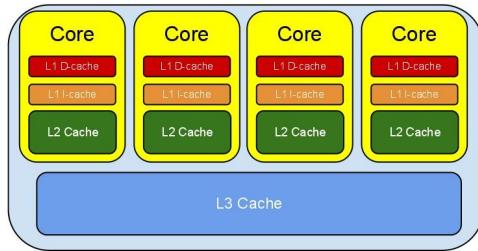
Moderne mangfold i hardwarearkitektur

Multiprosessorsystem

Mange moderne datasystemer har flere prosessorer. Et multiprosessor-OS skal gi samme funksjon som et multiprogrammert OS, med ekstra funksjonalitet for å imøtekommne flere prosesser. Dette inkluderer mer kompleks håndtering av samtidige prosesser og tråder, tidsstyring, synkronisering og minnehåndtering. Fordeler med multiprosessorer er økt pålitelighet, økt ytelse og reell parallelitet. Ulempene er at det medfører kompleks programvare og at det blir større ytelsesgap i tillegg til utfordringer knyttet til implementering, korrekthet og skalerbarhet.

Multikjernesystem

I et multikjernesystem er det flere prosesseringskjerner i hver prosessor. Dette tilbyr nyttig caching på flere nivåer (se figur) hvorav flere kjerner kan dele cache. Bruk av flere kjerner innenfor en prosessor gjør det mer komplisert for OS å fordele ressurser og å koordinere hendelser, da flere ting kjører samtidig. Dette kan løses med synkronisering. I tillegg er det utfordringer knyttet til effektiv utnyttelse – virtuell maskin vs. Sentral støtte.



Distribuert system

I et distribuert system befinner komponentene i systemet seg på datamaskiner i et nettverk. Disse kommuniserer ved å sende meldinger.

Fra tett til løs kobling

At to entiteter er løst koblet betyr at entitetene samhandler, men man kan endre den ene komponenten uten at dette medfører at man må gjøre endringer i den andre. For tette koblinger derimot er det ofte kostbart å gjøre endringer da man må endre begge entitetene. Det finnes også grader av de to. Løse koblinger er det mest grunnleggende verktøyet man har for å gjøre integrasjoner endringsdyktige.

Kapittel 3 og 4: Prosesser og Tråder

Prosesser representerer et program under utførelse. Dette inkluderer f.eks. kode, stakk, data og kontekst. En oppgave som skal løses på en datamaskin kan deles opp i flere mindre, delvis uavhengige underoppgaver, og en tilordner gjerne en prosess til hver slik for å få en effektiv utførelse av den samlede oppgaven.

- Effektiv utnyttelse av flere prosessnivåer
- Riktig synkronisering og kommunikasjon innen avanserte applikasjoner

Grunner for å opprette prosesser inkluderer: ny batch job, interaktiv pålogging (bruker), hvis OS ønsker å utføre en tjeneste på vegne av brukerprogrammet eller en eksisterende prosess danner nye prosesser for å utnytte modularitet eller parallelitet.

Grunner for å terminere prosesser inkluderer: Swapping, OS-grunn (som at prosessen er mistenkt for å skape et problem), interaktiv brukerforespørsel, timing eller at foreldreprosessen termineres.

Tråder miniprosesser som tilhører en overordnet prosess, deler kontekst med den og er billig å opprette og håndtere. En prosess oppretter gjerne flere undertråder for å kunne utføre ulike deloppgaver i parallel og samtidig få til kommunikasjon og håndtering av slike deloppgaver på en billigere måte enn med fulle prosesser. Ulempen er at det medfører mer kompleks kontroll og styring.

Pluss	Minus
<ul style="list-style-type: none"> • Skiller ressurseierskap og CPU-bruk • Passer naturlig applikasjonsstruktur • Gir ny kommunikasjon via deling av lager • Gir færre tidkrevende prosessskifter 	<ul style="list-style-type: none"> • Flere gjensidig avhengige konsepter • Mer kompleks kontroll og styring



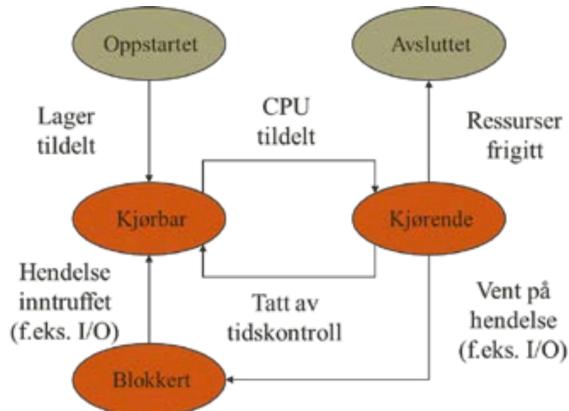
Prosesstilstander / Swapping

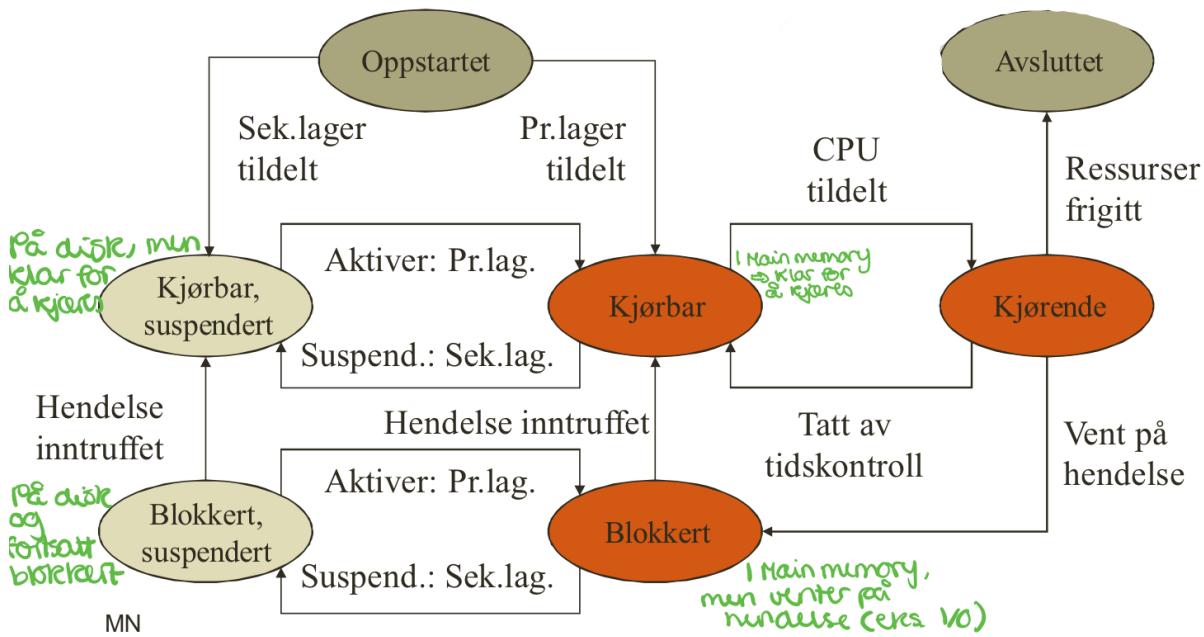
Swapping er når en del av en prosess (eller hele) flyttes fra primærminnet til sekundærminnet. Når ingen av prosessene i primærminnet er i «Ready» state, swapper OS en av de blokkerte prosessene til sekundærminnet (disk) til en suspendert-kø. OS tar deretter inn en annen prosess fra suspendertkøen eller tar inn en ny prosess på forespørsel. Dette innebærer I/O-til disk, men totalt sett er ikke dette mer krevende enn å vente på prosesser, og det vil derfor øke performance. Det kan dog være vanskelig å bestemme hvilken prosess som skal hentes inn. Som regel henter vi fra suspendertkøen, men hvis de er blokkerte kan vi hente en ny.

Avbruddssystem

Avbruddssystemet i en datamaskin sørger for at maskinvaren kan sende signaler til prosessoren for å indikere at en hendelse (event) har oppstått. Et avbrudd varsler prosessoren om at en tilstand med høyere prioritet krever at den utførende koden på prosessoren må avbrytes. Avbruddssystemet har innvirkning på effektivitet og funksjonalitet i datamaskinen, og det skaper både utfordringer og muligheter ved utviklingen av operativsystem. Siden OSet har oppgavene å blant annet tidsstyre prosesser, håndtere minnet, I/O og mer er det viktig at OS-utviklere har kjennskap til maskinens avbruddssystem.

Basismodell

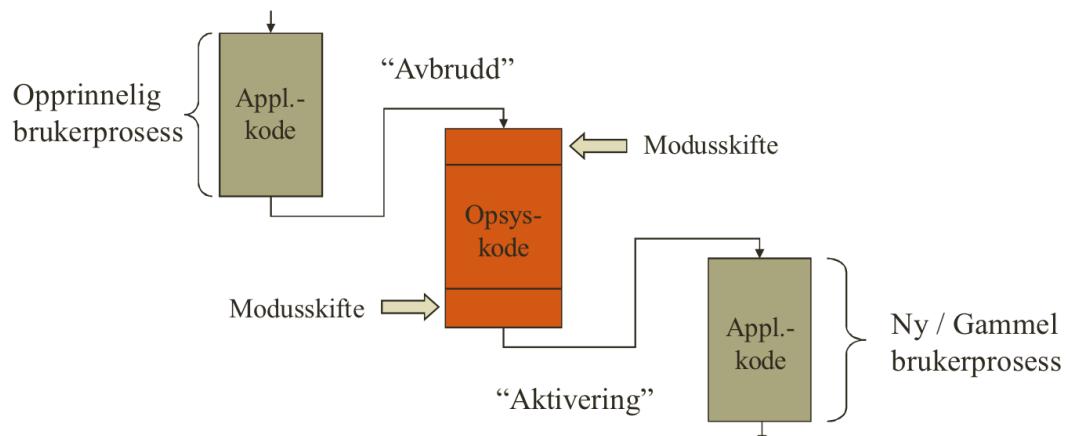




Prosess-skifte

Prosessskifte innebærer å bytte CPUen fra en prosess til en annen, og inkluderer tilstandsendringer for to prosesser. Dette gir dermed kontroll til operativsystemet for å bestemme hvordan skiftningen skal skje. Ett prosess-skifte inkluderer **to modusskifter inn og ut av operativsystemet, og krever derfor mye arbeid**. Et prosessskifte kan oppstå på ethvert tidspunkt OS har fått kontroll fra en pågående kjørende prosess. Det er typisk tre tilstander som medfører et prosessskifte:

- Klokkeavbrudd
- I/O-avbrudd
- Minnefeil: f.eks. at prosessoren støter på en virtuell adresse som ikke ligger i main memory.



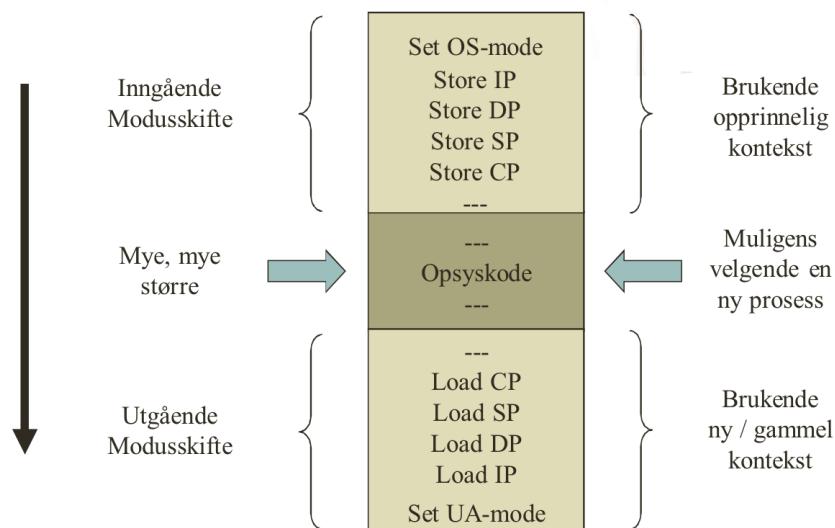
Modusskifte

Et modusskifte endrer tilgangen til adresselageret, hvor mye av lageret man har tilgang til, og instruksjonsssettet, hvor mange instruksjoner man kan utføre. Et eksempel er å skifte fra

kjernemodus som er mer privilegert, til brukermodus som er mindre privilegert. Typiske funksjoner i kjernemodus inkluderer styring av prosesser, minnehåndtering, I/O-styring og støttefunksjoner. Disse funksjonene har ikke vanlige prosesser i brukermodus tilgang til.

Hendelser som kan gi OS kontroll (modusskifte og prosessskifte):

- **Avbrudd** – eksternt (uavhengig av den pågående prosessen) som respons på en I/O-forespørsel hos en annen prosess
- **Trap/felle** – error eller exception generert i prosessen.
- **Supervisor call** – internt, som en forespørsel om en I/O-operasjon



Multithreading

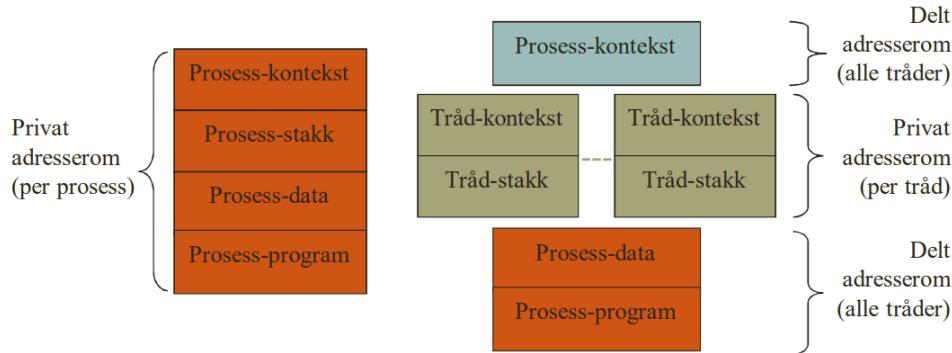
Multithreading er operativsystemets evne for å støtte flere, samtidige utførelser innenfor en enkelt prosess. I et multithreaded miljø defineres en prosess som en enhet for ressursallokering og beskyttelse. Assosiert med prosesser er:

Virtuelt adresserom	Holder prosess image med <i>prosess-kontekst, prosess-stack, prosess-data og prosess-program</i> .
Beskyttet tilgang til ressurser	Som prosessorer, andre prosesser (for intreprosesskommunikasjon), filer og I/O-ressurser.

Innenfor en prosess kan det være en eller flere **tråder** assosiert med:

Tilstand	Ready, Running, Blocked etc.
Kontekst (når den ikke kjører)	En måte å se på en tråd er som en uavhengig program-peker som opererer innenfor en prosess.
Stack	Stack som brukes under utførelse av program.
Statisk minne for lokale variabler	
Minneaksess	Tilgang til minne og ressurser til prosessen tråden tilhører. Dette deles med andre tråder i den samme prosessen.

Dette betyr at hver prosess fremdeles har en prosesskontrollblokk, og et bruker-adresserom, mens hver tråd har sin egen bruker- og kjernetråd, i tillegg til en trådkontrollblokk som holder et register, prioritet og informasjon relatert til den tråden. En enkelt-trådet prosess trenger ikke en trådkontrollblokk.



Fordeler med tråder: *knyttet til utførelse*

1. Det tar mindre tid (en faktor på 10) å opprette en ny tråd i en eksisterende prosess enn å opprette en ny prosess.
2. Det tar mindre tid å terminere en tråd.
3. Det tar mindre tid å bytte mellom tråder enn å bytte mellom prosesser.
4. Øker effektiviteten i kommunikasjon mellom ulike eksekverende programmer. I de fleste OS krever kommunikasjon mellom uavhengige prosesser innblanding av kjernen for beskyttelse og andre mekanismer for kommunikasjon. Ettersom at tråder deler minne og filer, kan de også kommunisere uten innblanding av kjernen.

Fire eksempler på bruk av tråder i et single-user multiprocessing program

Forgrund- og bakgrundsarbeid – For eksempel i et regnearkprogram kan en tråd vise frem menyer og lese brukerinput, mens en annen kjører bruker-kommandoer og oppdateringer i regnearket.

Asynkron prosessering – Asynkrone elementer i et program kan implementeres som tråder. For eksempel kan en beskyttelse mot strømavbrudd implementeres som at RAM lagres til disk én gang i minuttet. En tråd kan ha dette som sin hovedjobb.

Parallelitet → økt hastighet – En flertrådet prosess kan beregne en bulk med data samtidig som en annen bulk med data leses fra en annen enhet. Parallelitet øker hastighet på utførelse av programmet.

Modulær programstruktur – Programmer med varierte kilder og destinasjoner av input og output, samt varierte aktiviteter kan være enklere å designe og implementere med tråder.

Trådfunksjonalitet

Tilstander

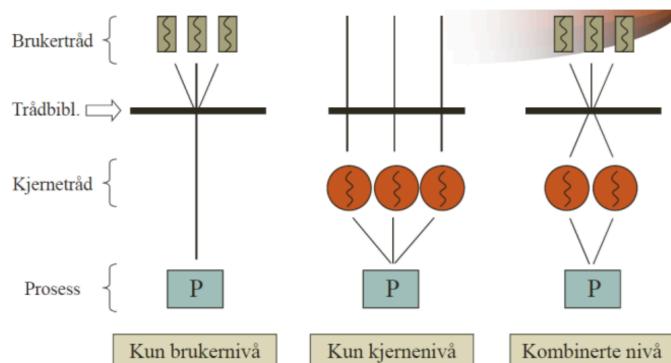
Nøkkeltilstander for tråder er *Running*, *Ready* og *Blocked*. Generelt gir det ikke mening å assosiere suspenderingstilstander med tråder, ettersom at dette er konsepter på prosessnivå. *Dersom en prosess byttes ut, vil alle tilhørende tråder også byttes ut siden de deler adresserom.*

Synkronisering

Enhver endring en tråd gjør på en ressurs påvirker miljøet til de andre trådene i prosessen. For å hindre at trådene forstyrrer hverandre eller ødelegge datastrukturene er det derfor nødvendig å synkronisere aktivitetene til de ulike trådene. Problemene som oppstår og teknikkene som brukes for å synkronisere tråder, er i hovedsak *den samme som for prosesser*.

Typer tråder

Ettersom at tråder er høyere oppe i «kjeden» enn prosesser ønsker vi å kontrollere trådenes aksess av OS-funksjonalitet.



User-level Threads (ULT)

Med kun ULT gjøres trådhåndtering kun av applikasjonen. Det vil si at det er et tråd-bibliotek som programmereren kan bruke til å opprette og terminere tråder, samt kommunisere mellom dem, tidsstyre, lagre og gjenopprette kontekster. En applikasjon begynner vanligvis med en enkelt tråd, og utvider til flere tråder ettersom at programmet utføres. Trådbiblioteket oppretter en datastruktur som holder trådene, kontrollen gis til tråder i tilstanden Ready, ved bruk av en *tidsstyringsalgoritme*. Når kontrollen gis til biblioteket igjen lagres trådkonteksten med innholdet i brukerregistre, program counter og stack-pekerne.

Fordeler

ULT har billigere systemkall enn KLT, i tillegg til at bytting mellom tråder *ikke* krever kjernemodus ettersom at all trådhåndtering gjøres innenfor brukeradresserommet til en enkelt prosess. Dette sparer overheaden av to modusskifter. En annen fordel er at tidsstyring av trådene kan skreddersys (være spesifikt tilpasset) den enkelte applikasjonen. For eksempel vil noen applikasjoner ha fordel av en round-robin-algoritme mens andre vil rangeres etter prioritet. I tillegg kan ULTer kjøres på ethvert OS. Det kreves ingen endringer i den underliggende kjernen for å støtte ULTer.

Ulemper

ULT har mange *blokkerende systemkall*. Det vil si at når ULT utfører et systemkall vil den tråden og alle andre tråder i prosessen blokkeres. En annen ulempe er at ved kun bruk av ULT *kan man ikke bruke multiprosessering* fordi kjernen kun kan gi en prosess til prosessoren av gangen, kun multiprogrammering kan benyttes.

Måter å overkomme ulempene ved ULT

Begge ulempene kan overkommes ved å skrive applikasjonen som flere prosesser, istedenfor flere tråder. Denne metoden eliminerer fordelen med tråder: at ethvert bytte blir et prosessbytte istedenfor trådbytte – noe som **resulterer i en stor overhead**.

Blokkerende systemkall kan løses ved å benytte en teknikk som heter *jacketing*. Hensikten med jacketing er å konvertere blokkerende systemkall til ikke-blokkerende systemkall via trådbiblioteket. For eksempel, istedenfor å direkte kalle en I/O-rutine, kan tråden kalle en I/O jacket-rutine på applikasjonsnivå. I denne rutinen er det kode som sjekker om I/O-enheten er opptatt. Dersom den er det, går tråden i blocked state og gir kontroll til en annen tråd.

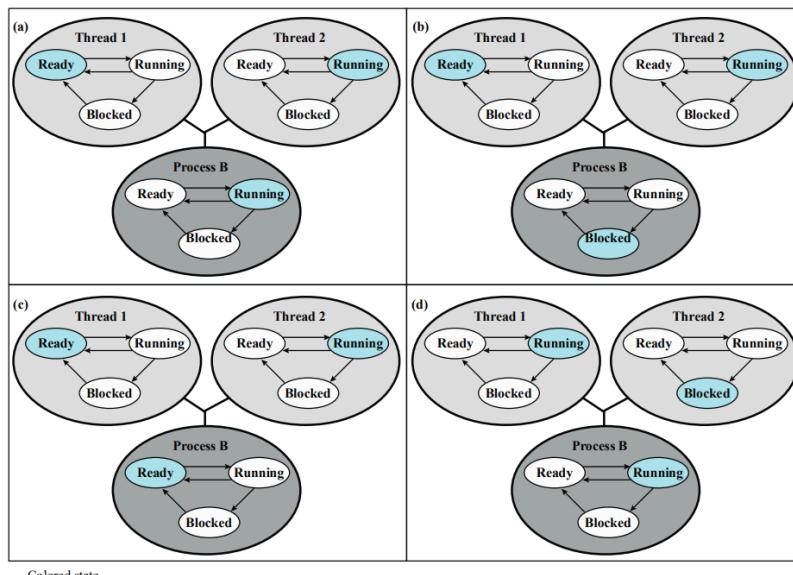


Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

Kernel-level Threads (KLT)

KLT er det motsatte av ULT, som vil si at all trådhåndtering gjøres av kjernen. Det er ikke noe trådhåndtering på applikasjonsnivå, kun et API for kjernetråd-fasilitering. Windows er et eksempel på et KLT-system. Kjernen opprettholder kontekstinformasjon for prosessen som en helhet og for de individuelle trådene i prosessen. Scheduling av kjernen gjøres på en trådbasis.

Fordeler

Fordelene med KLT er blant annet at man kan utnytte multiprosessorer. Siden tidsstyring gjøres på trådbasis kan kjernen sette opp flere tråder fra samme prosess på ulike prosessorer. I tillegg er det ikke blokkerende systemkall, som vil si at dersom en tråd er blokkert så kan kjernen sette opp en annen tråd fra samme prosess til å kjøres.

Ulempor

Ulempene med KLT er hovedsakelig 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.

I tabellen under ser man at det er en betydelig økt hastighet ved å bruke KLT multithreading i forhold til single-threaded processes, men at ULT gir en ekstra økt hastighet i tillegg til dette. Hvilk type som er raskest i praksis avhenger av applikasjonen. Hvis de fleste trådskiftene krever aksess til kjernemodus, kan det være at KLT gir bedre ytelse enn ULT.

Table 4.1 Thread and Process Operation Latencies (μs)

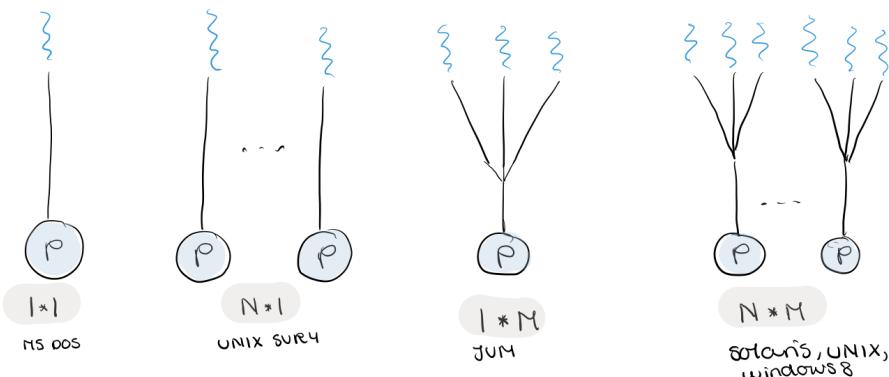
Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Kombinasjon av ULT og KLT

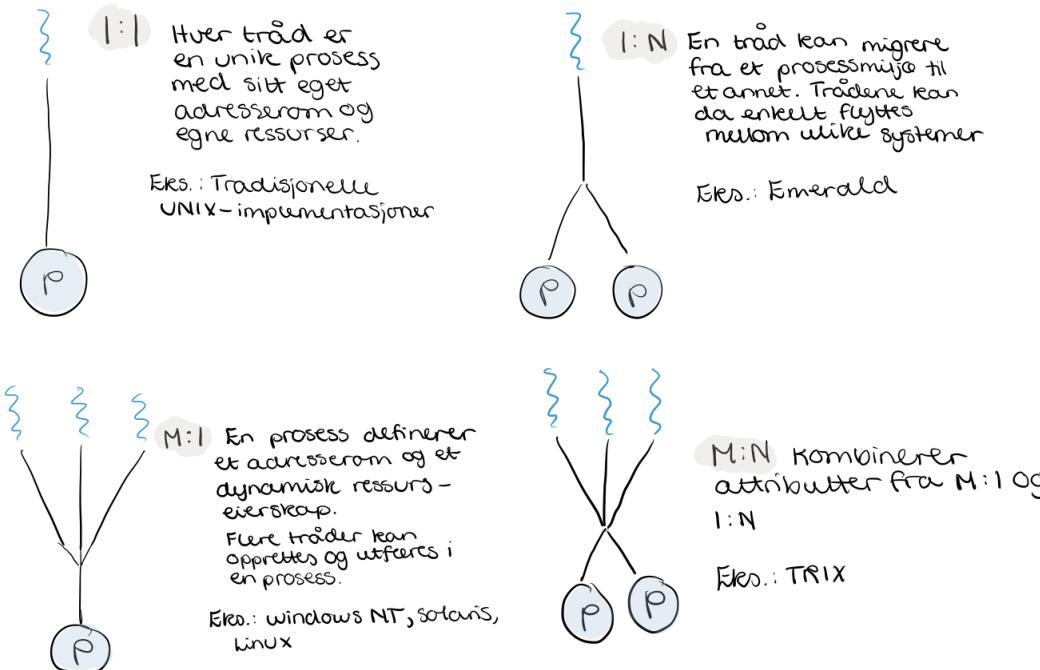
I et kombinert system er hele opprettelsen av tråder gjort i brukerrommet, som det meste av tidsstyring og synkronisering i applikasjonen. ULTene fra en enkelt applikasjon mappes til et (mindre eller likt) antall KLTer. Antall KLTer kan tilpasses for en bestemt applikasjon og prosessor for å oppnå de beste helhetlige resultatene.

Med kombinasjon kan flere tråder fra samme applikasjon kjøre i parallel på flere prosessorer, noe som løser problemet med blokkerende systemkall. Denne metoden kan kombinere fordelene og unngå ulempene av KLT og ULT dersom den designes riktig.

Enkle kombinasjoner



Avanseerte kombinasjoner



M:N

Mange-til-mange forhold har blitt utforsket i OSet 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 stack, tilstand og informasjon om scheduling.

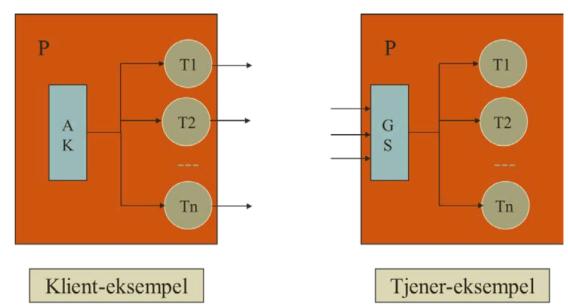
Flere tråder kan eksekveres innenfor et enkelt domene som gir effektivitetsfordeler. I tillegg er det mulig for en applikasjon å utføres i flere domener. I dette tilfellet eksisterer det en tråd som kan flyttes fra et domene til et annet.

1:N

Innenfor distribuerte operativsystemer (for distribuerte datasystemer), har det vært interesse for konseptet tråd som primært en entitet som kan flyttes langs adresserom.

Bruk av tråder i prosess-tjener-eksempler

En **klientapplikasjon** kan sette i gang flere parallele aktiviteter ved å operere med en separat tråd/prosess for hver av dem. En **tjenerapplikasjon** kan også respondere med flere parallele 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.



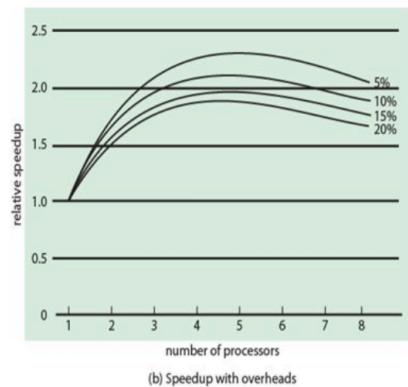
Multikjerne

Software pådrar seg typisk en ekstra overhead som et resultat av kommunikasjon og distribusjon av arbeid på flere prosessorer samt cache-overhead. Dette resulterer i en kurve der ytelsen peaker og deretter synker litt. Det eksisterer dog flere systemer som har stort utbytte av multicore, som for eksempel database management systems og flere typer servere som håndterer flere relativt uavhengige transaksjoner i parallel.

Amdahls lov gir mulig speedup for multithreading kjørt på flere kjerner:

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} = \frac{1}{(1-f) + \frac{f}{N}}$$

Der f er andelen kode som er uendelig paralleliserbar. Med $f = 0.9$ på 8 kjerner er ytelsesfordelen på 4.7.



Fordeler

Fordelene ved å bruke multikjerne er økt ytelse ettersom at CPU-ene kan ha ulike oppgaver. Det gir også reell parallelitet og økt pålitelighet. Hvis en CPU dør, kan de andre fungere som backup

Ulemper

Hovedulempen ved å bruke multikjerner er kompleks programvare. I tillegg resulterer det i et større ytelsesgap ettersom at lageret/minnet er tregt, mens CPU er raskt. Totalt sett vil det dermed ikke nødvendigvis gi økt ytelse, som for eksempel ved mange I/O-operasjoner til minnet.

Nye utfordringer

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

Mikrokjerne

Fordeler

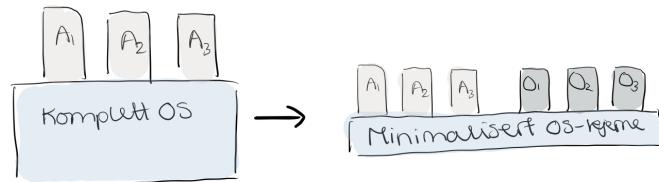
Fordelene ved bruk av mikrokjerne er en enklere programvare som er lettere å tilpasse, utvide og flytte (som nevnt tidligere). Dette støtter også en mer distribuert og objektorientert tankegang.

Ulemper

Eksisterende systemer må endres, og det kan være unødvendig for mindre systemer.

Nye utfordringer

- Riktig å skille mellom kjernekomponenter og komponenter utenfor kjernen.
- *Effektivt* samarbeid mellom kjernekomponenter og komponenter utenfor kjernen.



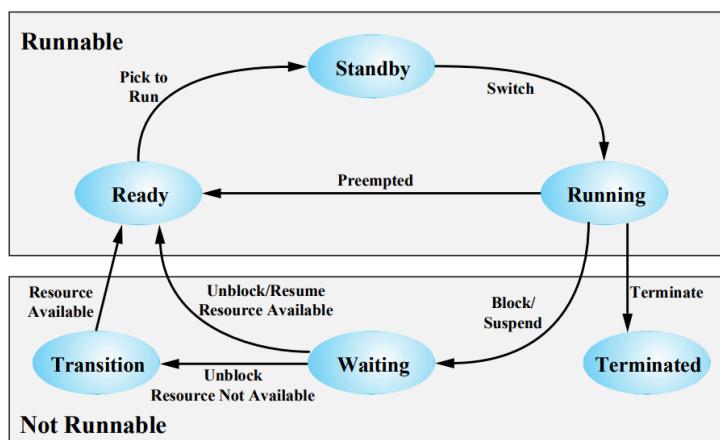
Prosess- og trådhåndtering for de ulike operativsystemene

Windows 8

Windows bruker moderne prosesshåndtering med **tråder**, **multiprosessering** og **mikrokjerner**. Strukturen til Windows er **objektorientert** med prosesser og tråder implementert som objekter med tilhørende attributter. En prosess kan opprettes som en ny prosess, eller som en kopi av en eksisterende prosess. En prosess inneholder en eller flere tråder, der både prosesser og tråder har innebygde **synkroniseringsmuligheter**.

Windows støtter samtidig utførelse blant prosesser fordi tråder i ulike prosesser kan utføres samtidig. I tillegg kan tråder fra samme prosess allokeres til ulike prosessorer og utføres samtidig. En multithreaded prosess oppnår samtidighet uten overhead av flere prosesser.

Tilstander



Tidlig UNIX

Tidligere UNIX-systemer tillot at felles data ble brukt uten mye kontroll på tvers av prosesser, i motsetning til prosess-tråd-modellen. I tillegg fantes det ikke tråder, kun prosesser i tidlig

UNIX, mens i prosess-tråd-modellen reserverer en prosess plass som tildeles trådene tilhørende prosessen.

Solaris UNIX

Solaris bruker også moderne prosesshåndtering med **tråder**, **multiprosessering** og **mikrokjerne**. Trådstrukturen til Solaris er **grensesnittbasert** med tre ulike nivå med tråder som er definert under. ULT-grensesnittet har et standard trådbibliotek og mappes til LWP (Leight-weight processes) som styres av OS og har definerte tilstander for utførelse. LWP er bundet til en kjernetråd med en-til-en-korrespondanse i tilstander. Dermed kan vi si at samtidskjøring og utførelse er håndtert på kjernenivået. I tillegg har applikasjonen tilgang til hardware gjennom et API for å utføre privilegerte oppgaver som read/write til fil, danne nye prosesser, allokkere minne osv.

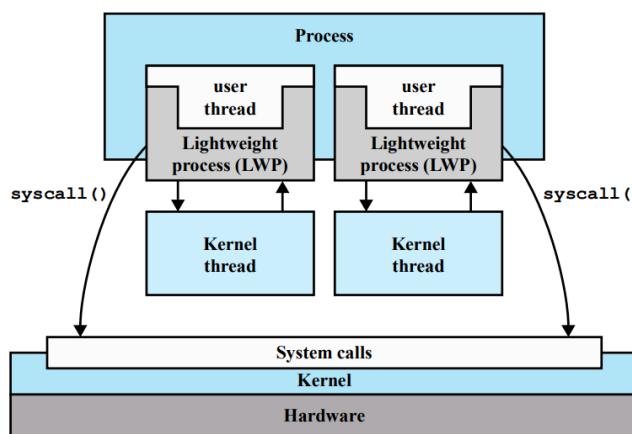


Figure 4.12 Processes and Threads in Solaris

Prosesser og tråder i Solaris

Prosess	Vanlig UNIX-prosess; inkluderer bruker-adresserom, stack og prosesskontrollblokk.
User-level threads (ULT)	Implementeres gjennom trådbibliotek i adresserommet til prosessen (usynlig for OS).
Lightweight processes (LWP)	Kan sees på som mappingen mellom ULT og KLT. Hver LWP støtter ULT og mapper til en kjernetråd. Disse planlegges av kjernen og kan kjøres i parallel på multiprosessorer.
Kernel threads (KLT)	Grunnleggende entiteter som kan kjøres på systemprosessorene. Det eksisterer én kjernetråd for hver LWP.

LINUX

Linux har delvis arvet fra UNIX med moderne prosess- og trådkonsepter og har en unik løsning som **ikke skiller mellom tråder og prosesser**. Ved å bruke en mekanisme lignende LWP i Solaris, mappes ULT 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; som filer, signalbehandlere og virtuelt minne. Når to prosesser deler virtuelt minne, *fungerer* de som tråder innenfor en bestemt prosess.

Til hver prosess i Linux er det assosiert et **navnerom**. 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 illusionen at de er den eneste prosessen i systemet, noe som støtter utnyttelse av multikjerner.

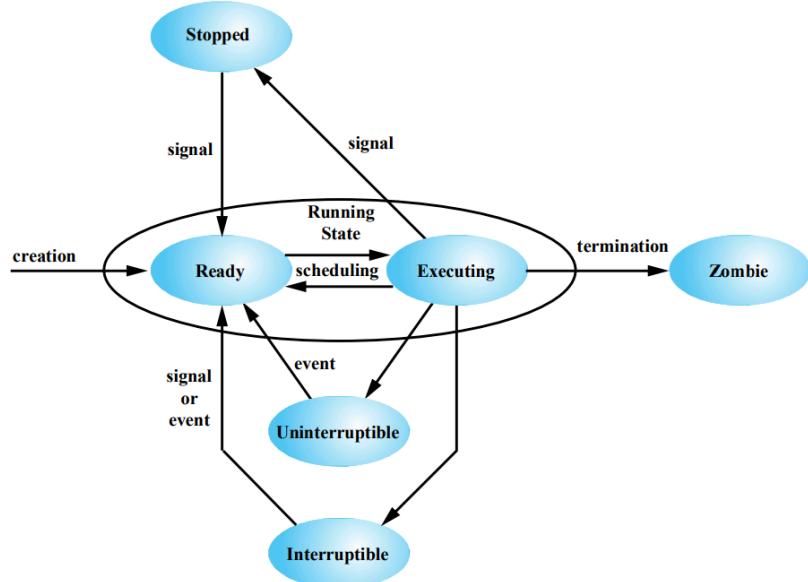


Figure 4.15 Linux Process/Thread Model

Android

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

En aktivitet samsvarer med et vindu synlig i UI som brukeren ofte kan interagere med. Eksempel: e-post-applikasjonen kan ha et vindu som viser listen av nye e-poster som en aktivitet. Aktivitetene ordnes i en LIFO-kø, med de nyeste brukte aktivitetene øverst.

Services

En Service utfører lang-tids operasjoner i bakgrunnen, og har ikke et UI. Dette sørger for raskere responstid fra main-tråden som brukeren direkte interagerer med.

Content providers

En content provider fungerer som et grensesnitt til applikasjonsdataen for å f.eks. hente/lagre data.

Broadcast receivers

Broadcast receivers responderer til system-brede kringkastingskunngjøringer. Dette kan være fra andre applikasjoner som f.eks. skal gi beskjed om at data har blitt lastet ned, som er tilgjengelig for applikasjonen å bruke.

Hver applikasjon kjører i en bestemt prosess som omfatter applikasjonen og en egen dedikert **virtuell maskin** for utnyttelse av multikjerner. Dette **isolerer** hver applikasjon slik at den ikke kan aksessere ressursene til en annen uten tilgang. Hver applikasjon fungerer som en separat Linux-bruker.

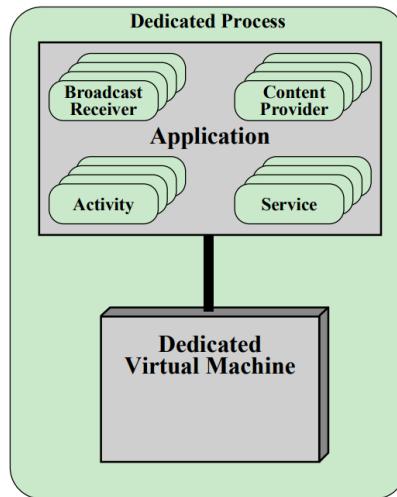


Figure 4.16 Android Application

Prosesser og Tråder

Hver applikasjon allokeres en prosess og en tråd til å begynne med som komponentene kjører i. Utvikleren kan danne flere tråder innenfor en prosess og/eller flere prosesser i applikasjonen for økt ytelse. I hvert tilfelle kjører alle prosessene og deres tråder til en applikasjon i den samme virtuelle maskinen. For å frigjøre minne i et system kan man drepe prosesser. Det er bestemt et presedenshierarki for hvilke prosesser som skal termineres først. I synkende orden er dette; forgrundprosesser, synlige prosesser, serviceprosesser, bakgrunnsprosesser og tomme prosesser.

Mac OS

MacOS 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)**.

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 enkelt utvidelse av Celler andre språk som C++ og har som hensikt å definere en selvstendig enhet for arbeid. Blokker lar programmereren innkapsle komplekse funksjoner sammen med argumenter og data slik at de kan refereres og sendes rundt i programmet som variabler. Blokkene legges i en FIFO-kø.

Bruken av forhåndsdefinerte tråder gir bedre ytelse for prosessering av blokker. Størrelsen på tråd-poolene defineres av systemet for å maksimere ytelse av applikasjonen ved bruk av GCD og minimere tråder som går på tomgang.

Kapittel 5: Samtidighet: gjensidig utelukkelse og synkronisering

Synkronisere

Å synkronisere prosesser er å ordne tiden til hendelsene i de forskjellige prosessene, spesielt for å sikre **gjensidig utelukkelse** mellom flere prosesser ved adgang til felles ressurser eller mer generelt ved for eksempel **kommunikasjon mellom prosesser**. Dette sikrer konsistent oppførsel og hindrer at parallell utføring av programmer/ delprogrammer gir gale resultater slik at enkeltoprasjoner kan ordnes i riktig rekkefølge/ at ulike sett med operasjoner kan utføres uten overlapp.

Samtidighet

- **Multiprogramming** – Handtering av flere prosesser i et uniprosessorsystem. Innebærer ikke reell parallelitet, men introduserer samme problemer som reell parallelitet, altså multiprosessering.
- **Multiprosessering** – Håndtering av flere prosesser i en multiprosessor. Reell parallelitet.
- **Distribuert prosessering** – Håndtering av flere prosesser som kjører på flere, distribuerte datasystemer. *Går ikke i dybden i dette temaet.*

Samtidighet: (concurrency) omfatter en rekke designproblemer, inkludert kommunikasjon mellom prosesser, deling av ressurser (som minne, filer og I/O-aksess), synkronisering av flere prosessers aktiviteter og allokering av prosessortid til prosessene. Samtidighet oppstår i kontekster som:

Multiprogrammerte applikasjoner: Deler prosesseringstiden dynamisk mellom et antall prosesser.

Strukturerte applikasjoner: Noen applikasjoner kan effektivt programmeres med prosesser som kjører samtidig, som en utvidelse til prinsippene om modulært design og strukturert programmering.

OS-struktur: De samme strukturelle fordelene gjelder for systemer. OS-er er i seg selv ofte implementert som prosesser eller tråder.

Nøkkeltermer relatert til samtidighet

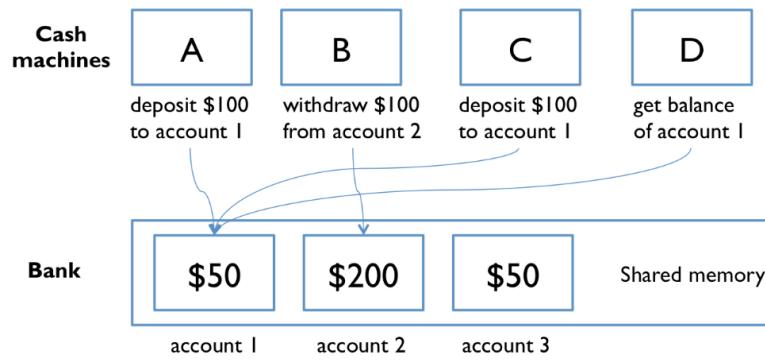
Atomisk operasjon	En eller flere instruksjoner som fremstår udelelig. En operasjon er atomisk hvis ingen eller hele instruksjonssettet utføres. Atomisitet garanterer isolasjon fra samtidige prosesser.
Kritisk seksjon	En beskyttet seksjon der delte ressurser kan aksesseres av én prosess av gangen.
Deadlock	To eller flere prosesser venter på at hverandre skal gjøre noe, som fører til at ingen kan gjøre noe.
Livelock	To eller flere prosesser endrer tilstanden sin kontinuerlig i respons til endring i de andre prosessene uten å gjøre nyttig arbeid.
Gjensidig utelukkelse (mutual exclusion)	Kravet at når en prosess er i en kritisk seksjon som aksesserer delte ressurser, skal ingen andre prosesser være i en kritisk seksjon som aksesserer noen av de delte ressursene.
Race condition	Flere tråder eller prosesser leser og skriver til delte data og det endelige resultatet avhenger av den relative tiden de utføres på.
Starvation	En situasjon der kjørbare prosesser oversees av scheduleren; selv om prosessen er klar for å kjøres, velges den aldri til å utføres.

Utfordringer ved multiprogrammering og multiprosessering

Problemet for multiprogrammering og multiprosessering stammer fra at man ikke kan vite den relative hastigheten til utførelsen av en prosess. Den vil avhenge av andre prosessers aktiviteter, hvordan OS håndterer avbrudd og tidsstyringen til OS. Utfordringer som oppstår er blant annet:

- Når flere prosesser leser og skriver til den samme globale variablene, som gjør at rekkefølgen av utførelse blir kritisk.
- Det blir vanskelig for OS å håndtere allokering av ressurser optimalt → deadlock.
- Det blir vanskelig å lokalisere error fordi oppførselen typisk ikke er deterministisk og reproducable.

Eksempel



Utfordringer knyttet til OS

OS må holde orden på de forskjellige prosessene, noe som gjøres ved hjelp av prosesskontrollblokker. Oppgaven til OS er å tildele ressurser til aktive prosesser. Disse ressursene inkluderer

- Prosessortid (tidsstyring)
- Minne (Virtuelt minne)

- Filer
- I/O-enheter

I tillegg må OS beskytte data og fysiske ressurser for hver prosess mot forstyrrelser fra andre prosesser. Funksjonen til en prosess og resultatet den produserer må være *uavhengig* av hastigheten til utførelsen i forhold til hastigheten til andre prosesser.

Former av parallellitet – *interaksjon mellom prosesser*

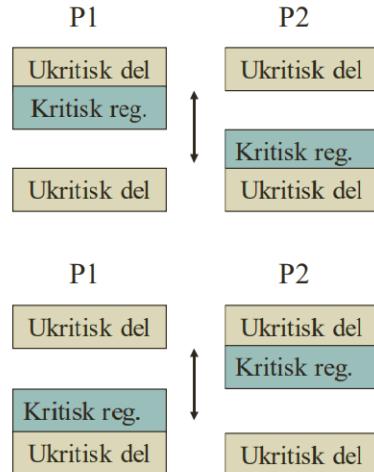
Type	Beskrivelse	Figur	Potensielle problemer
Uavhengige	Uavhengig.	(P) (P)	
Konkurranse	<p>Resultatet av en prosess er uavhengig av handlingen til andre prosesser. Timingen til prosessen kan bli påvirket.</p> <p>Prossessen vet ikke om de andre prosessene og må dermed ikke endre tilstanden på delte ressurser.</p>		Gjensidig utelukkelse, Deadlock (renewable resource), Starvation
Samarbeid via deling	<p>Resultatet av en prosess kan avhenge av informasjon opprettholdt av andre prosesser. Timingen til prosessen kan bli påvirket.</p> <p>Data kan aksesseres gjennom read/write, og kun write krever gjensidig utelukkelse.</p>		Gjensidig utelukkelse, Deadlock (renewable resource), Starvation, Dataavhengigheter
Samarbeid via kommunikasjon	Resultatet av en prosess kan avhenge av informasjon opprettholdt av andre prosesser. Timingen til prosessen kan bli påvirket.		Deadlock (consumable resource), Starvation

Prossesser vil ofte ha aspekter fra både konkurranse og samarbeid.

Gjensidig utelukkelse

Krav for gjensidig utelukkelse

1. Kun en prosess har tilgang til den kritiske seksjonen av gangen. Det vil si at **kritiske regioner må ikke overlappe.**
2. **Ikke forhindre andre;** En prosess som stopper i sin ikke-kritiske seksjon, må gjøre det uten å forstyrre andre prosesser.
3. Det skal ikke være mulig for en prosess som krever tilgang til en kritisk seksjon å måtte vente evig; **ingen deadlock eller starvation.**
4. Når ingen prosess er i en kritisk region, skal enhver prosess som ønsker aksess få det **uten forsinkelse.**
5. **Ingen antagelser** skal gjøres om den relative hastigheten til prosesser eller om antall prosesser.
6. En prosess forblir i den kritiske seksjonen kun i en **begrenset tid.**



Gjensidig utelukkelse: Hardware-basert løsning

Forhindring av avbrudd (interrupt disabling)

I et *uniprosessor-system* er det tilstrekkelig å **forhindre avbrudd** i en prosess for å garantere gjensidig utelukkelse. Denne formen kan defineres i OS-kjernen for å slå på og av avbrudd. Metoden har en høy kostnad fordi kjernen ikke kan flette prosesser. Det vil heller ikke fungere for en multiprosessor fordi flere prosesser uansett kan kjøre samtidig på ulike prosessorer – et tilfelle hvor å slå av avbruddsmekanismen ikke vil hjelpe.

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Special Machine Instruction

På hardware-nivå, vil aksess til en minnelokasjon ekskludere enhver annen aksess til den samme lokasjonen. Dette danner grunnlaget for maskininstruksjoner som utfører to handlinger atomisk, som read/write eller read/test. Ved utførelse av instruksjonen, blokkeres tilgang til minnelokasjonen for enhver annen instruksjon som refererer til denne lokasjonen.

Fordeler

Fordelen med å forhindre avbrudd er at det både fungerer for uni- og multiprosessorer som deler primærminne. Metoden er enkel, noe som også gjør den lett å verifisere. Den tøtter flere kritiske seksjoner fordi hver kritisk seksjon kan definieres som en egen variabel.

Ulemper

Ulemper med å forhindre avbrudd er at en prosess blir satt i *busy waiting* når den venter på tilgang til kritisk seksjon, slik at den fortsetter å oppta prosessortid. Det er også mulig med utsulting (starvation). Når en prosess avslutter kritisk seksjon og mer enn en prosess venter er valg av prosess tilfeldig, som kan resultere i at en prosess aldri får tilgang. Dersom en innfører prioritet av prosesser er også deadlock mulig. Generelt sett er ikke forhindring av avbrudd en god metode.

To av de vanligste implementerte instruksjonene er *Compare & Swap* og *Exchange instruction*:

Compare&Swap Instruction (CAP)

Compare-and-swap (CAP) er en *atomisk* operasjon på to steg (**sammenligning** og et **bytte**) og er brukt i multithreading for å oppnå synkronisering. Atomisiteten garanterer at den nye verdien er beregnet basert på oppdatert informasjon – hvis verdien har blitt oppdatert av en annen tråd i mellomtiden, vil write-operasjonen feile. Instruksjonen sjekker en minneadresse (***word**) mot en testverdi (**testval**). Hvis verdien på adressen er **testval**, byttes den ut med **newval**, ellers forblir den uendret. Den gamle verdien returneres alltid, og man vet at verdien er endret dersom den har samme verdi som **testval**.

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

Exchange Instruction

Instruksjonen bytter innholdet i et register med innholdet i en minneadresse.

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

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

(a) Compare and swap instruction

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Figur 5.2 (a): En delt variabel **bolt** er initiert til 0. Prosessen kan kun gå inn i den kritiske seksjonen hvis **bolt** = 0. Alle andre prosesser som forsøker å gå inn i den kritiske seksjonen går i en **busy waiting**-modus. I denne modusen blir prosessen satt på vent med en kontinuerlig funksjon som sjekker om den kritiske seksjonen er «ledig». Når en prosess går ut av den kritiske sonen settes **bolt** til 0 igjen og den neste ventende prosessen kan gå inn i kritisk seksjon.

Figur 5.2 (b): En delt variabel **bolt** er initialisert til 0. Hver prosess bruker en lokal variabel **key** som initiert til 1. Den eneste prosessen som kan gå inn i den kritiske seksjonen er den som finner **bolt** = 0. Hvis **bolt** = 0 er én prosess i den kritiske seksjonen – den som har **key** = 0.

$$bolt + \sum_i key_i = n$$

Semaforer

Merk at det er mye mer kode i boken på semaforer. Blant annet eksempel på dårlig bruk.

Semafor

En semafor er en heltallsverdi som brukes for å sende signaler via prosesser. Tre *atomiske* operasjoner kan utføres på en semafor: initialiser, decrement og increment. Decrement-operasjonen kan resultere i blokking av prosessen, mens inkrement-operasjonen kan resultere i avblokking av prosessen. Semaforer er lavnivå → lett å gjøre feil!

- **initialisering** settes til et ikke-negativt tall.
- **semWait** reduserer verdien.
- **semSignal** øker verdien.

Hvis prosessen venter på andre prosesser som skal fullføres kan de kalle **semWait(s)**. Hvert tall **s** reduseres med representerer antall prosesser som venter på en ressurs. Prosessen de venter på kan kalle **semSignal(s)** for å øke **s**, slik at en prosess kan utføres.

I figuren antas semWait og semSignal å være atomiske.

```
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 */
    }
}
```

Binær semafor

En binær semafor er en semafor som kun tar 1 og 0-verdier.

- **Initialisering** setter semaforen til 0 eller 1.
- **semWaitB** sjekker semaforverdien. Hvis den er 0 er prosessen blokkert. Hvis den er 1 skiftes verdien til 0 og prosessen fortsetter utførelsen.
- **semSignalB** sjekker om noen prosesser er blokkert på denne semaforen. Hvis det er det, avblokkes en prosess. Hvis ingen prosesser er blokkert, settes semaforen lik 1.

```
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(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */
        /* place process P on ready list */;
    }
}
```

Mutual exclusion lock (mutex) Mutex er et binært flagg som brukes til å låse og frigi et objekt. Den settes til låst (typisk 0) dersom et objekt som ikke kan deles er tilegnet en prosess, og dermed blokkeres fra andre som ønsker å bruke det.

Forskjellen mellom mutex og binære semaforer er at prosessen som låser mutoken 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.

Strong semafores er semaforer med FIFO-kø. Prosessen som har vært blokkert lengst kommer ut av køen først. Dette hindrer utsulting (starvation).

Weak semafores er semaforer som ikke spesifiserer rekkefølge på prosessene og dermed kan resultrere i starvation.

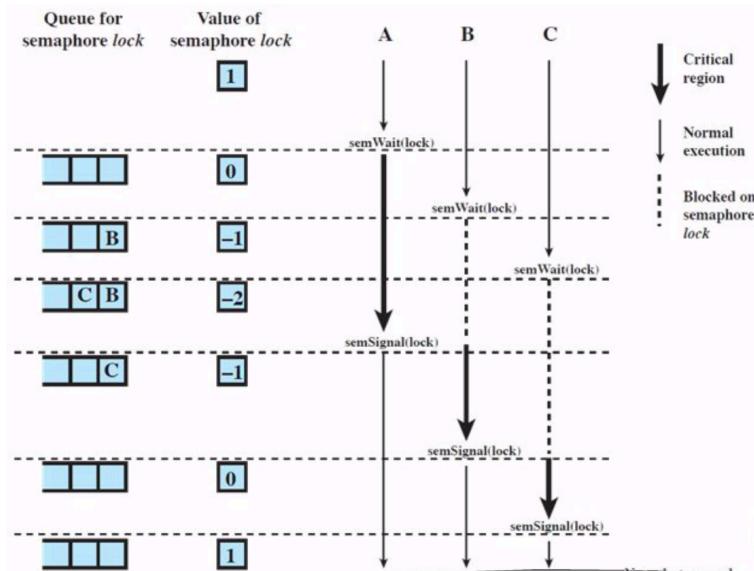
Gjensidig utelukkelse med semafor

Vi ser på n prosesser identifisert i listen P(i) som har behov for samme ressurs. For hver prosess utføres en semWait(s) før den kritiske seksjonen. Hvis s blir negativ, blokkeres prosessen. Hvis verdien er 1, reduseres den til 0 og prosessen inntar den kritiske seksjonen. Siden s ikke lenger er positiv vil ingen andre prosesser kunne gå inn i den kritiske seksjonen.

```
/* 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));
}
```

Figur: Processes Accessing Shared Data Protected by a Semaphore

Merk at en normal utførelse kan fortsette i parallel, men at kritiske regioner serialiseres.



The Producer/Consumer problem

En eller flere prosesser, **producers**, genererer data til en buffer, mens en prosess, **consumer**, tar dataen ut av bufferen og forbruker den. De kommuniserer via en kø av maksimal størrelse N og har begrensningene:

- Kun én bufferoperasjon kan skje av gangen.
 - Consumer må vente på producer hvis køen er tom.
 - Producer må vente på consumer hvis køen er full.
- De to nederste går på synkronisering.*

Semaforløsningen til problemet benytter tilstanden til to semaforer: e: antall tomme plasser i køen, og n: antall elementer i køen. Semaforen s sikrer at integriteten til køens tilstand ikke er i fare, for eksempel ved at to producers forsøker å legge til objekter til en tom kø samtidig.

Eksempelkode:

1. En consumer går inn i kritisk seksjon. Siden **n** = 0, blokkeres consumeren.
2. Flere producers går inn i den kritiske seksjonen for producers. Maksimalt **sizeofbuffer** kan produsere.
3. En producer av ganger får aksess til bufferen via **s** og legger inn data i køen.
4. Så fort den første produceren går ut av den kritiske seksjonen sin, økes **n** og et signal sendes slik at en consumer kan gå inn i kritisk seksjon.

```
/* 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);
}
```

For semaforer er *både* ansvaret for gjensidig utelukkelse og synkronisering lagt på programmereren.

The Barbershop Problem

Monitorer

Monitor

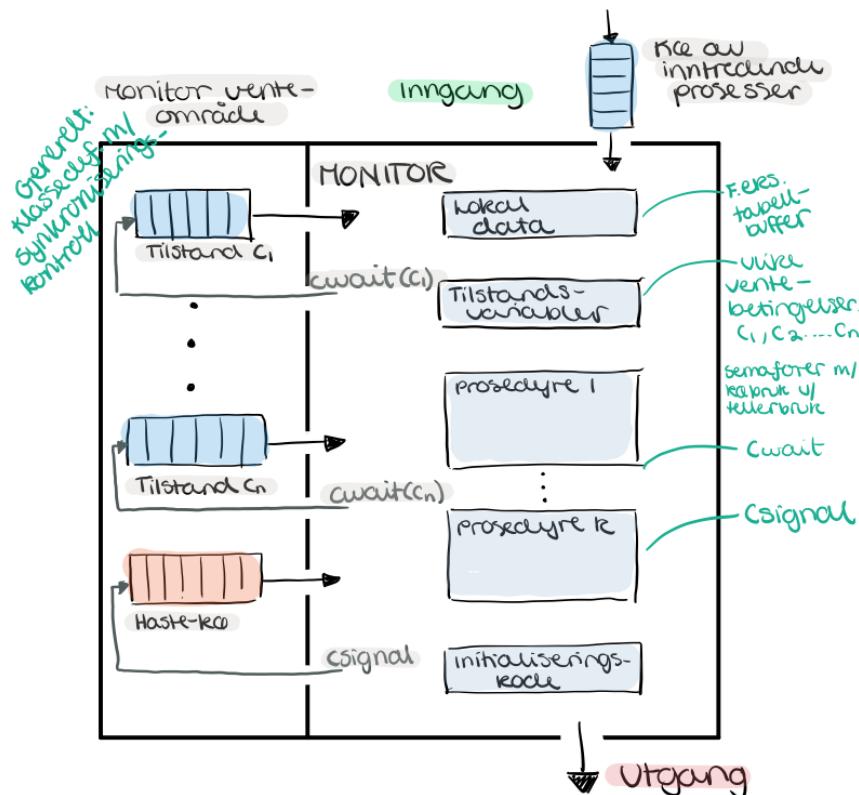
En monitor er en del av et programmeringsspråk som innkapsler variabler, aksesserer prosedyrer og initierer kode innenfor en abstrakt datatype. Vi kan derfor se på monitorer som en utvidelse av klassekonseptet. En monitors variabler kan kun aksesseres gjennom bestemte aksessprosedyrer og kun en prosess kan aksessere en monitor av gangen. Dette sikrer gjensidig utelukkelse mellom metoder som utføres på vegne av prosesser. Aksessprosedyrene er *kritiske sekSJoner*, og en monitor kan opprettholde en kø med prosesser som venter på aksess. Monitorer er mer *høy-nivå* enn semaforer, og det er derfor lettere å vise korrekthet.

Monitor med signal

Hovedegenskapene til en monitor er følgende:

- 1) de lokale datavariablene kan kun aksesseres ved prosedyrene til monitoren.
- 2) En prosess inntrer monitoren ved å benytte en av dens prosedyrer/metoder.
- 3) Kun én prosess kan utføres i monitoren av gangen → **Automatisk gjensidig utelukkelse**.

De første to punktene minner om objektorientert struktur, og kan enkelt implementeres i et objektorientert språk ved å legge til litt ekstra egenskaper.



For å kunne utføre parallelle prosesser, må monitoren ha synkroniseringsverktøy. Dette gjøres ved bruk av **tilstandsvariabler** som aksesseres via monitoren. Det tilbys metoder som la en prosess synkronisere seg selv (**cwait** – midlertidig venting) og en annen/andre prosesser (**signal** – oppstart igjen) når betingelser tilsier behov for det.

- **cwait(c)** suspenderer den kallende prosessen på tilstand **c**. Monitoren er da ledig for en annen prosess.
- **csignal(c)** fortsetter utførelsen av en prosess blokkert av **cwait** på den samme tilstanden. Hvis det er flere slike prosesser, velg en av dem; hvis det ikke er en slik prosess, gjør ingenting.

Merk at disse funksjonene er forskjellig fra semaforer. Hvis en prosess i en monitor kjører signal og ingen oppgave venter på tilstandsvariabelen, er signalet tapt.

Når en prosess er i monitoren kan den midlertidig blokkere seg selv ved å kalle **cwait(x)** på tilstanden **x**; den vil da plasseres i en kø med prosesser som venter på at tilstanden til **x** skal endres. Hvis en prosess som kjører i monitoren oppdager en endring i tilstandsvariabelen **x** påkaller den **csignal(x)**, som gir beskjed til den korresponderende køen.

```
/* 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 */
}
{
    nextin = 0; nextout = 0; count = 0;        /* monitor body */
    /* 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);
}
```

Monitormodulen **boundedbuffer** kontrollerer bufferen som brukes til å lagre data. Monitoren inkluderer to buffervariabler (deklarert med **cond**): **notfull** som er **True** når det er plass til minst en ekstra data i bufferen, og **notempty** som er **True** når det er minst en data i bufferen. En producer kan kun legge til data ved å bruke **append**-funksjonen til monitoren. Produceren sjekker først tilstanden **notfull** om det er plass i bufferen. Hvis det ikke er det, blokkeres prosessen på denne tilstanden. En annen prosess kan nå bruke monitoren. Senere når bufferen ikke lenger er full kan den blokkerte prosessen fjernes fra køen for å fortsette prosesseringen. Etter å plassere data i bufferen kaller prosessen **csignal(notempty)**. En lignende forklaring kan gjøres for consumerfunksjonen. Utfordringen ligger i å plassere **cwait** og **csignal** riktig innenfor monitoren slik at prosesser

ikke forsøker å hente fra en tom buffer eller legge til i en full buffer. Ansvaret for synkronisering ligger på programmereren, men gjensidig utelukkelse skjer automatisk.

Fordeler med monitor over semafor

En fordel med monitorer over semaforer er at synkroniseringsfunksjonene er begrenset til monitoren. Derfor er det lettere å verifisere at synkroniseringen er gjort riktig og å detektere feil. Dessuten har monitorer automatisk gjensidig utelukkelse, mens for semaforer må alle prosesser som aksesserer ressursen være riktig programmert.

Monitor med Notify og Broadcast (Mesa)

Monitor med signal krever at dersom det er minst en prosess i tilstandskøen, må en prosess fra den køen kjøre umiddelbart når en annen prosess kaller **csignal** for tilstanden. Derfor må prosessen som kaller **csignal** enten ut av monitoren umiddelbart, eller blokkeres. Dette introduserer to ulemper:

1. Hvis prosessen som kaller **csignal** ikke er ferdig, resulterer det i to ekstra prosessbytter: en for å blokkere prosessen og en for å fortsette den når monitoren er tilgjengelig.
2. Scheduling av prosesser assosiert med et signal må være 100% pålitelig. Når **csignal** er påkalt, må en prosess fra tilstandskøen aktiveres umiddelbart og schedulern må sørge for at ingen andre prosesser inntrer monitoren før aktivering. Hvis ikke kan tilstanden endres. For eksempel i koden over: når **csignal(notempty)** kalles vil en prosess fra **notempty**-køen aktiveres før en ny consumer inntrer monitoren.

Løsningen er en *Mesa-monitor-struktur*. I denne strukturen er **csignal** byttet ut med **cnotify**. **cnotify(x)** vekker tilstandskøen til **x**, men signalprosessen fortsetter utførelsen. Resultatet er at prosessen foran i tilstandskøen fortsetter når monitoren blir ledig igjen. Ettersom at det ikke er noen garanti for at en annen prosess inntrer i monitoren før dette, må den ventende prosessen sjekke tilstanden på nytt. Prosedyrene i **boundedbuffer** vil nå ha koden under. If-setningen er byttet ut med en **while**-løkke som gir minst en ekstra evaluering av tilstandsvariabelen, og ingen ekstra prosessbytte eller restriksjon for når den ventende prosessen skal kjøre etter **cnotify**.

```
void append (char x)
{
    while(count == N) cwait(notfull); /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    cnotify(notempty);
}

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

For å hindre starvation av en prosess dersom en annen prosess feiler før den rekker å signalisere introduseres et timeout-intervall. En prosess som har ventet så lenge blir aktivert uavhengig av den har blitt signalisert etter timeout. Prosessen sjekker da tilstandsvariabelen og fortsetter dersom den er tilfredsstilt.

Med notifikasjon over tvangsreaktivering kan en ny funksjon legges til: cbroadcast. Dette plasserer alle prosesser som venter på en tilstand i Ready-tilstand. Dette er nyttig når en ikke vet hvor mange prosesser som trengs å reaktiveres, eller når prosessen ikke vet akkurat hvilken prosess den skal reaktivere. Eksempelvis er en memory manager som har et visst antall bytes ledig, men vil vekke opp mange prosesser slik at de selv kan sjekke om det er nok minne.

Fordeler med Mesa

Denne metoden har mindre error enn *monitor med signal*. Dette fordi den signaliserte prosessen sjekker tilstanden ved while-løkken. En prosess kan signalisere feil uten å skape problemer siden tilstanden sjekkes før den eventuelt fortsetter utførelse. I tillegg er metoden mer modulær da Mesa har fasiliteter for å organisere programmer i moduler som kommuniserer gjennom veldefinerte grensesnitt som prosessene på passer inn i (s.263 eller <https://people.eecs.berkeley.edu/~brewer/cs262/Mesa.pdf> for nærmere forklaring).

Vanlig monitor	Mesa-monitor
<p>Den oppvekkede aktøren (som har kalt på csignal) vil midlertidig tre ut av monitoren slik at den påkalte aktøren (vekket opp) som tidligere har gjort cwait, kan fortsette uten å teste betingelsene på nytt</p> <p>Det manglende behovet for retesting av betingelser er billig for applikasjonene, mens de mange resulterende tråd/prosessskiftene er dyrt for systemet.</p>	<p>Den oppvekkede aktøren (som har kalt cnotify) vil ikke midlertidig tre ut av monitoren for den som tidligere har kalt cwait.</p> <p>Retesting av systemet er dyrt, mens de få resulterende prosess/tråd-skiftene er billig for systemet.</p> <p>Mesa har tillegg en broadcast-funksjon hvor en aktør kan vekke opp alle aktører som venter på tilstanden av gangen. Dette er som nevnt tidligere veldig nyttig ved f.eks. påkalling av prosesser for å sjekke om det er plass i frigitt minne.</p>

Meldinger

Meldinger er en måte for to prosesser å utbytte informasjon og kan benyttes for synkronisering. Meldinger kan benyttes til distribusjon (uten delt lager). Det minimale settet med operasjoner som er nødvendig for meldingskommunikasjon er:

- `send(destination, message)`
- `recieve(source, message)`

Designkarakteristikker for meldingssystemer

Synchronization	Format
Send blocking nonblocking	Content Length fixed variable
Receive blocking nonblocking test for arrival	
	Queuing Discipline
	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** – både sender og mottaker er blokkert inntil meldingen er levert. Kombinasjonen refereres til som *Rendezvous* og gir tett synkronisering mellom prosesser
- **Ikke-blokkerende sender, blokkerende mottaker** – Mottaker blokkeres inntil meldingen er mottatt. Dette er sannsynligvis den mest nyttige kombinasjonen og lar prosesser sende en eller flere meldinger til flere destinasjoner så raskt som mulig. Et eksempel er en server som eksisterer for å gi tjenester til andre.
- **Ikke-blokkerende sender og mottaker** – Ingen av partene trenger å vente.

Den ikke-blokkerende senderen er den mest naturlige i mange samtidsprogrammeringsoppgaver. En potensiell ulempe med den ikke-blokkerende sendfunksjonen er at en error kan medføre en situasjon der prosessen sender gjentatte genererte meldinger, som resulterer i overbruk av systemressurser som prosessortid og bufferplass. I tillegg vil den ikke-blokkerende send-funksjonen plassere ansvaret om bekreftelse på mottagelse av meldinger på utvikleren; acknowledgements.

For mottaker er det vanlig med den blokkerende varianten. Utfordringen her er at dersom en melding er tapt, vil mottakeren risikere å bli blokkert. Dette vil ikke hende med en ikke-blokkerende mottaker. Problemet med den ikke-blokkerende er at hvis en melding er sendt etter at en prosess allerede har utført en matchende receive, vil meldingen gå tapt.

Adresseering

Prosesser kan adresseres ved direkte eller indirekte adressering. Ved **direkte adressering** inkluderer senderen en spesifikk identifikator til destinasjonsprosessen. Ved **indirekte adressering** sendes meldinger via en delt datastruktur som består av køer som midlertidig holder meldinger. Disse køene kalles gjerne mailbokser. En styrke ved indirekte adressering er at det kobler sender og mottaker fra hverandre, noe som gir større fleksibilitet i bruken av meldinger. Forholdet mellom sender og mottaker kan være en-til-en, mange-til-en, en-til-mange eller mange-til-mange.

- **En-til-en** – privat kommunikasjonslink mellom to prosesser.
- **En-til-mange** – lar en sender ha mange mottakere; broadcast for eksempel.
- **Mange-til-en** – nyttig i klient/server-interaksjon; der en prosess gir en tjeneste til et antall prosesser.
- **Mange-til-mange** – lar flere serverprosesser gi samtidige tjenester til flere kunder.

Prosessers assosiasjon til mailbokser kan være **statisk** eller **dynamisk**. Dersom det er statisk er en prosess permanent assosiert til en port. Dette er vanlig for en-til-en-forhold for eksempel.

Dersom det er mange sendere kan dynamisk være mer vanlig, der assosiasjonen til mailboksen oppstår dynamisk. Da er det vanlig å bruke `connect` og `disconnect`.

Gjensidig utelukkelse med meldinger

Koden under er et eksempel på gjensidig utelukkelse med meldinger. Vi antar bruk av blokkerende `recieve` og ikke-blokkerende `send`. Et sett med parallelle prosesser deler mailboks, `box`, som brukes av alle prosesser til å sende og motta. Mailboksen initieres med en tom melding. En prosess som ønsker å gå inn i en kritisk seksjon forsøker først å motta en melding. Hvis mailboksen er tom blokkeres prosessen. Så fort en prosess har tatt til seg en melding, går den inn i kritisk seksjon og plasserer meldingen tilbake i mailboksen. Dermed fungerer den tomme meldingen som en *token* mellom prosesser.

```
/* 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));
}
```

Denne løsningen antar at dersom mer enn en prosess gjennomfører `recieve`-operasjonen samtidig, så:

- Hvis det er en melding, leveres den til kun én av prosessene, og den andre blokkeres.
- Hvis meldingskøen er tom, blokkeres alle prosessene; når en melding blir tilgjengelig, aktiveres kun en av prosessene for å motta meldingen.

Løsning til producer/consumer-problemet med meldinger

Koden under sender data og signaler som meldinger. To mailbokser brukes. Ettersom at en producer genererer data, sendes dette som meldinger til mailboksen `mayconsume`. Så lenge det er minst en melding i mailboksen, forbruker consumeren. `mayconsume` har dermed rollen som *buffer*; bufferen er organisert som en meldingskø, og størrelsen er gitt av `capacity`. Til å begynne med er mailboksen `mayproduce` fylt opp med antall null-meldinger lik kapasiteten. Antall meldinger i `mayproduce` reduseres for hver production og øker med hver consumption.

```

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);
}

```

Denne metoden er veldig *fleksibel*. Det kan være flere producers og consumers, så lenge alle har tilgang til begge mailboksene. Systemet kan også være distribuert, med alle producer-prosesser og **mayproduce**-mailboksen på en side, og alle consumer-prosessene og **mayconsume**-mailboksen på den andre.

Readers/Writers problem

Det er et omeråde med data, delt av mange prosesser. Dataområdet kan være en fil, en main memory block, eller en samling av prosessorregister. Det er et antall prosesser som kun kan lese data, readers, og et antall prosesser som kun kan skrive data, writers. Følgende betingelser gjelder:

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

Det vil si at readers er prosesser som ikke trenger å ekskludere noen prosesser, mens writers er prosesser som skal ekskludere alle prosesser.

NB: på side 270 står det tydelig hva som skiller de ulike generelle problemene.

Begge løsningene under bruker semaforer.

Readers har prioritet

Løsningen viser en instans for reader og en for writer, men løsningen er den samme ved bruk av flere. Semaforen **wsem** brukes for gjensidig utelukkelse. Så lenge en writer aksesserer det delte dataområdet, kan ingen andre aksessere det. Reader bruker også **wsem** for gjensidig utelukkelse. For å tillate flere readers av gangen, kreves det at når det ikke er noen readers som leser – må den første må vente på **wsem**. Når det allerede er en reader som leser, trenger ikke de resterende readersene å vente.

Writers har prioritet

Problemet med forrige løsning er at den kan medføre starvation for writers. Følgende løsning løser dette.

- En semafor **rsem** hemmer alle readers så lenge det er en writer som vil aksessere dataen.
- En variabel **writecount** kontrollerer innstillingen av **rsem**.
- En semafor **y** kontrollerer oppdateringen av writecount.

I tillegg er det en ekstra semafor, **z**, for at ikke rsem skal bli full; slik at writers skal få plass. **z** er en kø for readers, umiddelbart før de venter i **rsem**. Tabellen under gir alle mulighetene.

Readers only in the system	<ul style="list-style-type: none"> • wsem set • no queues
Writers only in the system	<ul style="list-style-type: none"> • wsem and rsem set • writers queue on wsem
Both readers and writers with read first	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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;
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);
}

/* 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);
}
```

Løsning ved bruk av meldinger – writers har prioritet

Her er det en kontrollerer-prosess som har aksess til det delte dataområdet. Andre prosesser som ønsker aksess til dataområdet sender en forespørsel til kontrolleren og får aksess ved en «OK»-melding som avsluttes ved en «finished»-melding tilbake. Kontrolleren har tre mailbokser, en for hver type melding som den kan motta.

Kontrolleren fullfører write-requester før read-requester. I tillegg på gjensidig utelukkelse opprettholdes. Dette gjøres ved bruk av **count**-variabelen som initialiseres til et tall større en maksimalt antall readers. I dette eksempelet bruker vi 100.

- Hvis count > 0, så er det ingen writer som venter. Det kan være readers som er aktive → Vent på alle «finished»-meldinger fra readers, deretter prioriter write-requests og til slutt read-requests.
- Hvis count = 0, så er den eneste gjenstående forespørselen en write-forespørsel → La writeren fortsette og vent på «finished»-meldingen.
- Hvis count < 0, så har en writer sendt en forespørsel og er satt på vent for å klarere alle aktive readers. Derfor skal kun «finished»-meldinger behandles.

<pre>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); } }</pre>	<pre>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++; } } }</pre>
--	--

Sammenligning mellom semaforer, monitorer og meldinger

Semaforer, monitorer og meldinger kan alle løse synkroniseringsproblemer, men passer bedre/dårligere for ulike formål eller typer problemer. Funkjsonelt er semaforer og monitorer likeverdige, og kan implementeres sammen. Semaforer mer generelle som vil si at de kan brukes til et bredt spekter av problemer, men det er 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 at monitorer har automatisk gjensidig utelukkelse er den spesielt bra for problemer hvor gjensidig utelukkelse er et essensielt behov. Meldinger kan også brukes uten delt lager, som vil si ved distribusjon og er særlig bra dersom generell tidsstyring er et essensielt behov.

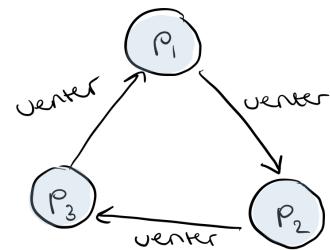
Kapittel 6: Samtidighet: deadlock og starvation

Starvation eller utsulting er en situasjon hvor en kjørbar prosess oversees av schedulern; selv om den kan kjøres, velges den aldri.

Livelock er en situasjon der en eller flere prosesser kontinuerlig endrer tilstanden sin i respons til endring i tilstanden til andre prosess(er) uten å gjøre nyttig arbeid.

Deadlock

En **deadlock**/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 vranglåst dersom hver prosess i settet er blokkert ventende på en hendelse som kan trigges av en annen prosess. Det er ingen generell løsning for deadlock.



Deadlock er et resultat av prosesssynkronisering (kontroll av ressurser / komunikasjon). Ulike synkroniseringsverktøy kan gjøre det enklere eller vanskeligere å håndtere vranglås, men i utgangspunktet må en legge til egne mekanismar for vranglåshåndtering: umuliggjøre, unngå, eller oppdage og rette opp.

Ved deadlock gjelder

Gjensidig utelukkelse	En ressurs kan kun tildeles en prosess av gangen.
Hold & wait	En prosess som holder en ressurs kan be om flere ressurser.
Ingen avbrudd	Prosesser kan ikke miste tilgang til en ressurs mot sin vilje.
Sirkulær venting	<p>Det må være en sirkulær kjede av to eller flere ressurser, som venter på en ressurs holdt av den neste prosessen i kjeden.</p> <p>Merk at dette kan være en konsekvens av de tre foregående punktenene som også gjør at deadlocken holder.</p>

Fatal region

Deadlock er kun uungåelig hvis to (eller flere) prosesser danner en sti som inngår den fatale regionen. Eksistensen av en fatal region er avhengig av logikken til de to prosessene. Se eksempel s. 292.

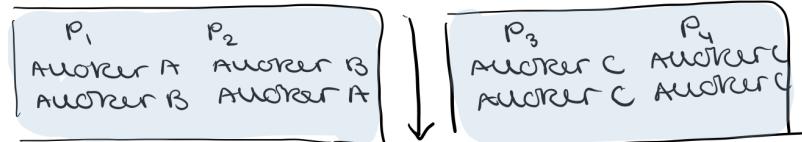
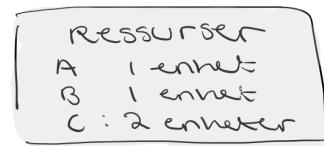
Fornybare ressurser

En **fornybar ressurs** er en ressurs som kan benyttes sikkert av én prosess av gangen, og som ikke blir oppbrukt. Eksempler: I/O-kanaler, main memory, sekundærminne, filer, databaser og semaforer.

*Eksempel: Vranglås – fornybare ressurser
Begge problemene kan håndteres **automatisk**.*

Løsning A og B

Man kan ordne rekkefølgen ressursene allokeres (f.eks. at A må allokeres før B) →
Innefektivt fordi en ofte må holde ressursene lengre enn nødvendig.



Løsning C

Del ut 2 enheter av gangen til en prosess
→ Samme problem (ineffektivt).

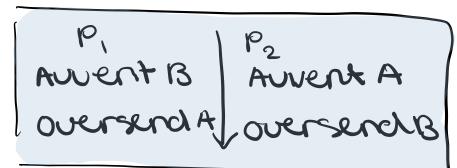
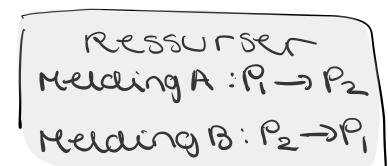
Konsumerbare ressurser

Konsumerbare ressurser er en som kan opprettes og forkastes etter bruk. Typisk er det ingen begrensning i mengde og en ikke-blokkert prosess kan opprette flere slike ressurser.
Eksempler: avbrudd, signaler, meldinger og informasjon i I/O-buffere.

Eksempel: Vranglås – konsumerbare ressurser

*Problemet må håndteres av **programmerer**.*

Deadlock oppstår dersom receive er blokkerende og receive skjer før send. Receive er blokkerende når den mottakende prosessen er blokkert inntil meldingen er mottatt.



Angrepsmåter

Metode	Grad	Ordninger	Fordeler	Ulemper
Umuliggjøre	Konservativ; lite parallelitet; forplikter ressurser.	Allokerer alle ressurser samtidig	Fungerer bra for prosesser som utfører en enkelt «burst activity». Ingen fortrinnsrett nødvendig	Ineffektivt Forsinker prosessinitiering. Fremtidige behov for ressurser må være kjent av prosesser.
		Ingen fortrinnsrett: fjernes av systemet	Nyttig ved bruk på ressurser som enkelt kan lagre og hente tilstand.	Gir fortrinnsrett til nye prosesser oftere enn nødvendig.
	Rekkefølge av ressurser	Kan opprettholdes ved å legge inn sjekker ved kompileringstid. Trenger ikke run-time- kompilering, siden	Forhindrer trinnvis ressurs- etterspørsel.	

			problemet løses i systemdesignet.	
Unngå	Mellomvei mellom umuliggjøring og oppdaging. Mer parallelitet.	Manipulering for å finne minst én trygg sti.	Ingen fortrinnsrett nødvendig.	Fremtidige behov for ressurser må være kjent av OS. Prossesser kan blokkeres for lengre perioder.
Oppdage og rette opp <i>Velges bl.a. hvis vranglås skjer sjeldent.</i>	Liberal; etterspurte ressurser gis tilgang der det er mulig	Legg inn sjekker periodisk som oppdager vranglåser.	Gir ikke forsinkelse i prosessinitiering. Fasiliterer online håndtering.	Iboende forutsetningstop? (Inherent preemption losses)

Umuliggjøre deadlock

Strategien for å umuliggjøre deadlock er å designe et system slik at muligheten for at det skal oppstå ikke er til stede. En indirekte metode er å forhindre en av de tre nødvendige punktene i tabellen under deadlock. En direkte metode er å forhindre sirkulær venting (punkt 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. Noen ressurser, som filer, kan gi aksess til flere prosesser ved lesing, men ikke for skriving.

Hold & Wait

At en ressurs holder på en ressurs og ber om en annen, kan forhindres ved å kreve at en prosess etterspør tilgang til alle ressursene den behøver på en gang og deretter blokkere prosessen inntil den har fått tilgang til alle ressursene samtidig. Dette er ueffektivt på to måter. 1) En prosess er blokkert mens den venter på alle ressurser, når den i mange tilfeller kunne fortsatt med ressursene den allerede har fått tilgang til. 2) Flere av ressursene allokkert til en prosess kan stå ubrukt i en periode, mens andre prosesser kunne utnyttet de. Et annet problem er at en prosess må vite på forhånd alle ressursene den trenger.

Ingen avbrudd

Denne tilstanden kan forhindres på flere måter.

1. En prosess som holder på en ressurs og får avslag på en annen ressurs, må frigjøre den ressursen den holder, og om nødvendig, be om tilgang igjen sammen med den ekstra ressursen.
2. Dersom en prosess etterspør en ressurs som holdes av en annen prosess, kan OS ta rettighetene fra den andre prosessen og kreve at den frigjør ressursen. Denne metoden forhindrer kun deadlock så lenge det ikke er to prosesser som besitter den samme prioriteten.

Denne metoden er kun praktisk når tilstanden til ressursene lett kan lagres og innhentes (som med prosessoren).

Sirkulær venting

Sirkulær venting kan forhindres ved å definere en **lineær ordning** av ressurstypene. Hvis en prosess har blitt allikert en ressurs av typen R, kan den i ettertid kun etterspørre ressurser av typen som kommer etter R i ordningen. Ulempen med dette er at det er ueffektivt ved at det senker farten til prosesser og unødvendig benekter tilgang til ressurser.

Unngå deadlock

Uunngåelse av deadlock tillater de tre tilstandene som kan medføre deadlock (i tabellen), men gjør kloke valg for å sørge for at deadlock ikke oppstår. Dette tillater høyere grad parallelitet. En beslutning tas dynamisk om en ressursallokering kan, dersom den gis, potensielt lede til deadlock. Dette krever derfor kunnskap om prosessers fremtidige behov for ressurser.

Det er to metoder for å unngå deadlock:

- Ikke start en prosess hvis prosessens behov kan medføre deadlock.
- Ikke innvilg en trinnvis ressurs-eterspørsel til en prosess hvis allokeringen kan medføre deadlock.

Avslag på prosessinitiering

Aksepter ny prosess: hvis problem aldri kan oppstå.

Ta i betrakting et system med n prosesser og m ulike typer ressurser. Vi definerer R som settet med alle ressurser i systemet, V som settet med antall av hver ressurs som ikke er allokkert til en prosess, C som matrisen med behov av prosess i for ressurs j, og A som matrisen med nåværende allokeringer til prosess i av ressurs j (s.301). Matrisen C har en rad for hver prosess og viser det maksimale kravet for hver ressurs. Denne informasjonen må erklæres på forhånd for å unngå deadlock. Følgende forhold må gjelde:

1. $R_j = V_j + \sum_{i=1}^n A_{ij}$ for alle j] Alle ressurser er enten tilgjengelige eller allokkert.
2. $C_{ij} \leq R_j$ for alle i,j] Ingen prosess kan kreve mer enn det totale antallet ressurser i systemet.
3. $A_{ij} \leq C_{ij}$ for alle i,j] Ingen prosess allokkeres flere ressurser enn den originalt krevede.

Enhver prosess får ikke starte dersom ressurskravene kan medføre deadlock. En prosess startes *kun* dersom det maksimale antallet krav av alle nåværende ressurser pluss kravet til den nye ressursen kan møtes. Det vil si:

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

Strategien er ikke optimal siden den antar at alle prosesser gjørde maks krav samtidig.

	R1	R2	R3	
P1	3	2	2	
P2	6	1	3	
P3	3	1	4	
P4	4	2	2	

	R1	R2	R3	
P1	1	0	0	
P2	6	1	2	
P3	2	1	1	
P4	0	0	2	

	R1	R2	R3	
P1	2	2	2	
P2	0	0	1	
P3	1	0	3	
P4	4	2	0	

(a) I initial state

	R1	R2	R3	
P1	3	2	2	
P2	0	0	0	
P3	3	1	4	
P4	4	2	2	

	R1	R2	R3	
P1	1	0	0	
P2	0	0	0	
P3	2	1	1	
P4	0	0	2	

	R1	R2	R3	
P1	2	2	2	
P2	0	0	0	
P3	1	0	3	
P4	4	2	0	

(b) P2 runs to completion

	R1	R2	R3	
P1	0	0	0	
P2	0	0	0	
P3	3	1	4	
P4	4	2	2	

	R1	R2	R3	
P1	0	0	0	
P2	0	0	0	
P3	2	1	1	
P4	0	0	2	

	R1	R2	R3	
P1	0	0	0	
P2	0	0	0	
P3	1	0	3	
P4	4	2	0	

(c) P1 runs to completion

motsatte, men merk at en utrygg tilstand \neq deadlock. Bankers algoritme tester en tilstand om den er sikker ved å simulere alloksering av de forhåndsbestemte allokseringene før den eventuelt kan fortsette.

	R1	R2	R3	
P1	0	0	0	
P2	0	0	0	
P3	0	0	0	
P4	4	2	2	

	R1	R2	R3	
P1	0	0	0	
P2	0	0	0	
P3	0	0	0	
P4	0	0	2	

	R1	R2	R3	
P1	0	0	0	
P2	0	0	0	
P3	0	0	0	
P4	4	2	0	

Figur 6.7 og 6.8 er forklart på s.304 og forklarer bankers algoritme.

(d) P3 runs to completion

I figur 6.8 etterspør P1 en ekstra enhet av R1 og R3. Denne tilstanden er *utrygg* fordi hver prosess ønsker en ekstraenhet av R1, som ikke er tilgjengelig. Dermed vil P1 sin forespørsel om tilgang benektes og P1 blokkeres.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2		5	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector **R**

	R1	R2	R3
	1	1	2

Available vector **V**

(a) Initial state

(b) P1 requests one unit each of R1 and R3

Figure 6.8 Determination of an Unsafe State

Figur 6.9 viser hvordan unngåelse av deadlock kan kodes. (b) viser hoved-algoritmen med tilstanden til systemet definert av **state**, mens **request[*]** er en vektor som definerer ressursene som er etterspurt av prosess i.

Først gjøres en sjekk om ressurs-forespørselen ikke er større enn kravet til prosessen. Hvis forespørselen er gyldig er neste steg å bestemme om det er mulig å innfri forespørselen (om det er tilstrekkelig med ressurser). Hvis det ikke er mulig, suspenderes prosessen. Hvis det er mulig er siste steg å bestemme om det er trygt. For å gjøre dette tentativt tildeles ressurser til prosess i for a danne **newstate**. Til slutt testes dette med algoritmen i (c).

Fordelene med unngåelse av deadlock er at du ikke trenger å fjerne eller rulle tilbake prosesser og at den er mindre konservativ og dermed mer effektiv enn umuliggjøring av deadlock.

Ulempene er at 1) prosessers maksimale krav på ressurser må vites på forhånd, 2) prosesser under vurdering må være uavhengige; dvs. At rekkefølgen de gjennomføres skal være ubetydelig, 3) Det må være et bestemt antall ressurser å allokere, og 4) Ingen prosess kan avsluttes mens den holder ressurser.

```
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,*])           /* total request > claim */
    < error >;
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 allocation algorithm

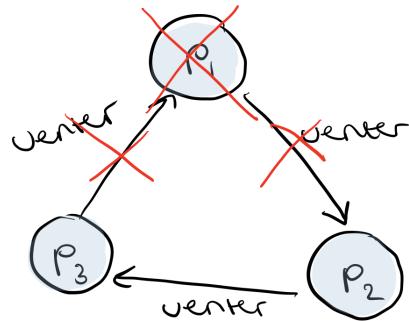
```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process P_k in rest such that
            claim [k,*] - alloc [k,*] <= currentavail[*]
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

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

Figure 6.9 Deadlock Avoidance Logic

Oppdag og rette opp deadlock

Oppdagelse av deadlock setter ikke grenser for ressursaksesser eller prosesshandlinger. Med oppdagelse av deadlock tildeles etterspurte ressurser til prosesser når det er mulig. OS gjennomfører en periodisk algoritme som sjekker for sirkulær venting (tilstanden som fører til deadlock). Sjekken kan gjøres så frekvent som for hver ressursforespørrelse eller mindre frekvent, avhengig av hvor sannsynlig det er at en deadlock oppstår. Hyppigere sjekker medfører tidligere deteksjon av deadlock i tillegg til at algoritmen er enkel, men krever mye prosessortid.



Oppdag deadlock

Finn ut om det eksisterer deadlock i systemet, og løs den opp hvis det er tilfelle. Søk gjøres periodisk. Utfør gjenopprettning.

Algoritmen er detaljert beskrevet på side 307.

Gjenopprettning

1. Aborter alle prosesser i deadlock.
 2. Back up hver deadlocked prosess til det forrige definerte check-pointet.
 3. Suksessivt aborter prosesser i deadlock til det ikke lenger eksisterer deadlock.
 4. Suksessivt fjern tilgang på ressurser til det ikke lenger eksisterer deadlock.
- Deteksjonsalgoritmen burde kjøres på nytt etter dette steget. En prosess som blir nedprioritert for en ressurs må bli backedup til før den fikk tilgang.

For trinn 3 og 4 velges prosesser ut fra at prosessen har:

- Minst prosessortid så langt.
- Minst produsert output.
- Mest gjenværende tid.
- Færrest totale ressurser allokeret.
- Lavest prioritet.

Integrt deadlock-strategi

Ofta er det mest effektivt å integrere de ulike metodene i designet av OS for ulike situasjoner. En metode: Grupper ressurser inn i et antall ulike klasser. Bruk en lineær ordningsstrategi for å forhindre sirkulær venting mellom ressursklasser. Bruk algoritmen som passer seg best innenfor de enkeltvise ressursklassene. Et eksempel er å dele inn i:

Swappable space

Består av blocker med sekundærminne, brukt for prosesser som byttes ut. Her brukes forhindring av deadlock ved å kreve at alle ressursene som skal brukes allokeres på en gang. Denne strategien er rimelig dersom de maksimale lagringskravene er kjent, noe det ofte er. Å unngå deadlock kunne også vært brukt her.

Process resources

Ressurser som kan tildeles (som filer, båndstasjoner osv). Her er den mest effektive metoden ofte å unngå deadlock, fordi det er rimelig å tro at prosesser oppgir hvilke ressurser de trenger på forhånd i denne klassen. Unngåelse ved å ordne ressurser er også mulig.

Main memory

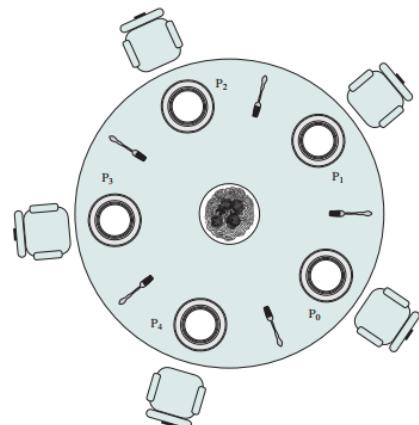
Her benyttes unngåelse av fortrinnsrett ofte. Når en prosess har mistet allokering til en adresse, flyttes den til main memory for at ikke deadlock skal oppstå.

Internal resources

Inkluderer bl.a. I/O-kanaler. Her kan unngåelse i form av ordning av ressurser brukes.

Dining philosophers problem

Det er fem filosofer som bor i et hus der et bord (se figur) er satt til dem. For at en filosof skal kunne spise, trenger hun to gaffler. Problemet er å lage en algoritme som lar filosofene spise. Algoritmen må tilfredsstille gjensidig utelukkelse (to filosofer kan ikke bruke den samme gaffelen samtidig) samtidig som algoritmen må utelukke deadlock og starvation.



Løsning med semaforer

En løsning er å kun tillate fire filosofer til bordet av gangen. Da vil minst en filosof ha tilgang til to gaffler. Denne løsningen forhindrer deadlock og starvation.

```

/* 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()
{
    par begin (philosopher (0), philosopher (1), philosopher (2),
               philosopher (3), philosopher (4));
}

```

Løsning med monitorer

En vektor med fem tilstander, én per gaffel, defineres. Tilstandsvariablene lar filosofer vente på at gaffelen blir ledig. I tillegg defineres en boolsk vektor som holder tilgjengelighetsstatusen til hver gaffel (true betyr at gaffelen er ledig). Monitoren består av to prosedyrer. Get_fork-prosedyren brukes av en filosof for å gripe venstre og høyre gaffel. Dersom en av gaflene er opptatt, plasseres filosofen i en kø på den passende tilstandsvariabelen. Dette gjør at en annen filosof-prosess kan entre monitoren. Release-fork-prosedyren brukes for å gjøre to gafler tilgjengelig. Løsningen gir ikke deadlock fordi kun en prosess av gangen kan være i monitoren (og ta to gafler). Det vil si at den første filosofen som entrer monitoren er garantert å 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 */
    }
}

```

Samtidighet for de ulike operativsystemene

Her er det hovedsakelig underoverskriftene som er viktig. Beskrivelsen er mest for forståelsen sin del. Det stilles som regel ikke detaljerte spørsmål om de ulike OS-ene på eksamen.

UNIX

UNIX støtter fleremekanismer for interprosessor-kommunikasjon og synkronisering. Det viktigste er pipes (rør), meldinger, delt minne, semaforer og signaler, der pipes, delt minne og meldinger kan brukes til å kommunisere data mellom prosesser, mens semaforer og signaler brukes til å trigge handlinger av andre prosesser.

Pipes; begrenset bufrende rør

Pipes er en sirkulær buffer som lar to prosesser kommunisere på producer-consumer-modellen. En pipe er defor en FIFO-kø som en prosess skriver til, og en annen leser fra. En prosess som forsøker å skrive til pipen utføres umiddelbart dersom det er plass, hvis ikke blokkeres den. Det samme gjelder for den lesende prosessen (bare med lesing av bytes). 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 enten motta meldinger i FIFO, eller ordne de etter type melding som spesifiseres av sender.

Ukontrollert delt lager

Den raskeste formen for interprosess-kommunikasjon er delt lager. Dette er en blokk med vitituelt minne, som deles av flere prosesser. Tilgangen er read-only eller read-write og bestemmes for hver enkelt prosess. Gjensidig utelukkelse er ikke en del av det delte lageret, og må derfor kontrolleres av prosesser som bruker det delte lageret.

Semaforer

UNIX bruker en generalisert versjon av semWait og semSignal, semaforer. Kjernen utfører alle operasjoner atomisk, som vil si at ingen andre prosesser kan aksessere semaforen før alle operasjonene er fullført. Semaforen har derfor en kø med blokkerte prosesser assosiert til seg. Generaliseringen av semaforer gir fleksibilitet i synkronisering og koordinering av prosesser.

Signaler

Et signal (en bit) er en software-mekanisme som informerer en prosess om forekomsten av asynkrone hendelser. Et signal ligner et hardwareavbrudd, og benytter ikke prioritet. Signalene behandles derfor likt; signaler som oppstår samtidig presenteres en av gangen til en prosess, uten en bestemt ordning. Prosesser kan sende hverandre signaler, eller kjernen kan sende signaler internt. En prosess responderer til et signal ved å utføre default action (f.eks. terminere), utføre en signal-håndteringsfunksjon eller ignorere signalet.

LINUX

LINUX inkluderer alle samtidsmekanismene til UNIX-systemer, som pipes, meldinger, delt lager og signaler. I tillegg støtter Linux en type signaler som heter real-time (RT) signaler. Disse er annerledes enn de vanlige UNIX-operasjonene på tre måter: 1) prioritet ved levering er støttet, 2) Signaler kan vente i kø, og 3) RT-signaler kan inneholde en verdi eller en peker.

Atomiske operasjoner

Linux har et sett med operasjoner som garanterer atomisitet på en variabel. Disse operasjonene kan brukes til å unngå race conditions. Det er to typer atomiske operasjoner i Linux: heltallsoperasjoner som opererer på heltallsvariabler og bitmap-operasjoner som opererer på en bit i et bitmap.

Spinlocks

Spinlocks er den vanligste teknikken i Linux for å beskytte kritiske seksjoner. En spinlock bygges på en heltallslokasjon i minnet som sjekkes av hver thread før den entrer den kritiske seksjonen. Hvis verdien er 0, setter tråden verdien til 1 og går inn i kritisk seksjon. Spinlocks er lett å bruke, men har ulempen at den setter en rekke tråder i busy-wait-modus. Den har også en innretning for lesere og skrivere som tillater høyere grad samtidighet.

Semaforer

På brukernivå implementerer Linux samme semaforgrensesnitt som UNIX SVR4, mens internt har Linux sin egen funksjonalitet for semaforer. Disse semaforene implementeres som funksjoner i kjernen, og er dermed mer effektive enn de på brukernivå. Linux har tre typer

semaforer på kjernenivå: **binære semaforer, generelle (counting) semaforer** og **read-write semaforer**.

Barrierer

I noen arkitekturen omordner kompilatoren og/eller prosessorhardwaren minne-aksesser i kildekoden for å optimalisere ytelse. Hvis man f.eks. skriver $a = 1; b = 1;$ har ikke rekkefølgen på minne-aksess noe å si, men dersom man skriver $a = 1; b = a;$ har det det. Barrierefunksjonen til Linux håndhever at rekkefølgen på minneaksess blir som spesifisert. Barrierefunksjonen relaterer til maskininstruksjonene load og store, og dikterer oppførselen til prosessoren og kompilatoren.

Solaris

I tillegg til samtidighetsmekanismene til UNIX SVR4, støtter Solaris trådsynkringiserings-metodene: gjensidig utelukkelse (MUTEX)-låser, semaforer, flere-lesere-en-skriver (readers /writer)-låser og tilstandsvariabler. Disse benyttes både for kjernetråder, og brukertråder.

Gjensidig utelukkelse (MUTEX)-låser

En mutex-lås brukes for å sikre at kun en tråd av gangen kan aksessere en ressurs. Tråden som låser mutexen må være den samme som låser den opp.

Semaforer

Solaris implementerer den klassiske (counting)-semaforen i tillegg til de to følgende.

Readers/writer-lås

Readers/writers-lås lar flere tråder ha samtidig lese-aksess til et objekt. Låsen lar kun en skriver aksessere objektet av gangen, og ekskluderer derfor alle lesere.

Tilstandsvariabler

En tilstandsvariabel brukes for å vente inntil en bestemt betingelse er sann. Tilstandsvariabler brukes i sammenheng med mutex-lås. Dette implementerer en monitor.

Windows

Hos Windows er prosesser og tråder direkte implementert som **objekter** som har innebygde synkroniseringsmuligheter. De viktigste metodene for synkronisering er Executive dispatcher-objekter, kritiske seksjoner for brukermodus, slanke leser-skriver-låser, tilstandsvariabler og låsbri synkronisering. Dispatcher-objekter benytter seg av ventefunksjoner.

Basis ventefunksjoner

Ventefunksjonene tillater en tråd å blokkere sin egen utførelse og returnerer ikke før et spesifikt kriterie har blitt oppfylt. Typen ventefunksjon bestemmes av kriteriet.

Dispatcher-objekter

Benytter normale synkroniseringsobjekter som mutexer, semaforer osv. listet på s. 326.

Kritiske regioner

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 locks (SWR) lar trader til en enkelt prosess aksessere delte ressurser; de er optimalisert for hastighet og okkuperer veldig lite minne, hvorav ordet *Slank* kommer fra. At den er *slank* betyr at den kun krever allokering av en enkelt del av minne på pekerstørrelse. I tillegg bruker Windows tilstandsvariabler som deklarereres av en prosess, og kan brukes med enten kritiske regioner eller SRW-låser.

Låsfri synkronisering

Windows benytter også låsbri synkronisering i stor grad. Denne typen synkronisering bruker hardware-funksjonalitet til å garantere at en minneadresse kan leses, modifiseres og skrives i en enkelt atomisk operasjon.

Android

Android benytter SVR4/Solaris sine metoder for prosesser og tråder, men legger til en ekstra funksjonalitet i kjernen for interprosess-kommunikasjon, 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 på samme system, men som kjører på ulike virtuelle maskiner.

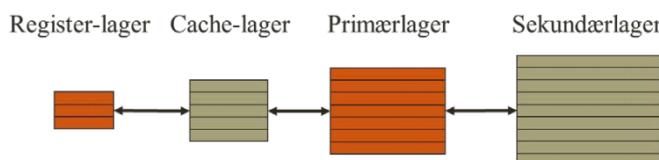
Kapittel 7: Minnehåndtering

Minnehåndtering tilsier å holde oversikt og tilby aksess til ulike deler av kode/data for ulike prosesser/tråder på forskjellige tidspunkt.

I de fleste datamaskiner er prosessorer kritiske ressurser som bør holdes aktive hele tiden. For å få til dette må flere (del)programmer være tilgjengelig for utføring i primærlageret til enhver tid. Dette kan gjøres ved:

1. Holde et mindre antall total-programmer i primærlageret og resten på sekundær-lageret, eller ved å
2. Holde et større antall del-programmer i primærlageret og resten på sekundær-lageret – mens delprogrammer som antas å ikke bli brukt i nærmeste fremtid kan byttes ut med delprogrammer som skal brukes i nærmeste fremtid.

Begge varianter medfører ulike oppgaver som må håndteres av operativsystemet, i form av lagerhåndtering.



- Når flyttes data nedover?
- Når flyttes data oppover?
- Hvor plasseres data?
- Hvor utbyttes data?
- Hvor mye data trengs?
- Hvor mange data tåles?

Krav for minnehåndtering

Minnehåndtering er tiltenkt å innfri følgende krav: **relokalisering, beskyttelse, deling, logisk organisering og fysisk organisering.**

Viktige termer

Frame	En blokk i main memory med bestemt lengde.
Page	En blokk i sekundærminnet (som disk) med bestemt lengde. En page kan midlertidig kopieres til en frame i main memory.
Segment	En blokk av variabel lengde som ligger i sekundærminnet. Hele segmentet kan kopieres til en tilgjengelig del av main memory (segmentering) midlertidig eller så kan segmentet deles opp i flere pages som individuelt kopieres til main memory (kombinert segmentering og paging)

Relokasjon

For å maksimere prosessorutnyttelse er det mulig å bytte prosesser inn og ut av main memory ved å ha en kø med prosesser klar for utførelse. Når et program flyttes til disk må det være mulig å **relokalisere** prosessen til et annet sted i minnet. Det ville vært veldig begrensende om den skulle plasseres på samme adresse i main memory igjen når den hentes inn.

Beskyttelse

Hver prosess burde beskyttes mot uvillig inferens av andre prosesser. Et program i andre prosesser burde derfor ikke ha mulighet til å lese eller skrive uten tillatelse. Alle minnereferanser generert av en prosess må sjekkes ved run time for at de kun refererer til minnerommet alllokert til den prosessen. Merk at dette tilfredsstilles på hardwarenivå, av prosessoren istedenfor av operativsystemet. Dette er fordi OS ikke kan forutse alle minnereferanser et program vil gjøre, og dessuten ville det vært ineffektivt.

Deling

Prossesser som samarbeider på en oppgave eller begge tilhører samme program, kan ha behov for samme datastruktur eller del av main memory. Beskyttelsesmekanismen må derfor ha fleksibilitet nok at flere prosesser kan aksessere samme porsjon av minnet.

Logisk organisering

Primær- og sekundærminnet er organisert som et lineært, en-dimensjonalt adresserom bestående av sekvenser av bytes eller ord. Programmer er derimot ofte organisert som **moduler**. **Segmentering** er en metode slik at operativsystemet og hardwaren håndterer programmer som moduler, og har følgende fordeler:

- 1) Moduler kan skrives og kompileres uavhengig.
- 2) Ekstra beskyttelse (read only, execute only) kan gis til ulike moduler med lite ekstra overhead.
- 3) Det er mulig å introdusere mekanismer slik at moduler kan deles mellom prosesser. Dette korresponderer også med måten brukeren ser problemet, som gjør det lettere for brukeren å spesifisere deling.

Fysisk organisering

Datamaskinens minne organiseres i minst to nivåer, sekundær- og primærminnet. En utfordring med denne ordningen er organiseringen og flyten av informasjon i systemet. Dette ansvaret ligger på systemet, og er essensen med minnehåndtering.

Minnepartisjonering

Den grunnleggende operasjonen til minnehåndtering er å hente prosesser til main memory for utførelse. I nesten alle multiprogrammeringssystemer involverer denne virtuelt minne som baseres på paging og segmentering. Vi skal først se på noen andre teknikker.

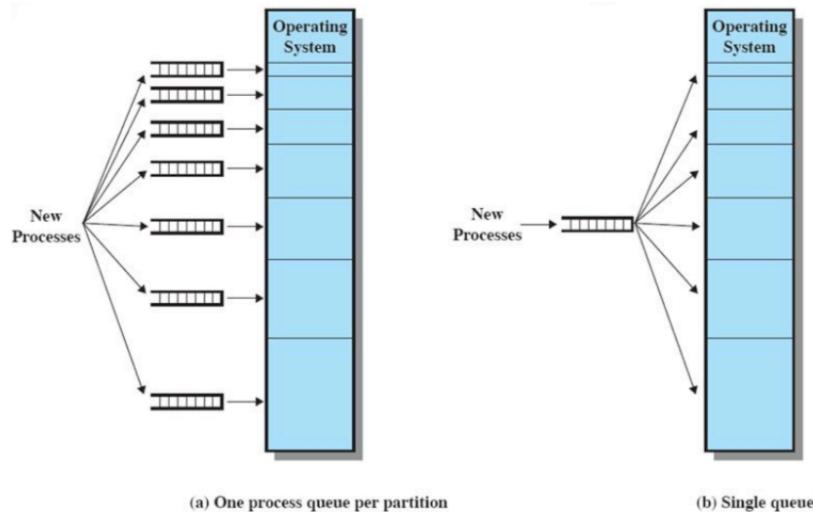
Fixed partitioning

Den enkleste formen for minnehåndtering er å dele minnet inn i begrensede regioner og brukes egentlig *ikke* lenger.

Størrelse: Et alternativ er å benytte samme størrelse på alle regionene, men dette introduserer to problemer. Et program kan være for stort til å plasseres i en region. Da må programmet programmeres slik at kun en del av det kan være i main memory av gangen. I tillegg medfører det intern fragmentering, at ikke all plass utnyttes. Begge disse problemene kan minskes, men ikke løses, ved bruk av ulik størrelse på blokkene.

Plasseringsalgoritme: For blokker med lik størrelse vil en prosess plasseres i main memory, så lenge det er ledig plass. Hvis det er fullt og plassen okkuperes av en prosess som ikke er klar for å kjøres, byttes prosessen ut for å lage rom til nye prosesser. Hvilken prosess som byttes ut er tidsstyrerens ansvar.

For regioner med ulik størrelse er det to måter å fordele minnet: Den enkleste måten er å gi hver prosess plass i den minste tilgjengelige regionen den vil passe. For dette trengs en kø for hver oppdeling som holder på utbyttede prosesser (a). Denne metoden minimerer intern fragmentering, men er ikke optimal for systemet i sin helhet. I tilfellet hvor det kun er små prosesser, vil de store regionene ende opp med å ikke bli brukt, mens mindre prosesser byttes ut kontinuerlig. En mer egnet metode er derfor å ha en enkelt kø for alle prosesser (b). Når en prosess skal lastes inn i main memory velges den minste og tilgjengelige regionen som har plass til prosessen. Dersom det er fullt gjøres en utbytting på størrelse eller f.eks. prioritet.

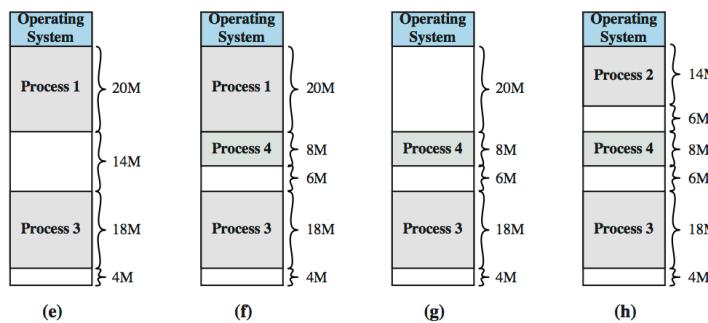
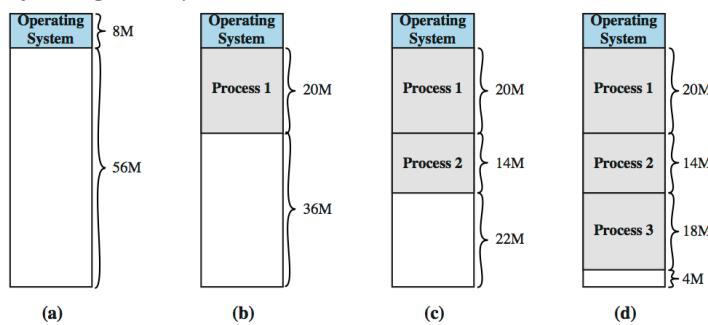


Fordeler med fixed partitioning med ulike blokkstørrelser er at det gir en viss grad med fleksibilitet. I tillegg er metoden relativt enkel og krever minimalt med OS software og prosesseringsoverhead.

Ulemper er at antall oppdelinger begrenser antall aktive prosesser i systemet, i tillegg til at små prosesser ikke utnytter plassen effektivt.

Dynamisk partisjonering

Med dynamisk partisjonering er delene i minnet av variabel lengde og antall. Når en prosess hentes inn til main memory, allokeres den den eksakte mengden minne som kreves. Dette medfører etterhvert **ekstern fragmentering**, som innebærer at minnet blir veldig oppstykket, med en rekke «hull» der ingen prosesser får plass. Se figur. Dette kan løses ved hjelp av en **sammenpressingsteknikk** som til tider skifter prosessene slik at de ligger kontinuerlig etter hverandre. Problemet med dette er at det er tidskrevende, bruker prosessortid og krever en dynamisk relokasjonsegenskap.



Plasseringsalgoritme: De tre plasseringsalgoritmene som brukes er best-fit, first-fit og next-fit.

- **Best-fit:** velger blokken som er nærmest i størrelse. Yter ofte dårligst av de tre da primærminnet ofte ender opp med deler som er for små til å allokerere.
- **First fit:** scanner minnet fra starten og velger den første tilgjengelige blokken som passer. Denne er enklest, ofte best og raskest.
- **Next-fit:** scanner minnet fra forrige plassering og velger den neste tilgjengelige blokken som passer. Denne er ofte mye dårligere enn first-fit, siden den største ledige blokken endes ofte opp med å fragmenteres.

Fordeler Ingen intern fragmentering, mer effektiv bruk av main memory.

Ulemper Ekstern fragmentering og ekstra overhead for sammenpressing.

Buddy-system

I et byddy-system er minneblokker kun tilgjengelig i størrelse 2^K ord, hvor $L \leq K \leq U$. Til å begynne med behandles hele minnet som en blokk av størrelse 2^U . Hvis en aksess forespørsel s er slik at $2^{U-1} < s \leq 2^U$, allokeres hele blokken. Ellers splittes blokken inn i to like «buddies» av størrelse 2^{U-1} . Hvis $2^{U-2} < s \leq 2^{U-1}$, forespørselen til en av «buddiesene». Hvis ikke splittes en av buddiesene igjen til en blokk som tilfredsstiller betingelsen er generert. Figur 7.6 illustrerer hvordan det fungerer. Splittingene er illustrert i figur 7.7

```
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>;
}
```

1 Mbyte block				
1 M				
Request 100 K	A = 128K	128K	256K	512K
Request 240 K	A = 128K	128K	B = 256K	512K
Request 64 K	A = 128K	C = 64K	64K	B = 256K
Request 256 K	A = 128K	C = 64K	64K	B = 256K
Release B	A = 128K	C = 64K	64K	D = 256K
Release A	128K	C = 64K	64K	256K
Release E	512K		D = 256K	256K
Release D				1M

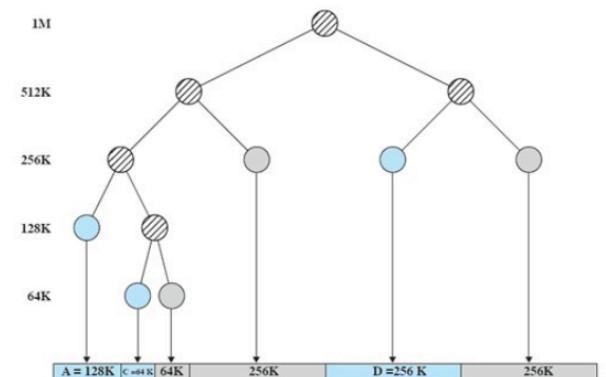


Figure 7.7 Tree Representation of Buddy System

Figure 7.6 Example of Buddy System

Fordeler Systemet har fordelene med fixed- og dynamisk partisjonering.

Ulemper Systemet har med noen ulemper, som intern fragmentering (størrelsen må være 2er-potens).

Relokalisering

Som nevnt tidligere må det være mulig å bytte prosesser ut av main memory for senere å hente de inn igjen. Det vil si at adressen (til instruksjoner og data) referert til av en prosess, ikke er fast. For å løse denne utfordringen benyttes ulike typer adresser:

- **Logisk adresse:** En virtuell adresse brukeren kan se og som dermed brukes som referanse. Denne genereres av programmet ved kjøretid.
- **Relativ adresse:** En relativ adresse er spesifisert ved å indikere avstanden til en annen adresse, ofte prosessor-registret.
- **Fysisk adresse:** Denne refererer til en lokasjon i minneenheten. Brukeren kan ikke se den fysiske adressen.

Det er nødvendig med en hardware-mekanisme som oversetter relative adresser til fysiske.

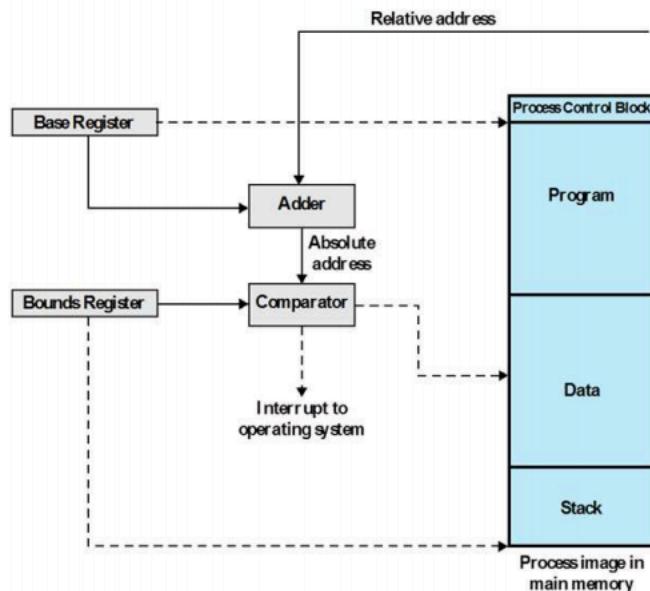


Figure 7.8 Hardware Support for Relocation

Simple paging

Main memory deles inn i mange små frames av samme størrelse, mens prosesser deles inn i pages av samme størrelse som frames. Størrelsen må være i toerpotens da dette gjør at den logiske og den relative adressen er lik. Små prosesser krever færre pages; større prosesser krever flere. Dette ligner på fixed partitioning, men framesene er mindre, et program kan okkupere flere frames uten at bitene er sammenhengende. Når en prosess hentes inn, lastes dens pages opp i tilgjengelige frames i tillegg til at en page-tabell settes opp som holder på adressen til framnen. OS opprettholder en liste med alle frames i main memory som er ledige. Det er ingen ekstern fragmentering, og kun en liten intern fragmentering. Paging er ikke synlig for programmereren.

Simple segmentation

Et program med assosiert data deles inn i et antall segmenter, som kan ha ulike størrelser. Segmentering ligner dynamisk partisjonering, men et program kan okkupere mer enn ett

segment, som ikke behøver å være sammenhengende. Segmentering eliminerer intern fragmentering, men har fortsatt litt ekstern fragmentering. I motsetning til paging er segmentering synlig for programmereren, som dermed må være klar over maksimal segmentstørrelse. En annen konsekvens av dynamiske blokkstørrelser er at det ikke er et enkelt forhold mellom logisk og fysisk adresse. En segmenttabell opprettholdes for hver prosess i tillegg til en liste med ledige blokker i main memory.

Kapittel 8: Virtuelt minne

Virtuelt minne (VM)

Virtuelt minne kan sees på som et *konsept* og er en minnehåndteringsfunksjon for et operativsystem. Minneallokeringen lar sekundærminnet adresseres som om det var en del av primærminnet. Virtuelt minne benytter seg av hardware og software for å kompensere for fysisk mangel på minne ved å overføre data fra RAM (Random Access Memory) til disk lagring. Virtuelt adresserom økes ved å bruke aktivt minne i RAM og inaktivt minne i harddiskstasjoner (HDD) for å danne sammenhengende adresser som både holder applikasjonen og dens data.

Virtuell adresse er adressen gitt en adresse i det virtuelle minnet for at den skal kunne aksesseres som om den var en del av main memory.

Virtuelt adresserom er den virtuelle lagringsplassen som tilegnes en prosess.

Adresserom er området med minneadresser tilgjengelig for en prosess.

Reell adresse er adressen til en lokasjon i main memory.

Hardware og kontrollstrukturer

Enkel oversikt over dette på side 373.

To egenskaper med paging og segmentering har laget grunnlaget for et viktig gjennombrudd innen minnehåndtering. Den første, at alle minnerefanser innenfor en prosess er logiske adresser som dynamisk oversettes til fysiske ved kjøretid. Dette gjør at prosesser kan byttes ut og inn av primærminnet og oppta ulike regioner av minnet under utførelse. Den andre er at prosesser kan deles inn i mange biter (pages og segmenter) som ikke trenger å ligge sammenhengende i minnet.

Disse to egenskapene la grunnlaget for at **det ikke er nødvendig å ha alle pagesene eller segmentene til en prosess i main memory under utførelse**. Så lenge delen som holder neste instruksjon og dataen som skal aksesseres ligger i main memory, kan utførelsen fortsette. Når utførelsen krever en del som ligger i sekundærminnet blokkeres prosessen mens OS gjør en disk I/O request. En annen prosess kan kjøre så lenge.

Det er to implikasjoner som medfører økt utnyttelse av systemytelsen:

1. **Flere prosesser kan holdes i main memory.** Siden kun noen biter av hver prosess ligger i main memory, er det plass til flere prosesser som medfører bedre utnyttelse av prosessoren.
2. **En prosess kan være større enn hele main memory.** Programmereren trenger ikke lenger å strukturere programmet slik at det kan lastes inn i main memory separat, dette løses nå av OS og hardwaren.

Minnet tilgjengelig for programmereren refereres til som **virtuelt minne**, og er større enn det **reelle minnet** som er main memory.

Lokalitetsprinsippet

Ettersom at kun en del av en gitt prosess ligger i minnet av gangen kan flere prosesser ligge i minnet. I tillegg spares det tid på å ikke hente prosesser ut og inn av minnet ofte. Når OS henter inn en ny del av en prosess, må en annen ut. Dersom OS kaster ut en prosess som snart skal brukes, går mer tid til å hente inn og kaste ut prosesser, enn det brukes tid på å utføre instruksjoner. Denne tilstanden kalles **trashing**. For å forhindre dette må OS benytte **lokalitetsprinsippet** for å beholde prosesser som med stor sannsynlighet vil brukes i nærfremtid. Lokalitetsprinsippet sier at *program- og data-referanser innenfor en prosess ofte klynges*. Derfor er antagelsen at mindre biter av en prosess i minnet av gangen gir bedre utførelse, gyldig.

For at virtuelt minne skal være praktisk og effektivt må det

1. Være **hardware-støtte** for paging og/eller segmentering
2. OS må inkludere **software** for å håndtere bevegelse av pages og/eller segmenter mellom primær- og sekundærminnet.

Paging

Paging matcher systemets fokus på faste blokkstørrelser osv. Som er en fordel ved bruk av sofistikerte minnehåndteringsalgoritmer. En page-tabell brukes også ved paging for virtuelt minne. Det er typisk å assosiere en tabell til hver prosess, men innholdet i tabellen er litt annerledes enn for simple paging da man må holde styr på blant annet hvilke biter av prosessen som ligger i minnet. En bit, P, indikerer om en page ligger i main memory, mens frame number oppgir hvor den ligger dersom $P = 1$. En annen bit M indikerer om innholdet ipagen har blitt endret siden den sist var lastet inn i main memory. Hvis ikke, trenger man ikke overskrive det som ligger i sekundærminnet når den flyttes ut av main memory. Gir ikke ekstern fragmentering, men noe intern fragmentering.

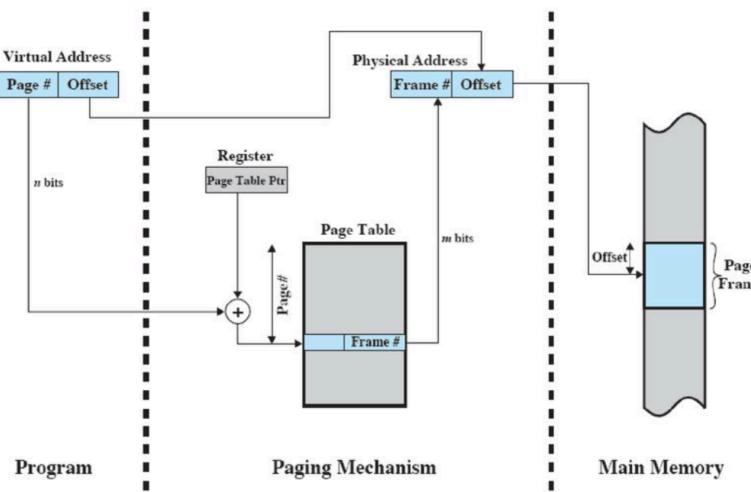


Figure 8.3 Address Translation in a Paging System

Page-tabellstruktur

Page-tabellen brukes til å konvertere mellom en virtuell (logisk) adresse og en fysisk adresse. Når en bestemt prosess kjører, holder et register startadressen til prosessens page-tabell. Pagenummeret til en virtuell adresse brukes til å indeksere tabellen og hente korresponderende frame-nummer. Dette kombinert med offset-delen danner den fysiske adressen. De fleste page-tabellene lagres i det virtuelle minnet for å spare plass. Derfor gjøres paging også på tabellen i seg selv. En del av tabellen må ligge i minnet når prosessen kjøres. En ulempe med denne tabellstrukturen er at størrelsen er proporsjonal med det virtuelle adresserommet. En alternativ tabellstruktur er **inverted page-tabell**. Her mappespagenummer og en virtuell adresse til en hashverdi ved å benytte en enkel hashfunksjon. Hashverdien er en peker til den inverterte page-tabellen. Det er kun en plass i tabellen for hver page i main memory, istedenfor virtuelt minne.

Struktur

Virtuell adresse	Pagenummer, offset
Rad i pagetabell	P- og M-flagg, kontrollbits, framenummer.

TLB – Translation Lookaside Buffer

I prinsippet kan hver virtuell minnereferanse gjøre to fysiske aksesser, en for å hente den passende pagetabellen, og en for å hente data. For at dette ikke skal doble minne-aksessiden har de fleste virtuelle minneskjemaene en Translation Lookaside Buffer (TLB). Dette er en høyhastighets cache for pagetabell-deler som har blitt brukt i nærmeste fortid.

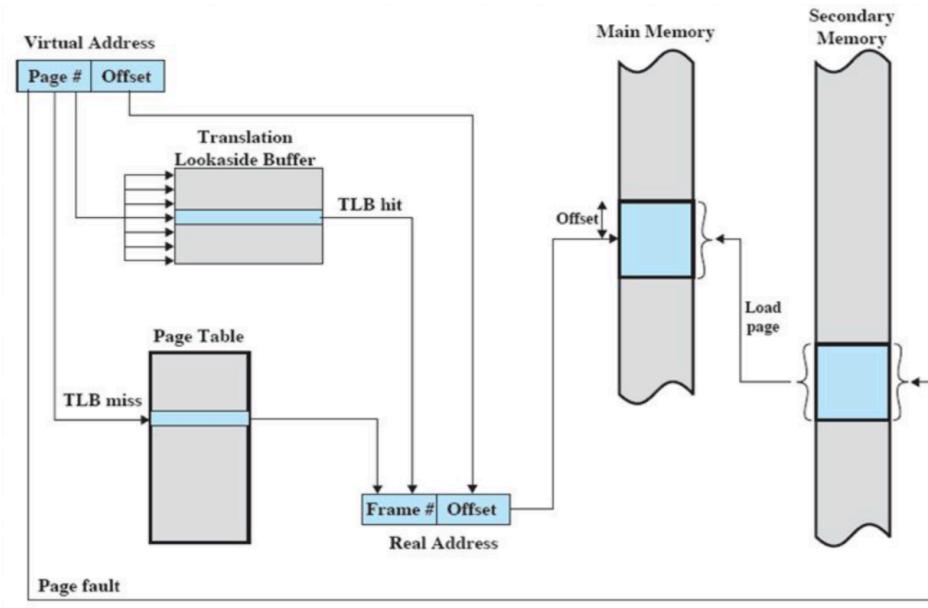


Figure 8.7 Use of a Translation Lookaside Buffer

Til slutt må mekanismen for det virtuelle minnet interagere med main memory cache-systemet.

Prosessen er som følgende. Først sjekker minnesystemet TLB for å se om den matchende pagetabelldelen ligger der. Hvis den er det, genereres den fysiske adressen ved å kombinere frame-nummer med offset. Hvis ikke, hentes delen fra pagetabellen. Når den fysiske adressen er generert sjekkes cache for å se om blokken vi adresserer er til stede. Hvis den er det, returnerer vi til prosessoren. Hvis ikke må blokken hentes fra main memory.

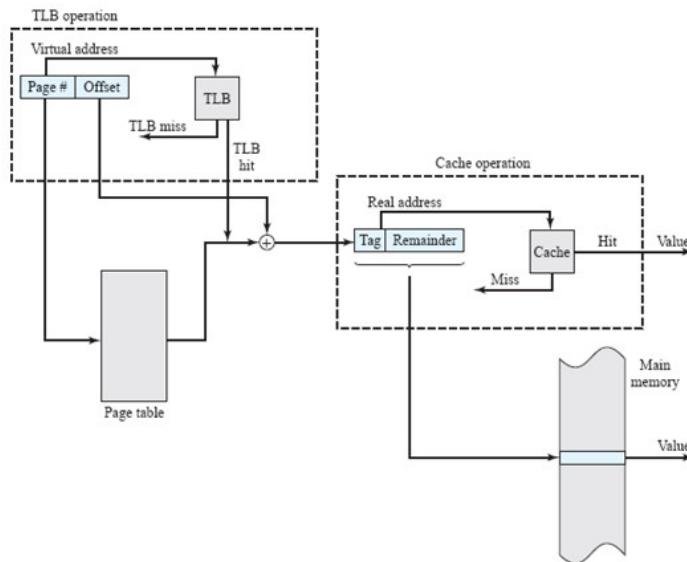


Figure 8.10 Translation Lookaside Buffer and Cache Operation

Page-størrelse

Ved veldig små page-størrelser er det plass til mange pages i main-memory for en prosess av gangen. Etter litt tid vil alle pages i minnet inneholde noen deler av nylig brukte referanser (lokalitetsprinsippet, figur (a)). Derfor vil sidefeilene være få. Ved større page-størrelse vil hver

individuelle page inneholder lokasjoner lengre fra nylige brukte referanser, som gjør at effekten av lokalitetsprinsippet synker og antall sidefeil økes. Etterhvert som sidestørrelsen nærmer seg størrelsen til prosessen, vil feilene falle igjen fordi hele prosessen passer i pagen. Figur (b) viser at for en fast page-størrelse, synker feilene ettersom antall pages i main memory vokser. Faktorer man må se på ved valg av størrelse:

1. **Intern fragmentering.** Ved mindre page-størrelse er det mindre intern fragmentering fordi man får utnyttet minnerommet.
2. **Antall pages som kreves per prosess.** Ved små page-størrelser kreves mange pages per prosess, noe som gir større page-tabeller. For store programmer må deler av denne tabellen da ligge i det virtuelle minnet. Dette kan medføre doblet sidefeil for en enkelt referanse.
3. **Sekundærminnets fysiske egenskaper** passer bedre til større page-størrelser for å transportere blokker av data mer effektivt

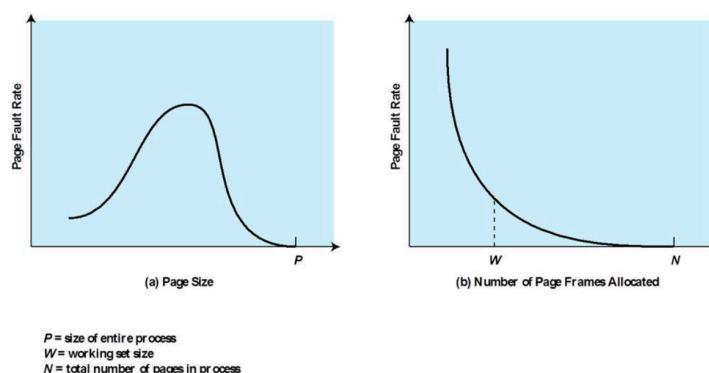


Figure 8.10 Typical Paging Behavior of a Program

Figur 8.10 a: Figuren reflekterer følgende:

1. Hvis page-størrelsen er stor nok rommer den en hel prosess/tråd i en enkelt side slik at sidefeil ikke vil oppstå. Ulempen med dette er at en ikke vil ha plass til så mange prosesser/tråder i primærlageret.
2. Hvis sidestørrelsen er liten nok, rommes mange nok deler av gjeldende prosess/tråd i tildelt primærminne-område som sjeldent vil gi sidefeil. Utfordringen er at en ikke har plass til så mange prosesser/tråder i primærlageret grunnet enorm overhead til page-tabeller etc.

Segmentering

Segmentering lar programmereren se på minnet som bygd opp av flere adresserom eller segmenter som kan være av ulike og dynamiske størrelser. Dette matcher brukerens fokus på delfunksjoner og hovedprogram som nevnt under. Minnereferansene består også her av segmentnummer og offset. Denne organiseringen har flere fordeler for programmereren:

1. **Forenkler håndteringen av voksende datastrukturer siden de er dynamiske.**
 Programmereren trenger ikke vite størrelsen til datastrukturen på forhånd.
2. **Tillater programmer å endres og rekompileres uavhengig, uten at hele settet av programmer skal omlinkes og omlastes.** Dette gjøres ved bruk av flere segmenter.
3. **Tillater deling mellom prosesser.** Programmereren kan definere et segment som kan aksesseres av andre prosesser.

4. **Tillater beskyttelse.** Programmereren eller systemadministrator kan gi aksess-privileger på et bestemt vis.

Struktur

Det er typisk å assosiere en segmenttabell med hver prosess. Denne er mer kompleks enn for enkel segmentering da det må være et P-flagg som sier om segmentet er i primærminnet eller ikke, i tillegg til en M-bit som sier om innholdet i segmentet har blitt endret siden det ble hentet. Hvis det ikke har blitt endret, trenger man ikke oppdatere framden den okkuperer. Andre kontrollbits kan også inkluderes, som for beskyttelse for eksempel

Struktur

Virtuell adresse	Segmentnummer, offset
Rad i segmenttabell	P- og M-flagg, kontrollbits, segmentlengde og segmentbase

Mekanismen

Mekanismen for å lese fra minnet involverer oversetting av en virtuell (logisk) adresse bestående av segmentnummer og offset, til en fysisk adresse ved bruk av segmenttabellen. Segmenttabellen må ligge i primærminnet for å kunne aksesseres fordi den har variabel lengde. Når en prosess utføres ligger startadressen til den tilhørende segmenttabellen i et register. Segmentnummeret til en virtuell adresse brukes til å indeksere tabellen for å hente den korresponderende main memory-adressen til starten av segmentet. Dette legges sammen med offset-delen til den virtuelle adressen for å danne den ønskede reelle adressen.

Gir ikke intern fragmentering, men noe ekstern fragmentering.

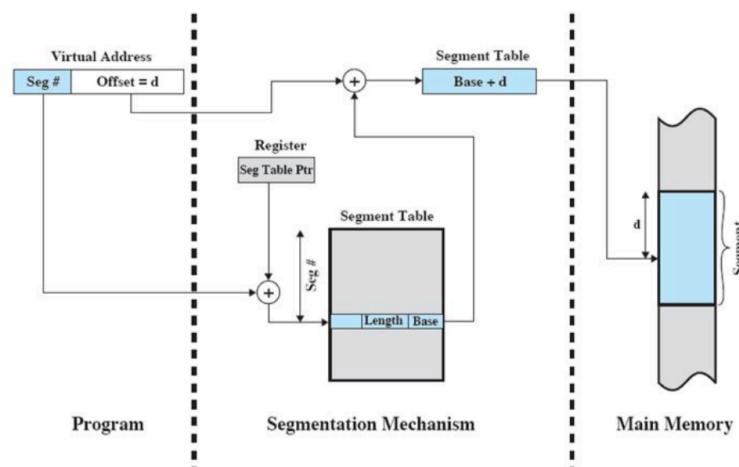


Figure 8.12 Address Translation in a Segmentation System

Kombinert segmentering og paging

En kombinasjon av segmentering og paging gir fordeler (og ulemper) fra begge metodene. Kombinasjonen gir også *større* overhead mht. Plassbehov for tabeller og oppslagsbehov i tabeller. Denne metoden kan derfor velges hvis man har råd til den ekstra overadden. Paging eliminerer ekstern fragmentering og gir effektiv bruk av minnet, i tillegg til at bestemte størrelser på pagene gjør det mulig med sofistikerte minnehåndteringsalgoritmer som utnytter

oppførselen til programmer. Segmentering har fordelene som er nevnt over inkludert håndtering av voksende datastrukturer, modularitet og støtte for deling og beskyttelse.

I et kombinert system, deles brukerens adresserom i et antall segmenter. Hvert segment deles inn i en rekke pages, som har samme størrelse som en main memory frame. Hvis segmentet er mindre enn en page, okkuperer det kun en page. Programmereren ser på den logiske adressen som segmentnummer og segment offset, mens systemet ser på segment offset som et page-nummer og page offset forpagen i segmentet. Hver prosess har assosiert en segmenttabell og et antall pagetabeller, en per prosess-segment. Et kombinert system gir fokus på responstid og gjennomstrømning på samme tid.

Struktur

Virtuell adresse	Segmentnummer, pagenummer, offset
Rad i segmenttabell	Kontrollbits, lengde og segmentbase
Rad i pagetabell	P, M, framenummer og kontrollbits

Programmeren ser på pagenummer og offset som et vanlig segment offset.

Segmentnummer og pagenummer brukes til å finne segmentet ogpagen i det segmentet.
Offset finner lokasjonen til pagen i framnen.

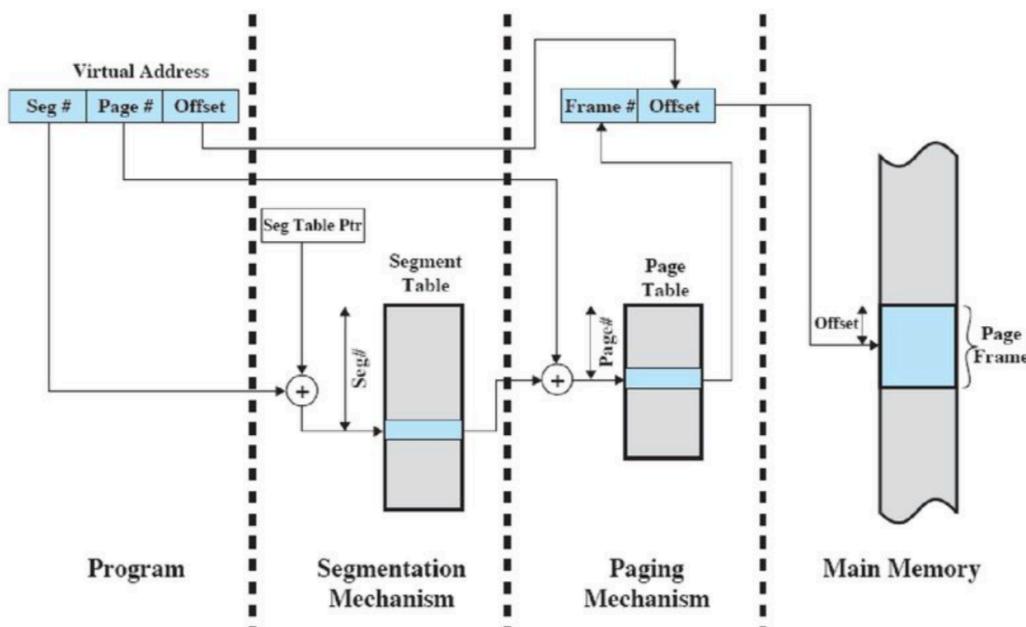


Figure 8.13 Address Translation in a Segmentation/Paging System

Operativsystem software

Tar utgangspunkt i bruk av pages.

Designet av minnehåndtering i OS avhenger av

- Om man skal bruke virtuelle minneteknikker
- Bruk av segmentering, paging eller begge
- Minnehåndteringsalgoritmene

Vi ønsker å minimere antall pagefeil fordi de skaper overhead. Denne overheaden inkluderer å bestemme hvilken page eller pages man skal bytte ut, I/O-bytte av bages. I tillegg må OS kjøre en annen prosess under I/O, som gir et prosessbytte. I tillegg ønsker vi å ordne slik at en prosess som utføres har liten sannsynlighet for å referere en manglende page.

Fetch Policy / Innhenting

Fetch policy bestemmer når en page hentes til main memory.

Demand paging

En page hentes til main memory ved behov, som vil si kun når en referanse til denpagen gjøres. Ettersom flere og flere pages hentes inn, gir lokalitetsprinsippet at de fremtidige referansene mest sannsynlig er de siste pagene hentet inn. Etter litt tid skal dermed antall sidefeil være lite.

Prepaging

Her hentes andre pages enn på etterspørsel etter pagefeil. Prepaging utnytter at de fleste sekundærminne-enheter har roterende ventetid. Det er dermed mer effektivt å hente inn et antall sammenhengende pages en gang, enn hver for seg. Denne metoden er ineffektiv dersom mesteparten av pagesene som er hentet inn ikke refereres. Denne metoden burde brukes enten når en prosess startes hvorav programmereren vil utpeke valgte pages, eller hver gang en page-feil oppstår. Den siste er foretrukket.

Placement Policy / Plassering

Placement policy bestemmer hvor i det reelle minnet en prosessdel skal oppholdes. For segmentering kan man bruke metodene som best-fit, first-fit osv. Beskrevet tidligere. For paging med eller uten kombinasjon av segmentering, er ikke dette nødvendig da hardware for main memory og adresseoversetting gjør denne funksjonen like effektivt.

Replacement Policy / Utbytting

Utbrytting omfatter valget av hvilken page i main memory som skal byttes ut når en ny page hentes inn. Hver metode har som mål å ta ut den prosessen som har minst sannsynlighet til å refereres i nærmeste fremtid. Merk at noen frames kan være låste, og kan dermed ikke byttes ut, f.eks. de som holder kjernen av OS osv.

Basic algoritmer for utbytting

- Optimal
- Last recently used (LRU)
- Least frequently used (LFU)
- First-in-first-out (FIFO)
- Clock

Optimal (kan ikke implementeres)

Optimal velgerpagen som har lengst tid til neste referanse. Det kan vises at dette medfører færrest pagefeil. Man har ikke all informasjonen som skal til for å implementere denne algoritmen, så den brukes derfor som et mål mot de andre metodene.

LRU

LRU bytter ut pagen som ikke har blitt referert til på lengst tid. Av lokalitetsprinsippet skal dette være pagen som har minst sannsynlighet å refereres til i nærmeste fremtid. Denne gjør det nesten like bra som optimal, men problemet er at den er vanskelig å implementere. Det ender ofte opp med mye overhead å lagre tiden for hver referanse, eller stack med referanser.

LFU

LFU bytter ut den minst frekvente refererte siden. Dette innebærer at man må holde en referanseteller for hver page tilhørende en prosess.

FIFO

FIFO behandler page frames allokkert til en prosess i en sirkulær buffer, der pages fjernes i en round-robin style. Dette er den enkleste formen å implementere, og krever lite overhead. Logikken er at pagen som har vært lengst i minnet, er den som burde byttes ut. Dette gjelder ofte ikke, da det er deler av programmer eller data brukes mye gjennom utførelsen av et program. Disse vil ende opp med å byttes ut og inn.

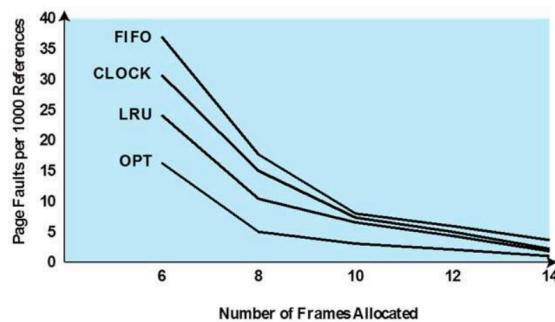
Clock

Klokemetoder er resultatet av forsøket i å implementere LRU uten så mye overhead. Algoritmene refererer til som klokker fordi framesene kan visualiseres som en sirkel, med pekeren i midten.

U-clock

Den enkleste formen for klokkealgoritme bruker en bit kalt *use bit (U)* assosiert til hver frame. Når en page lastes inn i framnen settes $U = 1$. Hver gang pagen refereres til i ettermidd settes også $U = 1$. En sirkulær buffer med en peker holder på pagesene som er kandidater for utbytting. Når en page byttes ut settes pekeren til å indikere den neste rammen i bufferen etter den som er byttet ut. For å bytte ut en page scanner OS bufferen for å finne en frame med $U = 0$. For hver frame med $U = 1$, resetter den biten til $U = 0$, og fortsetter. Dersom en av framesene har $U = 0$, velges denne for utbytting, den nye pagen legges inn og U settes lik 1 igjen før pekeren flyttes til neste frame. Dersom alle har $U = 1$ gjør pekeren en sykel, setter alle til 0 og velger den originale posisjonen for å plassere pagen.

Strategien ligner FIFO, bortsett fra at en frame med $U = 1$ vil passeres.

**Figure 8.16 Comparison of Fixed-Allocation, Local Page Replacement Algorithms**

UM-clock

Klokkealgoritmen kan gjøres kraftigere ved å øke antall bits brukt. I alle prosessorer som støtter paging, er det en modify bit assosiert til hver page i main memory, og dermed også til hver frame i main memory. Denne trengs slik at når en page er modifisert, byttes den ikke ut før oppdateringen også er gjort i sekundærminnet. Denne kan utnyttes i klokkealgoritmen. Hver frame faller inn i en av fire kategorier:

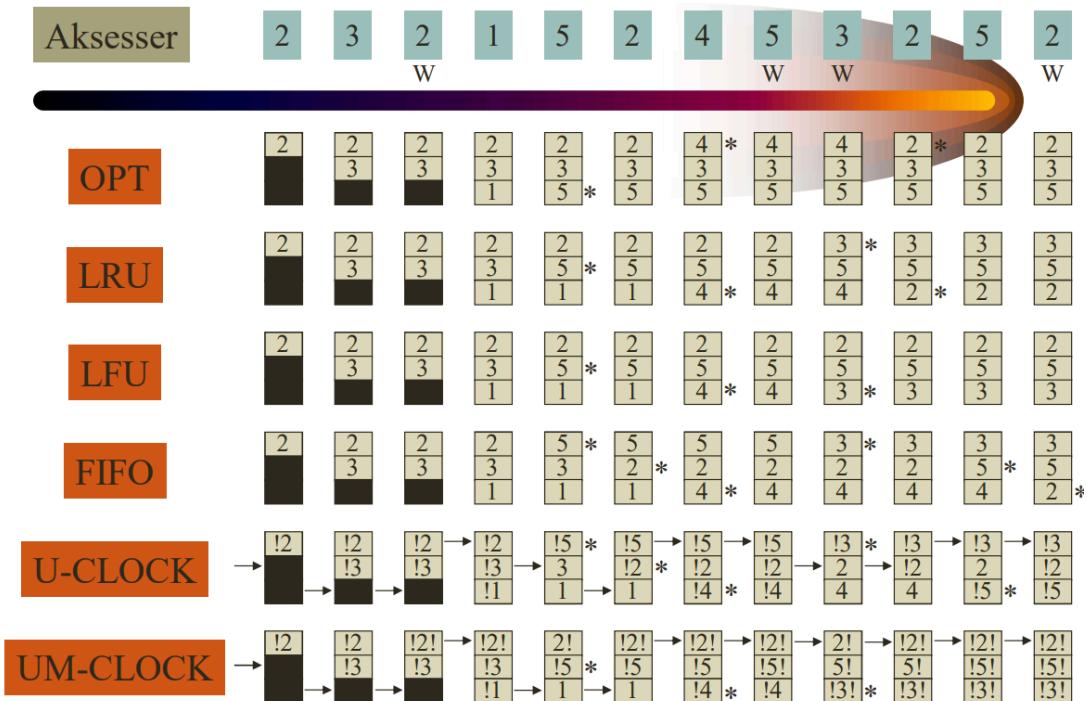
- Ikke aksessert nylig, ikke modifisert ($U = 0; M = 0$)
- Aksessert nylig, ikke modifisert ($U = 1; M = 0$)
- Ikke aksessert nylig, modifisert ($U = 0; M = 1$)
- Aksessert nylig, modifisert ($U = 1; M = 1$)

Med denne klassifiseringen opptrer UM-clock slik:

1. Starter ved posisjonen til pekeren og scanner frame-bufferen. Under denne scanningen røres ikke U-biten. Den første rammen med $U = 0, M = 0$ velges for utbytting.
2. Hvis steg 1 failer, scan igjen for å se etter $U = 0, M = 1$. Hvis det finnes, velges denne for utbytting. Under denne scanningen, sett alle $U = 0$ for hver forbipassering
3. Hvis steg 2 feiler, skal pekeren ha returnert til sin originale posisjon og alle frames skal være 0. Gjenta steg 1, og hvis nødvendig, gjenta steg 2.

Fordelen med denne algoritmen er at en page som er umodifisert foretrekkes å byttes ut. Dette sparer tid, siden den ikke må skrives til disk. Dersom det ikke er noen umodifiserte pages, velges den som ikke er aksessert nylig. Selv om man bruker tid på å skrive denne til disk, vil den ha en stor sannsynlighet for å ikke refereres til i nærmeste framtid (av lokalitetsprinsippet), som gjør at vi kanskje ikke må hente den inn igjen.

Sammenligning av algoritmer



- W: Skriver
- *: Bytter
- !x: U-merke
- Y!: M-merke
- ->: Peker
- Eks.: Første Read av 5
- OPT: Best
- FIFO: Dårlig
- LRU/LFU: Bra, dyre
- U-CLOCK/UM-CLOCK: Bra, billige

Resident Set Management / Mengde per prosess

OS må bestemme hvor mye main memory som skal allokeres til en bestemt prosess. Dette avhenger av flere faktorer, som:

- Mindre main memory, gir plass til flere prosesser, som dermed gir mindre swapping.
- Hvis et relativt lite antall pages av en prosess er i main memory, vil antall pagefeil være høyt på grunn av lokalitetsprinsippet.
- Over en viss størrelse, vil ikke ekstra main memory gi en merkbar endring for en prosess sine pagefeil grunnet lokalitetsprinsippet,

	Lokal	Global
Gitt allokering	OK. Ved gitt allokering bestemmes antall frames som allokeres en prosess, på forhånd. Det følger dermed atpagen som skal swappes ut, velges blandt prosessens allokerete frames.	Ikke mulig.

	Ulempene er at dersom allokeringen er for liten, medfører det mye page faults og dermed et tregt system, mens dersom allokeringen er for stor er det for få programmer i main memory som gir mye prosessor-tomgang, eller mye tid brukt på swapping	
Variabel allokering	<p>Avansert, men ofte bedre. Antall frames som allokeres til en prosess kan endres fra tid til annen for å opprettholde arbeidssettet til prosessen. Ved å reevaluere allokeringen for så å øke eller senke den, økes den overall ytelsen i systemet.</p> <p>Pagen som byttes ut velges blant framesene allokkert til prosessen.</p>	<p>OK. Page som skal swappes ut velges blant alle tilgjengelige frames i minnet; dette gjør at mengden av en prosess som ligger i main memory, varierer. En prosess som opplever page faults vil gradvis øke i størrelse, som skal redusere antall page faults i systemet. Kan forbedres ved å bruke page-buffer.</p>

Størrelse

Gitt: gir en prosess et fast antall frames i main memory som bestemmes ved prosessdannelse. Antall vil avhenge av prosesstypen, eller bestemmes av programmerer.

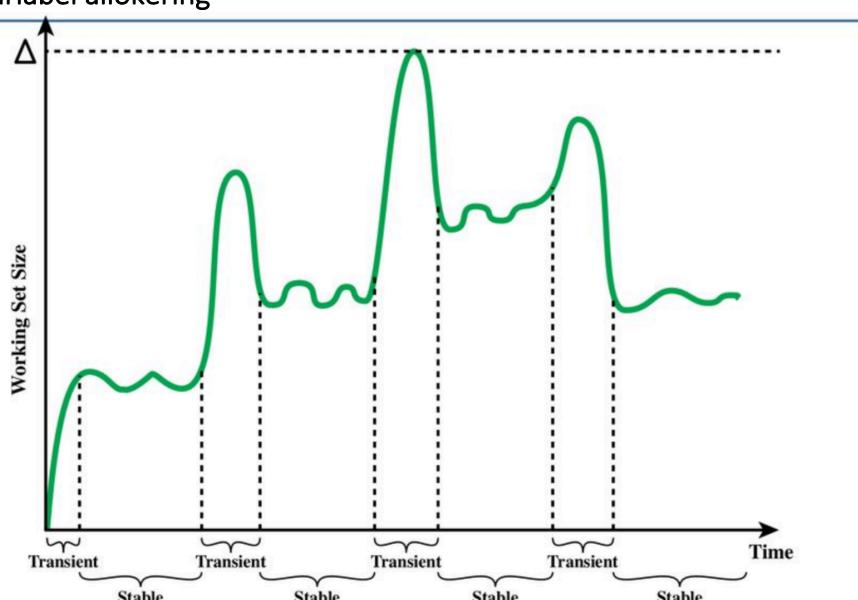
Variabel: lar en prosess allokeres et antall page frames som varierer gjennom levetiden til prosessen. Denne er mer kraftig, men krever at OS vurderer oppførselen til prosesser.

Omfang

Begge aktiveres av en pagefeil når det ikke er noen ledige page frames.

Lokal: velger kun blant de pagesene av prosessen som genererte pagefeilen.

Global: velger blant alle ulåste pages i main memory som er kandidater for utbytting, uavhengig av hvilken prosess som eier en page.

Variabel allokering

Arbeidssett er et modelleringstverktøy som har som mål å fastlegge de og bare de sidene som til enhver tid trengs lett tilgjengelig for ulike prosesser/tråder. Man betrakter da gjerne de sidene som har vært brukt av prosessen/tråden over en viss tidsperiode. Omfanget av slike sider avhenger av valgt tidsperiode og vil svinge over tid, men med lange stabile perioder innimellom.

I figuren speiler arbeidssett-størrelsen omfanget av det settet med sider som aksesseres til enhver tid. Lokalitetsprinsippet tilsier at arbeidssettet og dermed også arbeidssett-størrelsen holder seg stabil over visse tidsrom. Med jevne mellomrom endrer dog arbeidssettet seg, og størrelsen vil svinge tilsvarende opp og ned. I slike overgangssituasjoner vil sidene tilhøre to arbeidssett, og den tilhørende arbeidssett-størrelsen vil da være større enn både før og etter.

Δ er et vindu med virtuell tid som en prosess observeres. Tid t måles i antall minnereferanser. WS er arbeidssettets størrelse som en ikke-sinkende funksjon av Δ . Merk at desto større Δ , desto større WS , som illustreres av relasjonen under.

$$W(t, \Delta + 1) \geq W(t, \Delta)$$

Et WS kan vokse like stort som antall pages N hvis ulike pages adresseres hurtig og vindusstørrelsen tillater det. Dermed har vi at:

$$\text{Min}(\Delta, N) \geq |WS(t, \Delta)| \geq 1$$

Konseptet WS brukes for å bestemme RS (resident set):

1. Monitorer arbeidssettet (WS) for hver prosess.
2. Periodisk fjern prosessens pages fra RS som ikke ligger i WS .
3. Kun prosesser der RS omfatter WS kan kjøre (dvs. At WS er i main memory).

Denne metoden utnytter lokalitetsprinsippet til å minimere pagefeil. Problemet i praksis er at fortid ikke alltid angir fremtid og at kontinuerlig målinger av WS samt valget av en optimal Δ er vanskelig.

Sidefeil tilpasses for hver prosess.

- For mange pagefeil → øk RS
- For få pagefeil → mink RS
- Passe → Behold RS

Cleaning Policy

En cleaning policy bestemmer når en page skal skrives ut til sekundærminnet. Dette gjøres ofte som **demand cleaning** eller **precleaning**. Ved demand cleaning skrives en page til minnet kun på etterspørsel, mens ved precleaning skrives modifiserte pages til sekundærminnet før framesene trengs, det vil si at pages kan skrives ut i bulker. En god løsning er å benytte pagebuffering med en modified og en unmodified liste.

Load control

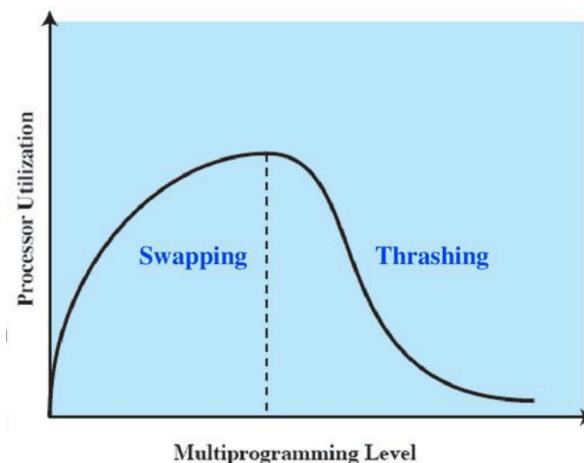
Load control bestemmer antall prosesser som skal ligge i main memory, og refereres til som multiprogrammeringsnivå. Dette er kritisk for prosessorytelsen.

Multiprogrammeringsnivå

Trashing er dersom OS kaster ut en prosess som snart skal brukes, og mer tid går til å hente inn og kaste ut prosesser, enn det brukes tid på å utføre instruksjoner. Dette kan løses på flere måter, blant annet ved å bestemme antall prosesser i primærminnet basert på en grenseverdi som gir høy ytelse. Dette kan være å sette gjennomsnittlig tid mellom pagefeil lik gjennomsnittlig tid det tar å ordne opp i en pagefeil.

Suspendering av prosesser

Dersom graden av multiprogrammering reduseres må en prosess suspenderes (swappes ut). Dette kan gjøres basert på *prioritet*, *høyest pagefault*, *minst resident sett*, *største prosess* eller *størst utførelsesvindu*.



For få prosesser/tråder i primærlageret gir dårlig CPU-utnyttelse da det ofte ikke er prosesser/tråder tilgjengelig for bruk av CPU-kraft. For mange prosesser i primærlageret vil også gi dårlig utnyttelse av CPU da det dermed brukes mye CPU-kraft på ut- og innsampling av hele prosesser/tråder. Et balansepunkt mellom de to kan etterstrebnes ved å overvåke sidefeilfrekvensen til hver enkelt prosess/tråd. Hver prosess/tråd må ha så mye data i primærlageret at dens sidefeilfrekvens holder seg under et visst tak. Antall aktive prosesser/tråder justeres da til enhver tid ut i fra svingningene i hver enkelt prosess/tråd sin tilhørende sidefeilfrekvens.

UNIX og Solaris minnehåndtering

Brukerdata: sidedelt og virtuelt lager

Systemdata: forsinket, buddy system.

Sideutbytting U-clock-variant

LINUX / ANDROID

Brukerdata og systemdata: buddysystem.

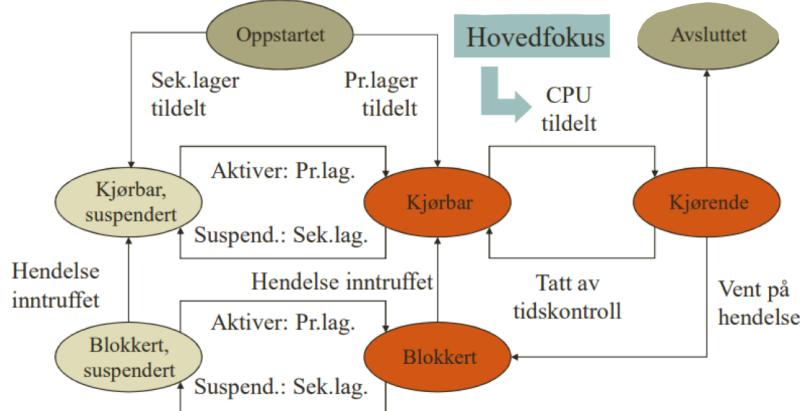
Windows 8

Brukerdata og systemdata: sidedekt virtuelt lager.

Kapittel 9: Uniprosessor Scheduling / Tidsstyring

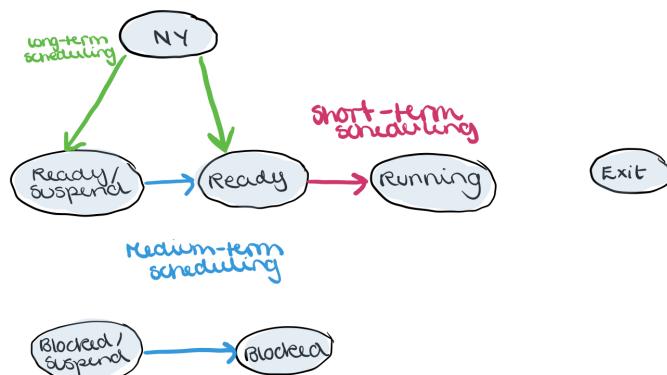
Scheduling eller tidsstyring er prosessen å tildele datamaskinens ressurser. Når det er flere prosesser som ønsker eller trenger flere ressurser enn hva som er tilgjengelig, må man bruke tidsstyring for å velge hvilken rekkefølge prosessene skal kjøres. Valget av rekkefølge kan gjøres basert på ulike kvantitative og kvalitative kriterier som gir ulike effekter på tilsvarende måleparametere. I et multiprogrammeringssystem er det typisk fire typer scheduling:

Long-term scheduling	Bestemmer utførende prosesser. Gjøres ved opprettelse av prosesser.
Medium-term scheduling	Bestemmer prosesser som er delvis eller fullt i main memory. Er altså en del av <i>swapping</i> -funksjonen.
Short-term scheduling	Bestemmer hvilke prosesser som skal utføres som neste av prosessoren.
I/O-scheduling	Bestemmer hvilken prosess sin I/O-forespørsel som skal håndteres av en tilgjengelig I/O-enhet.



Type processor-scheduling

Hensikten med prosessor scheduling er å tildele prosesser som skal utføres av prosessoren(e) over tid på en måte som møter systemets objektiver, som responstid, throughput og prosessoreffektivitet. Dette brytes ofte inn i long-, medium- og short-time scheduling. Scheduling er grunnleggende hvordan man styrer køer for å minimere køforsinkelse og optimere ytelse.



Long-term scheduling

Long-term tidsstyreren 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 omgjøres det til en prosess og legges til short-time-scheduleren. I noen systemer kan en ny prosess begynne i swapped-out tilstand, og i dette tilfellet legges den i køen til medium-term schedulern.

Det er to beslutninger som gjøres; tidsstyreren må bestemme når OS kan ta inn en ekstra prosess, og hvilken jobb eller jobber som skal aksepteres og gjøres om til prosesser.

Når kan en prosess legges til?

Dette spørsmålet bestemmes hovedsaklig av graden multiprogrammering. Desto flere prosesser som opprettes, desto mindre prosessortid per prosess. Long-term scheduleren kan dermed begrense antallet prosesser for å gi høyere tilfredsstilling av prosessene som allerede er opprettet.

Hvilken prosess legges til?

Dette kan bestemmes på flere måter, for eksempel ved *first-come-first-served*, *prioritet*, *I/O-krov* som prosessor-bundede prosesser, dvs. at de hovedsakelig bruker prosessor-ressurser, og ikke I/O-ressurser etc.

Medium-term scheduling

Medium-term scheduling er en del av swappe-funksjonen, og diskuteres i kap. 3, 7 og 8. Typisk baseres swappe-funksjonen seg på graden av multiprogrammering. For et system som ikke benytter virtuelt minne, er også dette en utfordring. Derfor må swappefunksjonen se på minnekrevene til de suspenderte prosessene.

Short-term scheduling

Short-term schedulern, også kalt **dispatcher**, utføres oftest og bestemmer hvilken prosess som skal kjøres neste. Dispatcheren involveres hver gang en hendelse oppstår som kan lede til blokkering av en nåværende prosess eller gi en mulighet for å fremkynne en prosess over en annen. Eksempler kan være *klokkeavbrudd*, *I/O-avbrudd*, *OS-kall*, eller *signaler* (eks. *semaforer*).

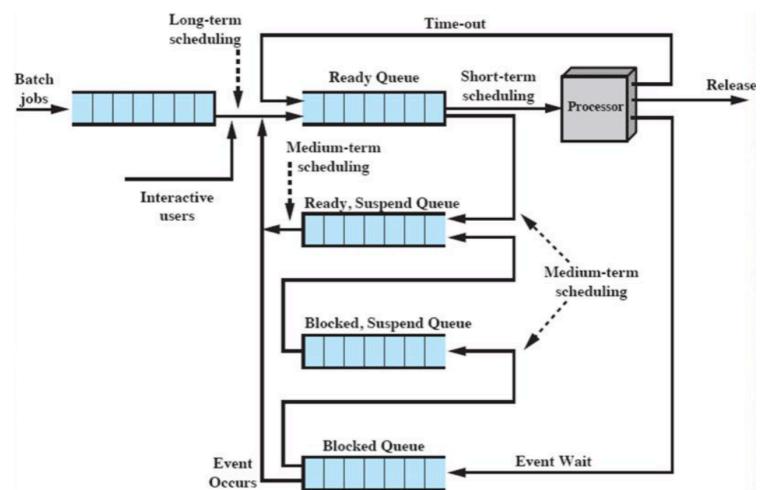


Figure 9.3 Queuing Diagram for Scheduling

Scheduling-algoritmer

Kriterie for short-time scheduling

Målet med short-time scheduling er å fordele ressurser slik at en eller flere aspekter av systemoppførselen optimaliseres. Kriteriene kan kategoriseres i to dimensjoner:

Bruker- eller systemfokusert

Brukervoksus relaterer til systemoppførsel sett fra én individuell bruker eller prosess, mens systemfokus på den andre side fokuserer på effektiv utnyttelse av prosessoren.

Ytelse- eller ikke ytelsesorientert

Ytelsesrelaterte kriterier er kvantitative og kan lett måles, som throughput og responstid. Kriteriene som ikke er ytelsesrelaterte er vanskeligere å måle og analysere. For eksempel forutsigbarhet.

Tabellen viser de viktigste tidsstyringskriteriene. Disse er gjensidig avhengige, og det er umulig å optimalisere alle på en gang.

	Brukerfokus	Systemfokus
Ytelsesorientert	<p><i>Turnaround time (Gjenomløpstid):</i> Tiden mellom innsending av en prosess og at den er ferdig.</p> <p><i>Responstid:</i> Tiden det tar fra innsending av en forespørsel inntil responsen mottas.</p> <p><i>Deadlines:</i> Når tidsfrister for gjennomførelse av prosesser kan settes, maksimerer tidsstyreren andelen tidsfrister som møtes.</p>	<p><i>Throughput:</i> Tidsstyreren burde forsøke å maksimere antall prosesser som fullføres per tidsenhet. Dette måler hvor mye arbeid som gjøres.</p> <p><i>Prosessorutnyttelse:</i> Dette er andelen tid hvor prosessoren er opptatt. Dette er viktig i et delt system.</p>
Ikke ytelsesorientert	<p><i>Forutsigbarhet:</i> En jobb burde kunne utføres med samme tid og kostnad uavhengig av systemets last. Stor variasjon i responstid og turnaround er forvirrende for brukere.</p>	<p><i>Rettferdighet:</i> Ved fravær av brukerveiledning skal alle prosesser behandles likt og ingen skal ende i starvation (utsulting).</p> <p><i>Prosessprioritet:</i> Tildeler ulik prioritet til ulike prosesser som tidsstyreren skal følge.</p> <p><i>Ressursbalanse:</i> Tidsstyreren skal sørge for at alle ressurser er i bruk.</p>

Bruk av prioritet

Ved bruk av prioritet eksisterer det en kø for hver prioritetsgruppe. Når en tidsstyringsbeslutning skal tas, sjekkes køen med høyest prioritet først, og deretter resten i synkende rekkefølge. Utfordringen her er at prosesser med lavere prioritet kan ende opp i starvarion (utsulting). For å løse dette kan for eksempel prioriteten til prosesser endres underveis.

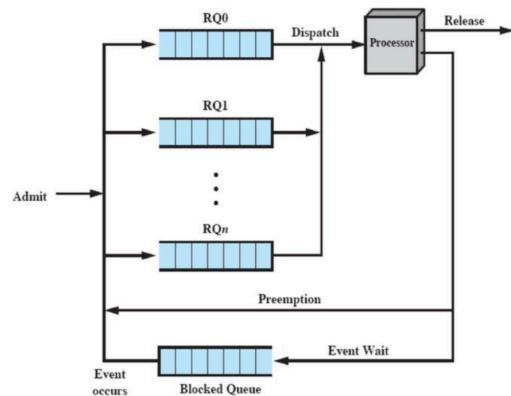


Figure 9.4 Priority Queuing

Alternative tidsstyringsregler Seleksjonsfunksjonen

Seleksjonsfunksjonen bestemmer hvilken prosess blant klare prosesser som skal velges for utførelse. Dette kan baseres på prioritet, ressurskrav, eller for eksempel hvordan prosessen utføres. I dette tilfellet ser man på:

- $w = \text{tid brukt i systemet så langt}$
- $e = \text{tid brukt under utførelse så langt}$
- $s = \text{total tjenestetid som kreves av prosessen (inkludert } e\text{), generelt må dette bestemmes av brukeren.}$

Valgmodus

Valgmodus spesifiserer ved hvilken tid seleksjonsfunksjonen skal utføres. Det varierer mellom to kategorier.

- **Ikke avbrytbar (nonpreemptive)** – Prosessen som kjører fortsetter inntil den terminerer eller blokkerer seg selv for I/O eller en annen OS-tjeneste.
- **Avbrytbar (preemptive)** – Den kjørende prosessen kan avbrytes og flyttes til ready-tilstanden av OS. Dette kan gjøres til fordel for en annen prosess når et avbrudd skjer fordi en ny prosess er i ready-tilstanden, eller ved klokke-avbrudd. Denne krever mer overhead, men kan gi bedre tjeneste til den totale mengden prosesser, fordi ingen prosess får monopol på prosessoren.

Scheduling-algoritmer

Turnaround Time – TAT

TAT er tiden T_r (r for resident) som et element bruker i systemet (ventetid + tjenestetid). Et nytig mål kan være normalisert turnaround time, som er turnover time over service time. Det indikerer den relative forsinkelsen som prosessen opplever.

First-Come-First-Served– FCFS

Har ikke avbrudd eller prioritering. Aktive prosesser kjøres til de er fullført eller til de blokkeres av forespørsel på I/O eller en annen OS-tjeneste. Denne metoden gjør det bedre ved prosessorbundet prosesser over I/O-bundet. FCFS medfører ueffektiv bruk av prosessor og I/O-enheter, og er som regel ikke bra alene i et uniprosessorsystem. Kombinert med *prioritering*, kan den gjøre nytte. Med en kø for hver prioritet, kan en FCFS benyttes på hver an köene.

Round Robin/Time slicing – RR

RR fordeler tiden på de ulike prosessene i periodiske intervaller. Ved avbrudd byttes prosessen ut i samme ordning som FCFS, bare at tidsfordelingen er mer rettferdig. Utfordringen med RR er å velge størrelsen på tidssegmentene. Håndteringen av avbruddene og tidsstyring samt dispatching inkluderer også noe overhead. Derfor burde veldig små segmenter unngås. Typisk settes segmentet lik tiden det tar for en typisk interaksjon eller prosessfunksjon, hvis ikke vil de fleste prosesser kreve to segmenter. En ulempe med RR er at den behandler prosessor-

bundede og I/O-bundede prosesser likt, selvom I/O-bundede generelt er kortere. Dette gir dårlig ytelse for I/O-bundede prosesser og økt variasjon i responstid. Kan løses noe ved bruk av Virtuell Round Robin, som har en egen kø for I/O-prosesser som prioriteres over prosessorbundet prosesser.

Shortest Process Next – SPN

SPN har ikke avbrudd, men prioriterer prosessen som forventes å ha kortest prosesseringstid til å utføres først. Overordnet prestasjon øker, særlig på responstid. Variabiliteten i responstiden vil dog økes, som gjør at forutsigbarheten synker. Dessuten kan det være vanskelig å vite prosesseringstid for hver prosess. En metode for å bestemme prosesseringstid kan være å bruke gjennomsnitt eller eksponentielt gjennomsnitt, som favoriserer nylige prosesser (som dermed også oftest er mer korrekt). SPN kan medføre utsulting for større prosesser, så lenge det er mange små prosesser.

Shortest Remining Time – SRT

SRT er en avbruddsversjon av SPN. Tidsstyreren favoriserer prosesser med minst gjenværende prosesseringstid, inkludert nye prosesser. SRT må dermed også ha et estimat på prosesseringstid, og det er en risiko for utsulting. I motsetning til RR er det ingen ekstra avbrudd, som reduserer overhead. At tiden må tas opp medfører derimot ekstra overhead. Alt i alt skal SRT gi bedre turnaround-performance enn SPN fordi korte jobber får umiddelbar prioritet.

Highest Response Ratio Next – HRRN

Generelt kan vi ikke vite service time på forhånd, men vi kan approksimere den. Enten basert på historie, eller input fra en konfigurasjonsmanager. Når den nåværende prosessen er fullført eller blokkert, velg den neste prosessen med høyest verdi av R (definert under). Dette tar hensyn til alderen til prosessen. Selv om små prosesser favoriseres (en mindre nevner gir større forhold), vil en eldende prosess som ikke velges få økt R for å hindre utsulting.

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

$$\begin{aligned} R &= \text{response ratio} \\ w &= \text{time spent waiting for processor} \\ s &= \text{expected service time} \end{aligned}$$

Feedback – FB

En annen måte å favorisere kortere jobber er å straffe jobber som har kjørt lenger. FB baseres dermed på utførelsestid så langt. Tidsstyring gjøres ved tidsavbrudd og en dynamisk prioritetsmekanisme. Når en prosess først inntrer i systemet legges den i RQ0 (se figur). Etter første avbrudd og prosessen er i ready-tilstand igjen, legges den i RQ1. For hvert avbrudd, legges prosessen i en kø av lavere prioritet. I alle køene bortsett fra den av lavest prioritet,

benyttes en FCFS-mekanisme. Når prosessen når køen av lavest prioritet legges den tilbake i RQ0 slik at den kan fullføre utførelse. Køene behandles dermed ved RR. FB kan medføre utsulting dersom mange mindre prosesser entrer systemet. For å kompensere til dette kan man variere tidsavbruddet for hver kø, som at RQ0 har tidsavbrudd etter 1 tidsenhets, RQ1 etter 2 enheter og RQi etter 2^i enheter. En annen metode som kan hindre utsulting er å sette en tidsgrense for utførelse, der en prosess vil få høyest prioritet etter en gitt tid.

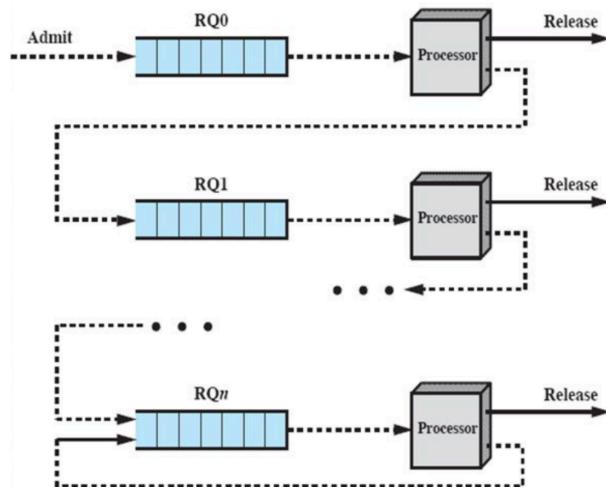
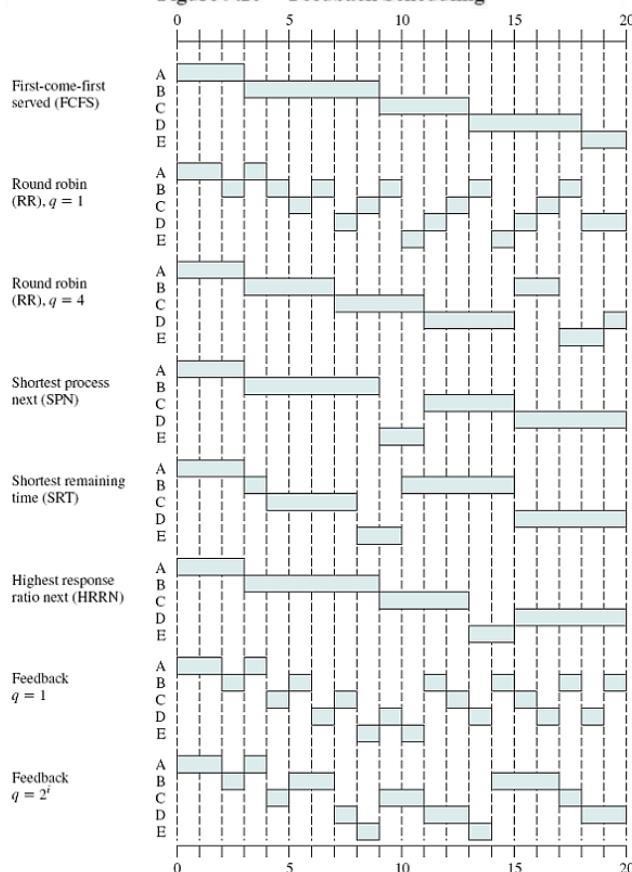


Figure 9.10 Feedback Scheduling



Sammenligning av de ulike algoritmene

Sammenligningen under er generell, og det er viktig å huske på at ytelsen avhenger av mange faktorer slik at den vil kunne være forskjellig i ulike tilfeller. Se antagelser gjort på s. 448.

KØ-analyse

Ved å favorisere kortere jobber ser man generelt at den gjennomsnittlige turnaround timen synker. Forbedringen er størst ved bruk av avbrudd, men man kan notere at den overordnede ytelsen ikke påvirkes mye av dette. Ved å skille prioritetsklasser kan vi øke ytelsen ytterligere. Ved bruk av prioritet uten avbrudd er forbedringene signifikante, noe som forsterkes enda mer ved bruk av avbrudd. Prosesser som er lengre og av lavere prioritet, vil få dårligere ytelse ved denne ordningen.

Algo- ritme	Respons- tid	Gjennom- strømning	Ut- sulting	Rett- ferdighet
FCFS	Last-avh.	Last-avh.	Nei	Små / IO: \div
RR	God: små	Kvant-avh.	Nei	Balansert
SPN	God: små	Høy	Mulig	Store: \div
SRT	God	Høy	Mulig	Store: \div
HRRN	God	Høy	Nei	Balansert
FB	Last-avh.	Kvant-avh.	Mulig	Ikke-IO: \div



Simuleringsmodell

Modell 9.14 viser normalisert turnaround time, mens modell 9.15 viser gjennomsnittlig ventetid. For turnaround time ser vi at FCFS gjør det dårligst, hvorav en tredjedel av prosessene har en normalisert turnaround time større enn 10 ganger service-timen; videre er dette de korteste prosessene. Når det kommer til ventetid er den uniformt fordelt på alle prosesser. RR har en gjennomsnittlig turnaround time på 5 (rettferdig for alle prosesser som er lengre enn et tidskvantum) ved bruk av $q=1$. SRT gjør det bedre enn SPN bortsett fra for de 7% lengste prosessene. Uten avbrudd har vi sett at FCFS favoriserer lange prosesser, mens SPN favoriserer korte. HRRN fungerer som et kompromiss mellom de to, noe som reflekteres i figurene. Til slutt ser vi at FB gjør det veldig bra for små prosesser.

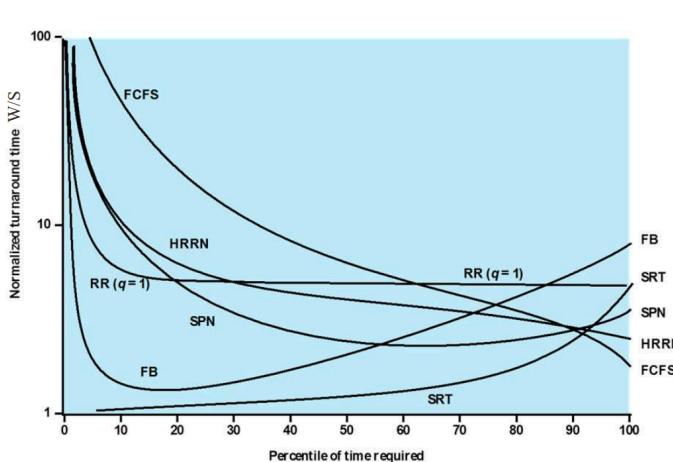


Figure 9.14 Simulation Results for Normalized Turnaround Time

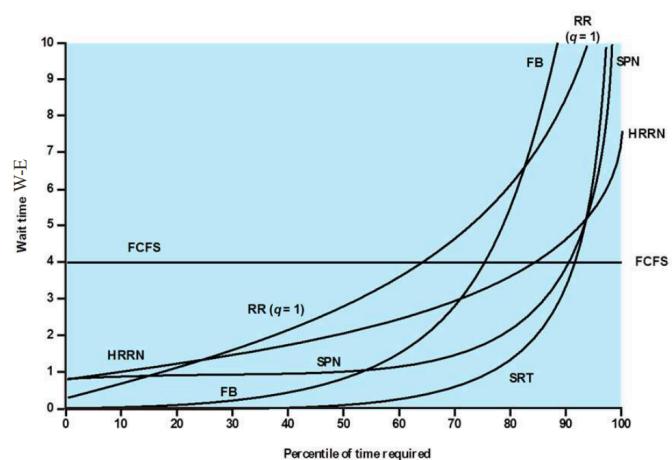


Figure 9.15 Simulation Results for Waiting Time

Fair-share scheduling (FSS)

I et multiuser system kan individuelle brukerapplikasjoner eller jobber være organisert i flere prosesser (eller tråder). I dette tilfellet ligger ikke fokuset på hvordan en bestemt prosess gjør det, men heller på hvordan settet av prosesser som utgjør en applikasjon opptrer. Derfor kan det være nyttig å lage en ordning som gjør tidsstyringsbeslutninger på grunnlag av hele settet. Dette kalles fair-share scheduling. Hver bruker har en vekt som definerer andelen av ressursene (prosessoren) som dedikeres til brukeren. Systemet deler brukerne inn i et sett med fair-share-grupper og allokerer andelen prosessorressurser til hver gruppe. Tidsstyringen gjøres på grunnlag av prioritet. Prioriteten til en prosess synker ettersom prosessen bruker prosessoren og ettersom gruppen prosessen tilhører benytter prosessoren.

Kapittel 10: Multiprosessor, multicore og samtids-scheduling

Tidsstyring med multiprosessor og multikjerne

Når et datasystem inneholder mer enn en enkelt prosessor introduseres flere nye designutfordringer. Multiprosessorsystemer kan kategoriseres i kategoriene:

- *Løst koblede eller distribuerte multiprosessorer, eller cluster*
- *Funksjonelt spesialiserte prosessorer*
- *Tett koblede prosessorer*

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

Detaljnivå på synkronisering

Detaljnivået eller frekvensen av synkronisering kan deles inn i fem kategorier:

Grain size	Beskrivelse	Synkroniserings-intervall	Eksempel
Fin	Parallelitet innebygd i en enkelt instruksjonsstrøm.	< 20	Representerer en mer kompleks bruk av parallelitet enn ved bruk av tråder.
Medium	Parallel prosessering eller multitasking innenfor en enkelt applikasjon (threading).	20-200	Threading – programmereren må eksplisitt spesifisere potensiell parallelitet i applikasjonen.
Grov	Multiprosessering av samtidige prosesser i et multiprogrammeringsmiljø.	200-2000	Programmer som håndteres enten ved multiprogrammering eller ved multiprosessering (samme som tidligere, så trenger ikke endre mye software).
Veldig grov	Distribuert prosessering på tvers av nettverksnoder for å danne et enkelt datamiljø.	2000-1M	Eks. Excelark.
Uavhengig	Flere urelaterte prosesser → Ingen synkronisering	Ingen	Banktransaksjoner. Operasjonene går raskere, men prosessene er uavhengige.

Designutfordringer ved tidsstyring på multiprosessorer

Metodene som brukes for å håndtere utfordringer avhenger som regel av detaljnivået på synkroniseringen.

1: Tildeling av prosesser til prosessorer

Den enkleste måten er å se på prosessorer som en samlet ressurs og å tildele prosesser til prosessorer on-demand. Da er spørsmålet bare om tildelingen skal skje statisk eller dynamisk. Dersom en prosess permanent får en prosessor til den er fullført (statisk tildeling) opprettholdes kun et korttidskø for hver prosessor. Fordelen er at det er lite overhead som går til tidsstyringsfunksjoner, men på kostnaden at en prosessor kan gå på tomgang mens en annen har en full kø. Man kan opprette en felles kø for å overkomme dette. *Det er vanlig å bruke både statisk og dynamisk tildeling med cache.*

2: Bruk av multiprogrammering på individuelle prosessorer

Burde individuelle prosessorer multiprogrammeres? For tradisjonelle multiprosessorer som håndterer synkronisering på grovere eller uavhengig detaljnivå burde hver prosessor kunne bytte mellom et antall prosesser for å oppnå høy utnyttelse. For applikasjoner med

synkronisering på medium detaljnivå som kjører på multiprosessorer med mange prosessorer er det derimot ikke like klart. Dersom flere prosessorer er tilgjengelige, er det ikke lenger et overordnet mål å holde hver prosessor så opptatt så mulig, men man ser heller på hva som gir god ytelse i gjennomsnitt for applikasjonene. En applikasjon med et antall tråder kan for eksempel gjøre det dårligere dersom ikke alle trådene er tilgjengelig for å kjøres samtidig.

3: Dispatching av prosesser

Den siste utfordringen er å velge hvilken prosess som skal kjøres. I motsetning til for uniprosessorer, er det unødvendig komplikt for multiprosessorer å benytte seg av prioritet og tidsstyringsalgoritmer basert på historie av bruk. For multiprosessorer brukes **enklere** metoder med **mindre overhead**, som vi tar for oss under.

Tidsstyring av prosesser i multiprosessering

Forskning har vist at spesifikke tidsstyringsdisipliner er i mindre grad viktig med flere prosessorer enn med kun én, og viktighetsgraden synker ytterligere med flere prosessorer. Derfor kan en enkel FCFS-ordning eller FCFS med prioritetsordning være tilstrekkelig for et multiprosessorsystem. Merk at dette er samme algoritme som brukes for uniprosessorer bortsett fra at valget faller på en av de enklere variantene. Dette viser at behovet til prosesser ikke avviker så mye i slike sammenhenger.

Tidsstyring av tråder i multiprosessering

For tråder er utførelse annerledes enn for prosesser. Fordelen med tråder kommer ytterligere frem i et multiprosessorsystem med reell parallelitet, noe som fører til at vi velger andre algoritmer enn for et uniprosessorsystem for å utnytte den reelle paralleliteten. Det er særlig to ting vi ønsker å utnytte med tråder. Den første er at vi kan ønske å sikre at en tråd alltid kjører på den samme prosessoren, for å utnytte bruken av cache. Den andre er at trådene kan kjøre på hver sin prosessor for å sikre samkjøring mellom dem og dermed god kommunikasjon mellom trådene. Med flere prosessorer tilgjengelig kan man fokusere ytterligere på slike hensyn enn i uniprosessering, der fokuset ligger på å holde prosessoren aktiv til enhver tid. Vi skal se mer på fire metoder for tidsstyring av tråder og tildeling av prosessorer:

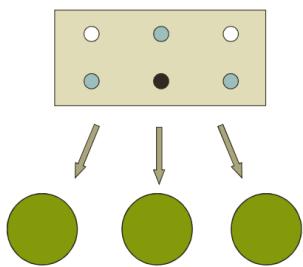
- *Load sharing*
- *Gang scheduling*
- *Dedicated processor assignment*
- *Dynamic scheduling*

Load sharing – selvtildeling av prosessorer

Den enkleste og kanskje mest brukte metoden er load sharing. Prosesser tildeles ikke til en bestemt prosessor. En global kø med klare tråder opprettholdes, hvorav hver prosessor henter tråder når den er ledig.

Fordeler:

- Lasten fordeles likt utover prosessorene, slik at prosessorene ikke går på tomgang
- Trenger ikke en sentralisert tidsstyrer fordi prosessoren selv velger den neste tråden når den er klar.
- En global kø kan brukes med en av ordningene fra kap. 9, som FCFS, Smallest number of threads first, eller preemptive smallest number of threads first.

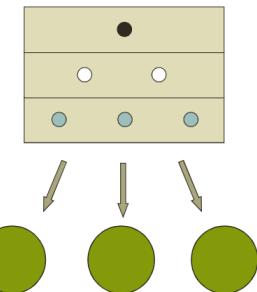


Ulemper:

- Den sentraliserte køen opptar en region i minnet som må aksesseres med **gjensidig utelukkelse** → **flaskehals**.
- Avbrutte tråder har liten sannsynlighet for å fortsette utførelsen senere på samme prosessor, som gir **dårlig utnyttelse av cache**.
- Hvis det er høy grad av kommunikasjon mellom tråder tråder i programmet, vil prosessbyttene gi veldig dårlig ytelse.

Gang scheduling – *lokalitetsprinsippet*

Gang scheduling er en tidsstyringsalgoritme der relaterte tråder eller prosesser kjører samtidig på forskjellige prosessorer. Vanligvis gjelder dette tråder som hører til samme prosess. Siden hver tråd får en prosessor **sikrer dette samkjøring** av trådene, og at de alltid er klare til å kommunisere seg imellom på samme tidspunkt. Dette medfører også **færre trådkifter** ved koordinering/kommunikasjon mellom trådene. Gang scheduling er derfor nyttig for applikasjoner som krever interkommunikasjon mellom tråder, som de som er medium/fin i synkroniseringsdetaljnivå.



Dedicated processor assignment

Dedikert prosessor-tildeling er en ekstrem form for *gang scheduling* og dedikerer en gruppe prosessorer til en applikasjon for utførelse. Hver av trådene i applikasjonen får sin egen prosessor til applikasjonen er fullført. Dette sikrer **samkjøring av trådene**, og gir **ingen trådkifter** – noe som videre medfører full hastighet på applikasjonen. Det er dog **dårlig utnyttelse av prosessorer**, ettersom at de blir stående i tomgang for eksempel ved en I/O-forespørsel. Både gang scheduling og dedicated processor assignment er gode til å unngå problemer som *trashing* og *prosessor-fragmentering*, som er situasjonen hvorav noen prosessorer står i arbeid, mens noen står på tomgang.

Dynamic Scheduling – *Fra brukerfokus til systemfokus*

Dynamisk tidsstyring er den mest effektive og minst brukte algoritmen. Antall tråder i en prosess kan endres under utførelse, noe som lar operativsystemet tilpasse lasten for å forbedre utnyttelse av ressursene. Det vil si at antall prosessorer til en prosess også kan variere tilsvarende. I Dynamisk tidsstyring kan og bør programmet og systemet samhandle om tilordning av tråder til prosessorer.

Tidsstyring ved multicore vs. multiprosessor

Mange multikjernesystemer og multiprosessorsystemer benytter samme algoritme for tidsstyring, hvorav load sharing ofte brukes. Lastdeling sikrer utnyttelse av prosessorer, men ettersom at trådene ikke utføres på samme prosessor gir det ikke full utnyttelse av caching. I multiprosessorsystemer er det viktig å sikre høy utnyttelse av prosessorkraft, mens i multikjernesystemer er det viktig å holde felles minneaksess innenfor brikken. Ved behov for felles lageraksess kan man bruke nabokjerner som deler cache.

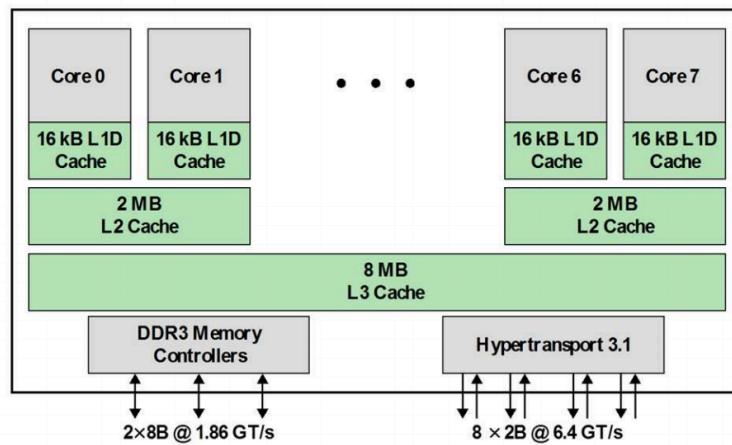


Figure 10.3 AMD Bulldozer Architecture

Tidsstyring ved sanntid (Real-time Scheduling)

Sanntidsproblemer

Sanntidsberegninger er beregninger hvorav korrektheten til systemet ikke kun avhenger av det logiske resultatet til beregningen, men også tiden resultatet produseres. Sanntidsoppgaver er ofte knyttet til en frist, og vi kategoriserer ofte oppgavene i **hard real-time tasks** som er oppgaver som må møte fristen, hvis ikke vil den gjøre skade eller skape en fatal error i systemet, og **soft real-time tasks** som har en frist som er ønskelig å nå, men ikke obligatorisk. Vi kan også dele inn oppgavene i aperiodiske og periodiske oppgaver. En **aperiodisk oppgave** har en frist for når den må starte og/eller fullføres, mens en **periodisk oppgave** kan gjøres enten «en gang per periode T» eller «med T enheter imellom».

Følgende karakteristikker er viktige i sanntidsoperativsystemer.

- Determinisme (forutsigbarhet), f.eks. tids-avbruddshåndtering
- Responsivitet, f.eks. behovsbehandling
- Brukerkontroll, f.eks. prioritetsfastlegging
- Pålitelighet, f.eks. tjenestegarantering
- Feiltilpasning, f.eks. tjenesteprioritering

I et sanntidssystem ligger fokuset på å fullføre alle hard real-time tasks innen fristen og så mange soft real-time tasks (eller starte) innen fristen. Ofte kan ikke OSet operere direkte med frister, men designes isteden til å være så responsive som mulig. En metode for å løse dette er prioritet kombinert med en klokkebasert avbruddsmekanisme, eller enda bedre å benytte

umiddelbart avbrudd. I dette tilfellet vil OSet reagere umiddelbart med mindre det er låst i en kritisk kode-seksjon. Tidsstyringsalgoritmer for sanntid kan deles inn i tre klasser:

Statisk tabelldrevede

Gitt et sett med oppgaver og deres egenskaper, beregnes tidsstyringen (tabellen) offline. Denne brukes for periodiske oppgavesett, og krever at hele tidsstyringen gjøres på nytt hvis oppgavesettet endres.

Statisk prioritetsdrevne med avbrudd

Gitt et sett med oppgaver og deres egenskaper, tildeles hver oppgave en fast prioritet. Prioritetene kombineres med en avbruddsmekanisme. Dette brukes også for periodiske oppgavesett. Primært benyttes tidligste tidsfrist først, og deretter av høyest prioritet først.

Dynamisk planleggingsbaserte

Gjennomførbarhet bestemmes dynamisk ved run time istedenfor statisk på forhånd. En oppgave aksepteres kun dersom det er gjennomførbart å møte tidsbegrensningene.

Dynamisk best effort tidsstyring

Gjør ingen gjennomførbarhetsanalyse på forhånd. Systemet forsøker å møte alle fristene og aborterer prosessene hvor fristene ikke kan møtes. Denne prioriterer tidligste tidsfrist først, og bruker deretter en FCFS-orden.

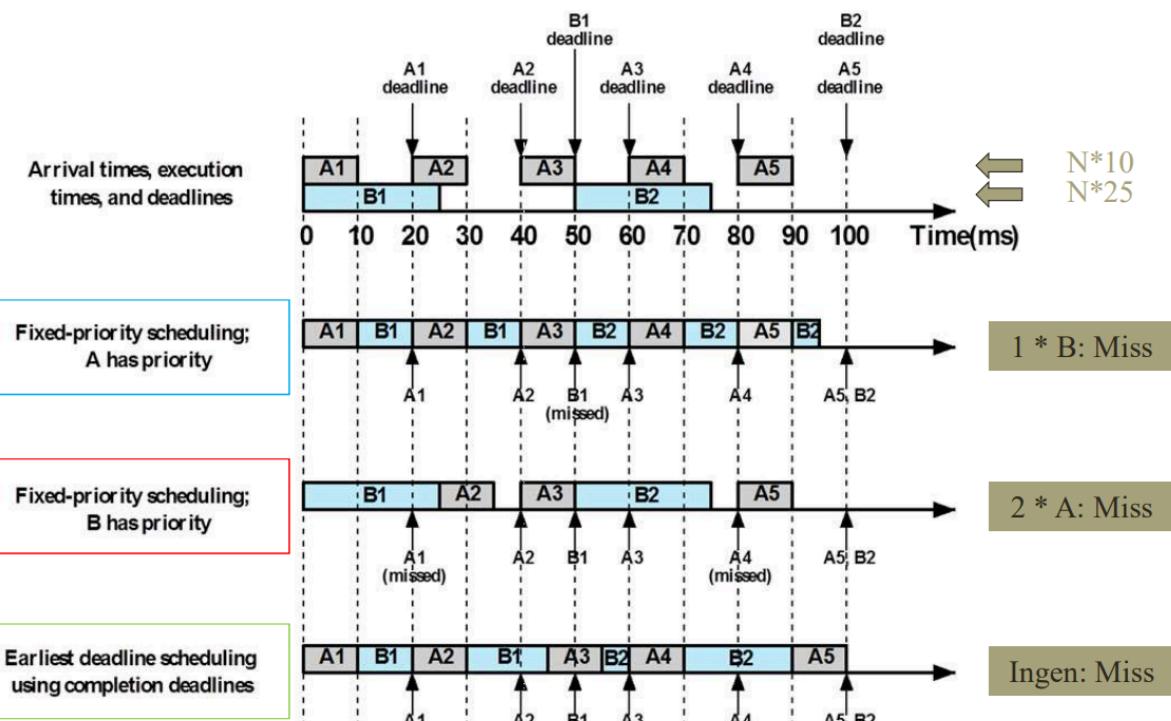


Figure 10.5 Scheduling of Periodic Real-time Tasks with Completion Deadlines (Based on Table 10.3)

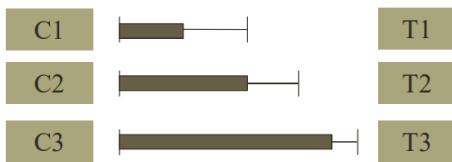
Rate Monotonic Scheduling (RMS)

RMS er en av de bedre metodene for å løse tidsstyring for periodiske oppgaver. RMS gir prioritet til oppgaver på basis av perioden deres hvorav en oppgave med kortere periode får høyere prioritet, og blir betjent først og en oppgave med lang periode får lavere prioritet. For å garantere at alle frister skal møtes må summen av prosessorutnyttelsen ($U = C/T$) for hver oppgave holde ulikheten under, hvor 1 er den totale utnyttelsen til prosessoren. RMS har ofte unyttelse inntil 90% og garanterer at den når alle frister så lenge følgende ulikhet holder:

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

Hvor C = utførelsestiden, andel prosessortid som kreves for oppgaven, mens T = perioden til oppgaven.

Informasjonen som kreves kjent for å rangere prosesser og å sjekke garantikravet er liten, men garantikravet i seg selv er ganske tøft da det forutsetter en slakk på opp i mot 30.7% = (1-0.693) i systemet (som er når antall prosesser, n , går mot uendelig).



Invertering av prioriteter

Invertering av prioritet 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 lavere prioritetsoppgave har låst en ressurs som en oppgave av høyere prioritet trenger. Problemet er mer kritisk ved **ubegrenset prioritetsinvertering** som er når varigheten til en protitetsinvertering avhenger av både tiden det tar å håndtere den delte ressursen og av de uforutsigbare handlingene til de urelaterte oppgavene. Det er to løsninger for dette:

- **Prioritetsarv** som gir oppgaver av lavere prioritet den samme prioriteten som en oppgave av høyere prioritet dersom den holder ressursen som den andre venter på. Prioriteten økes så fort oppgaven av høyere prioritet er blokkert på ressursen, og senkes tilbake så fort den frigir ressursen.
- **Prioritetstak** har en prioritet assosiert til hver ressurs. Prioriteten til ressursen er høyere enn prioriteten til brukeren med høyest prioritet. Tidsstyreren tildeler denne prioriteten dynamisk til oppgaver som aksesserer ressursen. Så fort oppgaven er ferdig med ressursen, resettes den til normal.

Kapittel 11: I/O-håndtering og disk scheduling

I/O-enheter

Eksterne enheter som brukes i I/O kan gruppertes i de tre kategoriene **human readable** (mus, keyboard), **machine readable** (USB, sensorer etc.) og **kommunikasjon** (passende for kommunikasjon med eksterne enheter, som moden). De tre klassene skiller fra hverandre med tanke på kapasitet, hastighet, representasjon og overføring, og resten i tabellen under.

Datarate	Det kan være forskjeller i dataoverføringsratene (bps).
Applikasjon	Bruken av enheten har innflytelse på softwaren i OS samt støtteverktøy.
Kompleksitet på kontroll	En printer krever et enkelt kontrollgrensesnitt, mens en disk er mer kompleks.
Enhett for overføring	Data kan overføres som en strøm av bytes eller characters eller i større blokker.
Datarepresentasjon	Ulike enheter representerer data ulikt.
Errorforhold	Error, hvordan de rapporteres, konsekvensen deres og tilgjengelige responser er veldig forskjellig.

Organisering av I/O-funksjonen

I/O kan utføres ved de tre teknikkene; programmert, avbruddsdrevet eller ved direkte minneaksess.

1 – Programmert I/O

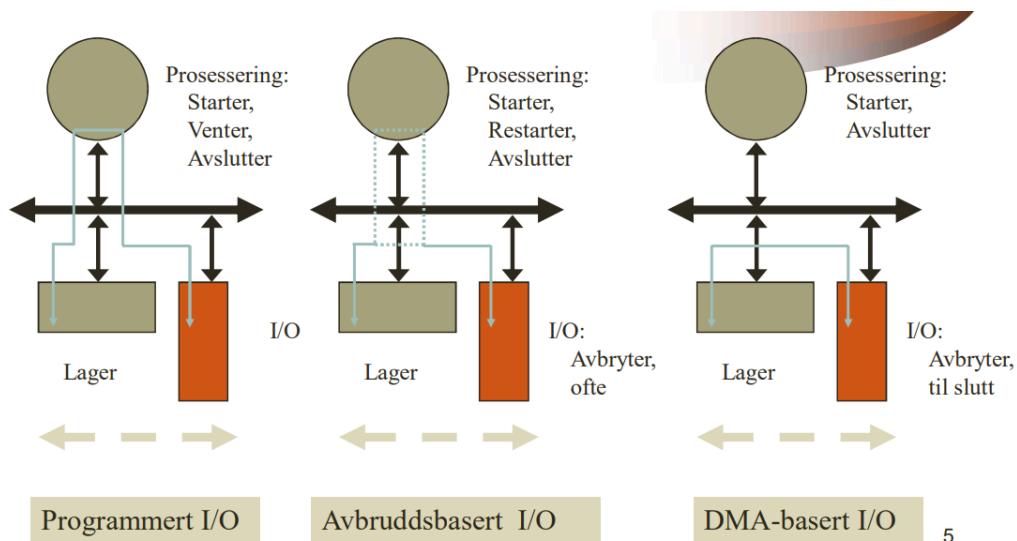
I programmert I/O etterspør prosessoren en kommando fra en I/O-modul på vegne av en prosess. Prosessen settes i *busy wait* til operasjonen er fullført før den fortsetter.

2 – Avbruddsdrevet I/O

Avbruddsdrevet I/O er når prosessoren etterspør en I/O-kommando på vegne av en prosess. Det gir to muligheter. Dersom I/O-instruksjonen fra prosessen ikke er blokkerende, fortsetter prosessoren å kjøre instruksjonene prosessen etterspurte. Hvis den derimot er blokkerende, setter OS prosessen i en blokkert tilstand og henter inn en ny prosess.

3 – Direkte minneaksess (DMA)

En DMA-modul kontrollerer flytting av data mellom primærminnet og I/O-modulen. Prosessoren sender etterspørsel for overføring av blokkdata til DMA-modulen og avbrytes kun når hele blokken er overført.

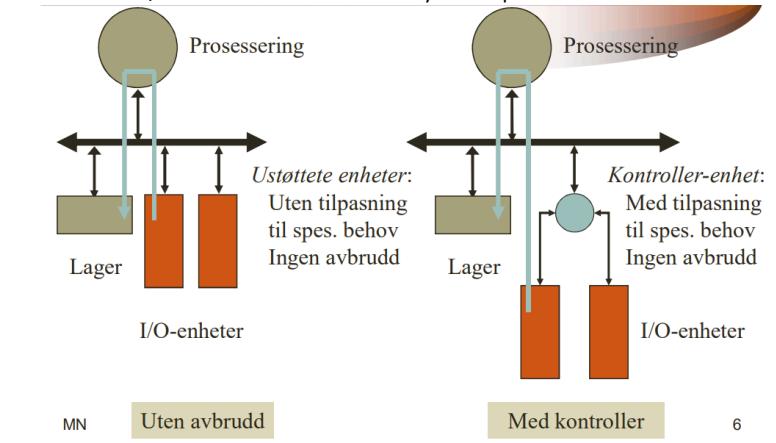


Evolusjon av I/O-funksjonen

Ettersom datasystemer har utviklet seg, har også I/O-funksjonen utviklet seg. Dette har etter hvert medført at I/O-modulen etterhvert har liten innblanding fra prosessoren, og fungerer som en selvstendig enhet.

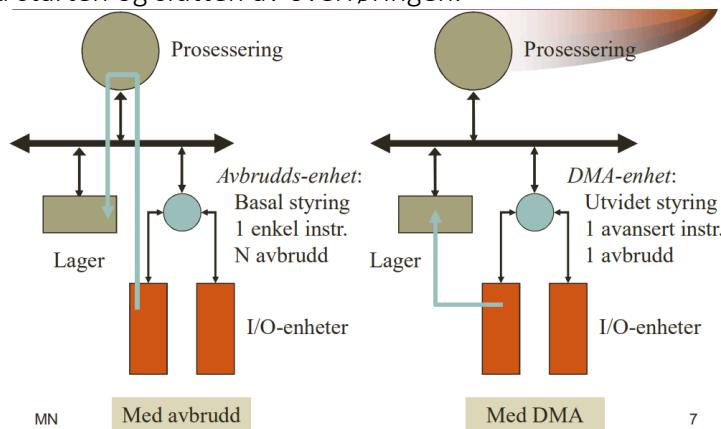
Kontroller

Kontroller-funksjonen uten avbrudd legger til rette tilpasning av spesielle behov. I dette steget skiller også prosessoren fra spesifikke detaljer på eksterne grensesnitt. Før kontroller-funksjonen ble eksterne I/O-enheter direkte styrt av prosessoren.



Avbrudd og DMA

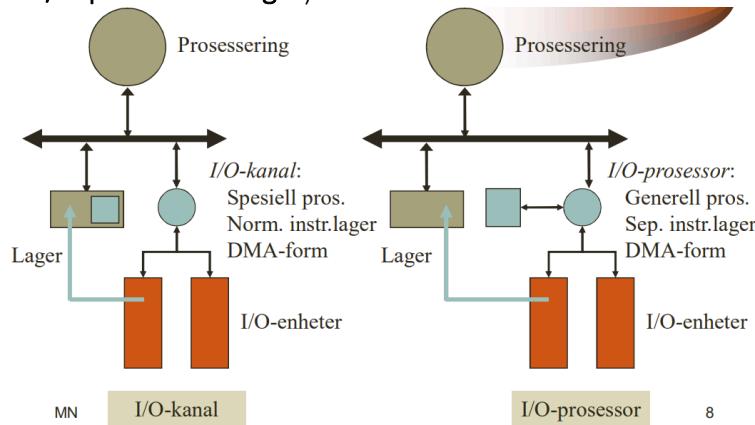
Med samme modell som over ble avbrudd introdusert. Dette førte til at prosessoren ikke lenger trengte å vente på gjennomføringen av I/O-operasjoner, noe som særlig økte effektivitet. Videre får I/O-modulen direkte kontroll over minnet ved bruk av **direkte minneaksess (DMA)**. Dette gjør at I/O-modulen kan flytte blokkdata til eller fra minnet uten å involvere prosessoren, bortsett fra ved starten og slutten av overføringen.



I/O-kanal og I/O-prosessor

Etter hvert ble I/O-modulen forbedret til en separat prosessor med et spesialisert instruksjonssett skreddersydd for I/O (dette steget kaller vi for **I/O-kanal**). CPUen kan da dirigere I/O-prosessoren til å utføre I/O-programmer i primærminnet. Både henting og utførelse kunne I/O-prosessoren gjøre uten innblanding av prosessoren. Slik kan prosessoren spesifisere sekvenser med I/O-aktiviteter, og kun avbrytes når hele sekvensen er fullført. Etter hvert får I/O-modulen

sitt eget **lokale minne** for at den skal kunne kontrolleres med minimal prosessorinnblanding (dette kaller vi for **I/O-prosessor-steget**).



Direkte minneaksess (DMA)

DMA kan ta kontrollen over systembussen som en prosessor. Dette er nødvendig for å kunne overføre data til og fra minnet over systembussen. Når en prosessor ønsker å lese eller skrive en blokk med data, sender den en forespørsel til DMA-modulen sammen med informasjon.

Designutfordringer i et operativsystem

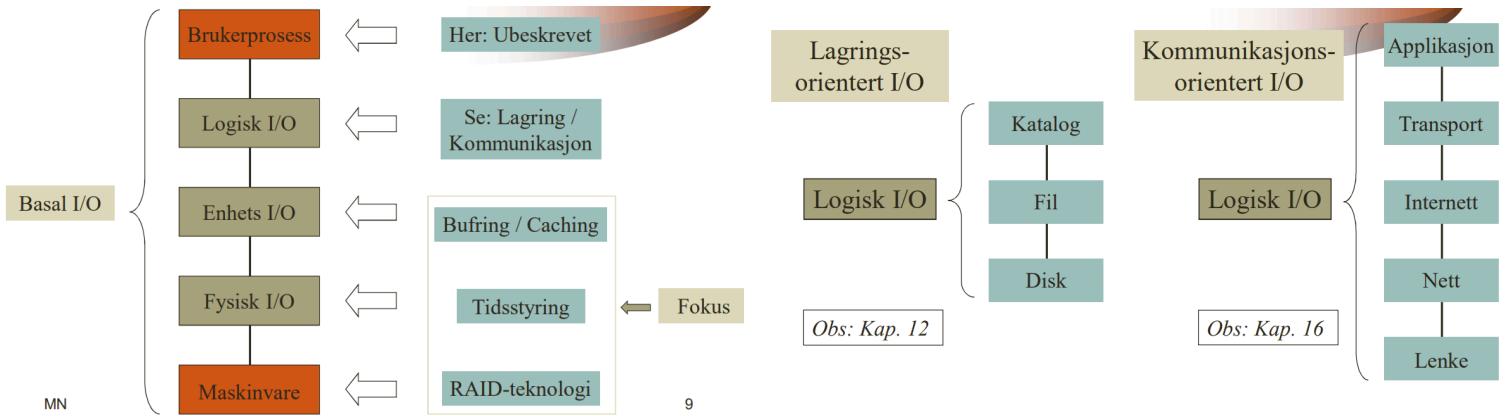
En stor utfordring i I/O-design er effektivitet, ettersom at prosessoren fra før av er mye mer effektiv enn I/O-modulen. Den ene delen går på effektivitet i disk I/O, mens den andre delen går på generalitet. Man ønsker å håndtere forskjellige enheter på en uniform måte for enkelhet og for å hindre error. Generalitet i åraksis kan være vanskelig.

For å oppnå generalisering ønsker man en hierarkisk, modulær ordning av I/O-funksjonen. Slik kan man gjemme unna en rekke av detaljene, slik at det kan sees på som generelle funksjoner som read, write open, close, lock og unlock.

I/O-struktur

I/O-strukturen er bygget opp hierarkisk som figuren under, med tre lag av abstraksjon: *logisk I/O*, *enhets-I/O* og *fysisk I/O* mellom brukerprosessen og hardwaren.

- **Logisk I/O:** håndterer de logiske ressursene og er dermed ikke med på den fysiske kontrollen av enheten. Dette inkluderer generelle I/O-funksjoner på vegne av brukerprosesser, som å benytte kommandoer som open, close, read og write.
- **Enhets I/O:** etterspurte operasjoner og data konverteres til passende I/O-instruksjoner, kommandoer og kontrollordre. Buffere/caching kan brukes for å forbedre utnyttelse.
- **Fysisk I/O (Tidsstyring og kontroll):** den faktiske tidsstyringen av I/O operasjonene i tillegg til kontroll av operasjoner skjer her. Dette laget håndterer avbrudd, og henter og rapporterer I/O-status.



I/O-buffering

For å unngå overhead og å øke effektivitet, kan det være nyttig å utføre input-overføringer før en forespørsel gjøres og output-overføringer etter at den er gjort. Dette er *buffering*. Uten I/O-buffering vil det oppstå blokkering av enheter og prosesser ved utførelse og swapping. Buffere hjelper til å jevne ut variasjonen i I/O-bruk mellom prosesser. Full utnyttelse av bufferen får man i systemer hvor det varierer mellom I/O-oppgaver og andre prosess-oppgaver. Bufferen vil da hjelpe å skjule variasjonen i hastigheten til de ulike oppgavene. Et eksempel på dette er multiprogrammeringssystemer hvor bufferen øker ytelsen til de individuelle prosessene. Ved kun I/O-oppgaver vil forsinkelsen etterhvert ta igjen tiden spart i bufferen, og man må uansett vente.

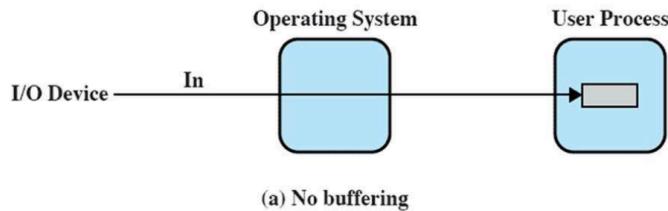
Vi skiller mellom to typer I/O-enheter. **Blokk-orienterte enheter** holder informasjon i form av blokker som vanligvis har en bestemt størrelse, mens **strøm-orienterte enheter** overfører data inn og ut av enheten som en bytesstrøm, uten blokkstruktur.

Uten buffer

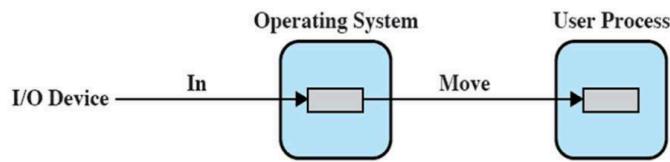
Uten bruk av buffer gir blokkering ved utførelse og swapping av prosesser.

Singel buffer

Singel buffer unngår blokkering ved utførelse og swapping. En enkelt buffer fungerer slik at input-overføringer gjøres til bufferen først. Dette heter *reading ahead* fordi det gjøres med forventningen om at det til slutt vil være behov for blokken som leses. Dette er rimelig, ettersom at data ofte aksesseres sekvensielt. Resultatet er økt hastighet, men med en mer komplisert logikk.



(a) No buffering



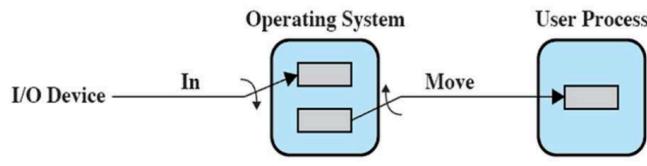
(b) Single buffering

Dobbel buffer

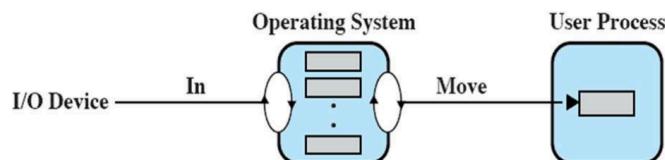
Dobbel buffer er en forbedring av singel-buffer og utjenvner mindre hastighetsvariasjoner. En prosess kan overføre data til/fra en buffer, mens OSet overfører til/fra en annen buffer. Kostnaden er også her at det er mer komplekst.

Sirkulær buffer

Sirkulær bufring utjenvner større hastighetsvariasjoner. Dobbel buffering kan være utilstrekkelig dersom prosessen utfører hyppige utbrudd med I/O. Vi kan da benytte mer enn to buffere, hvor hver buffer er en enhet i den sirkulære bufferen. Dette er samme modell som i bounded-buffer producer/consumer i kapittel 5.



(c) Double buffering



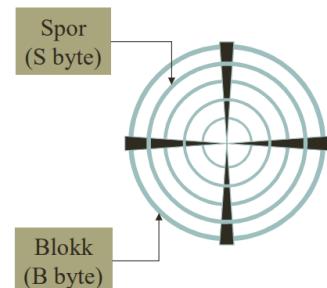
(d) Circular buffering

Tidsordning av disk

Gapet mellom disk og prosessorer og main memory er veldig stort, noe som gjør det enda viktigere å ha en effektiv tidsordning av disk.

Aksessformen på disk er 1) å søke frem til sporet, 2) rotere fram til blokken, og 3) overføre hele blokken. Dette gir følgende aksesstid:

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



Tidsstyringsregler

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

First-In-First-Out – FIFO

Den enkleste formen for tidsstyring som har fordelen at det er rettferdig. FIFO har tilnærmet samme ytelse som tilfeldig tidsstyring, som ikke er veldig bra.

Prioritet – PRI

Et system som benytter prioritet er ikke designet for å utnytte disk, men for å møte andre krav fra OS. Ofte prioriterer man kortere jobber, for at de skal kunne gå gjennom systemet raskt og gi god interaktiv responstid.

Last-In-First-Out – LIFO

LIFO viser seg å være bedre til disk, fordi det har tendenser til mindre søking etter riktig spor (utnytter lokalitet). Problemet er at det er mulig med utsulting (starvation).

Shortest-Service-Time-First – SSTF

Valget i SSTF faller på disken som krever minst bevegelse av diskarmen fra den nåværende posisjonen. Dette gir god ressursutnyttelse.

SCAN

SCAN-algoritmen fungerer mye på samme måte som en heis. Diskarmen beveger seg kun én vei, frem til den har tilfredsstilt alle forespørsler i den retningen. Deretter snus service-retningen til motsatt retning, og scanningen gjentas. Denne oppfører seg nesten likt som SSTF, men utnytter ikke lokalitet like bra.

C-SCAN

For C-SCAN er scanningen kun rettet i en retning. Etter en fullført scan returnerer den dermed til andre siden av disken, for å gjennomføre scanningen på nytt. Dette reduserer maksimal forsinkelse for nye forespørsler.

N-step-SCAN og FSCAN

N-step-SCAN segmenterer diskforespørselskøen inn i subsekvenser med lengde N. Subsekvensene prosesseres én av gangen ved bruk av SCAN. Når en kø prosesseres legges nye forespørsler i en annen kø. Dersom mindre enn N forespørsler er tilgengelig på slutten av scannen, så prosesseres alle ved neste SCAN. Denne har samme ytelse som SCAN for store verdier av N og garanterer tjeneste.

FSCAN er en rutine som bruker to subsekvenser. Når en scan begynner er alle forespørslene i en av køene, men den andre tom. Under scanningen legges alle nye forespørsler i den tomme køen. Derfor må de nye forespørslene vente på at de gamle prosesseres før de betjenes.

RAID

RAID (Redundant Array of Independent Disks) er en standard ordning for et multidisk-databasesystem som består av syv nivåer (fra null til seks). Disse nivåene er ikke hierarkisk ordnet, men utpeker designarkitekturen som deler tre egenskaper:

1. OS seer på RAID som en enkel logisk driver.
2. Dataen distribueres på tvers av fysiske de fysiske driverne.
3. Redundant diskkapasitet brukes til å lagre paritetsinformasjon, noe som garanterer gjenopprettelse av data dersom det oppstår diskfeil.

RAID-strategien bruker flere diskdrivere og distribuerer data på en slik måte at samtidig aksess av data fra flere drivere er mulig, noe som øker ytelsen av I/O og gir lettere inkrementell økning i kapasitet. Ved bruk av mange enheter øker sjansen for feil, som er grunnen for at RAID lagrer paritetsinformasjon for å kunne gjenopprette data. Av de syv nivåene beskrevet, brukes nivå 0,1,5 og 6 vanligvis.

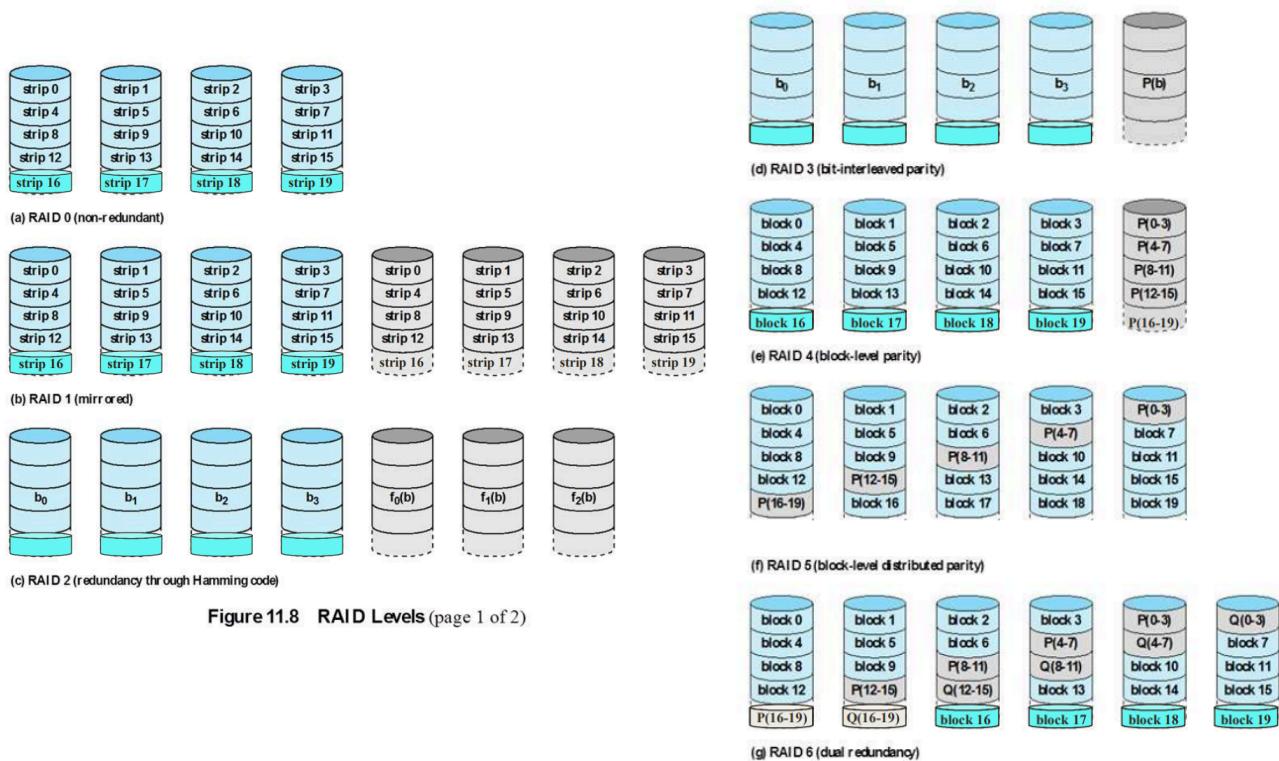
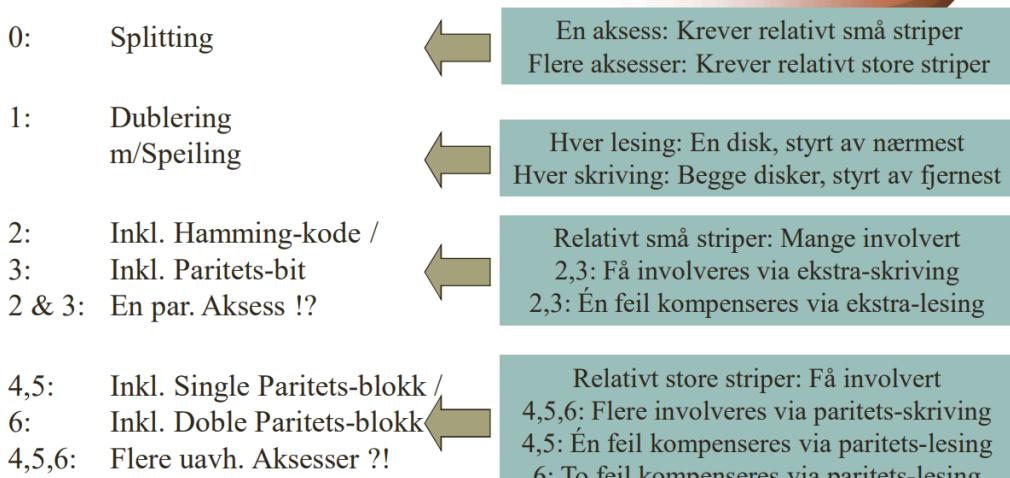


Figure 11.8 RAID Levels (page 1 of 2)

Figure 11.8 RAID Levels (page 2 of 2)



MN

Bruk av sek. lager: Ekstra HW-støtte for økt SW-ytelse

24

Level 0

Level 0 inneholder ikke redundans for å øke ytelse eller beskytte dataen. I RAID 0 distribueres bruker og systemdata på tvers av diskene i listen. Dette gjør at data distribuert på forskjellige diskene kan aksesseres samtidig. Videre *stripes* dataen på tvers av tilgjengelige diskene. Bruker- og systemdata sees på som om de er lagret på en logisk disk. Denne logiske diskene deles opp i *strips* som mappes ved round robin til fysiske diskene i RAID-arrayen. Fordelen er at i en n-disk array, er de første n logiske stripsene fysisk lagret som den første stripene på hver av de n diskene, noe som danner en *stripe*, den andre stripen ligger på den andre stripene i hver disk osv. Dersom en I/O-forespørrelse består av flere logiske etterfølgende strips, kan de håndteres i parallel.

Level 1

I RAID 1 oppnås redundans ved en hensiktsmessig duplisering av all dataen. Dette kalles for speiling (mirroring) hvor hver logisk stripe mappes til en separat fysisk disk slik at hver disk i arrayen har en speildisk med samme data. Fordelen med speiling er at lese-forespørslar kan gjøres til begge diskene, der den med minst tid velges. I tillegg må skrive-forespørslar gjøres til begge diskene, men dette kan gjøres i parallel. Siden det ikke er noe paritetsbit, vil dette gå fortare enn for RAID 2-6. Dersom det oppstår feil er dette også lett å ordne siden dataen kan aksesseres fra den andre diskene. Ulempen med RAID 1 er overhead. Vi velger å bruke RAID 1 i kritiske systemer som trenger backup av dataen eller i systemer som har mange lese-forespørslar.

Level 2 & 3

RAID 2 og 3 bruker en parallel aksessteknikk. Typisk er alle driverne synkronisert slik at hver disk head er i samme posisjon på en gitt tid. RAID 2 og 3 bruker også striping, men disse er relativt små.

RAID 2 bruker en hamming-kode for å korrigere single-bit error og detektere dobbel-bit error. RAID 2 krever også ganske stor plass, med en overhead på log av antall data diskene inneholder. Den vil kun være effektiv i bruk dersom feil forekommer ofte.

RAID 3 bruker kun en enkelt redundant disk. I tillegg bruker RAID 3 kun en paritetsbit istedenfor error-korreksjonskode. Siden små strips brukes kan RAID 3 oppnå høy dataoverføringsrate. På

den andre siden kan kun en I/O-forespørsel gjøres av gangen, som gjør at RAID 3 ikke er så bra i transaksjonsorienterte miljøer.

Level 4

RAID 4, 5 og 6 benytter en uavhengig aksessteknikk. Siden hver disk opererer uavhengig, kan I/O-foespørsler behandles i parallel. I RAID 4-6 er stripsene relativt store, og få er involvert. RAID 4 benytter en bit-by-bit-paritetsblokk. Hver gang en skriving skjer, må både dataen og paritetsbiten oppdateres. Dette introduserer en potensiell flaskehals.

Level 5

RAID 5 distribuerer paritetsbitene på alle diskene, som typisk allokeres ved round-robin. distribusjonen av paritetsstrips på driverne unngår en potensiell I/O-flaskehals som RAID 4 har. Dessuten vil ikke tap av en disk i RAID 5 gi tapt data.

Level 6

I RAID 6 benyttes to ulike paritetsberegninger som lagres i separate blokker på forskjellige disker. Derfor har en RAID 6 array hvor en bruker krever N disker, totalt N + 2 disker. P og Q er to ulike datasjekkalgoritmer, hvor en bruker OR, som i RAID 4 og 5 mens den andre er uavhengig. Dette gjør det mulig å regenerere data selv om to disker med brukerdatafeil oppstår. Fordelen med RAID 6 er at den gir svært høy datatilgjengelighet hvorav tre disker må feile innen MTTR (mean time to repair) for at data skal gå tapt. RAID 6 har dog en høy straff for skriving, fordi hver skriving påvirker de to paritetsblokkene. Sammenlignet med RAID 5 kan RAID 6 ha en overall drop på 30% i skrivingstid.

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

Disk cache

Cache refererer til et minne som er raskere og mindre enn primærminnet og ligger mellom primærminnet og prosessoren. Hensikten er å redusere aksesstid ved bruk av lokalitetsprinsippet. *Disk cache* er en buffer i primærminnet for disksektorer.

- Least Recently Used (LRU) – Krever tidsmerke for hver page til en prosess.
- Least Frequently Used (LFU) – Krever referanseteller for hver page til en prosess.

- Frequenced Based Stack (FBS) – Best i dette tilfellet. Krever seksjonert stakk for hver prosess.

Frequenced Based Stack – FBS

Blokkene er organisert i en stack som med LRU. En øvre del av stacken er utpekt til en ny seksjon. Når det er et treff i cache, vil den refererte blokken flyttes til toppen av stacken. Hvis blokken ikke allerede var i den nye seksjonen, økes referansecounten med 1. Siden den ikke økes hvis blokken lå der fra før av, vil ikke referansecounten til blokker som har etterfølgende referanser innenfor et kort tidsintervall økes. Dette løser et av problemene med LFU, med at blokker som har mange referanser på veldig kort tid får en høy referansecount, når de i utgangspunktet ikke burde ha det.

En utfordring er at når blokken kommer over i den eldre seksjonen, vil den raskt swappes ut hvis den ikke refereres raskt ettersom at den har lavest referencecount av de som er der. Dette kan løses ved å introdusere en middels-seksjon.

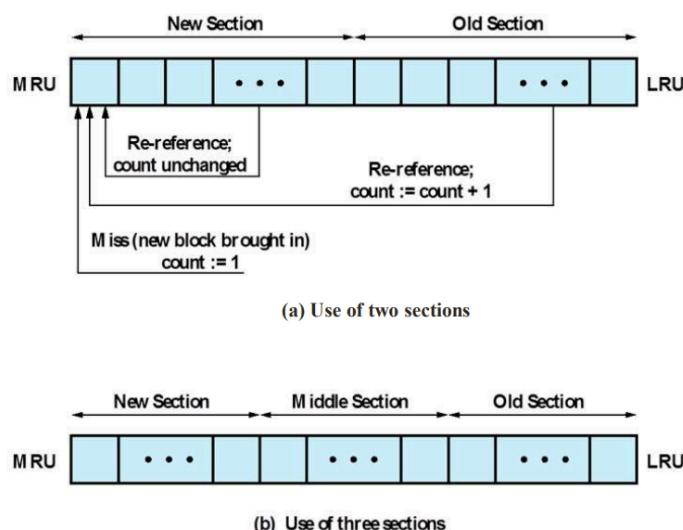


Figure 11.9 Frequency-Based Replacement

Kapittel 12: Filhåndtering

For en bruker er **filsystemet** en av de viktigste delene i et operativsystem. Et filsystem lar brukeren opprette filer med følgende egenskaper:

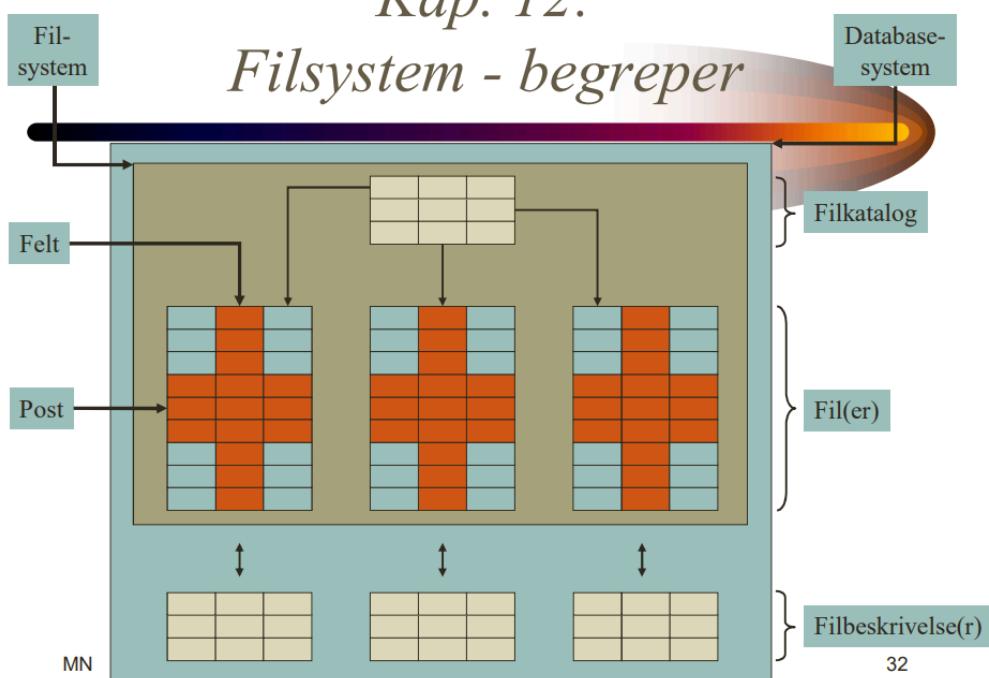
- **Langtidseksistens:** Filer lagres i sekundærminnet og slettes ikke når brukeren logger av.
- **Deling mellom prosesser:** Filer har navn og kan ha tildelte aksesser som tillater deling.
- **Struktur:** En fil kan ha en intern struktur tilpasset en applikasjon. I tillegg kan filer organiseres hierarkisk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

I tillegg har filsystemet en rekke funksjoner som kan utføres på filer. Blant annet Create, Delete, Open, Close, Read og Write.

Filstruktur

En fil består ofte av **felt**, **post**, **fil** og **database**. Et felt inneholder et enkelt element av data, en post er en samling av relaterte felter, mens en fil er en samling av lignende poster som behandles som en entitet. Aksesskontroll (hvem som har tilgang til hva) skjer ofte på filnivå. En **database** er en samling av relatert data.

Kap. 12: Filsystem - begreper



Filsystem (File Management System)

Et filhåndteringssystem er et sett med system-programvare som gir tjenester til brukere og applikasjoner ved bruk av filer. Som for eksempel aksessering av filer. Et filhåndteringssystem har oppgavene;

- Møte brukerens datahåndteringsbehov som lagring og filoperasjoner (nevnt over). Dette inkluderer kontrollert aksess, flytting av innhold mellom filer, angre og gjenopprette filer og å identifisere filer gjennom navn (ikke id).
- Sørge for at dataen er gyldig.
- Optimere ytelse både via responstid (bruker) og throughput (system)
- Støtte I/O med lagringsenheter.
- Minimere eller eliminere muligheten for tapt eller ødelagt data.
- Gi et standardisert sett med rutiner for I/O grensesnitt til brukerprosesser.
- Gi I/O-støtte for flere brukere.

Filsystemarkitektur

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

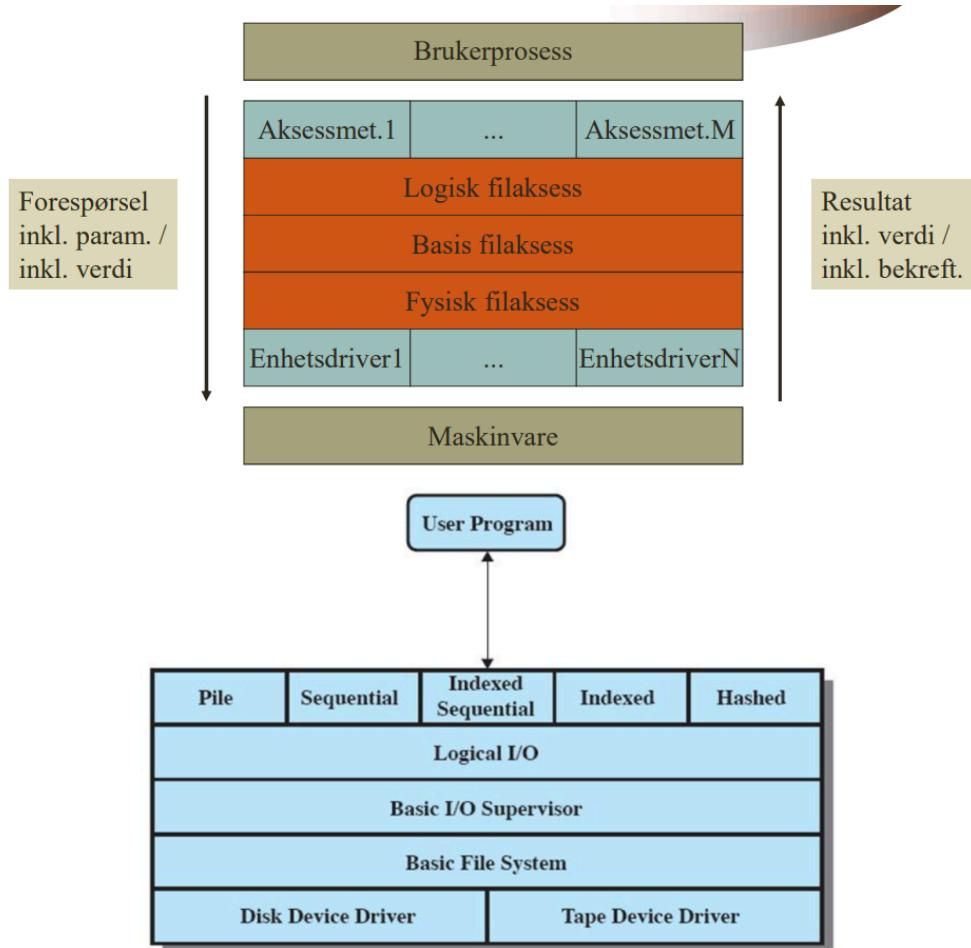
Enhetsdriverer er det nederste laget og kommuniserer direkte med eksterne enheter, kontrollere eller kanaler. Enhetsdriveren starter I/O-operasjoner på enheter og prosesserer fullførelsen av en I/O-forespørsel. En enhetsdriver er for eksempel disk.

Fysisk filaksess er primærgrensesnittet til enheter utenfor datasystemet. Dette nivået utveksler datablokker med enhetsdrivere og har som oppgave å plassere disse blokkene.

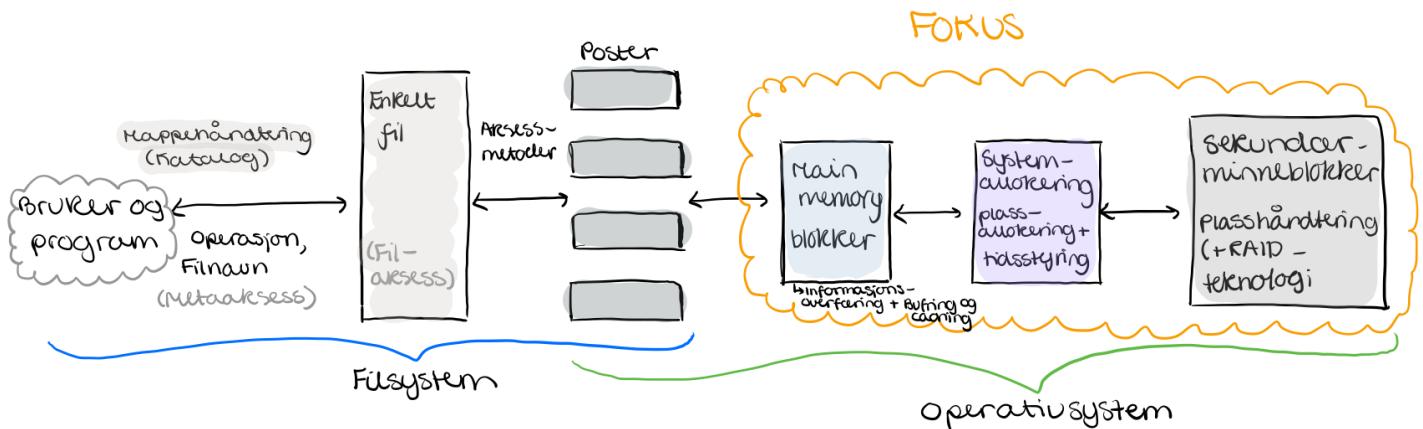
Basis filaksess har ansvar for all I/O-filinitiering og terminering. Her opprettholdes kontrollstrukturer for å håndtere enhets-I/O, tidsstyring og filstatus. Basis filaksess har ansvar for tidsstyring av aksesser til enhetsdrivere for å optimalisere ytelse.

Logisk filaksess sørger for at brukere og applikasjoner kan aksessere poster, og jobber dermed med hele filer. Logisk filaksess har dermed generelle metoder for postene, samt grunnleggende data om filene.

Aksessmetoder gir standard grensesnitt mellom applikasjoner og filsystemet og driverne som holder dataen. Ulike aksessmetoder reflekterer ulike filstrukturer.



Organisering av filsystemet

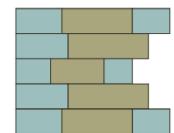


Filorganisering og aksess

Filorganisering er den logiske organiseringen av poster som bestemmes av måten de aksesseres. Viktige faktorer i valg av filorganisering inkluderer *kort aksesstid, enkel oppdatering, forsiktig håndtering av lagringsplass, enkelt vedlikehold og pålitelighet*.

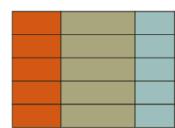
Haugfil / Ustrukturert fil

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. Å gjøre søk i ustrukturerte filer er veldig ineffektivt. Disse brukes dermed når det er vanskelig å organisere data. Fordelen er at de er plasseffektive og lette å oppdatere, men brukes dog ikke så mye.



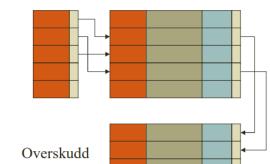
Sekvensiell fil

Sekvensielle filer er den vanligste formen for filer. De har en bestemt lengde på poster med et bestemt antall felter i en fast rekkefølge og baseres på et nøkkelfelt. Sekvensielle søk er lite effektivt, og ikke ideelt hvis man skal gjøre enkeltsøk på nøkkel eller andre kriterier.



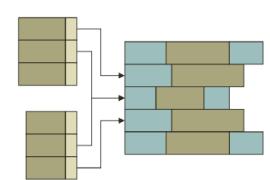
Indeksert sekvensiell fil

Den indekserte sekvensielle filen overkommer en rekke av ulempene med den sekvensielle filen ved å innføre en indeks til filen for å støtte random aksess og en overflow-fil med peker til. For å finne et spesifikt felt, søkes det i indeksfilen for å finne den høyeste nøkkelverdien som er lik eller mer enn den ønskede nøkkelverdien. Søket fortsetter i hovedfilen i lokasjonen til pekeren.



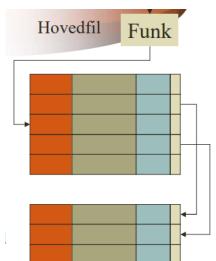
Indeksert fil

Dersom en ønsker å søke basert på noe annet enn nøkkelfeltet er indekserte filer uegnet. For å oppnå denne fleksibiliteten kan man strukturere på flere indekser. I en indeksert fil aksesseres poster etter indeksene sine. Resultatet er at det ikke er begrensninger i plasseringen av posten, så lenge pekeren til minst en indeks peker på posten. I tillegg kan postene være av variabel lengde.



Direkte eller hashet fil

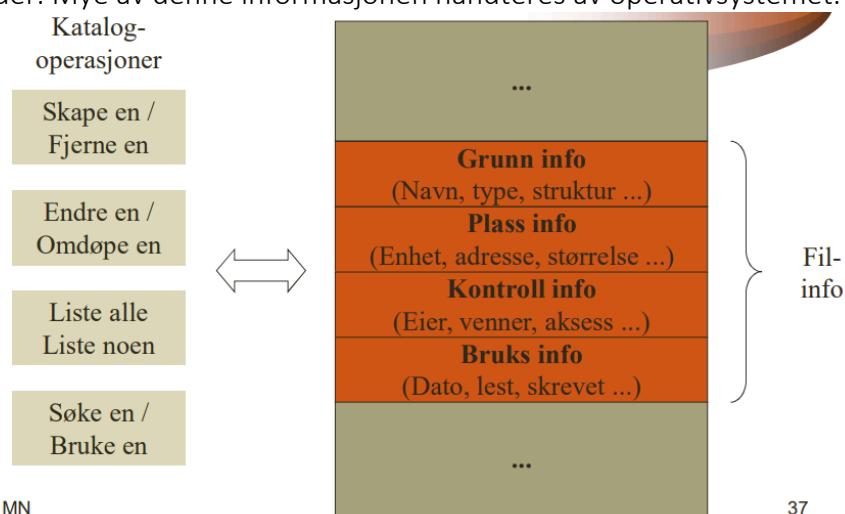
En direkte eller hashet fil utnytter evnen disker har til å direkte aksessere en blokk med kjent adresse. Det kreves derfor et nøkkelfelt, men sekvensiell ordning benyttes ikke. Disse brukes gjerne ved direkte aksess med poster av fast lengde.



B-trær – Man kan også gå fra flate indekser til å benytte B-trær for mer effektiv aksessering.

Kataloger

Kataloger inneholder informasjon om filer, inkludert attributter, plassering og eierskap som vises i figuren under. Mye av denne informasjonen håndteres av operativsystemet.



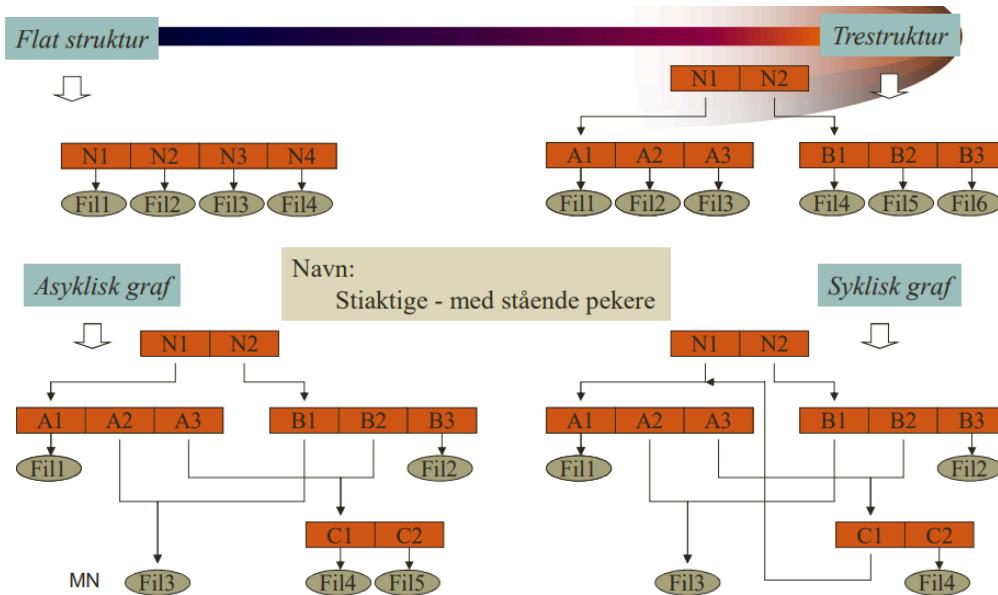
Katalogstruktur

Når en bestemmer strukturen til katalogen må man ta hensyn til de ulike filoperasjonene som skal gjøres; **søk**, opprettelse, terminering, områdesøk og oppdateringer.

Flat struktur (liste): Den enkleste formen for katalogstruktur er en liste med filer. Dette kan representeres ved en sekvensiell fil med filnavnet som nøkkel. Dette er dog ikke en effektiv løsning for operasjonene nevnt over. Det støtter blant annet ikke å ordne filer etter tema og har generelt dårlig struktur.

2-nivå ordning: En 2-nivå struktur har en katalog for hver bruker, og en masterkatalog med adresse- og aksessinformasjon. Dette gjør at navn kun trenger å være unike innenfor en katalog, men det er heller ikke noen form for struktur.

Trestruktur: Trestrukturen er mer fleksibel og benytter også en masterkatalog, som har et antall brukerkataloger, som hver kan ha flere underkataloger med filer. Hver underkatalog kan for eksempel ordnes som en sekvensiell fil.



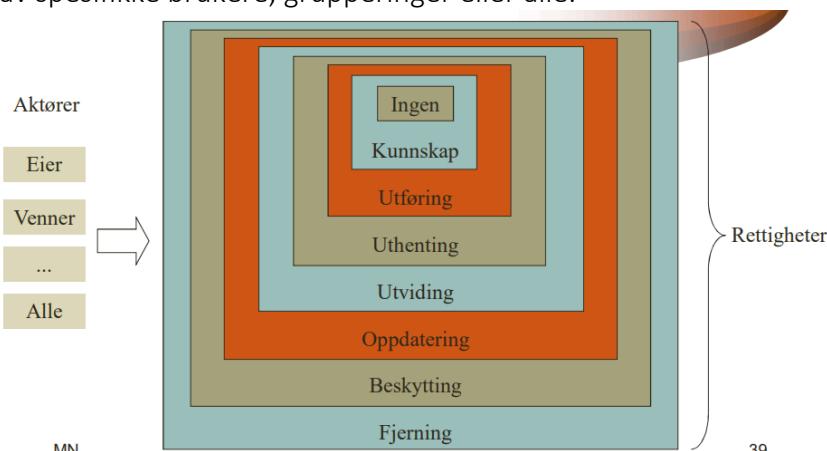
Fildeling

Det er to utfordringer med fildeling; aksessrettigheter og håndtering av samtidig aksess.

Aksessrettigheter

Filsystemet må gi alternative muligheter til aksesskontroll av ulike filer. Vanligvis er det brukere eller grupperinger av brukere som har en tildelt aksess. Ulike former for aksessrettigheter vises i figuren under. Forklaring på side 571 av de ulike. Noen av aksessrettighetene er hierarkiske, som vil si at dersom du har fått aksess for å oppdatere en fil, har du også fått aksess for kunnskap, utførelse, lesing og oppdatering.

Vanligvis har filer en eier, som regel den som opprettet filen. Eieren kan også gi aksess til andre brukere, i form av spesifikke brukere, grupperinger eller alle.



Samtidig aksess

Det er flere måter å kontrollere samtidig aksess. Ved oppdatering av filer kan en bruker låse hele filen, eller bestemte poster i filen. Dette problemet er essensielt readers-writers problem fra kapittel 5.

Post-blokker

Dersom I/O skal utføres må postene organiseres som blokker. Dette introduserer nye spørsmål som om blokkene skal ha fast eller variabel lengde, og størrelsen på dem. Gitt størrelsen til blokkene er det tre metoder som kan brukes:

- **Faste blokker:** Faste lengder brukes, og et integrert antall poster lagres i en blokk. Dette kan medføre intern fragmentering.
- **Variabel lengde, lenket blokking:** poster med variabel lengde brukes og pakkes inn i blokker. Det er ikke noe ledig område, som vil si at noen poster spenner over flere blokker og henger sammen via pekere (lenker).
- **Variabel lengde, uten lenker:** Kun fulle poster ligger i blokkene, som gir noe ubrukt område.

Som regel har blokker fast lengde blokker ettersom at dette forenkler I/O, buffer-allokering i primærminnet og organisering av blokker til disk. Størrelsen på blokkene er som regel store sammenlignet med den gjennomsnittlige poststørrelsen. Dette gir flere poster plassert i hver blokk, som kan sendes i I/O-operasjoner. Hvis blokkene prosesseres eller søkes sekvensielt er dette en fordel.

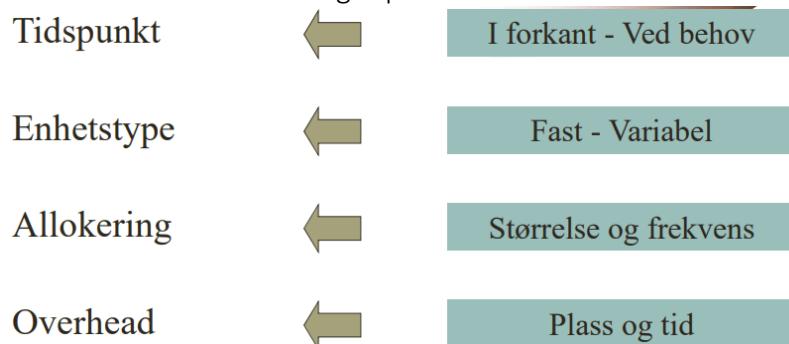
Hver metode har fordeler og ulemper ved ulik bruk. Faste blokker er mest brukt, og er bra ved sekvensielle søk- Variabel lengde er plasseffektivt, men med lenket blokking kreves det 2 I/O-aksesser for henting av blokker.

Minnehåndtering av sekundærminnet

I sekundærminnet består en fil av en samling med blokker. OS eller filsystemet er ansvarlig for allokering av blokker.

Plassallokering

Ved filallokering må man ta hensyn til tre ting: tidspunkt; skal allokeringen skje i forkant, eller ved behov, enhetstype: skal størrelsen på enheten som allokeres en fil være fast eller variabel? Og til slutt, hvilken datastruktur skal vi lagre plassen som er tildelt filene?



Preallokering vs. Dynamisk allokering

Ved preallokering må man vite hvor mye plass man har behov for på forhånd. I noen tilfeller er dette veldig vanskelig, som ofte ender med at en allokerer for mye plass som er veldig lite effektivt. Detfor kan det være en fordel med dynamisk allokering.

Fast eller variabel plass

Man må bestemme om hele filen skal lagres på samme plass, kun en blokk på en plass, eller en kombinasjon. Når man bestemmer hvilken må man gjøre en tradoff. Kontinuitet av plass øker ytelse og mange små blokker øker størrelsen på tabellene som skal holde informasjonen. Fast størrelse gjør det enklere å reallokere, mens variabel størrelse utnytter plassen bedre. Det er to hovedmetoder; **variable, store kontinuerlige deler** som gir god ytelse, men er vanskelig å gjenbruke, eller **blokker** som gir mer fleksibilitet men tar mye plass. Ved variabel lengde kan man benytte *first fit*, *best fit* eller *nearest fit* for plassallokering.

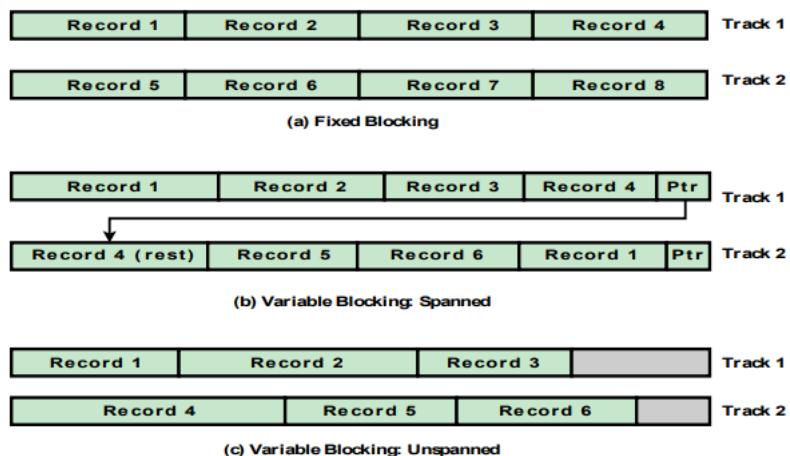


Figure 12.8 Record Blocking Methods

Filallokeringsmetoder

Det er hovedsakelig tre metoder for plassallokering; **Kontinuerlig, kjedet** og **indeksert**.

Kontinuerlig allokering

Kontinuerlig allokering er en preallokerings-metode som allokerer ett enkelt sett med blokker til en fil ved opprettelse. Denne metoden unngår mye overhead i allokeringstabellen fordi den kun trenger en plass per fil. I tillegg gjør den at aksessering av en fil er enkelt siden alt ligger etter hverandre. Ulempen er ekstern fragmentering, at det ikke nødvendigvis er plass til å legge en fil mellom andre filer, og at det dermed blir stående mye ledig plass. En annen ulempe er at en må vite størrelsen til filen på forhånd.

Bruken av en compaction-algoritme fra tid til annen vil samle de forskjellige filene og redusere ekstern fragmentering ved å frigi plass.

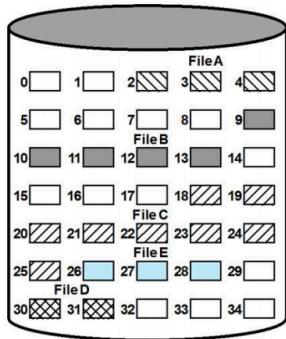


Figure 12.9 Contiguous File Allocation

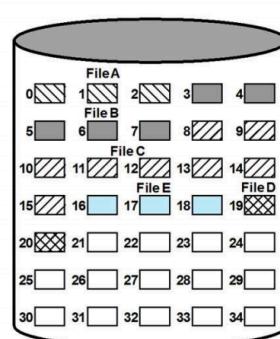


Figure 12.10 Contiguous File Allocation (After Compaction)

Kjedet allokering

På den andre siden av skalaen er kjedet allokering, hvor allokering typisk gjøres på individuelt blokknivå. Kjedet allokering tillater både preallokering og dynamisk allokering (som er mest vanlig). Det gjør det enkelt hvis du skal aksessere en spesifikk blokk, men allokeringstabellen må holde samme informasjon som over, pluss pekere som gir mye overhead. Ettersom at dataen lagres som en lenket liste kreves det også sekvensielt søk, noe som er tidskrevende. Dette gjør at kjedet allokering passer best for sekvensielle filer. En fordel er at det ikke gir ekstern fragmentering og dermed har god utnyttelse av plass.

En konsekvens av kjedet allokering er at lokalitetsprinsippet *ikke* er utnyttet. Man må aksessere mange deler av diskens selv om man ønsker etterfølgende blokker. For å overkomme dette kan man også her innføre en periodisk algoritme som samler filer. Dette vil gi færre diskaksesser.

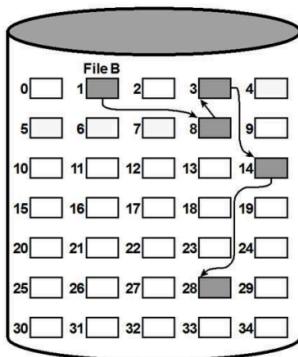


Figure 12.11 Chained Allocation

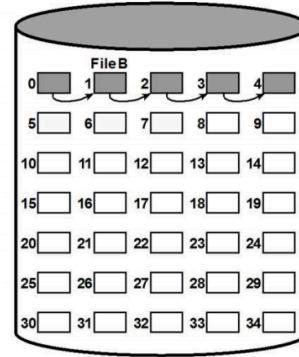


Figure 12.12 Chained Allocation (After Consolidation)

Indeksert allokering

Indeksert allokering løser mange av problemene ved kontinuerlig og kjedet allokering. Allokeringstabellen inneholder en in indeks for hver fil som gir plassen allokert til hver fil. Disse lagres som oftest i en egen blokk som allokeringstabellen peker til. Allokeringen kan være både fast eller variabel. Fast allokering eliminerer ekstern fragmentering, mens variabel lengde

utnytter lokalitetsprinsippet. Indeksert allokering støtter både sekvensiell og direkte aksess til filer og er dermed mest brukt.

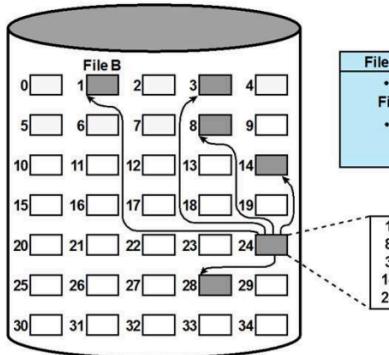


Figure 12.13 Indexed Allocation with Block Portions

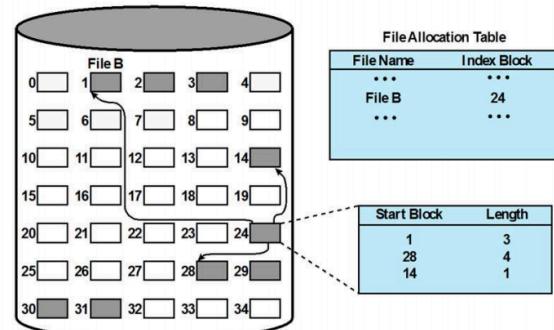


Figure 12.14 Indexed Allocation with Variable-Length Portions

Aspekt	Sammenhengende	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

MN

Tid for allokering:
Fast: Lange lenker
Variabel: Korte lenker

Plass for allokering:
Fast: Små indeksinnslag
Variabel: Store indeksinnslag

53

Håndtering av ledig plass

For å kunne allokkere plass til filer, må vi vite hvilken plass som er ledig. For dette trenger vi en diskallokeringstabell, i tillegg til filallokeringstabellen.

Bit-tabeller bruker en vektor med en bit for hver blokk på disken. Der 0 representerer ledig og 1 representerer tatt. Dette tabellen tar relativt liten plass og gjør det lett å søke etter kontinuerlig eller diskrete plasser. Det kan uansett ta en del plass, og sekvensielle søk er tidskrevende.

Kjedede ledige plasser: Ledige plasser kan være kjedet sammen ved en peker og en lengdeverdi for hver ledig plass. Dette krever neglisjerbart med overhead fordi det ikke er behov for en diskallokeringsstabell, men vil kunne gi fragmentering i tillegg til at søker er tidskrevende fordi man må lese hver blokk for å finne pekeren til neste ledige blokk.

Indeksering Indekseringsmetoden behandler ledige plasser som indekser i en fil og bruker en indekstabell lik den i filallokering. For effektivitet burde denne brukes på plasser med variabel lengde. Det vil gi en plass i tabellen for hver ledig del på disken. Indeksering er effektivt.

Free block list Hver blokk får et tall og listen av ledige blokker opprettholdes i en reservert del av disken.

Mangler notater fra sikkerhetsdelen av pensum.