# Bufferhåndtering i multimediasystemer

Pål Halvorsen
Universitetsstudiene på Kjeller (UNIK)
Universitetet i Oslo

14. mai 1997

# Forord

Dette er en hovedfagsoppgave i databehandling ved Institutt for Informatikk, Universitetet i Oslo. Oppgaven er tatt ved Universitetsstudiene på Kjeller (UNIK), og førsteamanuensis II Vera Goebel har vært min veileder.

Å være hovedfagsstudent ved UNIK har vært en veldig fin erfaring. Her er det et fint miljø, meget gode arbeidsforhold og mange faglige utfordringer som jeg har lært mye av.

Jeg vil først rette en stor takk til min veileder, Vera Goebel, for hennes gode veiledning, oppmuntring og støtte gjennom dette hovedfagsstudiet. Videre vil jeg takke medstudenter for hjelp med oppgaven og drift for hjelp med mange lange simuleringer. Tilslutt og ikke minst vil jeg takke og tilegne denne oppgaven til mine foreldre og min forlovede, Ann Kristin, for konstant støtte og oppmuntring i denne tiden.

Universitetsstudiene på Kjeller, mai 1997

Pål Halvorsen

# Sammendrag

Multimediaapplikasjoner med for eksempel video og audio har store, komplekse, kontinuerlige og tidsavhengige dataelementer som skiller seg en del fra dataelementene i for eksempel tradisjonelle databasesystemer. Håndteringen av disse dataene stiller derfor spesielle krav til systemkomponentene, og for at brukeren skal kunne tilbys en bestemt tjenestekvalitet, må alle hardware- og software-komponenter støtte de samme kravene. Bufferhåndteringen som er en delkomponent av et datahåndteringssystem, det vil si en software-komponent som er ansvarlig for lagring, henting, sending og endring av data, er et viktig punkt i lenken av funksjonelle enheter i et multimediasystem. De tradisjonelle bufferhåndteringsmekanismene kan ofte ikke støtte multimedia-applikasjoners krav tilfredsstillende, og det oppstår et behov for nye mekanismer for håndtering av bufferet.

I denne oppgaven ser vi på bufferhåndteringen i en valgt eksempelapplikasjon, det vil si et datahåndteringssystem for det elektroniske klasserommet. For å finne en bufferhåndteringsmekanisme som er egnet for denne applikasjonen, har vi sett på de krav en slik applikasjon stiller til en bufferhåndteringsmekanisme. I multimediaapplikasjoner med tidsavhengige, kontinuerlige dataelementer, trenger vi stor båndbredde og god gjennomstrømming av data mellom disken og bufferet, samt en kontinuerlig, jevn responstid under en viss øvre terskel.

Resultatene vi har kommet frem til i denne oppgaven, viser at en bufferhåndteringsmekanisme for multimediaapplikasjoner med kontinuerlige dataelementer, kan utnytte forhåndskunnskap om dataenes aksessmønster ved å forhåndshente data fra disken til bufferet. Videre kan antall diskaksesser reduseres ved at vi benytter en presentasjonsrelevant sideutbytting. Vi har funnet en bufferhåndteringsmekanisme, "least/most relevant for presentation", som tilfredsstiller de kravene vår valgte eksempelapplikasjon stiller. Denne bufferhåndteringsmekanismen har vi implementert og testet i et simuleringsprogram. Simuleringsresultatene viser at vår valgte bufferhåndteringsmekanisme kan støtte presentasjon av kontinuerlige datastrømmer. Videre får vi færre sideutbyttinger sammenlignet med tradisjonelle sideutbyttingsalgoritmer som "least recently used" og "random". Den valgte bufferhåndteringsmekanisme kan derfor gi en stor ytelsesforbedring sammenlignet med tradisjonelle bufferhåndteringsmekanismer for multimediaapplikasjoner à la vårt datahåndteringssystem for det elektroniske klasserommet.

# Innhold

# Figurer

# Tabeller

# Kapittel 1

# Innledning

## 1.1 Motivasjon og bakgrunn

Multimediaapplikasjoner består av store, komplekse, kontinuerlige og tidsavhengige dataelementer som for eksempel video, audio og animasjoner kombinert sammen med bilder, grafikk og tradisjonelle dataelementer som for eksempel vanlig tekst [Steinmetz et al. 95], [Rakow et al. 95], [Christodoulakis et al. 95]. Håndtering av slike data i et multimediasystem (MMS) stiller en del nye krav til systemkomponentene. For det første kan data i multimediaapplikasjoner variere veldig i størrelse og kompleksitet avhengig av datatype. Store mengder data må derfor kunne hånteres på en effektiv måte [Rakow et al. 95]. Videre består multimediaapplikasjonenes data ofte av flere datatyper som må presenteres sammen, og alle disse datastrømmene må få sin del av ressursene. Samtidig må vi ofte ta hensyn til relasjoner mellom datastrømmene, det vil for eksempel si at bilde og lyd må synkroniseres [Steinmetz et al. 95]. Et videre krav fra multimediaapplikasjoner er at presentasjon og eksekvering av kontinuerlige, tidsavhengige medietyper som video og audio ofte må gjøres under sanntidsbetingelser [Tokuda 94], [Steinmetz 95], [Özsoyoğlu et al. 95]. Et bestemt videobilde må komme til en bestemt tid for at en kontinuerlig presentasjon av dataene kan støttes.

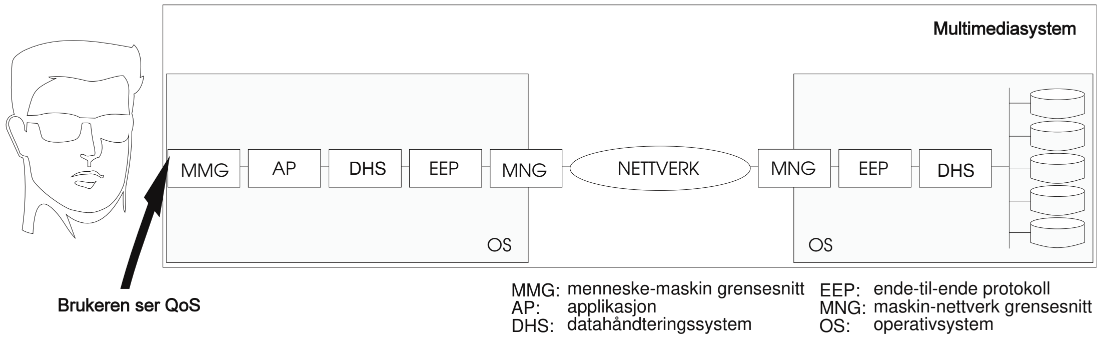
Som vi ser har nye applikasjonstyper og innføringen av nye datatyper gitt mange nye krav til både hardware- og software-komponenter. Tradisjonelle hardware- og software-komponenter kan ikke støtte disse kravene tilfredsstillende. I software-komponenter hvor dataelementene tradisjonelt har vært små med enkle strukturer, oppstår det nå behov for nye mekanismer for håndtering av multimediadata.

For å beskrive kravene til multimediaapplikasjoner på en side og ytelsen til multimediakomponentene på den andre, brukes begrepet *tjenestekvalitet* (quality-of-service, QoS) [OSI 88]. Opprinnelig kommer begrepet fra datakommunikasjon, men etterhvert som tidsavhengige datatyper har kommet innen MMSer, har begrepet blitt utvidet til også å dekke andre komponenter i MMSene [Vogel et al. 95].

MMSer består ofte av mange funksjonelle enheter som nettverk, ende-til-ende protokoller, datahåndteringssystemer (DHSer), applikasjoner, brukergrensesnitt og operativsystemer (OSer). For at et system kan støtte en viss QoS, må alle de funksjonelle kompnentene i dette systemet støtte samme QoS. Et eksempel på dette er vist i figur 1.1. Denne figuren viser et eksempel på en lagdelt arkitektur av et MMS for en "Video-on-Demand" (VoD, video på forespørsel) applikasjon. For at brukeren skal få se videoen han[1] har bedt om, må videodata sendes fra DHSet via ende-til-ende protokoller og et nettverk til applikasjonen og menneske-maskin grensesnittet. Som man ser av figuren kan hele systemet sees på som en lang lenke av komponenter som starter ved DHSet som lagrer videodataene på sekundære lagringsenheter som disker og slutter ved

[1] "Han" må leses som "han eller henne" i resten av denne oppgaven.

menneske-maskin grensesnittet hvor videoen blir presentert til brukeren. Gjennom sammarbeidet mellom alle disse komponentene får brukeren en samlet QoS fra systemet. QoS er derfor blitt et stadig viktigere punkt innen distribuerte MMSer siden alle komponentene i systemet må støtte samme QoS [Plagemann et al. 95], [Goebel et al. 96a].




Figur 1.1: Et eksempel på en lagdelt arkitektur av et multimediasystem [Plagemann et al. 95].

En viktig komponent i denne lenken av komponenter er DHSet, det vil si alle typer software-komponenter som håndterer lagring, henting, sending og endring av data. Eksempler på DHSer er databasesystemer (DBSer), filservere og filsystemet i et OS [Goebel et al. 96a]. For DHSer er det særlig viktig at data kan leveres raskt og i store mengder fra sekundære- til primære lagringsenheter, slik at multimediaapplikasjoners krav til håndtering av store mengder tidsavhengige data kan støttes.

En viktig del av DHSene er bufferhåndteringen. Bufferhåndteringsmekanismen styrer hvordan vi bruker minne til å oppbevare data som applikasjonene bruker, det vil si at denne komponenten styrer hvordan vi henter inn og bytter ut data i primærminnet ved hjelp av bestemte mekanismer. Sideutbytting (paging) [Patterson et al. 94] er en måte å håndtere minnet hvor dataene byttes ut i datablokker av en bestemt størrelse kalt sider (pages). Tradisjonelle sideutbyttingsalgoritmer er ikke designet med tanke på multimediaapplikasjoners krav til behandling av store datamengder, slik at nye mekanismer for henting og utbytting av sider er nødvendige.

## 1.2 Problemstilling

Multimediaapplikasjoner som inneholder kontinuerlige medietyper som video, audio, animasjoner og så videre, stiller store krav til blant annet håndteringen av data. Det er enorme mengder med data som skal til for å kunne gi en presentasjon av for eksempel en video med en akseptabel kvalitet. Videre holder det ikke å bare kunne lagre dataene. De må også kunne leveres videre til prosessoren for eksekvering, og siden data ikke kan eksekveres direkte fra disken, er bufferhåndtering en viktig komponent i systemet.

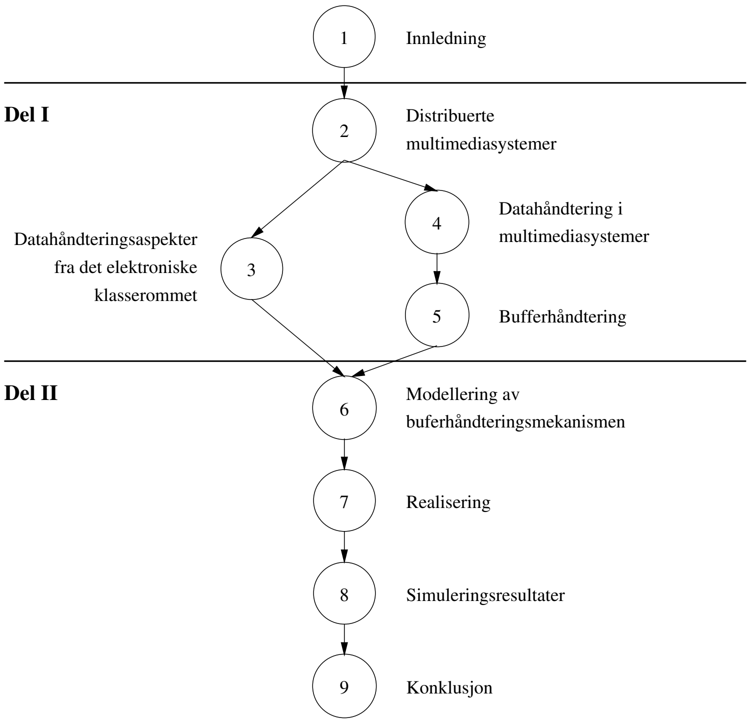
I dag brukes “langsome” disker som sekundære lagringsenheter. Dette er fordi en disk har mye større lagringskapasitet enn hovedminnet, har en lavere kostnad per MByte lagringsenhet og er et varig lagringsmedium. For så å imøtekomme multimediaapplikasjoners krav til stor båndbredde og raske, presise dataleveringer, må bufferhåndteringen gjøres så effektiv som mulig. Diskaksesser er i dag mye langsommere enn aksesser til hovedminnet, og for så å kunne få så stor gjennomstrømming av data som mulig, må antall diskaksesser reduseres mest mulig, det vil si at vi må finne en fornuftig måte å hente inn og bytte ut data i primærminnet. Tradisjonelt har primærminnet vært håndtert ved å bytte ut data på grunnlag av kriterier som for eksempel hvor hyppig data har blitt aksessert og hvor lang tid det har gått siden forrige referanse. I multimediaapplikasjoner er det ofte slik at vi i motsettning til i tradisjonelle applikasjoner, ofte vet i hvilken rekkefølge data aksesseres, og dette gir mange muligheter for å optimere bufferhåndteringen.

Ved håndtering av så store datamengder som i MMSer, hvor sanntidsaspekter og synkronisering er viktige punkter, er det store utfordringer i å optimalisere bufferhåndteringen. I denne oppgaven ser vi på bufferhåndteringen for en bestemt multimediaapplikasjon, og som en eksempelapplikasjon tar vi for oss et DHS for det elektroniske klasserommet vi har her på UNIK (Universitetsstudiene på Kjeller) i dag. På bakgrunn av vår valgte eksempelapplikasjons krav til systemet, vurderer vi forskjellige bufferhåndteringsmekanismer og forsøker å finne en bufferhåndteringsmekanisme som støtter eksempelapplikasjonens krav. Denne bufferhåndteringsmekanismen må støtte presentasjon av kontinuerlige dataelementer ved at gjennomstrømmingen økes ved hjelp av forhåndshenting av data fra sekundær- til primærminnet samt at antall diskaksesser minimeres ved en tilpasset sideutbytting.

## 1.3 Oppgavens struktur

For å komme frem til en bufferhåndteringsmekanisme som er godt egnet for en multimediaapplikasjon, holder det ikke bare å ha kunnskaper om forskjellige bufferhåndteringsmekanismer med forskjellige sideutbyttingsalgoritmer. For det første må bufferhåndteringen ta hensyn til hva slags applikasjon den skal brukes i. Det er derfor viktig å ha kjennskap til hvordan multimediaapplikasjonene refererer til data, hvor mye data som overføres og hvor fort dataene må leveres. Videre er ikke bufferhåndteringen en isolert komponent, men den påvirkes direkte av andre komponenter. Det er derfor viktig å ta hensyn til hvordan komponentene er avhengige av hverandre.

Denne oppgaven er grovt delt inn i to deler som vist i figur 1.2 hvor vi først bygger opp en teoretisk bakgrunn og ser litt på lignende arbeider. Deretter modellerer, realiserer og tester vi en bufferhåndteringsmekanisme for vår eksempelapplikasjon.




Figur 1.2: Flytskjema for oppgaven.

I *del I - bakgrunn* beskriver vi i kapittel 2 til kapittel 5 den teoretiske basisen vi trenger for vårt arbeid med multimediaapplikasjoner. I kapittel 2 tar vi for oss distribuerte MMSer hvor vi kommer inn på generelle aspekter ved distribusjon og multimediaapplikasjoner og deres krav til systemet. Vår eksempelapplikasjon, et planlagt DHS for det elektroniske klasserommet, beskrives i kapittel 3. Her ser vi på dagens elektroniske klasserom og presenterer vår fremtidsvisjon av dette systemet. Videre viser vi i kapittel 4 eksempler på datahåndteringen i forskjellige DHSer, samt at vi kommer inn på en del andre komponenter i OSet som påvirker bufferhåndteringen og som derfor må støtte multimedias krav. I kapittel 5 beskrives minnehåndteringen med utgangspunkt i et OS. Her tar vi for oss spesielt håndtering av hovedminnet og presenterer både tradisjonelle - og mer moderne sideutbyttingsmekanismer designet for MMSer.

I *del II - modellering, realisering og evaluering* viser vi i kapittel 6 til kapittel 9 hvordan vi har modellert, realisert og testet bufferhåndteringmekanismen vi har valgt for det elektroniske klasserommet. I kapittel 6 drøftes forskjellige konsepter i designet av en bufferhåndteringsmekanisme, samt at vi modellerer en algoritme for sideutbytting i DHSet for det elektroniske klasserommet. I kapittel 7 beskriver vi hvordan vi realiserer den bufferhåndteringsmekanismen vi har modellert. Vi viser hvordan vi har testet algoritmen i et simuleringsprogram, og hvordan vi kan realisere systemet ved integrering i et eksisterende OS. Resultatene fra simuleringen presenteres i kapittel 8, hvor vi også diskuterer hva disse resultatene betyr for et multimedia-DHS benyttet i det elektroniske klasserommet. I kapittel 9 oppsummerer og konkluderer vi hele oppgaven, samt at vi beskriver en del ubesvarte spørsmål og fremtidig arbeid.

# Del I

# Bakgrunn

# Kapittel 2

# Distribuerte multimediasystemer

For å få en generell forståelse av hvordan distribuerte multimediaapplikasjoner er bygd opp, hvilke krav de setter til et MMS, hvordan distribusjon av ressurser håndteres og så videre, vil vi i dette kapittelet beskrive en del generelle aspekter ved et distribuert MMS.

Multimedia er et begrep som per i dag ikke har noen generell, akseptert definisjon. Noen ser på multimedia som et forsøk på å bruke datamaskiner til å kombinere tekst, grafikk, bilder, video og audio for å presentere informasjon til brukere. For andre er multimedia bare en kobling mellom televisjon (TV) og datamaskiner, mens en tredje gruppe ser på det som en ny generasjon applikasjoner som trenger ny software- og hardware-arkitektur [Rodriguez et al. 95]. Christodoulakis og Koveos [Christodoulakis et al. 95] definerer multimedia løst som et system som kan brukes til å presentere informasjon i *mer enn en* form, men i følge Steinmetz og Nahrstedt [Steinmetz et al. 95] er denne definisjonen for løs siden den bare tar kvantitative egenskaper i betraktning. Man må også se på de kvalitative egenskapene, og selv da kvalifiserer ikke alle kombinasjoner av media til å kalles multimedia. Ved bruk av uttrykket multimedia bør både kontinuerlige og diskrete datatyper brukes i applikasjonen. I denne oppgaven defineres derfor et MMS etter Steinmetz og Nahrstedt som følger:

> *Et MMS er karakterisert ved datamaskinkontrollert, integrert produksjon, manipulasjon, presentasjon, lagring og kommunikasjon av uavhengig informasjon som er omkodet gjennom kontinuerlige (tidsavhengig) og diskrete (tidsuavhengig) medier.*

Et MMS må altså kunne lagre, håndtere og hente frem informasjon i forskjellige media, kunne håndtere relasjoner mellom forskjellige datatyper og kunne presentere disse dataene. Konsepter for tidsavhengigheter og synkronisert presentasjon må legges til og integreres i spørrespråket. Videre krever presentasjoner og kontroll av disse i brukerens arbeidsstasjon en klient-tjener arkitektur, bufferkonsepter og at nettverket tilbyr kontinuerlige eller isokrone transport protokoller[1], [Rakow et al. 95].

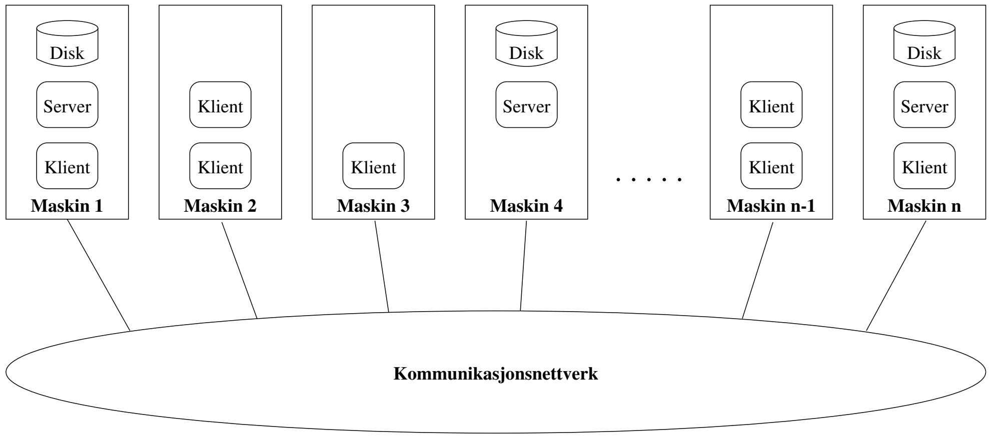
Et MMS som beskrevet over krever blant annet store lagringsmuligheter og stor prosessorkapasitet. Data og software i MMSer spres derfor ofte over mange forskjellige maskiner i et distribuert system hvor

> *Et distribuert system er en samling av uavhengige datamaskiner sammenkoblet over et nettverk som brukeren opplever som en maskin* [Tanenbaum 95].

Ved en slik distribusjon trengs det spesiell støtte fra både hardware og software. Fra hardware'ens side vil dette si at maskinene ikke er avhengige av hverandre. Maskinene kan brukes hver for seg.

[1] I en isokrone transport modus defineres ikke bare en øvre grense for ende-til-ende forsinkelse til hver datapakke, men også en nedre grense. Dette gir at jitter'et for hver pakke er bundet mellom disse grensene [Steinmetz et al. 95].

Software'en (det vil ofte si OSet) gjør at brukeren kan se på og bruke systemet som en enkelt maskin. Figur 2.1 viser et slikt system. Vi har data og applikasjoner spredd på servere på forskjellige maskiner. Disse maskinene snakker sammen over et kommunikasjonsnettverk.




Figur 2.1: En enkel modell for et distribuert system.

I dette kapittelet beskriver vi (avsnitt 2.1) først enkelte generelle aspekter ved distribuerte systemer. Videre ser vi nærmere på en del multimediaapplikasjoner (avsnitt 2.2) og QoS (avsnitt 2.3). Disse multimediaapplikasjoner gir store, nye krav til systemet, og multimediaapplikasjonenes systemkrav legges frem i det siste avsnitt i dette kapittelet (avsnitt 2.4).

## 2.1 Design av distribuerte systemer

Når vi skal designe et distribuert system [Tanenbaum 95] er det mange viktige punkter vi må passe på som hvordan ressurser[2] skal distribueres, hvordan organisere systemet og så videre. Mange av disse taler hverandre litt imot, og det er derfor viktig at vi finner en slags "gyllen middelvei" slik at systemet sett under et, best mulig støtter kravene til applikasjonen som skal brukes.

### 2.1.1 Transparens

I den løse "definisjonen" av et distribuert system i starten av dette kapittelet, ble det sagt at systemet skulle opptre som en enkelt virtuell prosessor, og et nøkkelpunkt i designet er transparens. Dette vil si at systemet skal skjule all distribusjon fra brukerene og også programmene. Brukeren skal ikke behøve å vite hvor de forskjellige ressursene ligger, men bruke systemet som om det bare bestod av en enkelt maskin.

Vi har flere former for transparens. Først har vi lokasjonstransparens. Brukeren skal for eksempel ikke behøve å vite hvor prosessorer, skrivere og filer er plassert. Videre er det ønskelig at en ressures kan flyttes uten at navnet til ressursen må forandres, og brukeren skal ikke måtte vite om det er flere kopier av en ressurs og eventuelt hvilken kopi han bruker. Systemet har da også ansvaret for å holde alle kopiene konsistente, slik at vi ikke har kopier av samme fil med forskjellig innhold. Systemet skal kunne ha flere brukere samtidig som bruker samme ressurser, og i tillegg bør flere aktiviteter kunne kjøres i parallell uten at brukeren merker det.

[2] Med ressurser menes her alle mulige systemkomponenter: prosessorer, minne, skrivere, filer etc.

### 2.1.2 Fleksibilitet

Distribuerte systemer er et relativt nytt fagfelt, og det vil i fremtiden kunne bli mange forbedringer både på hardware- og software-komponenter. Det er da viktig at systemene vi konstruerer nå er fleksible. Ved innføring av nye komponenter må disse kunne integreres i det eksisterende systemet slik at ikke helt nye løsninger må lages hver gang. I denne forbindelse er mikrokjerne-OSer et viktig konsept. Disse inneholder minimal funksjonalitet med bare det helt nødvendigste. Resten av OSet legges i egne subsystemer, som kan være forskjellige fra maskin til maskin. Mikrokjernen gir best fleksibilitet og er enklere å bygge ut enn en stor monolittisk kjerne som inneholder hele OSets funksjoner.

### 2.1.3 Pålitelighet

Et distribuert system må være pålitelig. En av fordelene med å ha flere maskiner sammenkoblet er at systemet som enhet skal kunne fortsette å kjøre selv om en maskin går ned. Hvis en maskin går ned, må en annen maskin kunne ta over jobben. Systemet skal hele tiden være tilgjengelig for brukeren, og det er derfor viktig å prøve å unngå å få kritiske komponenter, som ved en feil vil føre til krasj, i systemet. En slik kritisk komponent kan for eksempel være en fil en bruker jobber med. Hvis det bare eksisterer en kopi av denne filen og maskinen den ligger på går ned, kan ikke brukeren jobbe videre før maskinen kommer opp igjen og da vil kanskje arbeidet han gjorde før sammenbruddet være tapt. En løsning er å distribuere data over flere maskiner (se [Elmasri et al. 94], [Korth et al. 91], [Tanenbaum 87] eller [Tanenbaum 95]).

Et annet aspekt av pålitelighet er sikkerhet. Filer og andre ressurser må beskyttes mot uautorisert bruk, og vi må derfor ha en måte å identifisere brukere på.

### 2.1.4 Ytelse

Ytelse er et viktig aspekt i designet av et distribuert system. Systemet vil aldri bli godtatt selv om det er transparent, fleksibelt og pålitelig hvis det har en dårlig ytelse. Det som ofte senker ytelsen er all kommunikasjonen over nettverket, og det er da et viktig punkt å begrense dette, samtidig som den nødvendige kommunikasjonen gjøres så raskt (men samtidig så pålitelig) som mulig.

En måte å bedre ytelsen er å ha flere kopier av ressurser slik at strekningen for kommunikasjonsmeldinger ikke blir større enn nødvendig. En annen forbedring er å være nøye med hva som parallelliseres. Små enkle programmer kjøres ofte raskere på en lokal maskin på grunn av stor overhead med meldingsutveksling, mens store tunge utregninger gjøres raskere i parallell på flere maskiner.

Et siste sentralt punkt når vi snakker om ytelse er å unngå å bruke sentraliserte komponenter når systemene blir store. Når vi har mange brukere, vil slike komponenter ofte bli store flaskehalser i systemet, og vi vil få sterkt forlengede responstider på våre forespørsler til systemet.

### 2.1.5 Kommunikasjon

I et distribuert system er et av nøkkelpunktene kommunikasjonen mellom maskinene, og kommunikasjonen er som regel basert på meldinger. Disse meldingene kan bygges opp på forskjellige måter, og det er mange bestemmelser som må gjøres om hvordan bit'ene skal representeres, hva er starten og slutten av meldingen og hvordan oppdage og rette feil.

For å gjøre dette lettere lages det egne regler for hvordan prosesser kan kommunisere med hverandre. Disse reglene kalles protokoller, og er en "regelbok" for kommuniserende parter om hvordan kommunikasjonen skal foregå. En slik standard, for eksempel "Open Systems Interconnection" (OSI) eller "Asynchronous Transfer Mode" (ATM), lar et åpent system kommunisere med hverandre. Et åpent system er et system som er forberedt på å kommunisere med et annet ved å bruke standard regler for formatet, innholdet og semantikken til meldingene.

I disse modellene deles kommunikasjonen inn i flere lag, og hvert av disse lagene har ansvaret for et spesifikt område av kommunikasjonen. Vi får dermed en modulasjon av kommunikasjonsproblemet i mindre deler som hver kan løses for seg. Hvert lag danner et grensesnitt til laget over og under seg hvor hvert lag inneholder et sett med operasjoner som sammen gir de tjenestene laget kan tilby brukeren.

I meldingsutvekslingen beskrevet over må en melding gjennom en mengde lag som hver enten genererer og legger til et hode til meldingen på vei ut fra en maskin eller som fjerner og undersøker hodet til meldingen som er på vei inn til en maskin. I et lokalt nettverk meldinger overføres i mye større hastigheter enn i for eksempel et nettverk over flere land, vil for mye tid gå med til å la meldingene gå opp og ned protokollagene, så her benyttes ofte andre løsninger som *klient-tjener modellen* eller *fjernprosedyrekall* (remote procedure call). Se for eksempel [Tanenbaum 95] for mer informasjon om kommunikasjon i distribuerte systemer.

### 2.1.6 Fordeler og ulemper med distribusjon

Et distribuert system med mange små sammenkoblede maskiner har mange fordeler over sentraliserte systemer med en stor, kraftig maskin. Det er billigere å sammenkoble flere mikroprosessorer enn å kjøpe en ny maskin med en større og kraftigere prosessorenhet hver gang vi trenger forsterket eksekveringskraft, det vil si at vi har et bedre forhold mellom pris og ytelse. Videre kan vi få større hastighet ved å distribuere instruksjonene på forskjellige maskiner og kjøre i parallell, og vi får økt pålitelighet for at systemet er oppe. Hvis en maskin går ned, kan systemet som enhet fortsatt være oppe.

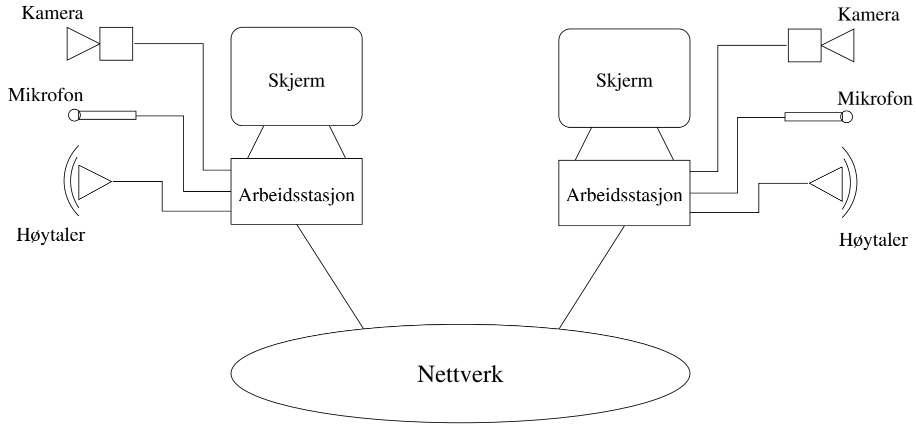
Videre har et distribuert system med sammenkoblede maskiner en fordel av deling av ressurser ovenfor en isolert maskin. I et distribuert system kan vi dele data mellom flere maskiner. I stedet for å måtte ha kopier av en fil på alle maskiner (med alle de problemene dette fører med seg) som vi må i sentraliserte systemer, kan vi i et distribuert system tillate at flere bruker samme filen. Vi kan også dele hardware-enheter som skrivere og tape-stasjoner, og vi kan lettere få en kommunikasjon mellom brukerene som for eksempel ved elektronisk post.

Et distribuert system har også sine ulemper. Systemet blir mer komplekst og trenger dermed mer avansert software. Vi må ha et nettverk for å kommunisere mellom maskinene hvor vi kan få brudd på forbindelsene og meldinger kan forsvinne. Ved å la flere brukere se på samme data, kan også sikkerhet bli et problem. Upriviligerte brukere kan få tilgang til informasjon de ikke skal ha.

## 2.2 Multimediaapplikasjoner

Utstyrt med spesiell hardware, kan arbeidsstasjoner produsere digital audio og vise bilder av høy kvalitet. Trenden innen MMSer er å bruke dette innen flere typer applikasjoner. Dette kan være applikasjoner som multimedia post, konferansesystemer og "virtual reality". Slike applikasjoner kalles *kontinuerlige multimediaapplikasjoner* [Tokuda 94]. Det eksisterer en mengde med forskjellige multimediaapplikasjonstyper [Steinmetz et al. 95], og her vil vi komme inn på noen av dem.

En type applikasjoner er *editeringsapplikasjoner* hvor man kan editere multimediaobjektene. I slike applikasjoner kan man slette, sette inn og erstatte data i multimediaobjektene. Her kan vi ha operasjoner som å hoppe fremover i objektet, spole frem og tilbake, hurtig avspilling (i for eksempel dobbel hastighet) og "klippe og lime" i tillegg til "vanlige" operasjoner som avspilling og stopping av presentasjonen.




Figur 2.2: Et eksempel på et video-konferansesystem.

Det er flere typer underholdningsapplikasjoner. Vi har "*virtual reality*" som skal gi brukeren en tredimensjonal, interaktiv virkelighetsfølelse, enten den er synlig, hørbar og/eller følelig. Andre typer multimediaapplikasjoner for underholdning er av gruppen "*Media-on-Demand*" (MoD, media på forespørsel). Her ligger for eksempel nyheter, musikk-CDer eller videofilmer lagret i et multimedia-DBS. Brukere kan så be om å få se/høre på den informasjonen som er lagret i DBS-et. Til slutt i denne gruppen har vi *spill*. Spillene og "virtual reality" er baserte på interaktivitet mellom bruker og maskin, hvor det kan være en eller flere, lokale eller ikke-lokale, samtidige brukere.

Videre kan vi ha applikasjoner for å holde *videokonferanser*. Vi bruker her video, audio og tekstinformasjon for å kunne holde konferanser mellom folk som jobber sammen på forskjellige geografiske steder uten at de må reise til et felles sted. Hver enkelt kan delta i konferansen lokalt på sitt arbeidssted. En slik videokonferanse kan også lagres på en sekundær lagringsenhet. På denne måten kan personer som ikke fikk deltatt, få konferansen på forespørsel og få den avspilt slik at de får med seg innholdet. Applikasjoner av samme type er *elektronisk klasserom* (se kapittel 3), *bildetelefoner* og *multimedia post*. Figur 2.2 viser et mulig oppsett for en videokonferanse hvor vi har skjerm, høytaler, kamera og mikrofon hos alle deltagere på konferansen.

## 2.3 Tjenestekvalitet

For å beskrive kravene til multimediaapplikasjoner på en side og ytelsen til multimediakomponentene på den andre, brukes begrepet tjenestekvalitet (quality-of-service, QoS) [OSI 88]. Begrepet oppstod opprinnelig innen datakommunikasjon for å beskrive visse tekniske parametere ved dataoverføring. For eksempel OSI's Reference Model har et antall QoS parametere som beskriver hastighet og pålitelighet ved dataoverføringen. Disse parameterene berører hovedsaklig de lavere protokollag i kommunikasjonssystemet og er ikke ment å være direkte observerbare for applikasjonene. Etterhvert som tidsavhengige datatyper har kommet innen multimediasystemer, holder derfor ikke denne QoS-beskrivelsen lenger [Vogel et al. 95].

MMSer inneholder som oftest de følgende funksjonelle enheter: nettverk, ende-til-ende protokoller, datahåndteringssystemer, applikasjoner, brukergrensesnitt og OSer. Med tanke på QoS i MMSer holder det derfor ikke å se bare på kommunikasjon og OSer, for i tillegg til å manipulere og overføre data, lagrer, henter og presenterer MMSer data til brukeren på en passende måte [Goebel et al. 96a]. Alle funksjonelle enheter i det distribuerte systemet må derfor kunne støtte QoS-kravene til en applikasjon.

En eksakt definisjon av begrepet QoS kan være vanskelig å finne. I [ISO/IEC 95] finner vi denne definisjon:

*Et sett av kvaliteter relatert til den kollektive oppførselen til et eller flere objekter.*

Vogel, Kerhervé, Gecsei og Bochmann [Vogel et al. 95] bruker følgende definisjon:

*QoS representerer mengden av de kvantitative og kvalitative egenskaper ved et distribuert system som er nødvendige for å oppnå en påkrevd funksjonalitet i en applikasjon,*

hvor funksjonalitet inkluderer både presentasjon av multimedia data til brukeren og brukerens tilfredsstillelse generelt.

### 2.3.1 Spesifikasjon

QoS i et system er uttrykt ved et sett av (parameter,verdi)-par hvor hver parameter er en typet variabel med en bestemt verdimengde. I motsettning til andre systemparametere i distribuerte systemer, kan systemets komponenter forhandle seg frem til en verdi på QoS-parameterene avhengig av maskinressursene [Vogel et al. 95]. Tabell 2.1 viser fem forskjellige kategorier med parametertyper.

| **Kategori** | **Eksempel på parametere** |
|---|---|
| *Ytelsesorienterte* | Ende-til-ende forsinkelse og bitrate |
| *Formatorienterte* | Videooppløsning, rammerate, lagringsformat og kompresjonsskjema |
| *Synkroniseringsorienterte* | Tidsforskjell mellom begynnelsen av video- og audiosekvenser |
| *Kostorienterte* | Kostnader for opprettelse av forbindelse og dataoverføring, copyright kostnader |
| *Brukerorienterte* | Subjektiv bilde- og lydkvalitet |

Tabell 2.1: Fem kategorier QoS parametere [Vogel et al. 95].

Generelt er det to måter å kvantifisere QoS-parametere [Plagemann et al. 95]. Brukeren kan spesifisere enten en bestemt verdi til parameteren eller et bestemt intervall hvor verdien skal ligge i. I praksis er det ofte et intervall verdien må ligge i, og denne verdien varierer ofte underveis [Vogel et al. 95].

### 2.3.2 Semantikk

Semantikken ( [Danthine et al. 93], [Plagemann et al. 95] og [ISO/IEC 95]) til QoS-parameterene er gitt ved graden av enighet mellom bruker og tjenesteyter. Det er generelt to ekstreme ytterpunkter i QoS-semantikk:

***Best mulig (best-effort)* QoS** representerer den svakeste formen av QoS-enighet. Alle funksjonelle enheter gjør deres beste for å yte den forespurte kvaliteten av tjenesten. Tjenesteyteren har ingen forpliktelser til å overholde den forhandlede kvaliteten.

***Garantert (guaranteed)* QoS** tilbys gjenom ressurs allokering, og den ønskede QoS overholdes alltid. Kvaliteten på tjenesten garanteres fra tjenesteyteren.

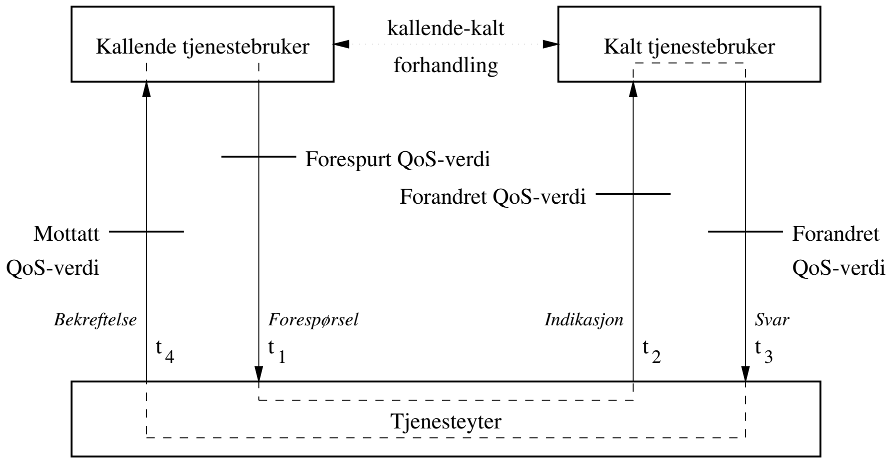
Videre har semantikker mellom disse ytterpunktene blitt introdusert:

***Tvungen (compulsory)* QoS** garanterer kvaliteten på tjenesten gjennom ressurs allokering og overvåkning av ressursene. Tjenesteyteren tilbyr tjenesten så lenge den kan mottas hos mottakeren, men bryter forbindelsen hvis den forespurte QoS ikke kan tilbys.

***Terskel (threshold)* QoS** er som tvungen QoS, men isteden for å bryte forbindelsen, sier den i fra til tjenestebrukerene når den finner ut at den forespurte QoS ikke kan tilbys.

I tillegg til semantikkene beskrevet over, har vi en annen type semantikk som angår kostnadene til en tjeneste:

***Maksimal (maximal)* QoS** gir en øvre grense for kvaliteten på tjenesten slik at ikke kostnadene blir for høye. En maksimal verdi for en QoS-parameter velges, og kvaliteten på tjenesten skal ikke overskride denne verdien.




Figur 2.3: Treangulær forhandling om verdien på QoS-parameterene.

### 2.3.3 Forhandling

Brukere av MMSer vil ha så god kvalitet på tjenesten som mulig. Det vil for eksempel si så høy rammerate som mulig for video og liten feilrate. For å kunne kjøre andre prosesser parallelt med multimediaapplikasjonene, må multimediaapplikasjonene koste så lite som mulig. Et MMS har begrensede ressurser, og det er ikke alltid at den forespurte QoS kan overholdes. Alle komponentene har egne QoS-parametere. For å finne frem til en QoS som kan tilbys, må vi ha en QoS-forhandling hvor alle involverte komponenters begrensninger taes i betraktning [Vogel et al. 95]. Målet med en QoS-forhandling er å finne verdier til parameterene som både brukerene og tjenesteyterene kan godta. Avhengig av skjema involverer forhandlingen to eller tre entiteter. Dette er som vist i figur 2.3 den *kallende tjenestebrukeren*, *tjenesteyteren* og den *kalte tjenestebrukeren* [Plagemann et al. 95]. Det skilles mellom tre forskjellige typer forhandlingsskjemaer:

**Treangulær forhandling** involverer som vi ser i figur 2.3 alle tre entitetene. Den kallende tjenestebrukeren sender en ønsket QoS-spesifikasjon til tjenesteyteren. Tjenesteyteren kan svekke verdien på parameterene og sender QoS-spesifikasjonen videre til den kalte tjenestebrukeren som igjen kan svekke verdiene. Den endelige QoS-spesifikasjonen sendes så tilbake til den kallende tjenestebrukeren. Både tjenesteyteren og den kalte tjenestebrukeren kan nekte å overholde tjenesten.

**Tosidig forhandling** foregår mellom tjenestebrukerene. Tjenesteyteren har ikke lov til å forandre verdiene i spesifikasjonen. Både tjenesteyteren og den kalte tjenestebrukeren kan likevel nekte å overholde tjenesten.

**Ensidig forhandling** er en forhandlingsmetode hvor den forespurte QoS-spesifikasjonen blir akseptert eller forkastes. Her gir den kallende tjenestebrukeren en forespørsel til tjenesteyteren og den kalte tjenestebrukeren. Tjenesteyteren og den kalte tjenestebrukeren kan ikke forandre verdiene, men bare må godta eller nekte på forespørselen.

En typisk realisering av allokering og håndtering av ressurser er basert på interaksjon mellom klienter og ressurshåndterere. Gjennom en QoS-spesifikasjon gir klienten ressurshåndtereren en forespørsel om ressurser. Ressurshåndtereren sjekker ressursbruken sin (og om nødvendig andres) for å se om forespørselen kan tilbys.

I etableringsfasen forhandles det mellom klient-applikasjonen og den forespurte ressurshåndtereren. Ved forhandlinger om ende-til-ende parametere over nettverk trengs ressursreservasjonsprotokoller hvor lokale ressurshåndterere i det distribuerte systemet allokerer de nødvendige ressursene.

Allokering og håndtering av ressursene deles i fire faser [Steinmetz 95]:

- *Schedulability-test*. Ressurshåndtereren beregner på grunnlag av de gitte QoS-parameterene, for eksempel gjennomstrømming og pålitelighet, om det er kapasitet nok til å kunne håndtere resten av forespørselen.
- *QoS kalkulasjon*. Den best mulige ytelsen som ressursene kan garantere, beregnes.
- *Ressursreservasjon*. Ressurshåndtereren allokerer den nødvendige kapasiteten for å tilfredsstille de gitte QoS-parameterene.
- *Ressurs-scheduling*. Innkommende meldinger schedules i følge de gitte QoS-garantier.

Når en QoS-spesifikasjon er ferdig forhandlet, må den godkjennes av alle parter med i forhandlingen, og hvis det ikke er kommet til noen enighet blir forbindelsen brutt. Ved et brudd i forhandlingene bør det gis en forklaring for bruddet til den kallende tjenestebrukeren, slik at forespørselen kan modifiseres og reforhandles eller eventuelt kanseleres.

Underveis i eksekveringen av en tjeneste kan arbeidsmengden til et system forandres, og en forhandlet og godtatt QoS-spesifikasjon må kanskje forandres og reforhandles. Eksempler på dette er at brukeren kan forandre sine krav til tjenesten eller at nettverket ikke lenger kan tilby den samme kvaliteten på grunn av økt arbeidsmengde [Plagemann et al. 95]. Ved siden av den initielle forhandlingen bør derfor et distribuert system planlegge QoS-overvåkning og reforhandling [Vogel et al. 95].

Et eksempel på et QoS-forhandlingssystem er "QoS Broker" [Nahrstedt et al. 95] som forhandler mellom applikasjonen, operativsystemet og transportprotokollene. Dette systemet er designet slik at vi har en kjøper og en selger som forhandler om ressursene.

### 2.3.4 Allokeringsstrategi

Ressurser kan reserveres på to måter: *pessimistisk* eller *optimistisk*. Ved pessimistisk allokering reserveres ressursen etter den verst mulige arbeidsmengden. Dette betyr at vi alltid har nok ressurser og kan garantere QoS. Optimistisk allokering av ressursen allokerer ressursen etter gjennomsnittlig arbeidsmengde. Dette kan gi en overbelastningssituasjon og dermed en feil, men ressursene blir her utnyttet maksimalt. Vi kan ikke garantere QoS, så resultatet gjøres så bra som mulig.

### 2.3.5 Brukergrensesnittet

QoS er synlig for brukeren på brukergrensesnittet. Det vil si at når brukeren setter opp en QoS-spesifikasjon, bør den bare inneholde parametere som berører selve presentasjonskvaliteten som bildestørrelse, bildeoppløsning og lydkvalitet. Brukeren skal ikke behøve å sette systemparametere. Disse skal skjules og settes av systemet selv på bakgrunn av brukerens ønsker. Når brukeren setter sine krav, bør disse mappes inn i systemparameterene. Dette er brukt i en metode kalt "Quality Query by Example", og det viser seg at metoden er god for presenterbare parametere som video, bilder og lyd, men ikke så godt egnet for parametere som responstid og synkronisering [Vogel et al. 95].

I et system er ikke parameterne uavhengige av hverandre. Ved valg av verdier som setter høye krav til for eksempel oppløsning, vil også kostnader og responstid øke. Derfor bør en bruker få vite om konsekvenser av verdivalg for parameterene, slik at han ikke automatisk velger best mulig kvalitet uten å tenke på konsekvensene.

Alle komponentene i et system er med på å begrense en brukers muligheter for kvalitetsvalg [Vogel et al. 95]. Manglende muligheter for sanntidseksekvering av data kan begrense presisjonen til synkroniseringen, langsomme segmenter kan begrense gjennomtrømmingen av data i transportsystemet og et DBS som bare har lavkvalitetsbilder kan ikke levere høykvalitetsbilder. Videre kan ikke en svart/hvit skjerm levere fargebilder og en maskin uten høytalere kan ikke levere lyd.

## 2.4 Multimediaapplikasjoners krav til systemet

Multimedia har tilført nye datatyper som *audio, video, grafikk, bilder, tale, musikk, animasjoner* og ulike sammensetninger av disse typene i tillegg til de "tradisjonelle" datatypene for nummeriske data, tekst, eksekverbare programmer og så videre [Steinmetz et al. 95], [Christodoulakis et al. 95]. I motsetning til data i tradisjonelle DHSer, kan disse multimediadata være veldig store, komplekse og variere dynamisk. Figur 1 i [Rakow et al. 95] viser blant annet at et fem minutter langt musikkstykke på CD tar 52,8 MBytes og et videoklipp i høy kvalitet på samme lengde kan være på 33 GBytes.

Data av typer som audio og video kalles ofte *kontinuerlige data*. Et videoklipp består av en sekvens rammer som presenteres ved kontinuerlig visning av rammene. Slike data er også *tidsavhengige*, og informasjonen er selv uttrykt som en funksjon av tiden. For å få et tilfredsstillende resultat, må hastigheten i fremvisningen av bildesekvensen være konstant. Videre fortsetter utviklingen og trenden innen multimediaapplikasjoner er i følge Steinmetz og Nahrstedt [Steinmetz et al. 95] at:

- Man går fra å gjøre om eksisterende applikasjoner til å lage nye typer applikasjoner. Problemet her er at man ikke vet hvordan fremtidige applikasjoner vil se ut og at nye applikasjonstyper vil kreve forandringer i brukergrensesnittet, nye integrasjonsteknikker og så videre.
- Multimediaapplikasjoner flyttes fra et enbrukersystem på en PC til et multibrukersystem.
- Multimediaapplikasjoner designes mer og mer for distribuerte systemer.
- Applikasjonene som eksisterer i dag er ofte platformspesifikke og systemavhengige. Trenden går mot en åpen løsning slik at fremtidens applikasjoner skal være portable over mange forskjellige platformer.
- Mediakommunikasjon går fra å være enveis til toveis informasjonsflyt, for eksempel interaktivt TV.

Dagens og fremtidens multimediaapplikasjoner gir, som vi ser over, nye og større krav til maskinsystemene. Christodoulakis og Koveos [Christodoulakis et al. 95] gir to hovedtyper med krav som multimediaapplikasjoner setter til et DBS. For det første må datamodellen kunne gi en naturlig og fleksibel definisjon og utvikling av skjema som kan representere en sammensetning av komplekse relasjoner mellom deler av et multimediainformasjonsobjekt. Det andre kravet er at man må kunne dele og manipulere multimediadata.

Disse kravene gjelder både selve arkitekturen og håndteringen av hardware. Videre må vi kanskje omgjøre brukergrensesnittet og lage nye integreringsteknikker. Systemene må kunne håndtere flere samtidige brukere samt håndtere distribusjon. Det er mange slike systemkrav, og i fra filsystemenes side gir kontinuerlige multimediadata følgende ekstra krav til systemet i forhold til diskrete [Steinmetz 95]:

- *Datamengder*. Som vi ser over kan filstørrelsen variere veldig avhengig av datatype. Filsystemet må organisere data slik at begrensede mengder lagringsplass blir brukt mest mulig effektivt.
- *Multiple datastrømmer*. Et MMS må samtidig kunne behandle ulike typer media. Det må passe på at alle får sin del av ressursene, og ta hensyn til relasjoner mellom strømmene, som for eksempel synkronisering av bilde og lyd.
- *Sanntids krav*. Henting, eksekvering og presentasjon av kontinuerlige datatyper er tidsavhengig. Lagrings- og hentealgoritmer må ta hensyn til dette.

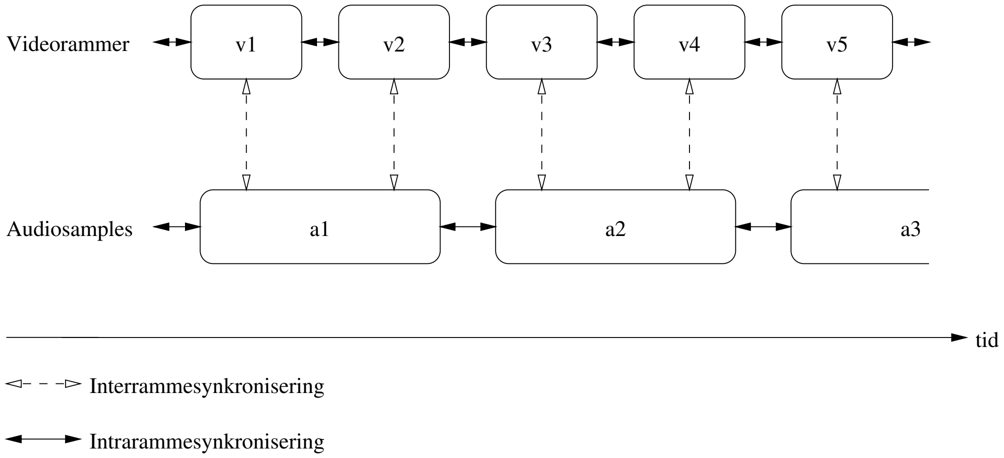
Disse kravene beskrives litt nærmere i avsnittene under.

### 2.4.1 Datamengder og datamodellering

Som nevnt over vil ukomprimert data fra grafikk, video og audio kreve veldig store mengder med lagringskapasitet. Videre vil et kommunikasjonsnettverk på grunn av store datamengder måtte ha veldig høy båndbreddekapasitet for å kunne tilby en enkelt ende-til-ende kommunikasjon. I tillegg er det ikke bare behov for stor lagringskapasitet, men også dataratene ved overføring av kontinuerlige datatyper over nettverk er store. De neste eksemplene vil vise mengden av data og båndbreddebehovet vi har for ukomprimerte, kontinuerllige datatyper [Steinmetz et al. 95]:

- Ukomprimerte audiosignaler med telefonkvalitet samplet med lydkodingen 8 bit/8 KHz gir et båndbreddebehov på 64 Kbits/s og et lagringsbehov på 3,84 Mbits for å lagre et minutt med tale.
- Et stereo audiosignal med CD kvalitet samples med lydkodingen 16 bit/44,1 KHz. Ukomprimert gir dette et båndbreddebehov på 705,6 Kbits/s og et lagringsbehov på 42,336 Mbits for å lagre et minutt med lyd.
- Ved å kode video etter den europeiske standarden "phase alternating line" (PAL) hvor man bruker 24 bits for å kode et punkt og ved å anta en bildeoppløsning på 640 X 480 punkter, vil ukomprimert video trenge 7372,8 Kbits for et bilde. Videre har PAL 25 bilder i sekundet som gir et totalt båndbreddebehov på 184,32 Mbits/s og et lagringsbehov på 11,0592 Gbits for å lagre et minutt. Ved å bruke "high definition television" (HDTV) i steden for PAL vil dataraten igjen øke med en faktor på 5,33 sammenlignet med dagens TV.

Som vi ser av tallene over, må vi for video ha sekundær lagringskapasitet i gigabyte-klassen og bufferlager i megabyte-klassen. Dagens teknologi støtter ikke slike datarater i et flerbrukersystem. For å kunne tilby brukbare, kostnadseffektive datamengde- og datarateløsninger bruker derfor de fleste MMSer en passende kompresjonsteknikk for blant annet video og audio.




Figur 2.4: *Intra-* og *inter*rammesynkronisering.

Forskjellie datatyper, krever ofte forkjellige kompresjonsteknikker. Noen slike er: **Joint Photographic Expert Group** (JPEG) for stillbilder, **H.261 (px64)** for video og **Moving Picture Expert Group** (MPEG) som kan komprimere både video og audio sammen.

Faren med å bruke datakodinger som er nevnt her, er at det kan gå ut over kvaliteten. Kvaliteten på kodet og senere dekodet data må være best mulig. Kompresjonsteknikken må ha minimal kompleksitet, og prosesseringen av algoritmen må ikke overstige en viss grense. Videre kan forskjellige teknikker brukes til forskjellige applikasjoner. For eksempel H.261 (px64) er godt egnet til videokonferanser og direkteoverføringer av data, mens MPEG er mer egnet til lagrede applikasjoner og fremvisning av disse [Steinmetz et al. 95].

### 2.4.2 Synkronisering

Synkronisering i MMSer refererer til temporale relasjoner mellom mediaobjektene. Som nevnt tidligere, kan multimediadata være sammensatt av flere forskjellige datatyper. Et presentasjonsobjekt kan inneholde flere dataobjekter, og når flere medier skal presenteres i parallell, må et MMS kunne synkronisere flere datastrømmer. For eksempel når bilde og lyd skal presenteres sammen, er det viktig at lyden passer med munnens bevegelser, og når vi har et lysbildeshow må bildene henge sammen med lyden. Det er derfor nødvendig å lagre tidsrelasjoner mellom mediene i tillegg til dataene selv slik at synkroniseringsmekanismen vet hvilke bilder som hører til hvilke samples i lyden.

Synkronisering er støttet av mange av systemkomponentene i et MMS. Dette inkluderer blant annet OSet, DBSet, kommunikasjonssystemet, multimediaobjektene og applikasjonene. Synkronisering må derfor betraktes på mange forskjellige nivåer. Et eksempel er en MPEG-applikasjon med bilde og lyd som igjen skal synkroniseres med visning av foiler i et elektronisk klasserom. MPEG inneholder selv mekanismer for å synkronisere bilde og lyd, mens andre mekanismer må ta seg av synkroniseringen mellom MPEG-applikasjonen og foilene.

Vi har, som vist i figur 2.4, to typer synkronisering. For det første har vi tidsrelasjoner i et tidsavhengig mediaobjekt selv. Disse brukes til *intraobjekt* synkronisering, det vil si at for eksempel selve datastrømmen til en videosekvens er synkronisert. Hvert bilde må komme i riktig rekkefølge. I figuren betyr dette at ramme v1 må presenteres før ramme v2 og så videre. Videre må tiden hvert bilde vises være like stor. Den andre typen er *interobjekt* synkronisering hvor vi synkroniserer forskjellige datastrømmer. Et eksempel her er synkronisering av bilde og lyd. I figuren betyr dette at audiosamplingene i ramme a1 presenteres sammen med bildene i rammene v1 og v2.

Synkronisasjon i et distribuert system er igjen mer kompleks enn i et lokalt miljø. Forskjellige lagringslokasjoner for mediene gir forskjellige kommunikasjonsavstander mellom presentasjonssted og lagringssted for mediaobjektene. Dette gir igjen ekstra (forskjellige) forsinkelser.

### 2.4.3 Sanntid

For å kunne presentere kontinuerlig, synkronisert data må et MMS kunne tilby sanntidseksekvering av data. En sanntidsprosess er en prosess som leverer resultatet av eksekveringen i løpet av en gitt tidsperiode [Steinmetz 95]. Eksekveringen må være ferdig før en bestemt *tidsfrist* som markerer det senest aksepterte tidspunktet for å levere ferdige resultater. Et sanntidssystem feiler derfor ikke bare hvis hardware og software feil oppstår, men også hvis systemet ikke greier å overholde en slik tidsfrist.

Vi har tre typer tidsfrister. Dette er *myke (soft)* -, *faste (firm)* - og *harde (hard)* tidsfrister [Goebel et al. 96a]. En prosess med en myk tidsfrist kan overtrede tidsfristen og likevel ha en monotont synkende verdi ettersom tiden går. En prosess med en fast tidsfrist har ingen verdi etter at tidsfristen har gått ut, men tidsfristen er her det tidspunktet hvor prosesser med myke tidsfrister ikke lenger har noen verdi [Özsoyoğlu et al. 95]. Myke- og faste tidsfrister kan overtredes hvis det ikke skjer for ofte eller med for mye tid, og de brukes derfor når konsekvensen av en overtredelse ikke gir et uakseptabelt resultat som systemkrasj. I motsetning til myke- og faste tidsfrister, kan ikke harde tidsfrister overtredes. En hard tidsfristovertredelse medfører systemfeil og aksepteres ikke.

Som nevnt over er audio og video kontinuerlige datastrømmer. For at disse skal kunne presenteres kontinuerlig må hver logiske dataenhet, for eksempel hver audiosample eller videoramme, være klar til en bestemt tidsfrist. Likevel er ikke sanntidskravene i MMSer like strenge som i andre sanntidssystemer. I mange multimediaapplikasjoner vil en overtredelse av tidsfristen ikke føre til en feil, men det bør selvfølgelig unngås. En videoramme som ikke rekker tidsfristen kan ofte ubemerket hoppes over. Krav til lyd er imidlertid noe strengere da det menneskelige øret er mer sensetivt til hull i lydstrømmen enn øyet er til videojitter. Multimedia tidsfrister kan derfor ofte sees på som myke.

# Kapittel 3

# Datahåndteringsaspekter fra det elektroniske klasserommet

I kapittel 2 så vi på multimediaapplikasjoner generelt. I dette kapittelet ser vi nærmere på en planlagt utvidelse av elektroniske klasserommet, det vil si et DHS for lagring av forelesninger, slik at vi vet nøyaktig hvilke krav som stilles og hvilke egenskaper som er godt egnet til en bufferhåndteringsmekanisme fra denne applikasjonen.

I dette kapittelet beskriver vi bakgrunnen for det elektroniske klasserommet (avsnitt 3.1), samt av vi presenterer hvordan det elektroniske klasserommet fungerer i dag (avsnitt 3.2) med ulike datatyper (avsnitt 3.3). Videre presenterer vi ønsket funksjonalitet i vår planlagte utvidelse av dagens applikasjon (avsnitt 3.4). Til slutt beskriver vi noen av de kravene et slikt DHS for det elektroniske klasserommet vil stille til en bufferhåndteringsmekanisme (avsnitt 3.5).

I [Goebel et al. 97] er det en mer omfattende beskrivelse av vårt planlagte DHS for det elektroniske klasserommet med alle typer krav til systemet og hva slags teknologistøtte vi har i dag for å kunne lage et slikt system.

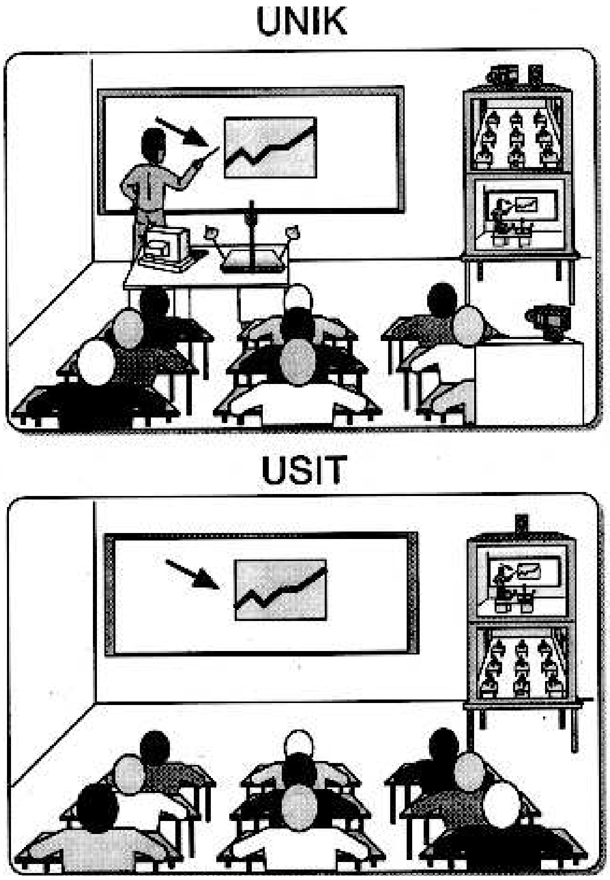
## 3.1 Bakgrunn

Norge har relativt spredt bosetning og lav befolkningstetthet, og det kan derfor vanskelig bygges opp undervisningsinstitusjoner på en slik måte at alle har det de ønsker av undervisningstilbud innen rimelig avstand fra sitt bosted. Til tross for dette, er det et utbredt ønske om et desentralisert høyere utdanningssystem. Dette ønsket, sammen med mulighetene moderne informasjonsteknologi gir, er bakgrunnen for at prosjektet med det elektroniske klasserommet ble startet [Bakke et al. 94].

## 3.2 Dagens elektroniske klasserom

Den konseptuelle ideen er et klasserom som integrerer video, audio og elektronisk tavle. Tidligere erfaringer fra fjernundervisning og videokonferanser har indikert et behov for å kunne dele informasjon, og det er dette den elektroniske tavlen sørger for. I tillegg er det lagt opp til at klasserommet skal være så likt et vanlig klasserom som mulig, slik at nye brukere ikke skal ha store problemer med å tilpasse seg. Til slutt har det vært ønskelig med en stor grad av automatisering, slik at det ikke er nødvendig med noe stort team rundt klasserommet for å holde undervisningen i gang [Bakke et al. 94].

Det elektroniske klasserommet er i dag beregnet på direkte overføring av forelesninger, det vil si at studentene må være til stede i et av klasserommene mens undervisningen faktisk pågår. Det er ingen mulighet for å spille av lagrede forelesninger. Den eneste mulighetene i så måte er




Figur 3.1: Det elektroniske klasserommet ved UNIK og USIT [Bringsrud et al. 94].

å ta opp audio og video ved hjelp av en vanlig videospiller. Dette vil imidlertid ikke gi noe godt bilde av den elektroniske tavlen, slik at nytteverdien av forelesningen vil bli redusert. Det er også et potensielt problem at det må kopieres og distribueres videokassetter.

Figur 3.1 viser hvordan det elektroniske klasserommet fungerer i dag mellom UNIK og universitetets senter for informasjonsteknologi (USIT) ved universitetet i Oslo. Virkemåten er slik at to (eller flere) klasserom som ligger på geografisk adskilte steder tar opp det som skjer i klasserommene under en forelesning ved hjelp av videokameraer og mikrofoner og sender dette til hverandre. Klasserommet her på UNIK har et kamera rettet mot foreleseren, og i tillegg har hvert klasserom et eller to kameraer rettet mot studentene. Mikrofonene er plassert i en matriseformasjon i taket, og i tillegg har foreleseren en egen "mygg". På denne måten kan studenter følge med på en forelesning selv om foreleser er på et helt annet sted. De kan også stille spørsmål og komme med kommentarer akkurat som om alt skulle ha foregått på samme sted.

Kameraene og mikrofonene i et klasserom sørger til en hver tid for å ta opp den personen som snakker høyest. Dette krever noe disiplin i klassen, men ved normalt bruk vil foreleser være i fokus det meste av tiden.

Foreleseren har også en del hjelpemidler til rådighet. Den elektroniske tavlen (whiteboard) brukes til å vise fram transparenter eller bilder som er skannet inn, og krittet er byttet ut med en lyspenn. Tavlen vises med lik tilstand ved begge klasserommene slik at det foreleseren gjør med lyspennen på et sted vil automatisk forplante seg til de andre tavlene. Tavlen kan også fungere som en "world wide webb"-browser (WWW-browser) med de fasiliteter dette medbringer.

## 3.3 Datatypene i det elektroniske klasserommet

Basert på den funksjonaliteten vi beskrev i forrige avsnitt, kan vi dele det elektroniske klasserommet inn i tre hoveddeler:

**Den elektroniske tavlen:** Dette er en vindusapplikasjon basert på en såkalt "shared window" teknikk basert på X11, som er et grafisk brukergrensesnitt for UNIX. Dermed viser de elektroniske tavlene i alle klasserommene det samme bildet, og enhver oppdatering som gjøres på en av tavlene vises umiddelbart på alle de andre. På tavlen vises det typisk lagrede "hypertext markup language"-dokumenter (HTML-dokumenter) eller "graphics image format"-bilder (GIF-bilder). Et slikt lagret dokument har samme funksjon som overhead-foilene som brukes i en tradisjonell undervisningssituasjon. Fremvisningen skjer ved hjelp av en semitransparent skjerm som belyses bakfra av en videokanon.

**Audiosystemet:** Her benyttes ukomprimert "Pulse Coded Modulation"-koding (PCM-koding, se [Steinmetz et al. 95]) med en 16 bits/16 KHz sampler (16.000 samplinger av 16 bits per sekund). Overføringen av audio mellom klasserommene er full duplex, med ekkokansellering [Sæthre 96].

**Videosystemet:** De analoge bildene fra videokameraene digitaliseres og kodes ved hjelp av komprimeringsalgoritmen CCITT H.261 (se [Steinmetz et al. 95]). Hvilket kamera som er aktivt til enhver tid, bestemmes av lyden [Sæthre 96].

Videre har vi som vist i tabell 3.1 forskjellige datatyper i det elektroniske klasserommet ut i fra det elektroniske klasserommets hoveddeler og planlagte utvidelser (avsnitt 3.4). Denne tabellen viser også hvilke "Input/Output"-enheter (I/O-enheter) som brukes. I avsnittene under beskrives hver av disse datatypene nærmere.

| Datatype | Input-enhet | Output-enhet |
|---|---|---|
| **Video** | Videokameraer | TV-skjermer |
| **Video** | Dokumentkamera | Tavlen |
| **Audio** | Mikrofoner | Høytalere |
| **Bilde** | Skanner | Tavlen |
| **Vektorgrafikk** | Lyspenn | Tavlen |
| **Tekst** | Tastatur | Tavlen, skjerm |

Tabell 3.1: Forskjellige datatyper i det elektroniske klasserommet.

### 3.3.1 Video

I det elektroniske klasserommet har vi to typer video: vanlig video mot foreleser og studenter, og et dokumentkamera som sender video av et dokument som det for eksempel tegnes på.

Vanlig video vises på TV-skjermer, og normalt er det bildene fra kameraet som er rettet mot foreleseren som vises. Ved spørsmål fra studentene, er det studenten som spør som blir filmet.

Dokumentkameraet gir foreleseren mulighet til å vise tekst og bilder på direkten. Bildene fra dokumentkameraet blir vist "online" på den elektroniske tavlen.

### 3.3.2 Audio

Audiodataene består av lyden av det foreleseren sier, samt spørsmål fra studentene. Lyden er et av de viktigste mediene i denne sammenhengen, og det er av stor betydning at lydkvaliteten er bra og uten avbrudd.

### 3.3.3 Bilde

Skanneren gjør det mulig for foreleseren å lese inn tekst og bilder som så kan hentes frem og vises på tavlen. Dette gjøres normalt som et ledd i forberedelsene til forelesningen hvor de innskannede bildene legges inn i HTML-dokumenter. Siden bildene fra skanneren må lagres før de kan vises, blir dette noe mer "offline" enn dokumentkameraet. Både dokumentkameraet og skanneren gjør det mulig for foreleseren å improvisere ved at ny informasjon enten kan vises direkte, ved hjelp av dokumentkameraet eller den skannes og lagres slik at den kan hentes frem senere.

### 3.3.4 Vektorgrafikk

På samme måte som man kan bruke tusj på overhead-foiler, gir en lyspenn foreleseren mulighet til å gjøre tilføyelser på det som vises på tavlen. Den gir også mulighet for å peke, ved at et lysende punkt kan flyttes rundt på tavlen, og plasseres der foreleseren ønsker.

Dataene skrevet eller tegnet med lyspenn lagres som vektorgrafikk knyttet til den enkelte foilen. På denne måten kan markeringer og tilleggsdata vises ved senere avspillinger av forelesningen.

### 3.3.5 Tekst

I de fleste kurs er det aktuelt med øvingsoppgaver. Disse legges normalt ut som tekstdokumenter som studenten så kan laste ned. I tillegg kan det være aktuelt med interaktive øvingsoppgaver hvor flere (geografisk adskilte) studenter jobber sammen på samme dokument og hvor oppgavene rettes av systemet.

Videre ligger det en del kursinformasjon som tekst omkring det aktuelle kurset, for eksempel kursbeskrivelse, pensumliste, litteraturliste og praktiske opplysninger. Dette lagres som ren tekst eller eventuelt som HTML-dokumenter.

## 3.4 Planlagte utvidelser i det elektroniske klasserommet

Til tross for sin distribuerte natur, er dagens elektroniske klasserom fortsatt til en viss grad sentralisert i den forstand at det (foreløpig) ikke finnes så mange slike klasserom, og studenten må møte opp ved ett av disse. Dessuten er studentene nødt til å møte opp når foreleseren holder forelesningen. Hvis en student av en eller annen grunn ikke har anledning til å møte opp ved en forelesning, har han ingen mulighet for å se denne på et senere tidspunkt.

For å endre på dette, foreslår vi en utvidelse av funksjonaliteten til det elektroniske klasserommet. Hovedkonseptet er at alle forelesninger lagres i et DHS, det vil si et multimedia-DBS, slik at de kan spilles av på et senere tidspunkt. Alle datatypene lagres separat, for å gi størst mulig fleksibilitet. For å identifisere de ulike brukernes behov, har vi delt dem inn i tre grupper: studenter, forelesere og administrasjon.

### 3.4.1 Studenter

Et slikt multimedia-DBS for et elektronisk klasserom er først og fremst beregnet på å kunne vise forelesningene i et kurs til personer som ikke har anledning til å følge forelesningene direkte, og studentene er derfor kanskje den viktigste brukergruppen. For å kunne ha et enda mer fleksibelt system tenker vi oss en ytterligere distribusjon av klasserommet hvor vi ser for oss muligheten for å spille av forelesninger over telenettet til en vanlig multimedia-PC eller arbeidsstasjon. På den måten kan en student som ikke har anledning til å følge en forelesning, enten det skyldes tid eller geografisk avstand, følge forelesningen på det sted og tidspunkt som passer best. Ved oppkopling mot DBSet for eksempel via en PC eller en arbeidsstasjon, undersøkes det hvilken QoS overføringsnettet kan tilby og hva studenten har behov for. Ut fra dette beregnes det hva studenten kan tilbys av tjenester. Resultatet blir formidlet til studenten, som så kan velge om han vil akseptere den tilbudte kvaliteten eller ikke.

En student vil kunne foreta en rekke operasjoner mot vårt DBS for lagrede forelesninger. Det viktigste er naturlig nok å spille av forelesninger. Dette kan foregå på flere måter:

- Forelesningen kan spilles av på vanlig måte, med alle medietyper i bruk.
- En døv student vil ikke ha glede av lyden, men nøye seg med video og den elektroniske tavlen.
- En student som leser før en eksamen kan tenkes å nøye seg med kun lyden, for å få en rask repetisjon. Eventuelt kan studenten hurtigspole frem og tilbake for å få spilt av noe på nytt.

Videre bør det være mulighet for å lete etter bestemte temaer innen en forelesningsserie. Et eksempel kan være å finne frem de forelesninger som inneholder stoff om Fourier-analyse. Her ser vi også for oss en mulighet for å søke etter figurer basert på stikkord.

Øvingsoppgaver hører med i de fleste kurs, og en student må derfor ha mulighet for å hente ned slike. I sin enkleste form er dette tekstdokumenter som studenten så besvarer og sender tilbake til systemet for eksempel via elektronisk post. En mer avansert versjon gir studentene mulighet for å delta i gruppearbeid ved at de kan arbeide sammen på samme dokument. Besvarelsen kan så rettes av systemet (flersvarsoppgaver) eller hentes ut av foreleseren for retting.

Det elektroniske klasserommet gjør det mulig å vise animasjoner og simuleringer, og studentene må ha mulighet for å spille av slike. Studenten kan også tenkes å ville se nærmere på en slik animasjon, så det bør være mulighet for avspilling i slow-motion.

Til slutt må studenten kunne hente ned praktiske opplysninger, slik som pensumlister, litteraturlister, eksamensdato og så videre.

### 3.4.2 Foreleseren

Det er foreleseren som blant annet legger til rette foilene (HTML-dokumentene) til den elektroniske tavlen for en forelesning. I dag foregår lagringen av foilene ved at dokumentene legges inn som filer i en diskkatalog. I vår fremtidige versjon av klasserommet vil foilene lagres i et DBS. På denne måten kan foreleseren på en enkel måte finne igjen foiler og oppdatere disse. Videre har foreleseren også ofte behov for å gå tilbake til en tidligere foil under en forelesning. I dag må dette gjøres ved å bla seg bakover eller ved å gå direkte til foilen dersom han husker foilnummeret. Dette har vist seg å kunne være litt problematisk fra tid til annen. Ved i stedet å lagre dem i et DBS, gis det mulighet for å presentere alle foilene som ikoner eller stikkord, for på den måten å gjøre det lettere å finne tilbake til riktig foil. Det blir også i langt større grad mulig med dynamisk krysslenking mellom dokumenter, slik at det er enkelt å endre rekkefølgen på foilene, og nye foiler kan legges inn mellom de gamle.

Etter en forelesning må foreleseren ha mulighet til en gjennomgang av denne. For det første kan det være aktuelt å registrere opplysninger omkring forelesningen som for eksempel hvor mange studenter som var til stede, notere viktige spørsmål fra studentene og så videre. Videre må det være mulighet for editering av video og audio. På denne måten kan unødvendige sekvenser som for eksempel hosteanfall fjernes ved at ny lyd kan legges på. I tillegg kan vi også tenke oss muligheten for å lese inn nye lydspor for hele forelesningen slik at studenten har mulighet for å velge hvilket språk han vil høre.

Foreleseren må også ha mulighet for å registrere øvingsoppgaver i DBSet, og knytte disse til den enkelte forelesning. Eventuelle besvarelser vil også bli lagret hvor de kan hentes ut for retting. Enkelte metadata som litteratur- og pensumlister vil også være foreleserens ansvar.

### 3.4.3 Administrasjon

Administrasjonen vil normalt kun ha behov for tilgang til kursenes metadata. Dette kan for eksempel være undervisningstidspunkter, eksamensdatoer, kursdeltakere og deres karakterer.

I tillegg tenker vi oss at brukerne kategoriseres og at det tildeles brukermodeller. En slik brukermodell beskriver brukerens behov, og er med på å avgjøre hvilke tjenester studenten skal tilbys. En brukermodell for blinde studenter vil for eksempel ikke tilby video, mens en modell for døve studenter ikke tilbyr lyd. Det vil være administrasjonens oppgave å administrere slike brukermodeller, samt å tildele dem til de enkelte studentene.

## 3.5 Krav fra det elektroniske klasserommet til en bufferhåndterings-mekanisme

Den planlagte utvidelsen av den "elektroniske klasserom"-applikasjonen som er beskrevet over, introduserer en rekke krav til DBSet. Det er enorme mengder med data som må håndteres, og DBSet må støtte en rekke forskjellig operasjoner på disse dataene som igjen vil si støtte for forskjellige referansestrukturer av dataene. Spesielt for bufferhåndteringen stilles det store krav til båndbredde, responstid og gjennomstrømning.

### 3.5.1 Båndbredde

Når det gjelder krav til båndbredde, er det audio og video som har de største behovene. Størrelsene her avhenger av parametere som kodingsteknikk, bilder per sekund, bildeoppløsning og -størrelse, antall bits per piksel eller sampel, antall samplinger per sekund og så videre.

I avsnitt 2.4.1 viste vi eksempler på båndbredder for ukomprimert audio. Videre så vi i avsnitt 3.3 at audio sendes ukomprimert fra en 16 bits/16 KHz sampler. Dette gir et båndbreddebehov på 32 KBytes/s for audio. Hvis vi benytter andre kodinger enn 16 bits/16 KHz, vil vi få andre båndbreddebehov. Kodingene 8 bits/16 KHz og 8 bits/8 KHz gir henholdsvis et båndbreddebehov på 16 KBytes/s og 8 KBytes/s. Videre er denne datastrømmen ukomprimert og jevn, det vil si at det sendes lyd selv om det er helt stille i klasserommet. Ved å komprimere lyden er det derfor mulig å redusere båndbreddebehovet noe.

Når det gjelder bilder fra videokameraene vil det, som dataene fra avsnitt 2.4.1 viser, være uholdbart å lagre og sende ukomprimerte datastrømmer for video. Båndbreddebehovet vil derfor også her variere med kodingsteknikk (bilderaten er satt til 24 bilder per sekund):

**H.261** gir de laveste kravene til båndbredde, men i følge [Steinmetz et al. 95] er denne kodingen beregnet på direkte overføringer og ikke så godt egnet for lagrede data. I følge et forsøk i det elektroniske klasserommet her på UNIK [Sæthre 96], gir denne teknikken en svært variabel datarate med svingninger fra 1,6 KBytes/s til 240 KBytes/s med et snitt på 65 KBytes/s. Hvis vi ønsker en garantert QoS, må vi gå ut fra det verste tilfellet og trenger dermed en båndbredde på 240 KBytes/s. Skal vi derimot gå ut fra en beste mulige leveranse av videostrømmen, kan vi gå ut ifra gjennomsnittet som er 65 KBytes/s.

**MPEG** gir også variable datarater med et snitt på circa 225 KBytes/s. Dette vil altså bli dataraten ved best mulig QoS. Hvis vi ønsker garantert QoS, må vi igjen gå ut fra verst tenkelig tilfelle. I følge [Steinmetz et al. 95] kan MPEG maksimum levere en datarate på 232 KBytes/s. En annen måte å se det verste tenkelige tilfelle er å se på dataraten når MPEG sender et helbilde (en såkalt I-ramme). Dette helbildet er et vanlig JPEG-kodet bilde, og vi må derfor gå ut fra samme datarate som for JPEG-koding (se under).

**JPEG** gir en kontinuerlig datarate siden det bare sendes helbilder. Denne blir imidlertid svært stor, det vil si hvis vi går ut i fra en bildestørrelse på overkant av 60 KBytes, blir båndbreddebehovet circa 1,5 MBytes/s. Siden dataraten er kontinuerlig, får vi ikke noe skille mellom garantert og best mulig QoS.

Disse tallene er imidlertid noe usikre. Både H.261 og MPEG har en variabel bitrate, og det er også muligheter for forskjellige kodinger avhengig av ønsket kvalitet. Videre kan MPEG også kode lyd sammen med bilde, og hver rammestørrelse avhenger også av bevegelsen i videoen (forskjeller fra bilde til bilde).

I tillegg setter animasjoner store krav til båndbredde. Her er imidlertid usikkerheten stor når det gjelder konkrete verdier, men vi kan gå ut fra de samme kravene som for video som en øvre skranke.

Når det gjelder de øvrige medietypene, er kravene til båndbredde beskjedne. Kun foilene som skal vises på den elektroniske tavlen har behov for båndbredde av noen betydning. En foil kan ha en størrelse på 400 KBytes, men med relativt lave krav til responstid (størrelsesorden 10 sekunder) blir kravet til båndbredde omtrent 40 KBytes/s. Henting av foiler vil dessuten ikke skje kontinuerlig, slik at ved å starte innlesingen av foilene før de trengs, kan kravene senkes ytterligere. De resterende medietypene har ubetydelige båndbreddekrav.

Tallene over viser oss at et gjennomsnittlig båndbreddebehov per bruker vil være i størrelsesorden 1-2 MBytes/s.

### 3.5.2 Responstid

Responstiden har stor betydning i mange applikasjoner. Når vi ser på responstiden sammen med DBSet for det elektroniske klasserommets funksjonalitet, vil det si at bufferhåndteringsmekanismen må garantere at sidene blir levert fra sekundær- til primærminnet innen en bestemt tidsfrist slik at vi ikke får forskjellige typer skjevheter i fremvisningen av de kontinuerlige mediene, samt at responstiden ved en interaksjon fra brukeren, det vil si tidsforsinkelsen mellom brukerens interaksjon og tiden det tar til systemet reagerer på denne interaksjonen ved å fortsette fremvisningen, er minimal. Responstiden bør derfor for hver av sidene være kontinuerlig under en viss øvre grense.

### 3.5.3 Gjennomstrømning

Samtidig med en rask og jevn responstid, må vi kunne ha en stor gjennomstrømming av data for å kunne håndtere flere datastrømmer og klienter samtidig. På grunn av medietypenes karakter henger dette kravet sterkt sammen med kravet om høy båndbredde. I et mulig scenario kan vi for eksempel ha flere kurs lagret i DBSet, og med mange studenter som følger hvert kurs, risikerer vi å ha mange samtidige brukere. Til sammen gir dette en betydelig ankomstrate av I/O-forespørsler, og bufferet må derfor støtte en stor gjennomstrømming av data.

### 3.5.4 Krav til maskinvare

Som nevnt tidligere er det behov for store lagringskapasiteter og dataoverføringer. Bufferet bør selvsagt være så stort som mulig, men hvor stort bufferet må være avhenger av hvor mye data vi har. Hvor mye data vi må håndtere avhenger igjen av hvilke kompresjonteknikker vi bruker. Ved bruk av lite kompresjon vil vi trenge mye lagringsplass og et stort buffer, mens ved en kraftig kompresjon av dataene vil dette kravet minske noe. Ulempen med sterk kompresjon er imidlertid at vi trenger mer prosesseringskraft for dekoding, så det gjelder her å finne en "gylden middelvei".

En annen faktor som påvirker kravet til bufferstørrelse er overføringshastigheten til andre komponenter som disker og nettverk. Hvis for eksempel disken er treg, vil vi måtte forhåndshente mye data, og disse dataene vil okkupere bufferplass.

# Kapittel 4

# Datahåndtering i multimediasystemer

I kapittel 2 har vi oppsummert multimediaapplikasjoner og deres krav til systemene. I dette kapittelet beskriver vi nærmere forskjellige typer DHSer, det vil si alle typer software-komponenter som håndterer lagring, henting, sending og endring av data, for å bygge opp en bakgrunn for hvordan multimediadata håndteres i forskjellige lagringssystemer.

Multimediadata kan lagres på flere måter. Et DHS må som sagt kunne lagre, hente og forandre persistente data i for eksempel filsystemer eller DBSer. For å illustrere funksjonaliteten og kunne se enkelte likhetstrekk mellom forskjellige DHSer som presenteres i denne oppgaven, brukes en lagdelt arkitekturmodell av et databasehåndteringssystem (database managment system, DBMS) som vist i figur 4.1. Hvert lag støtter et sett av operasjoner og inneholder flere typer mekanismer.

Først tar vi for oss multimedia-OSer (avsnitt 4.1). Her har vi også tatt med en del andre aspekter ved OSer, for datahåndteringen er ikke et isolert system, men blir påvirket og består av en rekke andre komponenter. Deretter beskrives DHSer som DBSer (avsnitt 4.2) og filservere (avsnitt 4.3) innen multimedia. Videre presenteres DHSenes QoS-parametere (avsnitt 4.4), og tilslutt ser vi (avsnitt 4.5) på systemers støtte for multimedia og QoS.

## 4.1 Multimediaoperativsystemer

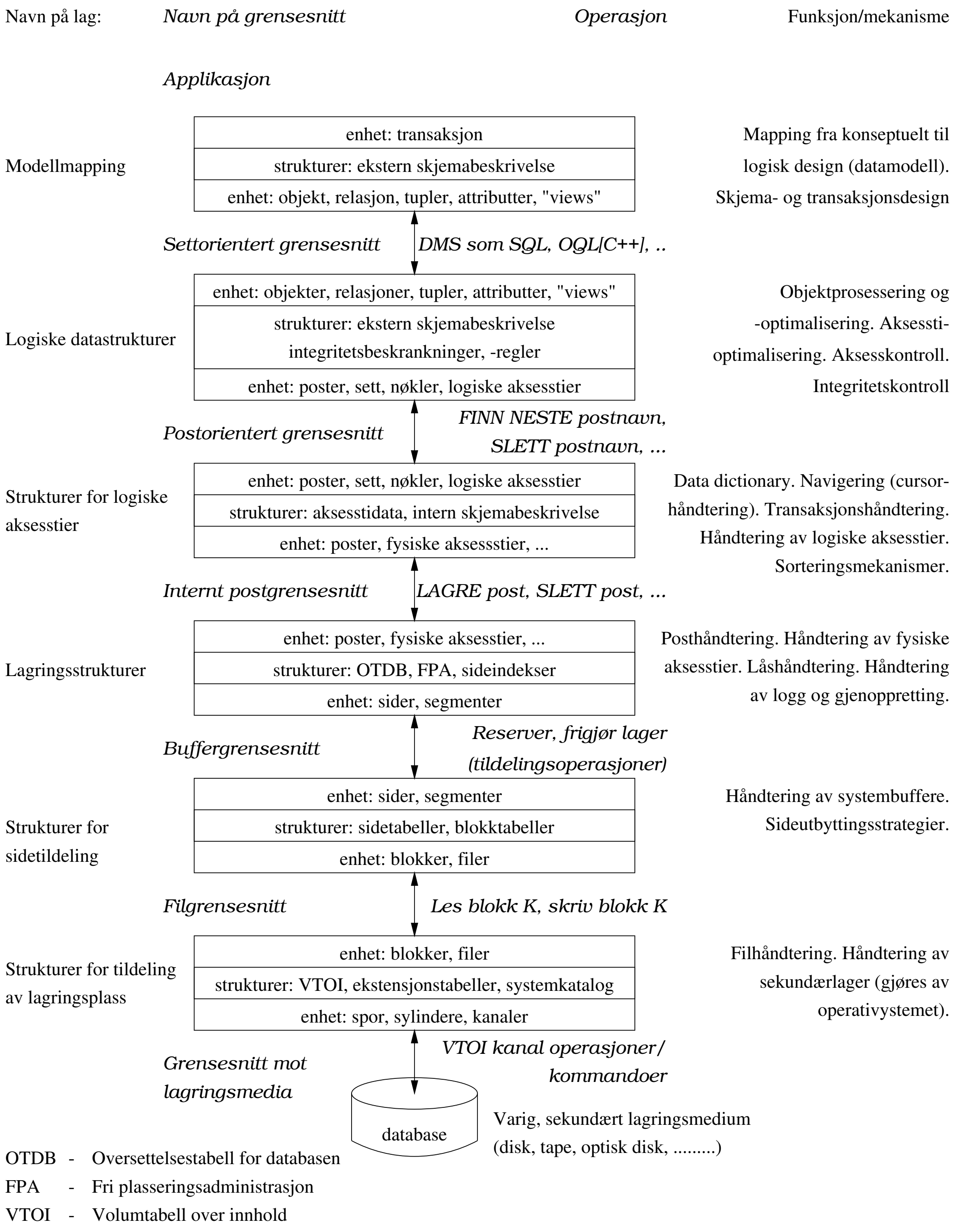
Et OS er et software-system som kan defineres ved å si at det har to oppgaver [Tanenbaum 87]:

> *Et OS skal gi brukeren en virtuell maskin å arbeide med, og det skal administrere maskinens ressurser.*

For det første betyr dette at et OS skjuler hardware'en og gir et pent og oversiktlig grensesnitt til maskinressursene. Brukeren kan dermed for eksempel lese en blokk fra disken uten å tenke på sektorer, spor, formatering og plassering på disken. OSet tilbyr altså tjenester relatert til prossesoren, internminnet, datalagring og alle "Input/Output"-enheter (I/O-enheter).

Videre består moderne maskiner av en mengde forskjellige enheter som prosessorer, minne, disker og skrivere. Oppgaven til et OS er å håndtere og tildele alle disse ressursene til programmene som konkurrerer om å få bruke dem. I et distribuert MMS inkluderer ressurshåndtering flere maskiner samt et kommunikasjonsnettverk. Den allokerer alle ressurser med i datatransmisjonen fra sender til mottaker. Ressurshåndtereren må derfor ved allokering av en ny ressurs, for eksempel ved opprettelse av en ny forbindelse, sjekke at det ikke oppstår vranglåser (se [Tanenbaum 87]) eller at ressurser taes fra andre prosesser og dermed bryter deres QoS-krav.

Begrepet ressurshåndtering kan utvides til å dekke prosesshåndtering, minnehåndtering, filsystemet og enhetshåndtering. Videre kan en ressurs sies å være en systementitet som prosesser trenger for manipulasjon av data. En ressurs' egenskaper klassifiseres ved at de er:




Figur 4.1: En lagdelt arkitekturmodell for et DBMS [Goebel et al. 96a].

- *Aktive eller passive*. En aktiv ressurs, som prosessoren eller en nettverkadapter, tilbyr tjenester, mens en passiv ressurs, som internminnet, er noe en aktiv ressurs trenger.

- *Eksklusiv eller delt*. Aktive ressurser er ofte eksklusive, mens passive ofte deles.

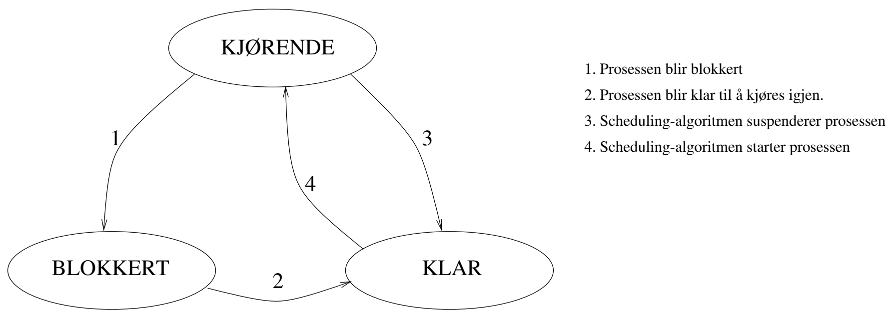
- *Singel eller multiple*. En ressurs som det bare finnes en av i systemet er singel, ellers er de multiple.

En ressurs' kapasitet måles i en arbeidsoppgaves mulighet til å yte i et gitt tidsrom ved bruk av ressursen. Med kapasitet menes her for eksempel prosessorkapasitet, frekvensområde og mengden av lagringsplass. Sanntids-scheduling tar bare hensyn til den tidsmessige delingen av en ressurs' kapasitet på sanntidsprosesser [Steinmetz 95].

Som vi så i avsnitt 2.4, vil multimediaapplikasjoner kreve tjenester som sanntidseksekvering og synkronisering av datastrømmer, og en bufferhåndteringsmekanisme påvirkes av disse komponentene. I dette avsnittet vil vi derfor komme inn på de viktigste aspektene (bortsett fra minnehåndtering som omtales i kapittel 5) og tjenestene til et multimedia-OS.

### 4.1.1 Prosesser

En **prosess** kan sammenlignes med et program under utførelse. Den består av et eksekverbart program, diverse registere og informasjon som programdata, programteller, stakk og stakkpeker, for å kunne kjøre programmet.




Figur 4.2: En prosess kan være *kjørende*, *klar* eller *blokkert*.

Mange OSer, blant annet UNIX, kan utføre flere prosesser "samtidig", det vil si i parallell[1]. I praksis kan en prosessor bare utføre en ting av gangen, så den faktiske eksekveringstiden på prosessoren må fordeles på en mest mulig rettferdig måte. Dette at prosessene ikke kjører hele tiden gir oss flere mulige tilstander for en prosess. Som vi ser av figur 4.2, er dette:

- KJØRENDE. Prosessen bruker prosessorenen.
- KLAR. Prosessen er klar til å kjøre, men er i en ventende tilstand hvor den venter på at prosessoren skal bli ledig.
- BLOKKERT. Prosessen er blokkert.

Figuren viser også at det er fire mulige overganger mellom tilstandene. Den første viser at en prosess som kjører blir BLOKKERT. Den venter på at en ekstern hendelse, som for eksempel skriving til skjermen, skal bli ferdig. Den andre viser at den eksterne hendelsen er ferdig, og at denne prosessen igjen er KLAR. Overgang 3 og 4 viser henholdsvis at scheduling-algoritmen suspenderer og starter en prosess. Prosessen får tilstanden KLAR i overgang 3 og KJØRENDE av overgang 4.

### 4.1.2 Scheduling

Flere prosesser kan være kjørbare samtidig, og det er et OSets oppgave å fordele prosessorressursene på de aktuelle prosessene etter en bestemt scheduling-algoritme. Ved valg av en slik fordelingsstrategi er det flere kriterier vi må ta hensyn til. I tradisjonelle OSer vil dette si at alle prosesser

[1] I et flerprosessorsystem kjører prosessene i ekte parallell, men hvis vi tidsdeler *en* prosessor, får vi kvasiparallell.

må få en viss del av den totale eksekveringstiden, at responstiden for brukere må minimaliseres og at scheduling-algoritmen må være effektiv slik at prosessoren ikke ligger ubrukt[2] [Tanenbaum 87]. Under ser vi både på tradisjonelle scheduling-algoritmer og algoritmer for sanntidseksekvering.

#### 4.1.2.1 Scheduling i tradisjonelle systemer

Det finnes flere slike scheduling-algoritmer som et OS kan benytte for å bestemme hvilken prosess som skal få kjøre. Hvilken som er best kan variere fra system til system avhengig av hva slags applikasjoner som skal kjøres. Tradisjonelt brukes ofte disse strategiene [Tanenbaum 87]:

***Round-Robin scheduling* (RR, lang lang rekke)** virker på den måten at når en prosess har brukt opp sin tid stiller den seg bakerst i køen og venter igjen på sin tur.

***Prioritetsfordelt scheduling*** gir hver prosess en prioritet, og den prosessen med høyest prioritet får lov til å bruke prosessoren.

***Prioritetsgruppe scheduling*** deler opp prosessene i prioritetsgrupper, og organiserer hver gruppe igjen etter RR.

Ved bruk av RR vil alle prosessene få sin del av eksekveringstiden, mens ved en prioritetsfordelt scheduling kan man risikere at bare prosesser med høy prioritet får kjøre. På den andre siden vil ikke viktige prosesser få mer eksekveringstid enn lavtprioriterte ved bruk RR. Det vil de med prioritetsfordelt scheduling. Prioritetsgruppe scheduling er en mellomting mellom de to andre. Her kjører hver prosess i den høyest prioriterte gruppa et intervall hver på rundgang. Først når alle prosessene i denne gruppa er ferdigeksekvert, får neste gruppe begynne. Her risikerer man på samme måte som over at de lavest prioriterte prosessene må vente "evig" ved stor tilgang på høytprioriterte prosesser, så ved prioritetsbasert scheduling bør prioriteten justeres etter hvert som prosessene venter på å få prosessortid.

#### 4.1.2.2 Scheduling i sanntidssystemer

Multimediaapplikasjoner må som tidligere nevnt eksekveres i sanntid, og den viktigste egenskapen til sanntidssystemer er behovet for riktighet. Dette gjelder ikke dette bare feilfri eksekvering, men også at resultatene blir presentert før tidsfristen går ut.

Ofte regnes hastighet og effektivitet som de viktigste egenskapene til et MMS, men det er ikke tilfelle. En avspilling av et videoklipp vil bare bli akseptert hvis den verken går for fort eller for sakte. MMSer må derfor også ta hensyn til forskjellige relaterte arbeidsoppgaver eksekvert samtidig som tidsavhengigheter og logiske avhengigheter. I konteksten av kontinuerlige multimediadatastrømmer er dette oppgaver som prosessering av synkronisering av audio og video hvor tidsavhengigheten mellom mediene må taes hensyn til.

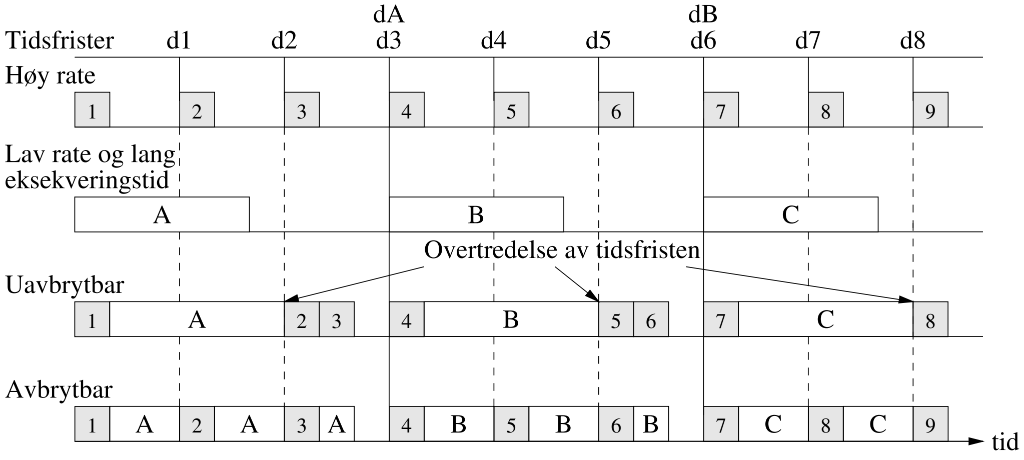
Siden operasjoner på kontinuerlige datatyper foregår om og om igjen i bestemte periodiske intervaller og alltid må overholde tidsfristen, må en sanntidsprosesshåndterer bruke en scheduling-strategi som kan gi eksekveringsgarantier. Problemet er å finne en god scheduling-algoritme som lar alle tidsavhengige prosesser levere sine resultater innen tidsfristene, for et system som fordeler arbeidsoppgaver innen multimedia må ta to motstridende mål i betraktning:

- Prosesser som ikke er avhengig av sanntidsprosessering må ikke bli utsultet på bekostning av sanntidprosesser.
- Tidsavhengige prosesser må aldri utsettes for prioritetsinverson, det vil si at lavtprioriterte prosesser tar prosessortid fra høytprioriterte prosesser.

[2] Ved tidsdeling av prosessoren vil man ikke kunne få 100% utnyttelse av prosessoren. Det går alltid bort litt tid til bytte av prosesser og lignende.

Målet med tradisjonelle scheduling-algoritmer for tidsdelte maskiner er optimal gjennomstrømming, optimal ressursbruk og en rettferdig kø. For sanntidssystemer er målet å kunne eksekvere så mange prosesser som mulig slik at flest mulig imøtekommer sine tidsfrister. Scheduling-algoritmen må fordele oppgaver på ressursene slik at alle kan overholde sine tidskrav.

En scheduling-algoritme kan være avbrytbar eller uavbrytbar. En avbrytbar prosess kan avbrytes og byttes ut med en prosess som har høyere prioritet, mens en uavbrytbar prosess ikke kan avbrytes hvis den ikke selv gir slipp på prosessoren. Ved bruk av sanntidseksekvering brukes vanligvis en avbrytbar algoritme, for som vi ser av figur 4.3 kan den uavbrytbare algoritmen bryte flere tidsfrister.




Figur 4.3: Avbrytbare scheduling-algoritmer er ofte bedre egnet enn uavbrytbare til sanntidseksekvering.

Det eksisterer flere scheduling-algoritmer for sanntidseksekvering, men de fleste er bare variasjoner av disse to [Steinmetz 95]:

***Earliest deadline first scheduling* (EDF, tidligste tidsfrist først)** eksekverer som navnet sier den prosessen som først må være ferdig. Når en ny prosess ankommer avbrytes eksekveringen, en ny prosesseringsrekkefølge beregnes og eksekveringen fortsetter etter den nye rekkefølgen.

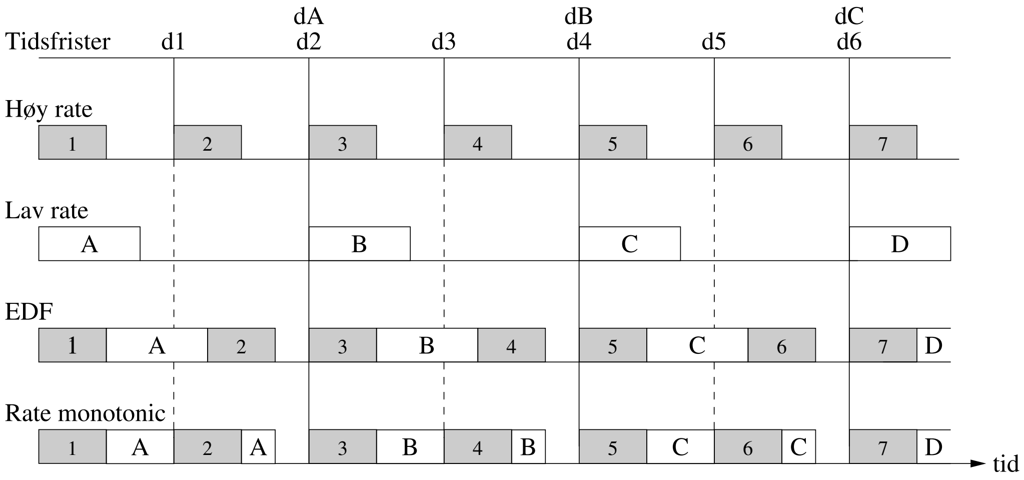
***Rate monotonic scheduling* (RM, ratemonoton)** gir hver prosess en prioritet ved oppstarten av prosessen. En prosess med kort periodetid, det vil si tiden fra en prosess starter til neste prosess av samme typen starter (fra prosess A starter i figur 4.3 til prosess B starter), får høy prioritet, mens en prosess med lang periodetid, får lav prioritet. Prosessene eksekveres så etter prioritet.

EDF er en optimal, dynamisk algoritme. En dynamisk algoritme scheduler hver instanse av innkommende oppgaver etter deres behov, noe som kan føre til at en periodisk arbeidsoppgave kan reschedules. Den er optimal ved at den kan schedule alle mulige prioritetstildelinger.

RM er en optimal, statisk, prioritesdrevet algoritme for avbrytbare, periodiske jobber. Optimal betyr her at ingen andre statiske algoritmer kan schedule arbeidsoppgaver som ikke RM kan.

EDF kan få en prosessorbruk på 100 prosent, det vil si 100 prosent minus tid til avbruddshåndtering, "context switching"[3] og så videre. RM har en verst mulig øvre begrensning på 69 prosent [Steinmetz 95]. Figur 4.4 viser hvordan disse algoritmene fungerer. Figuren viser også at ved to eller flere parallelle datastrømmer, vil RM få flere context switch'er en EDF.

[3]En "context switch" har vi når vi bytter kjørende prosess i prosessoren. Vi må da bytte ut dataene til den kjørende prosessen med den nye prosessens data i registere etc.




Figur 4.4: Av de avbrytbare scheduling-algoritmene gir RM flere "context-switch'er" enn EDF.

Ved siden av EDF og RM har andre scheduling-algoritmer for sanntidseksekvering vært utprøvet. Den mest lovende er [Steinmetz 95]:

***Least laxity first* (LLF, minst laxity først)** som eksekverer den prosessen som har minst laxity. Laxity er den tiden mellom tiden nå og tidsfristen minus gjenstående eksekveringstid.

Denne algoritmen er dynamisk og optimal for eksklusive prosesser. Det negative er at den hele tiden må regne ut "laxity'en"[4], og vi får stadig nye "context switch'er", noe som gir mye mer overhead enn EDF. I fremtidige MMSer med multiple prosessorer, kan LLF vise seg bedre.

Foreløpig har ingen andre algoritmer vist seg så passende for multimedia som EDF og RM. På grunn av at RM er en statisk algoritme og sjelden omgjør prioritetene, gir den liten overhead ved å bestemme neste jobbs prioritet. RM viser seg derfor særlig god til kontinuerlige multimediadata siden den har en optimal bruk av periodíteten [Steinmetz 95].

### 4.1.3 Kommunikasjon

Prosesser må ofte kommunisere med hverandre. Det kan være at en prosess' resultater brukes videre i en annen, eller at en bruker ber om en bestemt fil av filprosessen som igjen ber diskprosessen om å lese en bestemt blokk på disken. Vi vil her kort oppsummere to mulige kommunikasjonsformer mellom prosesser på en lokal maskin, mens prosesskommunikasjon mellom prosesser i et distribuert system ble kort beskrevet i avsnitt 2.1.5.

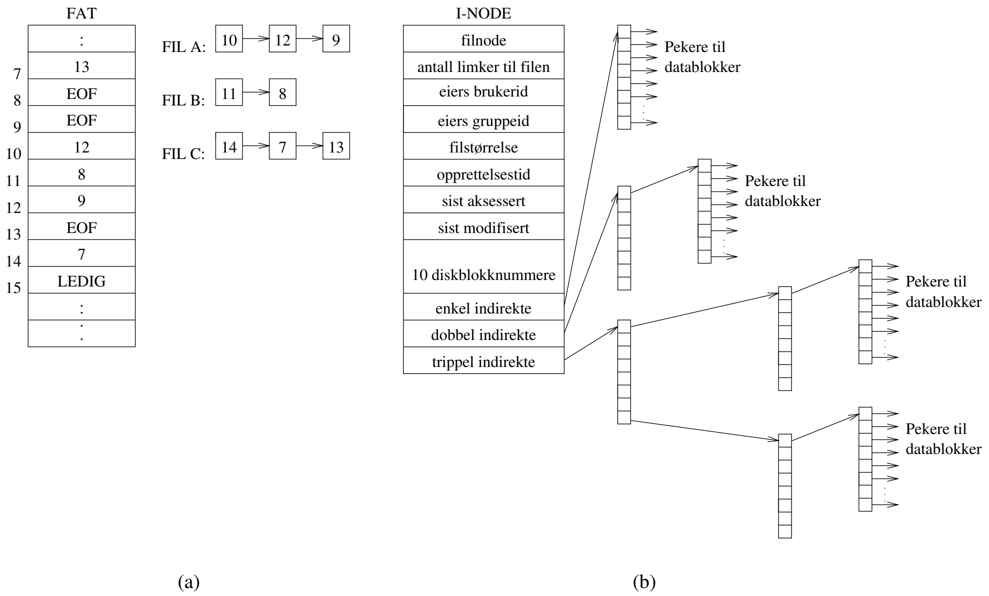
En mulig kommunikasjonsform er å la prosessene ha *delt minne* hvor alle prosessene kan lese og skrive til felles variable. En prosess som utfører en beregning som igjen skal brukes i en annen prosess, skriver resultatet inn i minnet hvor så den andre prosessen kan lese dette senere.

En annen kommunikasjonsform er å *sende meldinger*. I motsetning til bruk av delt minne hvor alle prosesser kan lese dataene, kan meldinger brukes til å gi informasjon direkte fra en prosess til en annen.

### 4.1.4 Filsystemer

Alle typer DHSer inneholder et filsystems basiske funksjoner, og disse tilsvarer det laveste laget i figur 4.1. Filsystemet er den delen av OSet som tilbyr tjenester for lagring og uthenting av data fra

[4]"Laxity" er den tiden vi har til overs når vi regner ut tiden det er igjen til tidsfristen går ut minus gjenstående eksekveringstid [(tidsfrist-nåtid)-gjenstående eksekveringstid].




Figur 4.5: (a) viser en kjedet liste av blokker som i MS-DOS. (b) viser en I-node brukt i UNIX.

disken. Det som er interessant fra multimedias side er hvordan systemet representerer informasjon i filer og hvordan disse blir aksessert på sekundære lagringsenheter. Filsystemene i multimedia-OSer må kunne håndtere alle datatypene som inngår i et MMS (se avsnitt 2.4), og dette setter nye krav til filsystemet som sanntidslesing og -skriving.

Tradisjonelt er målet med et filsystem å gi brukeren et komfortabelt grensesnitt for filaksesser og å utnytte lagringskapasiteten til disken best mulig. Målet med filorganisering i multimedia-OSer er å tilby en konstant, tidsavhengig uthenting av data. Designet av filsystemet avhenger derfor av bruken av systemet.

I tradisjonelle OSer [Tanenbaum 87], som MS-DOS og UNIX, plasseres data ofte der det er plass, og OSet må holde orden på hvilke blokker som tilhører hvilke filer. MS-DOS bruker en form for kjedet liste for hver fil, hvor hver blokk har en peker til neste blokk i filen. En fil-allokeringstabell (FAT) for hver disk holder orden på disse listene. Figur 4.5(a) viser et eksempel på en FAT i MS-DOS. Her begynner fil B i blokk 11. Denne blokken inneholder en peker til neste blokk som er 8 som igjen er merket "end of file" (EOF). En annen metode er å lagre blokkinformasjonen i tabeller. Et eksempel er UNIX' i-node som er vist i figur 4.5(b). Dette er en tabell med informasjon om filen og pekere til datablokker brukt av filen. I små filer lagres all informasjonen i i-noden selv, ellers må det i tillegg brukes indirekte pekere.

Filorganiseringen beskrevet over passer for "tradisjonelle"systemer hvor mye opprettelse, modifikasjon og sletting av filer forekommer. Fra multimedias side er dataene ofte av typen "*write-once-read-many*" (WORM, skriv en gang, les mange) med krav til sanntidseksekvering. Siden ytelsen til lagringsenheter bare er blitt marginalt forbedret sammenlignet med prosessorer og nettverk, har vi etter som forskjellen har blitt større forsøkt å finne nye lagringsmetoder.

Steinmetz [Steinmetz 95] nevner to måter å tilby lagring av kontinuerlige datatyper i filsystemer. Vi kan enten beholde organiseringer av filer på disk slik den er nå, hvor jitter unngås ved å ha en stor bufferkapasitet, og sanntidsstøtte oppnås ved å legge til spesielle algoritmer for disk-scheduling, eller vi kan optimalisere organisasjonen av audio- og videofiler. I den siste metoden økes gjennomstrømmingen og I/O-båndbredden maksimeres ved "striping"[5], og diskens transmisjonstid av dataene minimeres ved "clustering"[6] og sortering av dataelementene.

[5] "Striping" vil si å fordele dataene på flere disker slik at man kan lese dataene i parallell for eksempel fra et disk-array.

[6] "Clustering" vil (på det fysiske nivået) si å samle alle data som hører sammen etter hverandre på disken slik at man slipper å hoppe rundt på disken for å finne alle segmentene.

Multimediaobjekters størrelse, oppbyggning og sekvensielle uthenting favoriserer den siste av Steinmetz' lagringsmetoder ved å optimere diskens layout og ved å bruke flere disker. Ved å gruppere filer som sannsynligvis hentes ut samtidig sammen på diskene, minskes diskenes transmisjonstid av dataene ved at søketiden blir mindre på grunn av færre forflyttninger av lese/skrivehodet. Videre kan vi ved bruk av flere disker ha flere hoder som kan lese i parallell, og transmisjonstiden minskes videre ved at dataenes overføringstid reduseres. Med en slik dataorganisering og disklayout, vil bufferkravene og den totale datatransmisjonstiden minske.

Selv med optimert dataorganisering og disklayout, vil ikke tradisjonell disk-scheduling passe i MMSer på grunn av sanntidskravene. Det overordnede målet for disk-scheduling i MMSer er å imøtekomme så mange tidsavhengige prosesser som mulig før deres tidsfrist går ut, og sterkt relatert er også kravet om å holde bufferkravene lave. I tillegg må aperiodiske forespørsler kunne tjenes uten å utsette deres tjenester i for lange tider. Disse kravene til en disk-scheduling-algoritme sier at vi må finne en balanse mellom effektivitet og tidsavhengigheter. Det eksisterer flere algoritmer for disk-scheduling, men disse presenteres ikke her, men er for eksempel omtalt i [Steinmetz et al. 95].

### 4.1.5 Tjenestekvalitetparametere i operativsystemer

Hver komponent i et MMS må, som det kommer frem i avsnitt 1.1, oppfylle QoS-kravene til applikasjonen og datastrømmen, og disse kravene må mappes inn i systemkapasiteten av ressurshåndtereren. Steinmetz [Steinmetz 95] har klassifisert transmisjon og prosesseringskravene til multimediaapplikasjoner ved fire QoS-parametere:

- *Gjennomstrømming* bestemmes av dataraten en forbindelse trenger for å tilby applikasjonens QoS-krav og størrelsen på dataenhetene.
- *Forsinkelse* (lokal og global). Den lokale forsinkelsen er tiden det tar for en ressurs å gjøre ferdig en bestemt arbeidsoppgave, mens global (ende-til-ende) forsinkelse er den totale forsinkelsen til en dataenhet å komme fra kilde til mottaker.
- *Jitter* er den maksimale, lovlige tidsdifferansen vi kan ha ved ankomst av data.
- *Pålitelighet* som angår mekanismer for feiloppdagelse og korreksjon. Feil kan ignoreres, indikeres eller rettes. I tillegg refererer pålitelighet til prosessorfeil forårsaket av forsinkelse i eksekveringen slik at et brudd på en tidsfrist forekommer.

Parameterene i endesystemene [Vogel et al. 95] kan ha stor innvirkning på den QoS en bruker mottar. En svart/hvit skjerm kan som nevnt ikke vise farger og hastigheten på prosessoren og bussen kan begrense rammeraten i en videopresentasjon. Slike parametere faller også inn under OSets parametere.

QoS i et OS er ofte forbundet med sanntidstjenester, og viktige parametere er her ytelse, scheduling og ressursallokering. Standard OSer, som for eksempel standard UNIX, tilfredsstiller ikke disse kravene. Den vanligste måten å løse dette på er å utvide det eksisterende systemet. Enkelte versoner av UNIX har for eksempel utvidet med threads som har sanntidsprioritet, noe som må sies å være et skritt i riktig retning for sanntidsstøtte. En annen mulighet er å benytte seg av mikrokjernearkitekturer slik som CHORUS. CHORUS har blant annet støtte for sanntids-scheduling.

Vi kan identifisere OS-relaterte QoS-parametere på forskjellige abstraksjonsnivåer. Lavnivåparametere inkluderer ytelse, scheduling og størrelse på internt- og virtuelt minne. Høynivå-parametere inkluderer gjennomstrømming og forsinkelse. Høynivå-parametere gir en bedre basis for QoS-forhandling [Vogel et al. 95].

## 4.2 Databasesystemer

Et DBS er et software-system for å håndtere data på en effektiv måte. Det ligger som regel på toppen av et OS og bruker en del av OSets funksjoner for blant annet håndtering av hardware. Et DBS er det mest avanserte og komplekse DHSet, og det består av en database (DB) og et DBMS.

### 4.2.1 Databaser

> *En DB er en samling av relaterte data hvor data er kjente fakta som kan lagres og har en mening* [Elmasri et al. 94].

Dataene skal på en måte gjenspeile virkligheten, gi et bilde av den virkelige verden. Et eksempel er navn, telefonnummere og adresser til personer du kjenner. Dette er en samling av meningsfulle, relaterte data, og datasamlingen kan derfor kalles en DB. I tillegg må dataene være til interesse for noen og ha en logisk sammenheng med hverandre. Noen må komme å hente dem fram igjen og bruke dem ved en senere anledning. Vi kan dermed si at en DB er en samling av data som representerer et aspekt av den virkelige verden og er til nytte og interesse for noen.

### 4.2.2 Databasehåndteringssystemer

Et DBMS er brukt til å håndtere en DB. Elmasri og Navathe [Elmasri et al. 94] har følgende definisjon:

> *et DBMS er en samling av programmer som gir brukeren muligheten til å opprette og vedlikeholde en DB.*

DBMSet er altså et software-system for å definere, konstruere og manipulere DBer for forskjellige applikasjoner. Det å **definere** en DB vil si å spesifisere datatyper, strukturer og restriksjoner[7] for dataene som skal lagres i DBen. **Konstruksjonen** av en DB involverer lagringen av data på et sekundært lagringsmedium kontrollert av DBMSet, mens **manipulajon** inkluderer funksjoner for spørsmålsprosessering mot DBen for å hente og forandre data.

Figur 4.1 viser en modell av et DBMS hvor alle lagene inngår i DBMSet. Figuren skal ikke vise et spesielt DBMS, men illustrere typiske DBMS-moduler. Denne modellen er sammensatt av seks lag [Goebel et al. 96a]:

**Modellmapping:** DBSer må ha språk for å definere (et *data-definisjonsspråk* (data definition language, DDL)) og manipulere (et *data-manipulasjonsspråk* (data manipulation language, DML)) applikasjonsdataene og for å kontrollere informasjonen (et språk for å spesifisere integritetsrestriksjoner). Datamodellen omfatter alle disse tre språkene. Ved siden av dette må alle de synlige datatypene i brukergrensesnittet mappes inn i interne objektrepresentasjoner.

**Logiske datastrukturer:** Typiske mekanismer i dette laget er *spørsmålsprossesering, optimering, aksesskontroll* og *logiske aksesstier* for å prosessere deklarative brukerspørsmål. Informasjonen aksesseres ved hjelp av DML-instruksjoner basert på verdier eller identiteter[8]. Sikkerhet er nødvendig for at applikasjonene skal kunne dele data blant brukere med forskjellige behov og privilegier.

**Strukturer for logiske aksesstier:** Her realiseres postorienterte strukturer og operasjoner, samt transaksjonshåndteringsmekanismer. I tillegg er det støtte for opprettholdelse av temporale relasjoner og sorteringsmekanismer.

[7] Restriksjoner (integritetsregler, -skranker) forteller noe om hvordan data er relaterte, hvilke verdier de kan ha etc.

[8] Den delen av et DML som involverer henting av informasjon kalles ofte *spørrespråk*. Eksempler på dette er SQL [Elmasri et al. 94], Objekt SQL [Annevelink et al. 95] og OQL[C++] [Blakeley 95].

**Lagringsstrukturer:** Dette laget støtter posthåndtering, adresseringsteknikker, fysiske aksesstier, "clustering"-mekanismer, låsing, logging og gjennoprettelse av systemet ved systemfeil. Her blir blokk- og sidestrukturer mappet inn i mer komplekse objektstrukturer og de tilsvarende operasjonene. Informasjon om sider og segmenter er skjult for den logiske aksesstien i det øvre grensesnittet.

**Strukturer for sidetildeling:** Denne komponenten støtter mekanismer for sideutbytting og dens optimering. Valget av en passende sideutbyttingsstrategi har stor innflytelse på gjennvinnings- og loggingsalgoritmene. Bufferhåndtering er en viktig komponent blant annet fordi vi har lokalitet i referansestrukturen til objektene (se kapittel 5 om bufferhåndtering).

**Strukturer for tildeling av lagringsplass:** Dette laget må holde kontakten med OSet. Typiske operasjoner er I/O av multiple blokker i en operasjon, en fleksibel filhåndtering og fysisk "clustering". Det dannes et enkelt grensesnitt til aksess av blokker på disken.

DBMSer tilbyr altså konsepter for pålitelig, langtidshåndtering av integrert data som oppfyller kravene spesifisert av applikasjonsdomenene og prosesseringsmiljøene.

### 4.2.3 Databasesystemer egnet for multimedia

Vanlige relasjonale DBSer kan være veldig ineffektive ved håndtering av multimediainformasjon. Disse DBSene er designet for å håndtere små, enkle datastrukturer og ikke store, høyt komplekse datastrukturer som omtalt i avsnitt 2.4. Multimediadata skiller seg fra andre "tradisjonelle" data ved sin kompleksitet, størrelse og kontinuitet, og dette gir nye krav til DBSene (avsnitt 2.4).

For å møte alle datamodelleringskrav til multimediaapplikasjoner, er **objektorienterte-DBS** (OODBS) (se for eksempel [Elmasri et al. 94], [Korth et al. 91] eller [Kim 95a]) en god basis [Christodoulakis et al. 95]. Den objektorienterte paradigmen er basert på innkapsling av data og kode til en enkelt enhet kalt et *objekt*. Dette er en av de viktigste egenskapene til en OODBS. Man kan lage komplekse objekter hvor objektet kan være konstruert av en mengde andre objekter slik at det blir stort av den grunn. Slike objekter kaller vi *strukturerte komplekse objekter*. Vi har også en annen type kalt *ustrukturerte komplekse objekter*. Dette er objekter som trenger store mengder av lagringsplass, for eksempel datatyper for å representere bilder og lyd. Videre legges alle operasjonene til disse objektene inn i objektet selv slik at brukerne ikke aksesserer dataene direkte, men igjennom operasjonene. Som ved vanlig programmering i objektorienterte språk, vil vi finne data som naturlig hører sammen på samme sted. Både variablene og funksjonene legges i objektene. Det blir dermed lettere å programmere, og vi finner lettere frem i koden. All fornyelse og forandring av et objekt vil foregå på et sted.

*Arv* er også et annet viktig konsept i objektorienterte systemer. Vi får større mulighet til gjenbruk av gamle definisjoner og vi kan unngå å lage flere nesten like strukturer. Vi lar da heller de nye strukturene arve egenskaper fra superklassene. Noen OODBS har også multigeten for å ha *multippel arv* og *selektiv arv* . Dette vil henholdsvis si at subklassen arver fra flere superklasser og at man lar en subklassene arve bare noen av egenskapene (velger ut hvilke de vil ha) til en superklasse.

**Sanntids-DBS** (se for eksempel [Goebel et al. 96a] eller [Özsoyoğlu et al. 95]) er et DBS som støtter enkelte av multimediaapplikasjoners QoS-aspekter. Sanntids-DBSer støtter responstid ved å ha tidsfrister. Ved transaksjoner med harde tidsfrister får vi garantert QoS, mens myke tidsfrister tilsvarer tvungen QoS. Tradisjonelle transaksjoner kan bare tilby best-mulig QoS. Gjennomstrømmingen måles i prosent av alle transaksjoner som fullfører før tidsfristen.

Mekanismer som støtter tidsfrister og konsistenskrav er transaksjonsprosessering, samtidighetskontroll, bufferhåndtering og disk-scheduling. Ved bruk av harde tidsfrister må den høyest mulige prosesseringstiden til en transaksjon estimeres, og alle disse mekanismene må være kjent på forhånd. For å prosessere transaksjoner med myke tidsfrister trenger man prioriteter til konfliktløsing og prosessor-scheduling. Samtidighet oppnåes ved låsingsprotokoller og multiple kopier av data.

## 4.3 Filservere

Filservere (multimedialagringsservere) er brukt til å lagre multimediadata som kan hentes over et nettverk. Hovedsaklig er dette komponenter i et distribuert miljø basert på klient-server interaksjon. Tanenbaum [Tanenbaum 95] karakteriserer en filserver som en prosess som eksekveres på en maskin og hjelper til med å implementere filtjenestene i et distribuert system. Filtjenestene er igjen en definisjon av hvilke tjenester filsystemet kan tilby klientene. Vanlige, enkle filservere tilbyr ofte de samme filtjenestene som et filsystem, mens for eksempel en videofilserver i tillegg kan tilby video-lignende tjenester. I figur 4.1 vil en filserver dekke de to til tre nederste lagene.

Mange nye designspørsmål må stilles ved bruk av filservere framfor filsystemer i et nettverk [Goebel et al. 96a]. For å bedre ytelse er caching og forhåndshentingsstrategier ofte lagt til, og i tillegg må filserverene håndtere problemet med autentifikasjon. På grunn av den enorme mengden med data i MoD-systemer og deres ytelseskrav, brukes ofte diskarrayer som sekundære lagringsenheter. Designet av diskarrayer inkluderer behandling av samtidige datastrømmer, en fornuftig bufferhåndteringsmekanisme for en jevn avspilling av datastrømmene og at tjenestene er pålitelige. Videre kan forskjellen mellom videoobjektstørrelser og diskkapasitet gi dårligere utnyttelse av lagringskapasiten og båndbredden til disken. De fleste disker trenger jevnlig å utføre en “thermal calibration”[9]. Under en slik prosess vil ikke disken tilby I/O-kommandoer, så arbeidsmengden bør fordeles nesten likt på alle diskene slik at ikke alle tjenester blir avbrutt og utsatt for jitter hvis en disk utfører en “thermal calibration”.

## 4.4 Tjenestekvalitetsparametere i datahåndteringssystemer

DHSer støtter følgende QoS-parametere [Goebel et al. 96a]: responstid, gjennomstrømming, integritet, konsistens, sikkerhet, pålitelighet og tilgjengelighet. Disse QoS-parameterene er hentet fra [ISO/IEC 95], [Date 95] og [Gray et al. 93], og her presenteres disse litt nærmere:

- *Responstid* er den tiden det tar fra en bruker sender forespørselen om en tjeneste til systemet til systemet gir respons tilbake. Denne QoS-parameteren har to komponenter. Disse er *ventetid* som for eksempel er tiden systemet må bruke for å vente på ledige ressurser og *arbeidstid* som er tiden det tar å eksekvere forespørselen. De fleste DHSer gir best mulig responstid, mens sanntidssystemer garanterer en maksimum responstid.

- *Gjennomstrømming* er raten av operasjoner utført i et git tidsintervall. I DHSer vil det si antall transaksjoner og i filsystemer vil det si antall I/O-operasjoner, mens i et MoD-system måles gjennomstrømmingen ofte i det største antall samtidige klienter eller datastrømmer.

Det er flere relevante mekanismer for responstid og gjennomstrømming. For det første er det spørsmålsprosessering og optimalisering. Videre har vi transaksjonshåndtering med låsing og loggemekanismer, og tilslutt har vi lagringshåndtering inkludert aksesstier, dataplassering og bufferhåndtering. Disse mekansimene velges og initieres ofte på forhånd, men noen utvidede DBMS tilbyr likevel muligheten for å forandre mekanismer som dataplassering underveis, men før initaliseringen av transaksjonen.

[9] For å flytte lesehodet på disken benyttes en tabell for å angi hvor mye strøm som skal brukes for å få riktig posisjon. Ved tempraturforandringer i disken, må verdiene i denne tabellen justeres, og denne justeringen kalles “thermal calibration”. En slik kalibrering tar typisk et halvt sekund, og i denne perioden kan ikke disken betjene forespørsler.

- *Konsistens* passer på at en data oppbevares og presenteres logisk korrekt uten mangler og motsetninger i et single-brukersystem.

- *Integritet* omfatter de pålitelige operasjonene til et DBS i et flerbrukersystem og tar seg av unikheten til identifikatorer, datarekkefølgen og forebyggelse av tap av data. Integritet refererer til riktigheten og gyldigheten til dataene.

Tradisjonelle DBMSer garanterer konsistens og integritet ved ACID-egenskapene[10], og DHSer uten transaksjonshåndtering kan bare tilby best-mulig konsistens. Transaksjonshåndtering er en sentral mekanisme som forsikrer konsistens og integritet. Tradisjonelt initieres disse på forhånd, men som for responstid og gjennomstrømming, kan enkelte utvidbare DBMSer forandre og integrere multiple transaksjonshåndteringsmekanismer som kan velges under kjøringen [Goebel et al. 96a].

- *Sikkerhet* innebærer alle tekniske, fysiske og organisatoriske mål for beskyttelse av data og programmer. Sikkerhet refererer til beskyttelse av data mot uautoriserte forandringer og ødeleggelse.

Sikkerhet er relevant i mekanismer som autentisering (identifikasjon), autorisasjon, aksessmatriser og kryptering.

- *Pålitelighet* er den gjennomsnittlige tiden mellom feil til å opprettholde de definerte QoS-kravene (se også avsnitt 4.1.5).

- *Tilgjengelighet* er et mål for hvor tilgjengelig en tjeneste er. Det er den mengden av tid som tilfredsstillende tjenester er tilgjengelige. Tilgjengelighet er for eksempel om man kan finne en annen rute gjennom nettverket hvis en del av nettverket går ned, eller data som er lagret på en server som får en feil tilgjengelig et annet sted.

Pålitelighet og tilgjengelighet er relaterte til hverandre. I enkelte tilfeller er det tilstrekkelig å spesifisere bare en av disse.

Mange DHSer gir ikke klienten mulighet til å spesifisere sine QoS-krav og ingen forhandling finner sted. Årsaken til dette er at QoS-parametere som responstid og gjennomstrømming ofte er "best-mulig", og at det ofte bare er en bestemt verdi som garanteres for en QoS-parameter. Applikasjonskrav til QoS er tradisjonelt bestemt i design og implementasjonsfasen av aktivitetene i DHSene [Goebel et al. 96a].

## 4.5 Støtte for multimedia og tjenestekvalitet

Tidligere er det blitt fokusert på hvilken kvalitet som ytes av kommunikasjonsnettverk og OSer. Et MMS består av mange komponenter, og for å gi ende-til-ende QoS i et distribuert MMS, må alle funksjonelle enheter støtte samme QoS.

I området om enhetshåndtering har flere forskergrupper foreslått at multimediaenheter som typisk er plassert på en arbeidsstasjons I/O-buss, skal plasseres direkte på et høyhastighets ATM-nettverk [Campbell et al. 95]. Coulson og Blair [Coulson et al. 95] foreslår OS-støtte ved tre arkitekturelle prinsipper for kontinuerlige mediaapplikasjoner i et mikrokjernemiljø. Dette er:

**Oppkallingsdrevet applikasjonsstruktur.** Systemets infrastruktur og ikke applikasjonene er ansvarlige for å iverksette kommunikasjon. Dette gir velstrukturerte sanntidsapplikasjoner, mindre arbeidsmengde for applikasjonen med opprettelse av threads og bufferallokasjon og bedre ytelse ved færre "context switch'er".

[10]"Atomicity, Consistency, Isolation and Durability" (ACID, atomiske transaksjoner, bevaring av DBens konsistens, isolerte transaksjoner og varige endringer) [Date 95].

**Splittnivå systemstruktur.** Ytelsen til viktige systemfunksjoner er delt mellom kjerne og bruker ved at mange systemfunksjoner settes i samme adresserom som applikasjonen selv. Dette kan gi minimal kommunikasjonsoverhead, men gjør også at kjernen mister global oversikt over ressursbruk.

**Skille av kontroll og dataoverføring.** Overføring av kontroll utføres asynkront med dataoverføring for å separat kunne optimalisere begge. Dette kan gi flere fordeler, for eksempel økt samtidighet ved asyknron kommunikasjon i distribuerte systemer.

Disse prinsippene er utprøvet i CHORUS, og de fungerer godt sammen eller kan implementeres alene.

I dag er det to DHSer som gir støtte for enkelte QoS-aspekter selv om de ikke fokuserer på terminologien QoS [Goebel et al. 96a]. Dette er sanntids-DBSer (avsnitt 4.2.3) som støtter responstid ved å ha tidsfrister, og multimedialagringsservere (avsnitt 4.3) som er utviklet for applikasjoner som håndterer og lagrer kontinuerlige mediatyper som VoD eller "News-on-Demand" (NoD, nyheter på forespørsel). Gjennomstrømmingen måles ofte i antall samtidige klienter, mens lavnivå evaluering inkluderer kriterier som I/O-operasjoner per sekund. Multimedialagringsservere støtter generelt to aspekter av QoS [Goebel et al. 96b]:

- aktualitet eller kontinuitet og
- synkronisering mellom forskjellige media for presentasjon av kontinuerlige mediatyper uten feil/svikt (glitch) i brukerens sammenhengende bilde.

I tillegg er OODBSer (avsnitt 4.2.3) godt egnet til modellering av høyt strukturerte, komplekse multimediadata [Christodoulakis et al. 95].

# Kapittel 5

# Bufferhåndtering

Ytelsen til alle komponentene i et MMS påvirkes som tidligere nevnt av hverandre. I tidligere kapitler har vi derfor sett på generelle aspekter ved håndtering av multimediadata slik at vi har fått kjennskap til hvordan komponentene i et MMS virker sammen. I dette kapittelet tar vi for oss den delen av et DHS, det vil si håndteringen av minnet eller bufferhåndtering, som vi i denne oppgaven har lagt mest vekt på.

Et DBS bruker i likhet med OSer "langsome" disker som lagringsenheter for data. Disse er for trege til å kunne brukes direkte til manipulasjon av data, så eksekvering foregår i hovedminnet. Dette kapittelet handler om minnehåndtering og da spesielt bufferhåndtering i OSer og DBSer.

Minnehåndteringen i et system er tilknyttet den fysiske oppbygningen av maskinen. Hardware tilbyr ofte egne bits til håndtering av minneenhetene, og enkelte ting kan helt implementeres i hardware. Et OS er ofte realisert delvis i hardware og delvis i software, og et DBS er som regel implementert i bare software. Likevel kan det være vanskelig å skille mellom hardware og software, for i mange systemer er det slik at noe er implementert i hardware med noe software liggende over. For eksempel kan et virtuelt minnesystem (se avsnitt 5.3) implementeres i bare hardware eller software, men dette vil gi en dårlig ytelse. Vi vil derfor ikke skille spesielt mellom hva som er implementert hvor, men bare gi en oversikt over minnehåndteringen i en datamaskin.

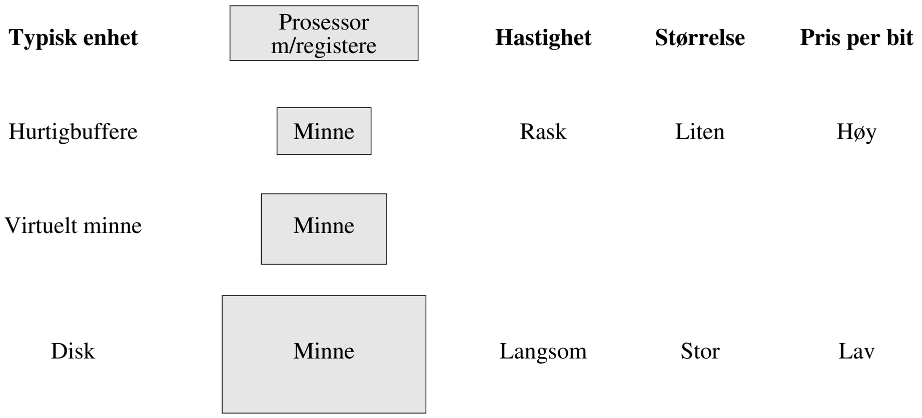
Et OS bruker stort sett det minnehierarkiet som presenteres under. Et DBS derimot ligger ofte over et OS og bruker en del av OSets mekanismer. Minnehåndteringen i et DBS fungerer i prinsippet på samme måte som OSets virtuelle minnesystem [Silberschatz et al. 97]. Likevel har DBSer ofte applikasjoner som refererer data på helt andre måter enn OSene. For å støtte DBSenes applikasjoner bedre og siden et DBS ofte bare er implementert i software, brukes mer kompliserte og komplekse mekanismer for håndtering av minnet. Et DBS har derfor ofte et eget buffer, ofte kalt et databasebuffer, som er lastet inn i minnet i brukeradresserommet.

I dette kapittelet presenteres bufferhåndteringen i OSer og DBSer sammen, og eventuelle forskjeller vil bli nærmere kommentert i i de avsnittene slike forskjeller er viktige. For å få en forståelse av hvordan data lagres i en datamaskin, gis det først en generell oversikt over minnehierarkier (avsnitt 5.1). Deretter gis det en introduksjon om hurtigbuffere ("cache memory") (avsnitt 5.2) og sideutbytting i hovedminnet ("paging"-systemer) (avsnitt 5.3). Videre beskrives forskjellige utbyttingsalgoritmer (avsnitt 5.4), samt av vi ser på bufferallokering (avsnitt 5.5). I tidligere kapitler kom det frem at multimediaapplikasjoner stiller høyere krav til systemet enn tradisjonelle applikasjoner. Vi tar for oss bufferhåndteringen i MMSer (avsnitt 5.6), og til slutt (avsnitt 5.7) ser vi på QoS-parametere i bufferhåndteringen.

Stoffet til avsnittene 5.1 til 5.3 er hovedsaklig hentet fra [Patterson et al. 94]. Videre er utbyttingsalgoritmene i avsnitt 5.4 og stoffet om bufferallokering i avsnitt 5.5 stort sett hentet fra [Effelsberg et al. 84]. Algoritmer fra andre kilder, stoffet om bufferhåndtering i MMSer og QoS-parameterene er spesielt merket med kildereferanser.

## 5.1 Minnehierarkier

For å få best mulig ytelse er det et ønske om at en datamaskin skal ha en ubegrenset mengde av *raskt* minne, men det raskeste minnet koster mer per bit og er derfor som regel mindre. For å gi en illusjon av ubegrenset raskt minne, er minnet bygd opp som et hierarki hvor vi har flere nivåer med minne med forskjellige hastigheter og størrelser. Hierarkiet er bygd opp slik at det raskeste minnet kommer nærmest prosessoren, og det blir billigere og langsommere når vi kommer til de lavere nivåene.




Figur 5.1: Typisk struktur av et minnehierarki.

Et typisk minnehierarki er vist i figur 5.1. Nederst i hierarkiet har vi disker (eller andre former for sekundære lagringsenheter som for eksempel tape). Dette er et varig og langsomt lagringsmedium hvor alle data er lagret. En disk er for langsom til å kunne brukes direkte til manipulasjon av data, så høyere opp i hierarkiet kommer raskere, men flyktige lagringsmedier. Videre blir dette hierarkiet brukt slik at dataene i de øverste nivåene, det vil si de nærmest prosessoren, er delmengder av nivåene under[1]. Dette vil si at de dataene du finner for eksempel i hovedminnet vil du finne også på disken.

Når prosessoren ber om data fra minnet, søkes det først etter i nivået nærmest, og hvis det ikke er der, letes det videre nedover til dataene er funnet. Når dataene så er funnet, kopieres disse inn i nivåene over.

Dataene blir kopiert fra nivå til nivå i små enheter, kalt *blokker*. Hvis vi finner de forespurte dataene i minnenivået vi søker i, kalles dette et *treff*. Hvis den blokken som er forespurt ikke finnes i dette nivået, får vi en såkalt *miss*. Ved en miss blir de lavere nivåer aksessert for å finne blokken med forespurt data. *Treffraten* er den delen av minneaksesser hvor den forespurte blokken ligger i nivået det søkes i, og *missraten* (1.0 - treffraten) er den delen hvor vi ikke finner blokken.

Hovedårsaken til å bygge et slikt minnehierarki er ytelse, og derfor spiller hastigheten til treffer og misser en stor rolle. *Trefftiden* er den tiden det tar å aksessere en blokk i det nivået vi søker i, noe som inkluderer den tiden det tar for å bestemme om det er et treff eller en miss. *Misstraffen* er tiden det tar å bytte ut en blokk i nivået vi er i med den forespurte blokken fra lavere nivåer (noe som tar lang tid), pluss den tiden det tar å levere denne blokken til prosessoren.

Når vi skal designe minnehierarkier, tar vi ofte hensyn til prinsippet om *lokalitet*. Ofte er det slik at vi bruker bare en liten del av adresserommet av gangen. En blokk som er nylig brukt eller er i nærheten av en nylig brukt blokk, har derfor høy sansynlighet for snart å bli brukt igjen. Vi har to typer av lokalitet:

[1]Nivået over kan inneholde nyere, oppdaterte versjoner av "samme" data avhengig av hvordan oppdateringer blir håndtert og skrevet tilbake til nivåene under. Dette kommenteres i avsnitt 5.1.2 senere i kapittelet.

- *Lokalitet i tid*. Når en blokk er aksessert, vil den sansynligvis bli aksessert snart igjen.
- *Lokalitet i rom*. Når en blokk blir aksessert, vil blokker som har adresse like ved den aksesserte blokken ha store muligheter for å snart bli forespurt.

Lokalitet er et viktig prinsipp, og minnehierarkiet bruker lokalitet i tid ved å ha nylig brukte blokker så nær prosessoren som mulig og lokalitet i rom ved å kopiere flere kontinuerlige blokker av data samtidig til høyere nivåer i minnehierarkiet.

### 5.1.1 Blokkplassering

En blokk tildeles ofte en plass i minnenivået etter bestemte plasserings- og utbyttingsstrategier basert på blokkens adresse. Av slike strategier eksisterer det to ytterpunkter. Den første er at en blokk kan plasseres på nøyaktig et sted. Denne plasseringsstrategien kalles *direkte mapping*. Posisjonen i bufferet er gitt ved ADRESSEN TIL BLOKKEN MODULO ANTALL BLOKKPLASSER I BUFFERET. Det andre ytterpunktet er *full assosiativ* plassering. Her kan en blokk plasseres hvor som helst i bufferet. Mellom disse ytterpunktene har vi mange muligheter for *sett assosiativ*[2] plassering som bruker prinsipper fra både direkte mapping og full assosiativ plassering. I en sett assosiativ plassering mappes en blokk inn i et sett, som igjen består av felter hvor en blokk kan plasseres. Innenfor hvert sett kan en blokk plasseres hvor som helst, men en blokk kan bare høre til et bestemt sett. Et sett assosiativ buffer med n lokasjoner for en blokk, kalles et n-veis sett assosiativt buffer. Settet til en blokk bestemmes ved ADRESSEN TIL BLOKKEN MODULO ANTALL SETT I BUFFERET. Siden en blokk kan plasseres hvor som helst innenfor et bestemt sett, må vi søke i alle lokasjonene i settet ved en forespørsel om en tilhørende blokk.

Når vi skal finne en blokk i bufferet, brukes igjen blokkens adresse til å finne riktig plassering. Et problem er at de lavere nivåene i hierarkiet er større, og det kan være flere mulige dataelementer som kan mappes til samme lokasjon i bufferet. Dette løses ved at vi bruker den øverste delen av en blokkadresse som *tag* i bufferet. Når vi så finner riktig lokasjon i bufferet, blir denne tagen sjekket mot adressen[3]. Hvis vi får en likhet, har vi fått en treff og dataene kan leses ut, og hvis ikke, får vi en miss, og dataene må hentes fra et nivå under. Videre vil det ved oppstart av maskinen ligge en tilfeldig samling av dataverdier i minnet. For ikke å feilaktig tro at vi har funnet riktig dataelement ved søk i bufferet, har vi et *gyldighetsmerke*, ofte et eget bit, som ikke blir satt før et "ordentlig" dataelement blir satt inn i dette feltet.

Lønnsomheten ved økt grad av assosiativitet avhenger av minnenivået. Fordelen med å øke assosiativiteten er som regel at det gir en lavere missrate, men jo større bufferet er jo mindre blir denne fordelen. Ulempen med økt assosiativitet er økte kostnader og dårligere aksesstid ved at vi må velge ut riktig element fra et sett, og derfor bruker for eksempel ikke hurtigbuffere full assosiativ plassering.

### 5.1.2 Utskifting av blokker

Når en miss oppstår i et n-veis assosiativt buffer, hvor $n > 1$, må vi bestemme hvilken av blokkene som skal byttes ut. I et fullt assosiativt buffer er alle kandidater for utskiftning, mens i et sett assosiativt buffer må vi velge en av blokkene i settet. Det eksisterer mange algoritmer for å velge ut hvilken blokk som skal byttes ut. Flere av disse presenteres i avsnitt 5.4 om utbyttingsalgoritmer.

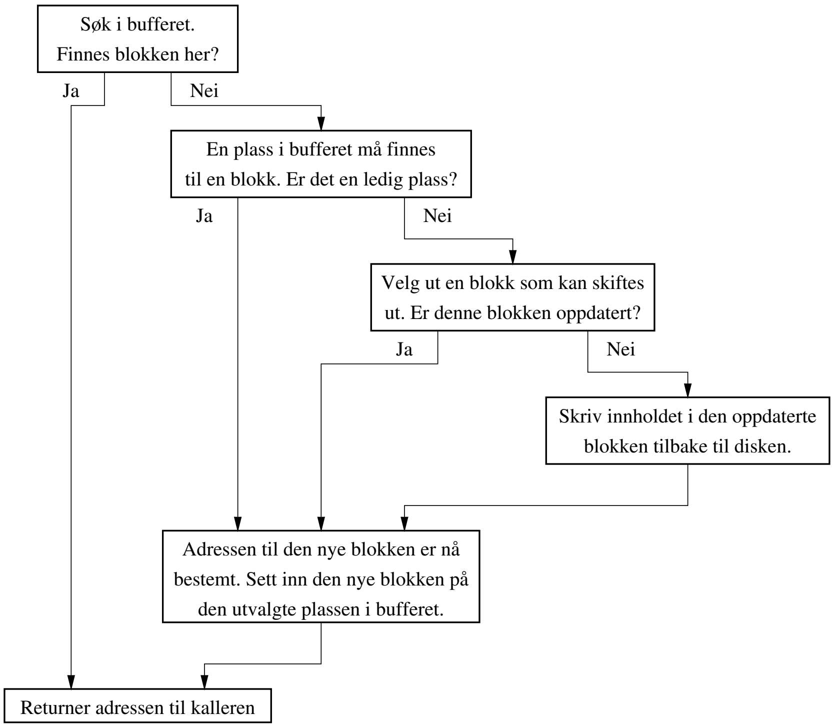
[2] Egentlig kan alle blokkplasseringsstrategier sees på som sett assosiativ, hvor et direkte mappet buffer blir en en-veis sett assosiativ og et full assosiativ buffer med x elementer blir et x-veis sett assosiativt buffer.

[3] På grunn av ytelse bør alle tagene i et sett assosiativt buffer søkes i parallell, for et serielt søk vil gi en for dårlig trefftid.

### Skriveproblemet

Når en blokk skal skiftes ut må vi ta hensyn til om blokken er oppdatert eller ikke. Blokker som bare er lest kan bare skrives over, mens oppdaterte blokker må håndteres på en annen måte. Dette kalles *skriveproblemet*, og vi har to hovedstrategier for håndtering av en oppdatert blokk. Den første er alltid å skrive de oppdaterte dataene tilbake til både blokken i dette nivået og de lavere nivåene, en strategi som kalles *write-through*. Dette er en strategi som alltid gir konsistens i minnehierarkiet, men har den overheaden at det tar lang tid hver gang det skrives til en blokk på grunn av at lavere og langsommere minnenivåer må aksesseres. Den andre strategien er *write-back*. Her blir ikke en oppdatering av en blokk skrevet direkte til nivåene under, men blir i stedet merket som oppdatert. Når en blokk skal skiftes ut i bufferet, sjekkes det om blokken er merket. Hvis ikke kan den skiftes ut med en gang, men er den oppdatert, må den skrives til nivået under først. Ved bruk av denne strategien, vil vi kunne ha en inkonsistens mellom nivåene i minnehierarkiet, og siden ikke alle minneenhetene er varige, kan vi risikere å miste data ved for eksempel strømavbrudd. Likevel kan det være en del tid å spare hvis en blokk som oppdateres flere ganger før den skiftes ut i bufferet, bare blir skrevet tilbake en gang.

I OSer brukes gjerne en av strategiene beskrevet over, men i et DBS er det ikke alltid like enkelt. Et DBS stiller i tillegg andre krav til blant annet konsistens og integritet ved ACID-egenskapene. Det er derfor ikke alltid at den ene eller den andre kan brukes, men en kombinasjon av disse. Videre vil vi se på hvordan et OS håndterer dette skriveproblemet i de forskjellige minnenivåene, og i avsnitt 5.3.7 kommer vi tilbake til punkter som samtidighet, gjenvinning av data og DBSers transaksjonshåndtering som også innvirker på valget av strategi for tilbakeskriving av data til disken.



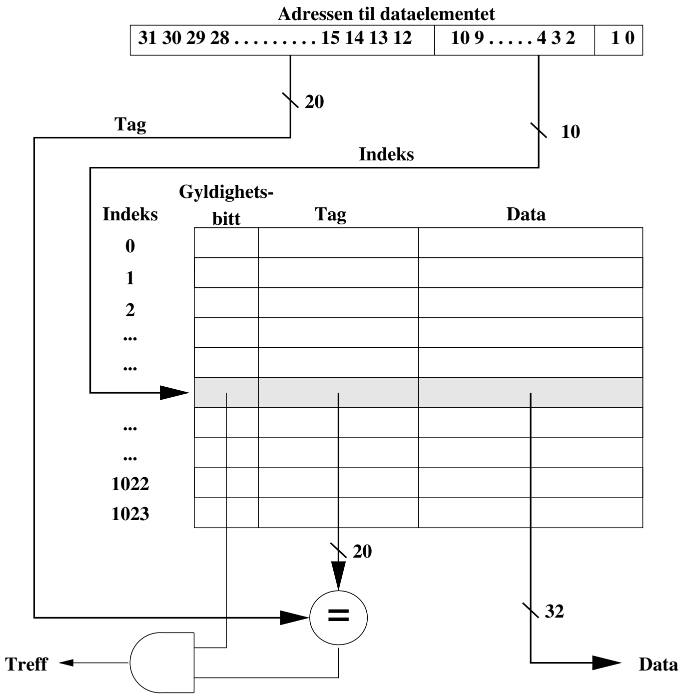

Figur 5.2: Buffermanagerens virkemåte.

### 5.1.3 Buffermanageren

Et buffer styres av en buffermanager [Gray et al. 93]. Ved en forespørsel om en blokk i bufferet, virker buffermanageren som vist i figur 5.2. Buffermanageren sjekker først om blokken finnes i bufferet. Gjør den det, returneres adressen til kalleren, og hvis ikke må blokken hentes fra disken. Da sjekkes det hvor den nye blokken skal plasseres. Er det noen ubrukte eller ugyldiggjorte plasser, brukes disse, ellers må en blokk i bufferet velges ut ved bruk av en utbyttingsalgoritme. Er den blokken som skal byttes ut modifisert, må den skrives tilbake til disken. Den nye blokken kan nå skrives inn i bufferet og adressen kan returneres til kalleren.

## 5.2 Hurtigbuffere

Under prosessoren har vi et hurtigbuffer, som er det hurtigste nivået i et minnehierarki. Det er mange måter å designe dette nivået, og her presenteres den enkleste mulige løsningen: En prosessors forespørsler er alle på et ord (på 4 Bytes), og blokkene i hurtigbufferet er også på et ord. Videre blir lokasjonen til blokken i hurtigbufferet basert på dens adresse i hovedminnet, og det er bare en plass i hurtigbufferet en blokk kan være. Dette er som nevnt i avsnitt 5.1.1 et direkte mappet hurtigbuffer.




Figur 5.3: Et direkte mappet hurtigbuffer.

Et eksempel på et slikt direkte mappet hurtigbuffer er gitt i figur 5.3. Dette er et hurtigbuffer på 4 KBytes, og hele hurtigbufferet med gyldighetsbit, tag og data er på tilsammen

$$2^{10} \times (1 + 20 + 32) = 53 \times 2^{10} = 53\text{Kbits}$$

($2^n\times$ (gyldighetfeltstørrelse + tagstørrelse + blokkstørrelse), hvor $2^n$ er hurtigbufferstørrelsen (antall indekser) i ord). Videre viser denne figuren hvordan adressen til et dataelement brukes til en hurtigbuffer-aksess som består av en tag og et dataelement på et ord. Hurtigbufferet har 1024 elementer og trenger derfor 10 bits til indeksen. De to nederste bit'ene brukes ikke her[4], og tagen er på de resterende 20 bit'ene. Hvis tagen og de 20 øverste bit'ene er like og gyldighetbit'et er satt, har vi et treff, og dataordet kan gis videre til prosessoren.

[4]Vi henter hele blokker på 4 Bytes (et ord). Disse bit'ene vil bli brukt hvis vi vil ha en enkelt byte ut av blokken.

Dette hurtigbufferet er det imidlertid mange muligheter for å forandre. Vi kan utnytte lokalitet i rom ved å ha større blokkstørrelser, og bruke en multiplekser til å velge mellom ordene i blokken. Vi kan utnytte lokalitet i tid ved å ha en grad av sett assosiativt plassering og bruke en utskiftningstrategi (se avsnitt 5.4 om utbyttingsalgoritmer) av blokker som tar hensyn til hvilke blokker som sansynligvis skal brukes. Andre design av hurtigbuffere kommenteres ikke her, men for mer avanserte og kompliserte hurtigbufferarkitekturer, se [Patterson et al. 94].

### 5.2.1 Skriveproblemet

Minnet er i dag implementert i såkalt med såkalt "random access memory" (RAM) hvor hovedminnet ofte bruker såkalt "dynamic RAM" (DRAM) og hurtigbuffere bruker "static RAM" (SRAM). Forskjellen i aksesstid er som vi ser i tabell 5.1 per idag ikke så stor mellom hurtigbufferet og hovedminnet, ofte bare et titalls klokkesykler, så flere varianter av **write-through** strategien er ofte brukt i hurtigbuffere. Likevel går fremskrittene med å lage raskere prosessorer fortere enn å lage raskere DRAM-baserte minnebrikker, så forskjellen i aksesstid vil etterhvert bli større. I fremtiden kan derfor en write-back strategi bli mer aktuell.

| Typisk minneteknologi | Typiske bruksområder | Typiske aksesstider |
|---|---|---|
| SRAM | hurtigbuffere | 8-35 ns |
| DRAM | hovedminne | 90-120 ns |
| Magnetisk disk | sekundærminnet | 10.000.000-20.000.000 ns |

Tabell 5.1: Typiske aksesstider (fra 1993) for forskjellige typer minneenheter [Patterson et al. 94].

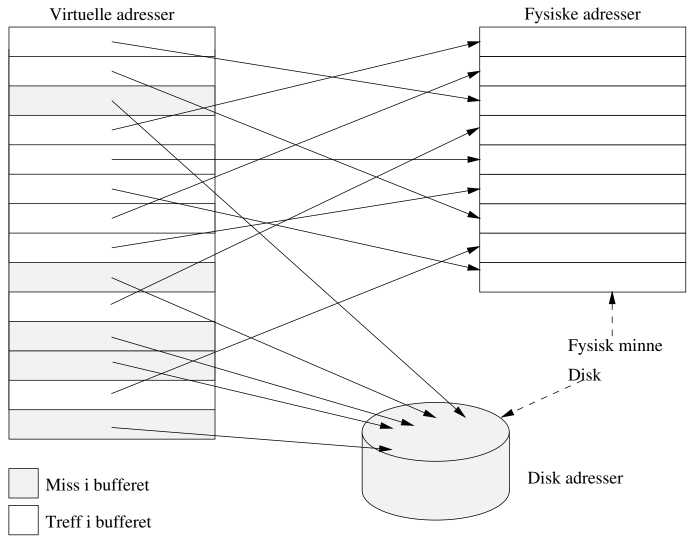
### 5.2.2 Bufferoppbygning

En annen viktig del av designet av et hurtigbuffer, er størrelsen vi skal ha på bufferet og blokkene i bufferet. Et 4 KBytes hurtigbuffer kan bygges opp på flere forskjellige måter som for eksempel et hurtigbuffer med 1024 blokker av et ord som i figur 5.3 eller som et hurtigbuffer med 256 blokker av fire ord. Som regel er det hva som gir best ytelse som styrer dette, og hva som passer best avhenger av hardware-arkitektur og applikasjonene som skal kjøres på maskinen.

Hvordan man velger å bygge opp et hurtigbuffer, må vurderes etter hva som gir best ytelse i hvert enkelt tilfelle. Har vi et hurtigbuffer med mange blokker, er det større sansynlighet for å få en treff, men det tar igjen lengre tid å finne frem riktig blokk. En måte å minske antall blokker er å gjøre hver enkelt blokk større. Da tar det kortere tid å finne en enkelt blokk, og vi får brukt fordelen med lokalitet i rom. Ulempen med store blokker er imidlertid at vi må ha mekanismer til å velge rett ord fra blokken (en multiplekser), og misstraffen blir mye høyere for hver miss siden blokken som skal skiftes ut er mye større. Ved design av et slik hurtigbuffer gjelder det å finne en slags gylden middelvei, slik at ytelsen blir best mulig når vi ser på alle parametere sammen.

## 5.3 Hovedminnet - sideutbytting

På samme måte som et hurtigbuffer gir oss en metode for rask aksess til nylig brukt kode og data i hovedminnet, kan hovedminne opptre som et "hurtigbuffer" for sekundære lagringsenheter (som en disk eller tape). Denne teknikken kalles *virtuelt minne*. Det virtuelle adresserommet kan være større en det faktiske fysiske minnet, slik at vi får en illusjon av å ha mer minne enn det vi egentlig har. Alle forespørsler går igjennom det virtuelle adresserommet, som igjen oversetter til den fysiske adressen i hovedminnet eller til adressen på disken. Figur 5.4 viser hvordan de virtuelle




Figur 5.4: I det virtuelle minnet blir sider oversatt fra virtuelle adresser til fysiske adresser.

adressene blir mappet til adressene i det fysiske minnet eller på disken. Både det virtuelle og det fysiske minnet deles i like sider slik at en virtuell side mappes direkte til en fysisk side. Hvis en side ikke finnes i det fysiske minnet, finnes disse dataene på disken[5]. Fysiske sider kan deles ved å la to forskjellige virtuelle sider peke til samme fysiske side.

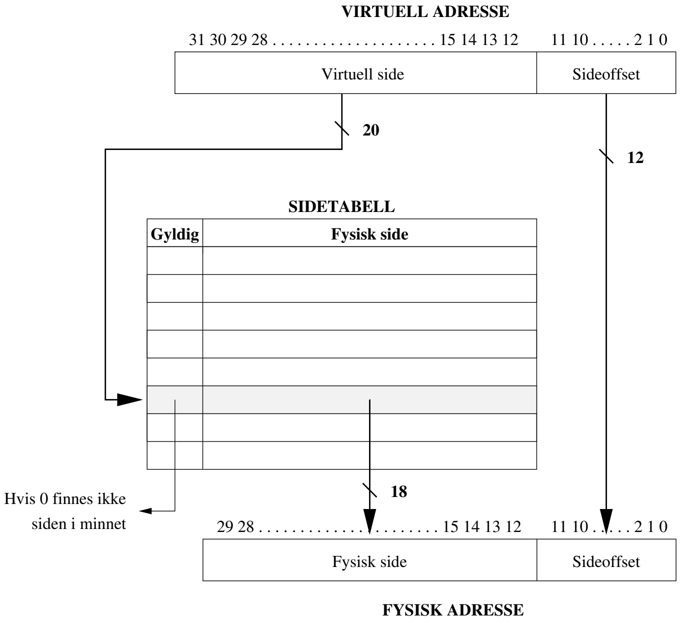
En virtuell blokk kalles en *side*, og en virtuell minnemiss kalles en *sidefeil*. Prosessoren produserer en virtuell adresse for sidene, som blir oversatt til en fysisk adresse, og det er denne adressen som bruker til aksess i hovedminne. En virtuell minneadresse er delt i et *virtuelt sidenummer* og et *sideoffset*. Ved en adresseoversettelse vil sideoffset'et være uforandret, mens det virtuelle sidenummeret oversettes til det fysiske sidenummeret. Antall bits i sideoffsetet bestemmer sidestørrelsen.

Hver forespørsel om en side i bufferet kalles en *logisk referanse*, og en aksess til en side på disken kalles en *fysisk referanse*. Sekvensen av logiske referanser til sider sortert på tiden, kalles en *logisk sidereferansestreng*, og den beskriver "referanseoppførselen" et system uavhengig av bufferstørrelser og sideutskiftningsalgoritmer.

Som tidligere nevnt og som det kommer frem av tabell 5.1, er en diskaksess veldig mye langsommere enn en aksess til hovedminnet, så en sidefeil er tidkrevende og kostbart. En sidefeil tar flere hundre tusen klokkesykler, og mange designvalg er derfor motivert av den høye misskostnaden. Dette har ledet til flere nøkkelvalg i designet av et virtuelt minnesystem:

- Sidene bør være store nok til å kompensere for stor aksesstid ved å utnytte lokalitet i rom. Sider fra 4 KBytes til 16 KBytes er typiske.
- Organisasjon av sidene bør være slik at vi får minimalt med sidefeil.
- Sidefeil kan håndteres i software, fordi overheaden blir så liten sammenlignet med en diskaksess.
- Siden sidefeil kan håndteres i software, kan mer avanserte algoritmer for sideutbytting brukes. Selv små reduksjoner i antall sidefeil vil få dette til å lønne seg i forhold til eksekveringskostnadene til slike algoritmer.

[5]Hvis siden finnes i det fysiske minnet, finnes det også som vi så tidligere i avsnitt 5.1 i lavere minnenivåer, det vil si disken.




Figur 5.5: Et eksempel på en sidetabell.

- Siden det å skrive til disken er så kostbart, kan man ikke bruke *write-through* i et OS. Vi må redusere antall diskaksesser.

### 5.3.1 Sideplassering og sidetabeller

På grunn av store kostnader ved sidefeil, er missrater avgjørende når vi skal bestemme hvordan sidene plasseres i bufferet. I et virtuelt minnesystem bruker vi alltid full assosiativ plassering. På denne måten kan OSet bytte ut hvilken som helst side ved bruk av en utvelgelsesalgoritme som gir minst mulig missrate.

Hvis en side kan plasseres hvor som helst, trenger vi en mekanisme for å finne dem igjen. Til dette bruker vi noe vi kaller en *sidetabell*. En sidetabell, som også er lagret i minnet, er indeksert med sidenummeret fra det virtuelle adresserommet og inneholder den fysiske adressen.

Et eksempel på en slik sidetabell er gitt i figur 5.5. Her er sidestørrelsen på $2^{12}$ Bytes = 4 KBytes. Det virtuelle adresserommet er på $2^{32}$ Bytes eller 4 GBytes, og det fysiske adresserommet er på $2^{30}$ Bytes eller 1 GBytes. Sidetabellen er indeksert med de 20 øverste bit'ene i den virtuelle adressen, noe som gir $2^{20}$ eller circa en million felter i sidetabellen. Feltet inneholder så de 18 øverste bit'ene i den fysiske adressen. De 12 nederste bit'ene er like og kopieres derfor bare over. Hvis ikke gyldighetsbit'et er satt, finnes ikke siden i minnet. I denne figuren ser man hvordan den virtuelle adressen kan bli oversatt til den fysiske adressen i minnet. Alle programmer har sin egen sidetabell, og dette kan ta mye plass i minnet. Med en 32 bits adresse med 4 KBytes sider og 4 Bytes per tabellfelt, blir den totale tabellstørrelsen

$$\frac{2^{32}}{2^{12}}\text{sider} \times 2^2 \frac{\text{Bytes}}{\text{side}} = 4\text{MBytes}$$

På denne måten kan alt minne gå til å lagre sidetabellene. En løsning på dette problemet er å ha dynamiske størrelser på tabellene, eller å tilføre en hashfunksjon på virtuelle adresserommet slik at strukturen på sidetabellen ikke behøver å være større enn det fysiske minnet.

### 5.3.2 Sidefeil

Som vi ser i figur 5.5, bruker vi et gyldighetsbit for å se om siden finnes i minnet. Er ikke dette bit'et satt, finnes ikke siden i minnet, og en sidefeil oppstår. Vi må nå finne den ønskede siden i det neste nivået i hierarkiet, som oftest er disken. Dette er en tidkrevende prosess, og vi må derfor være nøye med hvilke sider vi bytter ut. Hvis ikke alle sidene i sidetabellen er i bruk, bruker vi selvfølgelig en av de ledige, men er alle brukt, må vi velge en å bytte ut. På grunn av den høye misstraffen, må vi minimere antall sidefeil ved å skifte ut en side vi ikke tror vil bli brukt i nærmeste fremtid. (Se avsnitt 5.4 for forskjellige utskiftningsalgoritmer.)

Ved en sidefeil får vi et sidefeil-unntak, og OSet overtar kontrollen. Disse forekommer midt i eksekveringen av en instruksjon. Siden denne instruksjonen kan ikke gjøres ferdig før de rette dataene er funnet, må instruksjonstelleren settes tilbake, slik at den avbrutte instruksjonen kan eksekveres på nytt når de rette dataene er funnet.

### 5.3.3 Søk i bufferet

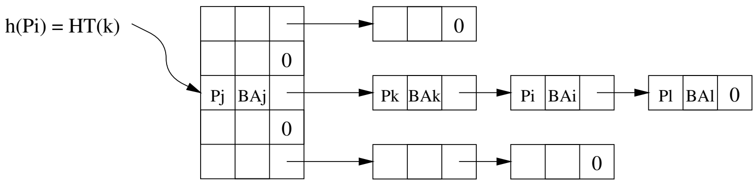
Hver gang en logisk referanse til en side forekommer, må bufferet søkes igjennom. Dette skjer ofte, og søkestrategien bør være effektiv.

Den enkleste formen for buffersøk er *direkte* søk, hvor man sjekker sidene i bufferet sekvensielt. Dette gir i et buffer med N sider gjennomsnittlig $\frac{N}{2}$ sider å søke igjennom ved en treff og N ved en miss. Hvis DBMS ligger i det virtuelle minnet i OSet, noe det som regel gjør, vil sidene ligge forskjellige steder i dette minnet. Dette kan gi mange sidefeil. Derfor brukes oftest en form for tabellsøk, noe som gir høyere lokalitet.

En tabell til et N-side stort buffer inneholder N felter. I gjennomsnitt får vi $\frac{N}{2}$ side å søke igjennom ved en treff. Ved å sortere tabellen og ha binære søk, vil vi kunne redusere søkslengden til $\frac{N}{2}$ også ved en miss, men dette vil gi større overhead ved oppdateringer. Enda bedre aksesstid, $\log_2$N for både treff og miss, kan oppnåes ved å indeksere tabellen eller organisere den som et balansert binært tre, men igjen er oppdateringskosten høyere.

Et tabell som er *kjedet* gir to fordeler over en kompakt tabell. Oppdateringer blir lettere siden ingen elementer må fjernes, og vi kan få representert informasjon om hvordan sider blir referert.

Et effektivt valg av organisering av sidetabellen og søking i denne, kan være å bruke en *hashteknikk* på sidenummerene. Dette minner om sett assosiativ plassering i avsnitt 5.1.1, men hvor hvert sett ikke må ha noen fast størrelse. Hashalgoritmen transformerer sidenummeret til en plassering i hashtabellen (sidetabellen). Figur 5.6 viser en slik hashtabell. Når vi skal søke etter side nummer i, Pi, bruker vi hashfunksjonen på sidenummeret og får ut plass i tabellen, HT(k). Vi vet nå at side Pi ligger i den lenkede listen vi finner her, og trenger bare å søke igjennom denne for å finne riktig side.




| | |
|---|---|
| Pi, Pj, Pk, Pl: | sidenummere |
| BAi, BAj, BAk, BAl: | bufferadresser (rammenummere) |
| h: | hashfunksjon |
| HT(k): | k'te plassen i hashtabellen |

Figur 5.6: Et eksempel på bruk av en hashtabell i et buffer.

### 5.3.4 Skriveproblemet

På grunn av den store forskjellen i aksesstid mellom minnet og disken, bruker vi i OSer alltid **write-back** strategien for sider som må skrives tilbake til disken. I tillegg til å ikke skrive tilbake sider før de skal skiftes ut, brukes også ofte en merking av oppdaterte sider ved å sette et oppdateringsbit ved oppdatering. Bare sider som er merket oppdatert blir skrevet tilbake. Umerkede sider blir bare skrevet over.

### 5.3.5 Bufferoppbygning

Når vi skal bygge opp et buffer i hovedminnet, er forskjellen mellom diskhastighet og bufferhastighet så stor at bufferet bør være så stort som mulig. Det som imidlertid kan diskuteres er størrelsen på hver enkelt side. Dette avhenger igjen av hva slags applikasjoner bufferet skal brukes på. Hvis det bare er små mengder data i hver side som skal brukes, er det lite lønnsomt med store sider. Da bruker vi tid til å hente data i sider som ikke brukes, og bufferplass okkuperes av unødvendige data. Bruker vi derimot alle dataene i siden, kan store sider lønne seg. Vi får da færre sidefeil, og henter mer data av gangen mens vi først er i gang.

Et DBS hadde i 1984 typisk sider på mellom 512 og 4 KBytes. Antall sider det er plass til i bufferet kan settes som en DBMS-parameter med verdier i dag mellom 16 KB til 12 MB (typisk mellom 128 KB og 256 KB) [Effelsberg et al. 84]. Nyere tids DBSer for blant annet multimedia har ofte større sidestørrelser. Reddy og Wyllie [Reddy et al. 94] mener at vanlige sidestørrelser på 4 KBytes er altfor små i et MMS med kontinuerlige datatyper. De antyder at fremtidige blokkstørrelser i filsystemer på 64 Kbytes eller mer, og dette indikerer også en tilsvarende sidestørrelse i bufferet.

### 5.3.6 Assosiativt minne

Sidetabellene er som tidligere nevnt store og er lagret i minnet. Dette betyr at vi må aksessere minnet to ganger for hver dataforespørsel: en for å finne den fysiske adressen i sidetabellen og en for å hente datasiden.

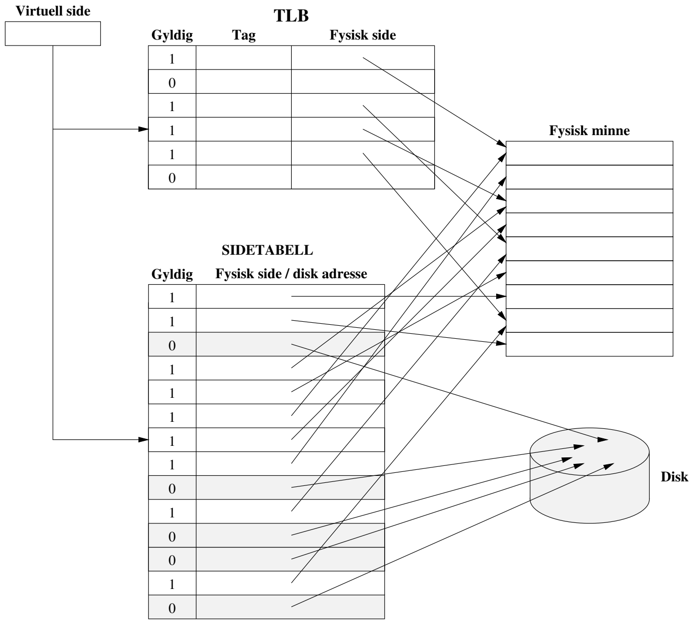
Dette kan igjen løses ved utnyttelse av lokalitet i tid på sidetabellen. Sider som nylig er blitt referert vil sansynligvis bli referert igjen, fordi dataene i en dataside har både lokalitet i tid og rom. Derfor har moderne maskiner et "hurtigbuffer" for nylig brukte oversettelser mellom virtuelle og fysiske adresser, kalt *assosiativt minne* eller "*translation-lookaside buffer*" (TLB).

Et eksempel på et slikt system er vist i figur 5.7 hvor en TLB i motsetning til en sidetabell bare inneholder mappinger til det fysiske minnet. Siden TBLen bare inneholder et subsett av alle mappinger for virtuelle-til-fysiske sider i sidetabellen, må vi ha med en tag for å sjekke om det er riktig side. I tillegg må vi ha et gyldighetsbit for å sjekke om vi eventuelt har en gyldig treff i TLBen. Sidetabellen mapper en virtuell adresse til en side i det fysiske minnet eller til en disk adresse. Er gyldighetsbit'et satt, leveres en adresse til en side i minnet, ellers leveres en adresse til dataene på disken. Siden sidetabellen har et felt for hver virtuelle adresse, trengs ikke noe tag.

Ved oppslag av virtuelle sider, sjekkes det først i TLBen. Får vi en treff kan siden hentes rett ut, men ved en miss må vi først se om det bare er en TLB-miss eller om det er en sidefeil. Dette betyr henholdsvis at siden finnes i det fysiske minnet eller at siden ikke finnes i det fysiske minnet, men på disken. Ved en TLB-miss eksisterer den forespurte oversettelsen mellom virtuell og fysisk adresse i sidetabellen og kan hentes derfra. Sidefeil håndteres som nevnt over.

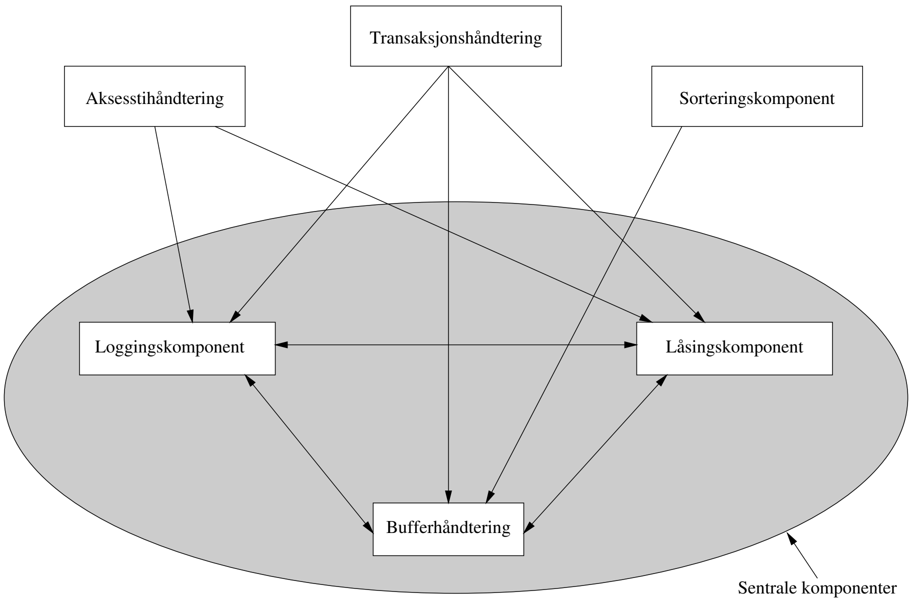
### 5.3.7 Kompliserende aspekter

Figur 5.8 viser de forskjellige avhengighetene vi har mellom sentrale - og høynivåkomponenter i et DBMS [Lockemann et al. 87]. De sentrale komponentene i denne figuren håndterer sine ressurser




Figur 5.7: TLBen fungerer som et hurtigbuffer for sidetabellen.

direkte i grensesnittet til OSet, mens høynivåkomponentene igjen bruker de sentrale komponentene. Dette gir igjen flere kriterier vi må ta i betraktning i designet av en bufferhåndteringsmekanisme. I underavsnittene under ser vi nærmere på avhengighetene mellom bufferhåndteringen og de andre komponentene i figuren samt datadistribusjon.




Figur 5.8: Avhengigheter mellom sentrale - og høynivåkomponenter i et DBMS.

#### 5.3.7.1 Låsingskomponenten

Data er ofte en delt ressurs, og flere brukere kan aksessere samme data samtidig. Det vil derfor være uheldig å bytte ut en side som er i bruk av andre. En løsning er å ha en teller med antall brukere for hver side, slik at siden ikke kan byttes ut før telleren er tilbake på null. Videre kan transaksjonene oppdatere data i en side, og hvis andre transaksjoner leser samme siden, kan vi risikere å få feil i systemet. Derfor kan vi bare la en manipulasjonstransaksjon bruke en side av gangen. Eventuelle andre transaksjoner kan aborteres (avhengig av samtidighetskontrollprotokollen) og startes på nytt etter at oppdateringen er ferdig. En måte å minske antall sidefeil i bufferet, er å la sidene til avbrutte transaksjoner bli liggende i bufferet, for disse sidene (hvis de ikke er blitt oppdaterte) vil vi trenge igjen når transaksjonen blir restartet.

Hovedoppgaven til låsingskomponenten er å låse sider i bufferet for å garantere et "logisk enbrukersystem". Dette krever at låsingskomponenten holder orden på hvilke sider som er låst og ikke slik at ikke flere transaksjoner oppdaterer og leser samme side samtidig. En manipulasjonstransaksjon garanteres derfor isolasjon gjennom låsing av sidene i bufferet.

#### 5.3.7.2 Loggingskomponenten

Når systemet går ned eller en transaksjon i et DBS aborterer, må dataene gjenvinnes slik at dataene blir holdt konsistente. Ved et systemkrasj vil oppdateringer av data i bufferet gå tapt hvis siden ikke blitt skrevet tilbake til en varig lagringsenhet (som en disk). Oppdateringer vil måtte omgjøres i de tilfeller hvor en prosess ikke fikk gjort seg ferdig og oppdaterte data likevel har blitt skrevet tilbake.

Til en eventuell gjenvinning av data i DBSer, har vi en logg hvor alle *operasjoner logges*. En metode som brukes ofte er "write ahead log" (WAL) hvor logginformasjon skrives tilbake til disk før de virkelige dataene (sidene) blir oppdaterte. I denne loggen gir vi alt et logg sekvensnummer (LSN) som angir hvilken versjon av en side eller post det refereres til. Loggen brukes til å gjennopprette DBen, og to typer operasjoner kan utføres på sidene som ligger på disken. Den første er å rulle tilbake transaksjoner (rollback) hvor poster med lik eller lavere LSN enn sidene må gjøres om igjen i motsatt rekkefølge for å omgjøre en transaksjons oppdateringer. Her har siden blitt skrevet tilbake til disken før transaksjonen har gjort seg ferdig som ved en transaksjonsabortering. Den andre operasjonen er når en transaksjon har avsluttet sitt arbeid, men oppdateringene har ikke blitt skrevet tilbake til disken før krasjet. Poster med høyere LSN enn sidene må gjøres om igjen for å repetere transaksjonenes oppdateringer.

Et alternativ til loggbaserte gjenvinningsteknikker er *skygget sidehåndtering* (shadow paging). Ideen bak denne teknikken er å ha to sidetabeller under levetiden til en transaksjon. Vi har den opprinnelige sidetabellen som brukes som vanlig og en ekstra sidetabell som ikke oppdateres og er en "skygge" av den andre. Hvis transaksjonen fullføres kastes den ekstra sidetabellen, men hvis transaksjonen aborterer kastes den opprinneige sidetabellen og kopien brukes videre.

Hvor mye det er å gjøre ved en datagjenvinning avhenger av når og hvordan vi skriver tilbake oppdaterte sider til disken. Hvis en side kan byttes ut og skrives tilbake til disken før transaksjonen som har oppdatert den er ferdig, vil vi ved en tilbakerulling måtte aksessere sider som ligger på disken. I motsatt fall vil disken ikke bli berørt ved en tilbakerulling, men bufferet må da være så stort at det kan holde alle oppdaterte sider i bufferet, og det vil medføre låsing av siden. Videre er ikke bufferet et permanent lagringsmedium, og for eksempel ved et strømavbrudd, vil alle data som ikke er skrevet tilbake til disken gå tapt.

Videre vil det være slik at hvis vi bruker et write-through strategi, vil ikke avsluttede transaksjoner gå tapt ved en krasj, for alle oppdaterte sider ligger på disken. I motsatt fall, ved bruk av en write-back strategi, vil ikke siden skrives tilbake til disken før den skal byttes ut, noe som gjør at alle oppdateringer på sider i bufferet vil måtte gjøres på nytt. Write-through vil gi en rask restart uten oppdateringer som må gjøres på nytt, men ytelsen blir dårligere. Transaksjonen vil heller ikke kunne avslutte før alle sidene er skrevet tilbake, noe som gir dårlig responstid.

For at DBSer skal kunne gjenopprettes etter et krasj, er det nødvendig å restriktere når sider i bufferet kan skrives tilbake til disken. En blokk som ikke tillates tilbakeskrevet sies å være "*pinned*", det vil si at siden er låst fast i bufferet[6] uten mulighet for å skrive dataene tilbake til disken. Mange OSer støtter ikke en slik mekanisme, men den kan være av avgjørende betydning i implementasjonen av DBS som er robuste mot krasjer. Videre er det situasjoner, som for eksempel etter at en transaksjon er fullført, hvor det er nødvendig å skrive tilbake data til disken. Dette er hovedsaklig på grunn av at bufferet ikke er et varig lagringsmdium og alle oppdaterte data i bufferet vil gå tapt ved en krasj.

En eventuell forbedring kan være å ha sjekkpunkter i loggen. Etter en viss tid, ved hvert sjekkpunkt, blir alle oppdateringstransaksjoner satt i kø og hele bufferet skrives tilbake. Ulempen med dette er igjen at hele systemet må vente ved hvert sjekkpunkt. Det er flere måter å forbedre sjekkpunktstrategien, men dette kommer vi ikke inn på her (se blant annet [Gray et al. 93]). En annen forbedring er å bruke ledig eksekveringstid til å skrive tilbake modifiserte sider. En tredje mulighet for forbedring er å optimere antall diskaksesser uten å miste data fra flyktig minne ved krasjer og strømavbrudd ved å bruke et såkalt varig RAM [Silberschatz et al. 97]. Dette er ofte implementert ved å bruke batterier som tar over strømforsyningen ved strømavbrudd slik at ikke dataene i dette minnet går tapt. Ideen er at når transaksjoner fullfører, skriver de resultatene til dette minnet i steden for disken, og disken aksesseres bare når det ikke er mer plass i dette minnet.

#### 5.3.7.3 Transaksjonshåndtering

Bufferhåndteringen er en sentral komponent med hensyn til systemets ytelse, og antall parallelle transaksjoner i et DBS har stor påvirkning på sideutbyttingen. Hvis en transaksjon øker sideutbyttingen mye, burde transaksjonshåndtereren utsette starten av denne transaksjonen til senere, og hvis mulig burde transaksjonshåndtereren av ytelsesgrunner gruppere transaksjonene på en "bufferorientert" måte slik at transaksjonene som bruker de samme buffersidene eksekveres samtidig [Lockemann et al. 87].

#### 5.3.7.4 Datadistribusjon

I et distribuert system kan alle maskinene ha egne disker med data distribuert på forskjellige måter. Hver slik maskin har også et eget lokalt buffer med kopier av nylig aksesserte datasider. Et problem er at når en slik side blir oppdatert i et system, blir alle andre kopier av samme side på andre maskiner i systemet *ugyldiggjort*. Ugyldiggjorte sider er ikke lenger brukbare.

Ytelsen til et system er sterkt avhengig av treffraten. I et enbrukersystem vil en større bufferstørrelse kunne gi høyere treffrate, men i et distribuert system vil dette også gi en økning i ugyldiggjøringsraten. Økningen i ugyldiggjøringsraten gjør at det er mindre å tjene på å gjøre bufferet større.

Et forsøk gjort av Dan, Dias og Yu [Dan et al. 90] på en DB-buffermodel for et datadelingssystem, viser at ugyldiggjøringsraten er avhengig av bufferstørrelsen og sannsynligheten for at en side skal oppdateres. For små buffere og liten oppdateringssansynlighet øker ugyldiggjøringsraten nesten proporsjonalt med antall noder i det distribuerte systemet. Ved høyere sannsynlighet for oppdatering blir sider ugyldiggjort raskere og igjen enda raskere ved større bufferstørrelser. Når ugyldiggjøringsraten når igjen bufferets missrate vil antall ledige plasser i bufferet øke siden ugyldige sider ikke kan brukes igjen og derfor regnes som ledige. Ugyldighetsraten vil ikke lenger øke med større bufferstørrelser.

[6] Dette vil også påvirke buffermekanismen. For mange låste sider vil igjen gi et mindre tilgjengelig buffer.

Sideutskiftningsraten (hvor gyldige sider fjernes fra bufferet) går den inverse veien av ugyldiggjøringsraten, for ugyldige sider regnes som ledige. Antall sider som må byttes ut minsker da med en økning av ledige sider i bufferet. Når ugyldiggjøringsraten når igjen bufferets missrate, vil sideutskiftningsraten bli null.

Sansynligheten for en treff i et lite buffer er lite påvirket av ugyldiggjøringsraten. Ved større buffere og flere noder vil ugyldiggjøringsraten bli større, og treffsansynligheten vil minske.

## 5.4 Utbyttingsalgoritmer

Når vi får en miss i bufferet og det ikke er flere ledige plasser igjen, må en blokk velges ut for utskiftning. For å designe optimale algoritmer for utbytting av blokker, må referansestrukturen til blokkene i hvert enkelt system undersøkes. Forskjellige typer blokker refereres ofte forskjellig. Et eksempel fra DBS [Effelsberg et al. 84] sier at selv om systemblokker er mange færre, kan referanseprosenten komme opp i hele 45 prosent, og oppdateringstransaksjoner ser igjen ut til å bruker enda flere systemblokker. Videre ser man ofte lokalitet i referansestrukturen. Alt dette er parametere som nøye må taes hensyn til i designet av en algoritme.

Det er derfor ikke en universell blokkutbyttingsalgoritme som passer for alle typer systemer. Det finnes for eksempel mange forskjellige DBSer. Disse har ulike transaksjoner og referansestrukturer avhengig av implementasjonsdetaljer, intern struktur av system- og brukerdata og så videre. Ved valg av strategi for blokkutbytting, bør man se på hvert enkelt tilfelle og modelere en algoritme som passer for den type applikasjoner som skal brukes.

Et annet viktig punkt er i hvilket nivå av minnehierarkiet algoritmen skal brukes. En stor og kompleks algoritme mellom to nivåer med liten forskjell i hastigheten, som for eksempel et hurtigbuffer og det virtuelle minnet, vil bruke for lang tid til å finne hvilken blokk som bør byttes ut. Det vil ta like lang tid å bytte ut blokken som det tar å finne ut hvilken som bør byttes. Er det imidlertid stor hastighetsforskjell vil en bedre algoritme kunne lønne seg slik at antall misser reduseres.

### 5.4.1 Blokkutbyttingsalgoritmer i hurtigbufferet

I et hurtigbuffer bør utbyttingsalgoritmen være rask og enkel. For å få best mulig ytelse implementeres utbyttingsalgoritmen til et hurtigbuffer ofte i hardware, og det er derfor viktig at den er så enkel som mulig. En avansert og kompleks algoritme vil bli for tidkrevende og komplisert å implementere i dette nivået i hierarkiet. [Patterson et al. 94] beskriver to hovedalgoritmer:

***Vilkårlig.*** Blokken som skal skiftes ut er vilkårlig trukket ut.

***Least recently used* (LRU, sist brukt).** Blokken som ble sist brukt (referert), altså har vært lengst i bufferet uten å bli brukt, byttes ut.

En vilkårlig utvelgelse er lett å implementere, mens LRU blir mer og mer kompleks og kostbar ettersom antall blokker i hurtigbufferet blir flere. I et to-veis sett assosiativ hurtigbuffer har vilkårlig seleksjon circa 1,1 ganger høyere missrate enn LRU [Patterson et al. 94]. LRU viser seg å bli enda bedre ved en større grad av assosiativitet, men er da vanskligere å implementere.

### 5.4.2 Sideutbyttingsalgoritmer i hovedminnet

Fysiske referanser til sider[7] er i alle typer systemer en av de dyreste operasjonene. I tillegg til at det koster mellom 10 og 20 ms i aksess tid for å finne siden på disken (tabell 5.1), må vi bruke

[7] Som nevnt i tidligere avsnitt kalles en blokk i det virtuelle minnet en side.

fra 2000 til 5000 instruksjoner i prosessoren for å håndtere en sidefeil [Effelsberg et al. 84]. Det er derfor viktig å optimere bufferhåndteringen ved å ha en en sideutbyttingsalgoritme som minimerer antall diskaksesser.

En utskiftningsalgoritme for et sideutbyttingssystem er helt eller i allefall delvis implementert i software. Vi kan derfor ha mer komplekse algoritmer som igjen kan gi bedre treffrate. Den første algoritmen vi vil nevne er OPT (av Belady, L., A.) som gir et optimalt ønskelig resultat. Algoritmen regner ut hvilken side det er lengst tid til skal brukes igjen, og deretter bytter den ut denne siden. Denne agoritmen er ikke realiserbar og er bare av teoretisk interesse, for den baserer seg på fremtidig bruk av sider. Den er interessant fordi den kan brukes som et mål for best mulig sideutskiftning.

I tillegg til algoritmene presentert i avsnittet over (avsnitt 5.4.1), er de neste algoritmene mer praktiske. Målet for alle er å minimere antall sidefeil. Algoritmene prøver å spå fremtiden ved å bruke viktige designparametere som lokalitet og referanse-oppførsel:

***First inn, first out* (FIFO, først inn, først ut)** skifter ut den siden som har vært i bufferet lengst uavhengig om siden er referert mye eller lite. Alderen til siden er eneste utskiftningskriterie, og algoritmen er derfor ikke brukbar for annet enn applikasjoner med sekvensiell aksess-oppførsel.

***Least frequently used* (LFU, lavest referansefrekvens)** bytter ut den siden som har lavest referansefrekvens. Hver side får en egen referanseteller som økes med en for hver referanse. Ved hver sidefeil sjekkes tellerene, og den siden med lavest referanseteller byttes ut. Fordi det bare er antall referanser som teller, brukes ikke ren LFU. En side som på et kort intervall brukes veldig mye, men aldri mer etter det, kan risikere å aldri bli byttet ut. LFU viser oss at alder er et viktig poeng i utskiftning av sider i bufferet, så både alder og referansefrekvens bør taes hensyn til.

***CLOCK*** er implementert som en ring av sider med en roterende peker. Hver side har et bruktmerke som indikerer om siden ble referert under siste rotasjon av pekeren. Ved en referanse settes merket. Ved en miss i bufferet, roterer pekeren rundt i bufferet. Hvis den kommer til en side med merket satt, fjernes dette merket, og første side uten satt merke byttes ut.

***GCLOCK* (generalisert CLOCK)** er en kombinasjon av LFU og CLOCK. Bruktmerket i CLOCK er byttet ut med referansetelleren i LFU og initialisert til en. Ved en sidefeil går pekeren rundt og senker verdien i referansetellerene med en. Den stopper ved første side hvor referansetelleren er lik 0. Denne siden byttes ut. Selv om dette er en god forbedring av LFU, er det likevel oftest de yngste sidene som byttes ut. Dette kan forbedres ved følgende varianter av GCLOCK:

- initialisere referansetelleren med en verdi større enn en.
- sette referansetelleren til en fast verdi hver gang den rerefereres.
- sette vekter på sidene, slik at “viktige” sider får høyere verdier i referansetelleren.

***Least reference density* (LRD, minst referansetetthet)** har to versjoner:

- LRD(V1) beregner en sides referansetetthet med antall referanser til siden og alderen til siden (gitt i antall referanser i hele systemet siden den ble satt inn i bufferet). Siden med minst verdi byttes ut. Her har vi en global variabel, GRC, som inneholder det totale antall referanser. Hver side har en aldervariabel som settes til GRC ved den første referansen samt en referanseteller. En sides referansetetthet kan da regnes ut ved

$$\frac{\text{referanseteller}}{\text{GRC-alder}}$$

En svakhet med LRD(V1) er at en høy referanseaktivitet i begynnelsen av referanseintervallet, kan holde siden i bufferet lengre enn ønsket. Dette har ført til en annen versjon av LRD, nemlig LRD(V2).

- LRD(V2) reduserer innflytelsen av gamle referanser på utvelgelsen av side som skal skiftes ut. Her blir alle referansetellere (RT) redusert etter et visst referanseintervall. For eksempel kan algoritmen designes slik hvor siden eldres etter et spesifisert referanseintervall:

$$RT = \begin{cases} RT - C1 & \text{hvis RT-C1}\geq\text{C2 når C1>0 og C2}\geq 0 \\ C2 & \text{hvis RT-C1<C2} \end{cases}$$

Ved en miss i bufferet blir siden med lavest referanseteller byttet ut.

***LRU-K*** [O'Neil et al. 93] baserer utskiftingen av en side på de siste *K* referansene til en side[8]. Til dette benytter den en *baklengs K-distanse* $b_t$*(p,K)* som er distansen bakover i referansestrengen til den K'te siste referansen til siden p. $b_t$(p,K) er $\infty$ hvis ikke K referanser eksisterer. Når vi trenger en ny bufferplass for en ny side, velger algoritmen den siden som har lengst $b_t$(p,K) i bufferet. Ved flere tilfeller av en $b_t$(p,K) lik $\infty$ må en tillegs algoritme velge mellom disse.

***2Q*** [Johnson et al. 94] er en algoritme som bruker tre køer for å beholde mye refererte og "populære" sider i bufferet. En Am-kø (organisert som LRU) for sider som er aksessert mer en to ganger, en A1in-kø (organisert om FIFO) for sider som er referert en gang og en A1out-kø (organisert som FIFO) som har lagret pekere til sider som har blitt aksessert en gang, men som har blitt kastet ut av A1in.

Algoritmen fungerer slik at når en side skal aksesseres sjekkes først Am-køen. Hvis den er der flyttes (LRU) siden først i køen. Er den i A1out-køen, finner man en plass til siden i bufferet, og den settes først i Am-køen. Hvis siden er i A1in gjøres ingen ting, og finnes den ikke i noen av køene, finner man en plass til den i bufferet, og så legges den til først (FIFO) i A1køen.

Når en side skal settes inn sjekkes det om det er plass i bufferet. Er derimot bufferet fullt, må en side fjernes og da sjekkes det først om A1in er full. I så fall fjernes den siste siden i A1in, og en peker til denne siden legges først i A1out. Hvis ikke A1in er full fjernes den siste siden fra Am.

Strategier som GCLOCK og LRD gir mange muligheter for design gjennom mange parametere. I et forsøk på to forskjellige sidereferansestrenger i en CODASYL DBMS-applikasjon [Effelsberg et al. 84], kommer det frem at lokal allokering med dynamiske partisjoner gir en missrate i bufferet ganske lik global LRU utbytting. Med en global allokering er bufferinnholdet ved bestemte referansestrukturer bestemt ene og alene av utbyttingsalgoritmen. Det viser seg at LRU og CLOCK gir gode resultater, noe som også kan vises ved bruk av LRD og GCLOCK algoritmer.

LRU-K algoritmen er en enkel og lite kostbar algoritme som ikke bare bruker aksesshistorien til en side som utskiftningskriterie, men også bruker aksessfrekvensen. Den er forskjellig fra LFU på den måten at den har en innebygget aldring hvor de siste K referansene brukes, og den skiller dermed mellom nylige og gamle referanser. Det gjør ikke LFU. LFU-baserte algoritmer som GCLOCK og LRD, bruker variabler som referansetellere. LRU-K bruker bare egne data og er uavhengig av parametere som varierer med arbeidsmengden. Simulasjoner viser at LRU-K (K$\geq$2) har betydelig bedere kost/ytelse forhold enn algoritmer som LRU [O'Neil et al. 93].

2Q er en algoritme som er bygd på samme prinsipper som LRU-K (LRU-2), men som prøver å minske overheaden LRU-K har med å manipulere prioritetskøen. LRU-K har log N (N = antall

[8]Merk at LRU-1 tilsvarer den klassiske LRU algoritmen.

sider i bufferet) arbeid ved aksess av en side, mens 2Q har konstant arbeidsmengde. Både LRU-K og 2Q er algoritmer som deler sidene i varme og kalde sider. Varme sider vil bli brukt flere ganger over lengre tid, mens kalde sider brukes en gang (eller flere ganger i starten) og deretter refereres de ikke flere ganger. Algoritmen prøver å beholde de varme, populære sidene så lenge som mulig, mens kalde sider byttes ut. Forsøk gjort av Johnson og Shasha [Johnson et al. 94] viser at 2Q er en god buffer-algoritme som har konstant overhead og krever ingen eller lite justeringer. Både 2Q og LRU-2 baserer bufferprioritet på langvarig popularitet i steden for enkelt aksesser, og det viser seg at 2Q gir bedre resultater enn LRU og GCLOCK og samme eller bedre resultater enn LRU-2.

Et annet viktig kriterie å ta hensyn til er om en side er oppdatert eller ikke. Ofte kan det være ønskelig å beholde oppdaterte sider lengre i bufferet med tanke på nye oppdateringer. På den andre siden er det da en risk for at vi skal få for få "bare-lese"-sider i bufferet. Videre er det ofte slik at det bare er små deler av en side som brukes. For å unngå å måtte hente bare hele sider, gjør Kemper og Kossmann [Kemper et al. 94] et forsøk med dobbel buffering (dual buffering) i en objekt-DB hvor både objekter og hele sider buffres avhengig av hvor mye av siden som brukes. De prøvde flere strategier med hensyn på hvordan objekter kopieres fra deres hjemmeside og hvordan objektene overføres tilbake (relokaliseres) til hjemmesiden. Dette viste seg å utvide den fleksibiliteten buffersegmentering tidligere har vist, og best resultater fikk de ved en sen objektkopiering og så rask relokasjon.

## 5.5 Bufferallokering

Noen systemer har en egen algoritme for å allokere plass i bufferet. En slik bufferallokeringsalgoritme i buffermanageren distribuerer tilgjengelige bufferrammer (sider) på prosessene i et OS og transaksjonene i et DBS. En slik algoritme er nært knyttet til utbyttingsalgoritmen, og som regel brukes bare en felles algoritme. På grunn av den nære sammenhengen mellom disse algoritmetypene, presenteres ingen egne allokeringsalgoritmer her, men bare de ulike klassene av algoritmer.

Bufferallokeringsalgoritmer kan grovt deles i tre kategorier: *lokale*, *globale* og *sidetypeorienterte*. I et DBS vil lokale og globale algoritmer allokere sidene til en bestemt transaksjon, hvor lokale ikke tar hensyn til referansestrukturen i samtidige transaksjoner. Sidetypeorienterte algoritmer deler bufferet etter sidetyper, og allokerer plass til samme type sider (for eksempel system-sider) på samme sted.

Lokale og sidetypeorienterte algoritmer kan igjen deles i *statiske* og *dynamiske* algoritmer. Statiske algoritmer tildeler hver transaksjon et fast antall sider som den har under hele sin levetid i systemet. Forskjellige typer transaksjoner kan ha ulikt antall sider. Dynamiske algoritmer har variabel størrelse på sidepartisjonene til transaksjonene. Hver partisjon kan økes eller minskes underveis avhengig av referansestrukturen og DBSets arbeidsmengde.

Siden transaksjonene i statiske algoritmer beholder alle de tildelte sidene til transaksjonen har terminert, blir systemet ineffektivt når systemets arbeidsmengde forandres. En transaksjon kan bli sittende med mange flere sider enn den trenger og ytelsen blir lavere enn den kunne ha vært. Statiske algoritmer blir derfor ikke brukt i bufferhåndtering [Effelsberg et al. 84].

## 5.6 Bufferhåndtering i multimediasystemer

På grunn av MMSers karakteristiske datatyper og deres bruk, krever multimediaapplikasjoner spesiell støtte fra de underliggende delkomponentene (avsnitt 2.4). Dette inkluderer støtte for plasskrevende, kontinuerlige og tidsavhengige datatyper som audio og video, sanntidseksekvering og synkronisering. Et kritisk punkt i behandlingen av kontinuerlige datastrømmer er bufferhåndteringen for presentasjonen av multimediadata.

**Buffer etter endt avspilling:**

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

**Vil så se fra side 5 og utover igjen:**

**LRU:**

Trenger 5 -> sidefeil. Bytt ut LRU -> 6

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

Trenger 6 -> sidefeil. Bytt ut LRU -> 7

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

Trenger 7 -> sidefeil. Bytt ut LRU -> 8

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 7 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

⋮

Trenger 19 -> sidefeil. Bytt LRU -> 20

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 20 |

Trenger 20 -> sidefeil. Bytt ut LRU -> 5

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

**Sidefeil totalt: 16**

**L/MRP:**

Trenger side 5 -> sidefeil. Bytt ut den minst presentasjonsrelevante siden -> 20

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 5 |

Trenger side 6 -> OK

Trenger side 7 -> OK

⋮

Trenger side 19 -> OK

Trenger side 20 -> sidefeil. Bytt ut den minst presentasjonsrelevante siden -> 5

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

**Sidefeil totalt: 2**

Figur 5.9: Et eksempel på hvorfor det er nødvendig med presentasjonsrelevant sideutbytting.

Minnehierarkiet som ble presentert tidligere i kapittelet, brukes også i MMSer. Dette gjelder også mange av utbyttingsalgoritmene for sider, men på grunn av multimedias spesielle datatyper og applikasjoner, er en del andre algoritmer også prøvd.

### 5.6.1 Sideutbyttingsalgoritmer for multimediasystemer

I avsnitt 5.4 ble det presentert algoritmer som LRU, FIFO, LFU og så videre. Fordi multimedia-applikasjoner er så forskjellige fra tradisjonelle databaseapplikasjoner, viser det seg raskt at ikke alle disse kan håndtere interaktive kontinuerlige datastrømmer. Et eksempel er vist i figur 5.9 hvor vi har LRU utskiftning og et buffer med 15 plasser [Moser et al. 95]. Ved en presentasjon av sidene 1 til 20, vil det ved slutten av presentasjonen være sidene 6 til 20 i bufferet. Hvis vi deretter ønsker å få sidene 5 til 20 presentert en gang til, må vi hente inn side 5 på nytt og velge ut en side som må fjernes. Etter LRU strategien er det side 6 som må ut. Etter side 5 trenger vi side 6, men denne er nå fjernet fra bufferet og må hentes inn igjen på bekostning av side 7. Dette eksempelet viser at LRU kan risikere å bytte ut den siden som vil bli referert neste gang.

Tilsvarende resultater kan bli funnet ved bruk av andre tradisjonelle utbyttingsalgoritmer i tilsvarende applikasjoner. På grunn av store datastrømmer i MMSer, trenger vi bedre minnehåndteringsmekanismer for å kunne optimere distribusjonen av datasider i bufferet mellom tidsavhengige og tidsuavhengige datastrømmer [Reddy et al. 94]. Det er derfor blitt prøvd en del utbyttingsstrategier spesielt for multimedia. Hva som passer best for et system vil også her avhenge av hva slags applikasjoner som støttes. En algoritme spesielt utviklet for presentasjon av kontinuerlige datastrømmer presenteres her [Moser et al. 95]:

***Least/Most Relevant for Presentation*** **(L/MRP, minst/mest relevant for presentasjon)** er en forhåndshenting og utbyttelses algoritme som bruker noe kalt interaksjonssett i algoritmen. Det er tre slike sett opprinnelig. Et sett med fremtidig refererte sider som inneholder sider som vil bli referert i presentasjonen senere. Et historisk sett som inneholder sider som har blitt referert, og et sett med sider som kan hoppes over avhengig av presentasjonsstatus. Hvert element i et slikt sett gis en verdi fra null til en av en relevansfunksjon for å angi elementets egnethet for utbyttelse og forhåndshenting, graden av viktighet for å beholde siden i bufferet. Disse verdiene er dynamiske, og de er avhengig av spesielle presentasjonsparametere som hvilken enhet som presenteres akkurat nå. Elementer med minst relevansverdi byttes ut, og de med størst relevansverdi forhåndshentes.

Når en enhet hentes for eksekvering, sjekkes alle enheter med relevans en i objektet som presenteres. Alle disse skal være i bufferet. De som ikke er det hentes. Hvis det ikke er plass til alle disse fjernes den enheten med minst relevans.

Hvis ingen avbrytelser finner sted og vi bare har en kontinuerlig presentasjon av data, kan en "bruk og kast"-algoritme gi en optimal sideutbytting, men ved bruk av applikasjoner hvor vi for eksempel snur presentasjonen og avspiller den baklengs, spoler den tilbake eller lignende, vil dette være en dårlig løsning siden vi har byttet ut sider vi igjen vil trenge.

Figur 5.9 viser også hvordan L/MRP vil skiftet ut sider i scenarioet beskrevet over. Som det kommer frem av figuren vil L/MRP gi en betydelig forbedring fra LRU's 16 sidefeil til L/MRP's 2 sidefeil.

L/MRP kan også sees på som en applikasjonsspesifikk algoritme spesielt designet for interaktive, kontinuerlige datastrømmer. Den skal minimere ventetiden ved interaksjoner av brukeren. Algoritmen garanterer at en forespurt side er i bufferet ved å forhåndshente, og bare en presentasjonsforandring kan gjøre at ikke den neste siden som skal refereres ligger i bufferet. Algoritmen kan lett utbygges til å inneholde flere interaksjonssett, slik at andre applikasjoner får bedre støtte. Basert på relevanser og presentasjonsstatusen byttes sider med minst relevans for presentasjonen ut, og de viktigste sidene for fremtidige referanser forhåndshentes.

I et ytelsesstudie gjort av Moser, Kraiß og Klas [Moser et al. 95] viser det seg at L/MRP har bedre ytelse i applikasjoner med både mye og lite interaksjoner. For eksempel er sidefeilraten bare tre prosent dårligere enn OPT uansett bufferstørrelse i et VoD-scenario, og jo større frekvensen av interaksjoner blir, jo bedre blir L/MRP i forhold til de andre algoritmene.

### 5.6.2 Bufferdeling og forhåndshenting

Ved siden av å finne en god sideutbyttingsalgoritme, er det andre ting som også kan forbedres i bufferbruken. I et system som har en fast, forhåndsbestemt diskbåndbredde og bufferstørrelse, viser Ng og Yang [Ng et al. 94] at gjennomstrømming og responstid kan bedres betraktelig ved å maksimere bruken av disken og bufferet.

Det kan være mye å tjene på å dele bufferet mellom datastrømmer. Hvis vi tenker oss at parallelle datastrømmer eksekveres etter hverandre i en sykkel, vil hver datastrøm trenge et vist antall buffere hver sykkel. Ng og Yang sier at en datastrøm ikke vil trenge alle disse bufferplassene samtidig. En enkel måte å minimere det totale bufferbehovet er å la datastrømmene dele buffer. På denne måten vil mer bufferplass bli tilgjengelig for andre datastrømmer og gjennomstrømmingen kan forbedres.

Videre kan det å forhåndshente data til ventende prosesser inn i bufferet gi bedre gjennomstrømming og responstid; det vil si hvis det gjøres uten å redusere tilgjengelig bufferplass og båndbredde for aktive prosesser. Forhåndshenting av data kan bedre ytelsen på flere måter. For det første kan vi få en raskere start på nye prosesser. Ved oppstart av en prosess vil ofte den nye prosessen legges til slutt i eksekverings- og datahentingssykkelen. Den nye prosessen har da ingen data å jobbe med, før det er dens tur til å lese fra disken. Ved å forhåndshente data inn i bufferet mens prosessen enda venter på å få begynne, kan eksekveringen av denne prosessen startes før, og responstiden bedres. Videre vil en prosess som har fått forhåndhentet data ikke behøve å hente så mye underveis. Dette gir igjen rom for at andre kan bruke den ledige kapasiteten, og gjennomstrømmingen vil øke.

Resultatene til Ng og Yang er hentet fra et NoD system hvor datastrømmene sjelden overstiger fem minutter. Det viser seg at det å dele bufferet mellom datastrømmene kan redusere bufferbehovet med opptil 50 prosent. Deres forhåndshentingteknikk, *intelligent preloading* (IP, intelligent forhåndshentning) som maksimerer forhåndshentingen og antall prosesser som kan startes, kan gi opptill 40 prosent bedre gjennomstrømming.

Andre studier av lengre datastrømer kan vise at det kan lønne seg å partisjonere bufferet for å bedre brukerens ventetid på en forespurt tjeneste. I et studie av bufferhåndtering i et video-DBS prøver Rotem og Zhao [Rotem et al. 95] tre forskjellige muligheter å partisjonere bufferet for en lang videosekvens:

- *Strategi S1: ingen partisjonering*. Hele bufferet brukes som en enkelt partisjon.
- *Strategi S2: statisk partisjonering*. Bufferet deles opp i et fast antall statiske partisjoner.
- *Strategi S3: tilpasset partisjonering*. Antall partisjoner og størrelsene på disse tilpasses etter når brukerene ønsker å se filmen. Når nye brukere kommer til får vi en ny repartisjonering av bufferet.

Innen hver partisjon er det en bestemt lengde med film vi har i bufferet, og brukere som "melder seg på" for å se filmen underveis, kan begynne å se med en gang hvis en partisjon ikke har begynt å bytte ut starten av filmen. I så fall må de vente til en ny partisjon blir ledig. S1 vil kunne ha et langt åpent intervall for nye seere, men ha lang ventetid hvis de kommer for sent. Ved å ha flere partisjoner vil ventetiden kunne reduseres med over 50 prosent, og S3 viser seg noe bedre enn S2 i de fleste tilfeller.

## 5.7 Tjenestekvalitetparametere i bufferhåndtering

Bufferhåndtering er en viktig komponent i et MMS, og bufferhåndteringen påvirker mange av de QoS-parameterene nevnt i avsnitt 4.1.5 og i avsnitt 4.4 om QoS-parametere i henholdsvis OSer og DHSer[9]. Ser vi imidlertid på bufferhåndteringen som et isolert system, er det eneste interessante om bufferhåndteringsmekanismen greier å hente inn data fort nok for eksekvering. Dette gir to særlig viktige QoS-parametere:

- *responstid* og
- *gjennomstrømming.*

Kort sagt betyr dette at bufferhåndteringsmekanismen må kunne levere flest mulig datasider i løpet av et gitt tidsrom.

For bufferhåndtering er det særlig ventetid-delen (avsnitt 4.4) av responstiden som er viktig. Bufferhåndteringen er viktig i prosessen med å hente frem data for eksekvering, men har ellers ikke så mye med selve eksekveringen å gjøre. Det er hvor lenge en prosess må vente på at dataene hentes fra disken til bufferet som teller. Gjennomstrømmingen er også viktig., for den sier litt om hvor mange datastrømmer vi kan håndtere av gangen, eller hvor mange klienter et system kan tilby tjenester til samtidig.

[9]For eksempel vil vi få forskjellige typer skjevheter i presentasjonen hvis data ikke leveres i tide.

For å kunne gi en bruker best mulig kvalitet på tjenestene som et system tilbyr, må, som nevnt flere ganger tidligere, alle komponenter i systemet støtte samme QoS. Hvis en komponent ikke kan støtte den ønskede QoS, vil den best mulige QoS begrenses av denne komponenten. For å få bufferhåndteringen best mulig, er det derfor viktig å finne bufferhåndteringsmekanismer som gir minst mulig responstid og størst mulig gjennomstrømming. Dette kan gjøres ved gode og fornuftige utbyttingsalgoritmer i forhold til hva slags applikasjoner som skal kjøres på systemet, en passende størrelse på sidene og et buffer som er så stort som mulig.

Videre kan både lav responstid og høy gjennomstrømming være vanskelig å oppnå. Dette kan være to motstridene krav ved at god responstid kan gi lavere gjemmomstrømming[10] og omvendt. I slike tilfeller må man bestemme hva som er viktigst i sitt system.

[10]Lave responstider kan for eksempel oppnås ved bruk av forhåndshentingsteknikker. Sider for fremtidig bruk vil da okkupere plasser i bufferet som igjen kan gi lavere gjennomstrømming.

# Del II

# Modellering, realisering og evaluering

# Kapittel 6

# Modellering av bufferhåndteringsmekanismen

I dette kapittelet modellerer vi en sideutbyttingsalgoritme for et bufferhåndteringssystem for den valgte eksempelapplikasjonen, det vil si et DHS for det elektroniske klasserommet, som ble presentert i kapittel 3. Siden fysiske aksesser til disken er så kostbart, ønsker vi å minimere antall diskaksesser, og det vi prøver å oppnå med den valgte sideutbyttingsalgoritmen er å bytte ut den siden som har minst relevans for det punktet vi er i fremvisningen av en multimediaapplikasjon. Videre ønsker vi å oppnå en stor gjennomstrømming av data og en rask responstid slik at bufferhåndteringsmekanismen ikke skal bli noen flaskehals i systemet for å kunne støtte en kontinuerlig presentasjon av en forelesning.

I avsnittene under vil vi se på sentrale punkter i en bufferhåndteringsmekanise. Vi diskuterer hvilke punkter som er viktige for vårt system og evaluerer forskjellige parametere og sideutbyttingsalgoritmer.

Først beskriver vi den delen av det elektroniske klasserommet vi vil ta for oss (avsnitt 6.1), og deretter beskriver vi på de generelle kravene til en bufferhåndteringmekanisme i et DHS for det elektroniske klasserommet (avsnitt 6.2). Videre undersøker vi hvordan buffer- og sidestørrelser innvirker på systemet (avsnitt 6.3), samt at vi drøfter bufferdeling mot det å splitte bufferet i flere partisjoner (avsnitt 6.4). For å bevise at nye mekanismer for bufferhåndtering i MMSer er nødvendig, viser vi at tradisjonelle bufferhåndteringsmekanismer som henter data på forespørsel ikke kan støtte kontinuerlig presentasjon av en forelesning (avsnitt 6.5). Deretter evaluerer vi en del av de sideutbyttingsalgoritmene vi så på i kapittel 5 (avsnitt 6.6). I slutten av kapittelet modellerer en algoritme som kan passe godt for vår valgte eksempelapplikasjon (avsnitt 6.7).

## 6.1 Det elektroniske klasserommet fra studentens side

I den planlagte utvidelsen av det elektroniske klasserommet er det studentene som vil gi den overveiende lasten i DHSet og dermed gir den største utfordringen. Vi tar derfor for oss et multimedia-DBS som best støtter behovene til det elektroniske klasserommet fra studentens side.

En student kan levere øvingsoppgaver eller følge en forelesning i det elektroniske klasserommet. Det scenarioet vi tar for oss i denne oppgaven er presentasjon av de datatypene som vil oppta de største delene av båndbredden, det vil si presentasjonen av en forelesning som inkluderer presentasjon av video fra vanlig videokamera og et dokumentkamera, presentasjon av audio og presentasjon av foiler på den elektroniske tavlen. Her må studenten ha mulighet til å få presentert hele forelesningen i sin helhet med alle de forskjellige datatypene, eller studenten må kunne velge bare en eller flere av dem. Videre tenker vi oss at studenten må kunne koble presentasjonen av en gitt datatype ut og inn etter som det passer han. Han må for eksempel kunne bla igjennom foilene, for så å kunne følge resten av forelesningen herfra.

I presentasjonen av forelesningen tenker vi oss at studenten tilbys følgende operasjoner:

- vanlig avspilling,
- stopp og pause av presentasjonen,
- hopp og
- rask avspilling (frem- eller bakover).

Dette gir en funksjonalitet som på en vanlig videospiller. Rask avspilling er som vanlig avspilling i for eksempel dobbel hastighet, og hopp er som å spole frem eller tilbake uten bilde. Ved rask avspilling og avspilling bakover kobles lyden ut, og vi ser bare bildet.

I denne delen av det elektroniske klasseromscenarioet har studenten bare lesetilgang på dataene. Dette gir en del forenklinger med hensyn på de kompliserende aspektene fra avsnitt 5.3.7. Studenten kan ikke gjøre noen oppdateringer av data som skal lagres i DBSet slik at samtighetskontroll og mekanismer for gjennvinning av data kan forenkles. Videre kan flere brukere benytte samme data samtidig, og loggings- og låsingskomponenter kan derfor også forenkles betraktelig.

## 6.2 Designparametere og krav

Håndtering av data i det elektroniske klasserommet stiller, som vi så i avsnitt 3.5, bestemte krav til båndbredde, responstid og gjennomstrømming i en bufferhåndteringsmekanisme. Disse og enkelte andre designparametere beskriver vi nærmere på i dette avsnittet.

### 6.2.1 Krav til båndbredde fra de ulike datatypene

I det planlagte multimedia-DBS for det elektroniske klasserommet tar vi nå for oss håndteringen av data fra vanlig video, audio, dokumentkamera og tavlen. Alle disse datatypene lagres for seg slik at man enkelt kan velge seg en eller flere av disse for presentasjon samt at en modifikasjon av data forenkles. I avsnittene under beskrives hvordan vi har valgt å se på de forskjellige datatypene. Disse tallene om båndbreddebehov som presenteres her er per samtidige bruker.

#### 6.2.1.1 Video

Som nevnt i avsnitt 3.3.1, har vi to typer video. Først har vi vanlig video fra klasserommet. Her antar vi at vi har en bildestørrelse på 640x480 med en bilderate på 24 bilder per sekund. Vi koder videoen med JPEG-koding hvor hvert bilde tar 60 KBytes med forholdsvis god kvalitet. Dette gir et konstant båndbreddebehov på circa 1,5 MBytes/s.

Den andre typen video er fra dokumentkameraet. Vi antar at dette kodes som vanlig video, men stiller ikke like store krav til bilderaten og bruker seks bilder per sekund. Videre tenker vi oss at foreleseren viser et slikt bilde i et minutt av gangen. Dette gir et konstant båndbreddebehov i hvert av disse minuttene på circa 400 KBytes/s.

#### 6.2.1.2 Audio

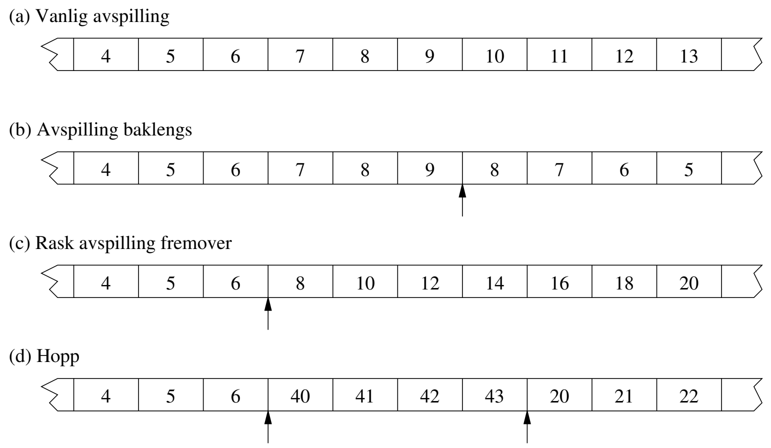
Audio spilles inn, som vi så i avsnitt 3.3, med en 16 bits/16 KHz sampler og lagres ukomprimert med PCM-koding. Dette gir et konstant båndbreddebehov på 32 KBytes/s.

#### 6.2.1.3 Foiler

En foil kan, som vi så i avsnitt 3.3, bestå av flere datatyper som samles til et HTML-dokument. Vi har her regnet med at en foreleser bruker 20 foiler på 200 KBytes per forelesning på 45 minutter. Under nedlastingen av disse foilene trengs en båndbredde på 40 KBytes/s.

### 6.2.2 Funksjonalitet og sidenes referansestruktur

For å bestemme seg for hva slags bufferhåndteringsmekanisme som bør brukes i et system er det mange parametere som må vurderes. Det første og kanskje det viktigste man må ta hensyn til, er hva slags operasjoner vi har i applikasjonen som skal brukes (avsnitt 6.1) og hva slags referansestruktur dette gir av sidene. I figur 6.1 vises typiske referansestrukturer til sider i bufferet ved presentasjon av de kontinuerlige mediene. Vanlig avspilling (a) gir en presentasjon av data i sekvensielle sider, og avspilling baklengs (b) benytter tidligere refererte sider. Ved hopp (d) i presentasjonen trenger vi sider man ikke kan forutsi på forhånd, mens ved rask avspilling (c) forover (eller bakover) brukes kanskje ikke alle sidene, og vi kan kanskje hoppe over for eksempel annenhver side.




Figur 6.1: Sidereferansestruktur for forskjellige funksjonaliteter i det elektroniske klasserommet.

Videre er det så strenge krav til å levere data i tide for de kontinuerlige, tidsavhengige mediene at det ikke ville være mulig å støtte en kontinuerlig fremvisning av en forelesning hvis vi skulle få en sidefeil for hver forespurte side i bufferet. Vi trenger en form for forhåndshenting av sider til bufferet. Ut i fra referansestrukturen presentert her, bør det i de fleste tilfeller være mulig å finne en rimelig bra forhåndshentingsmekanisme. Interaksjoner fra brukeren som lange hopp og så videre, er så å si umulige å forutsi, men store deler av en slik presentasjon vil være en kontinuerlig avspilling av data. Vi vet hvilke data (sider) som vil bli brukt neste gang, og bør utnytte dette ved å hente disse på forhånd.

### 6.2.3 Tjenestekvalitetparametere

Det kom frem i avsnitt 5.7 at QoS-parameterene som påvirkes av bufferhåndteringen spesielt er *responstid* og *gjennomstrømming*. Når vi ser på disse parameterene i sammenheng med et multimedia-DBS for det elektroniske klasserommets funksjonalitet vil det si at bufferhåndterings-mekanismen må garantere at sidene blir levert innen en bestemt tidsfrist slik at vi ikke får for-skjellige typer skjevheter i fremvisningen av de kontinuerlige datatypene, samt at responstiden ved en interaksjon fra brukeren er minimal. Responstiden bør derfor for hver av sidene være jevn og kontinuerlig slik at dataene verken presenteres for fort eller for sakte.

Samtidig med en rask og jevn responstid, må vi kunne ha en stor gjennomstrømming av data for å kunne håndtere flere datastrømmer og klienter samtidig. Dette gir som vi ser i avsnitt 6.2.1 et stort båndbreddebehov. I et elektronisk klasseromscenario vil det være de kontinuerlige datatypene som video og audio som vil kreve mest ressurser. Det er de kontinuerlige mediene som gir de største dataratene, og bufferhåndteringsmekanismen må derfor legge størst vekt på å støtte disse medienes krav. Likevel må vi ikke glemme at det er andre datastrømmer i tillegg. Det skal ved siden av presenteres bilder og foiler samt markeringer med blant annet lyspenn på tavlen. En operasjon som det å hente bare foilene til en forelesning må også kunne støttes.

## 6.3 Buffer- og sidestørrelser

Bufferetstørrelsen spiller en stor rolle i valget av sideutskiftningsalgoritme. Bufferet bør, som nevnt i avsnitt 3.5.4, være størst mulig, men resultater fra forsøk (se [Effelsberg et al. 84]) viser at ytelsene til forskjellige algoritmer, eller i allefall forskjellen i ytelse mellom forskjellige algoritmer, kan variere avhengig av bufferstørrelsen.

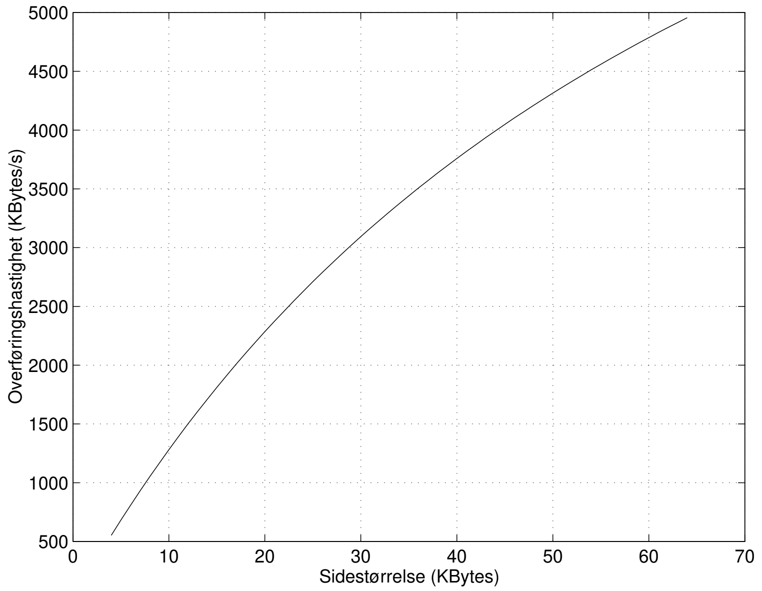
| | |
|---|---|
| Kapasitet | 23,2 GBytes |
| Antall sylindere | 6882 |
| Antall spor per sylinder | 28 |
| Antall blokker per spor | 235 |
| Blokkstørrelse | 512 Bytes |
| Spor til spor søketid (lesing) | 1,1 ms |
| Gjennomsnittlig søketid (lesing) | 13 ms |
| Søketid for fullt utslag (lesing) | 28 ms |
| Rotasjonshastighet | 5400 runder/minutt |
| Midlere rotasjonsforsinkelse | 5,56 ms |
| Overføringsrate | 10,75 - 15,5 MBytes/s |

Tabell 6.1: Spesifikasjoner for harddisken SEAGATE ELITE 23 [Seagate 97].

En av parameterene som er viktige er størrelsen på hver enkelt side i bufferet (avsnitt 5.3.5). I det elektroniske klasserommet er det behov for store mengder data, og vi vil ved å bruke mange små sider trenge mange sider og kunne få mange sidefeil. En typisk side i et tradisjonelt system er på 4 KBytes, og i et enbrukersystem med typiske båndbreddebehov på over 1-2 MBytes/s, vil dette bety at vi trenger circa 250-500 sider i sekundet. Siden det ofte er søketiden og rotasjonsforsinkelsen og ikke selve overføringen som tar lengst tid ved henting av en ny side fra disken, vil det ofte være mye tid å tjene ved å la sidestørrelsene være større. Hvis vi ser på et eksempel hvor vi antar at for hver side vi trenger, vil vi få en sidefeil, og at vi bruker data fra tabell 6.1[1] for å beregne hvor lang tid det tar å hente en side[2], vil den totale øverføringstiden for en side på 4 KBytes være 7,2 ms mens en side på 64 KBytes vil bruke 12,9 ms. Vi ser at ved å bruke en datarate som over og sider

[1] Dette er egentlig en disk med lik datatetthet over alt, det vil si at vi har flere spor ytterst enn innerst. Vi får altså høyere dataoverføringshastighet hvis dataene ligger ytterst på disken. Her regner vi imidlertid med at vi har like mange spor i alle sektorene, og at dataoverføringshastigheten er konstant for hele disken.

[2] Vi antar at blokkene ligger etter hverandre slik at den først leser alle blokkene i ett spor før den går til neste spor (som ligger ved siden av det første). Den totale overføringstiden til en side er så beregnet slik:
$\text{total overføringstid} = \text{gjennomsnittlig søketid} + \text{rotasjonsforsinkelse} + \frac{\text{rotasjonshastighet}}{\text{antall blokker per spor}} \times \text{antall blokker}$




Figur 6.2: Forholdet mellom sidestørrelse i bufferet og datarate.

på 4 KBytes, vil disken i tabellen bruke fra 1,8 til 3,6 sekunder på å hente et sekunds data[3]. Ved å bruke 64 KBytes sider, vil uthenting av tilsvarende datamengde ta fra 201,6 til 403,1 ms. Figur 6.2 viser forholdet mellom sidestørrelsen og dataraten med det forbeholdet at vi bruker en disk av samme type som i tabell 6.1 og at vi som over ikke bruker noen form for forhåndshenting av sider. Denne figuren viser at ved bruk av større sider, vil vi få en større dataoverføring per tidsrom.

Ved uthenting av data til foiler og lyspenn, vil kanskje sider på for eksempel 64 KBytes bli for "store", det vil si at dataene ikke fyller hele siden, og vi vil få okkupert, ubrukt plass i bufferet. Likevel vil andelen av sider brukt på medier som video og audio bli så mye større enn for andre datastrømmer at det vil lønne seg med store sidestørrelser. I følge Reddy og Wyllie [Reddy et al. 94] bør et filsystem allokere multimediadata i blokker på 64 KBytes eller større, og dette kan igjen indikere en anbefalt bufferside på samme størrelse.

## 6.4 Bufferdeling versus partisjonering av bufferet

Rotem og Zhao [Rotem et al. 95] viser et eksempel fra et video-DBS hvordan partisjonering av bufferet kan gi gode resultater. Disse resultatene er basert på visning av en film à la en tv-kanal slik at alle trenger de samme dataene, og bufferet er partisjonert slik at alle skal få kortest mulig responstid fra de ber om å få se filmen til filmen vises. I det elektroniske klasserommet har vi imidlertid et litt annet scenario. Vårt elektronisk klasseromscenario vil trenge egne datastrømmer for hver enkelt bruker, og det ligner derfor mer på Ng og Yangs NoD-scenario [Ng et al. 94]. Ved å partisjonere bufferet må hver bruker få en partisjon stor nok til å håndtere den største mulige datastrømmen han vil trenge, men over et gitt tidsrom, er det ikke alltid slik at alle brukerene vil trenge maksimalt med bufferplass. Ved partisjonering vil "alltid" noe bufferplass ligge ubrukt ved at denne plassen er tildelt en bruker som kanskje ikke trenger den akkurat nå. Denne bufferplassen kunne heller blitt brukt til å øke gjennomstrømmingen av data og den totale båndvidden. Det totalt bufferbehovet vil ligge en del lavere enn at alle brukere skal få maksimale partisjoner, og Ng og

[3] Ligger imidlertid alle sidene etterhverandre på disken, og vi bruker en form for forhåndshenting for å hente ut alle disse sidene samtidig, vil dette bare ta fra 101,4 til 196 ms for å hente et sekunds data.

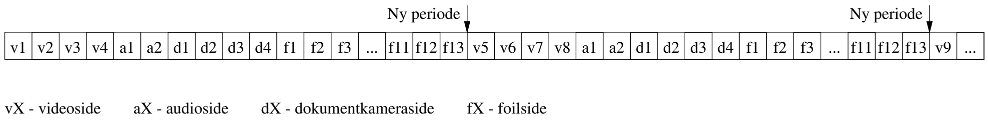
Yang foreslår derfor å minimere det totale bufferbehovet og maksimere utnyttelsen av bufferet ved å dele bufferet mellom brukerene i steden for å dele bufferet opp i partisjoner. I en bufferdelingsanalyse kommer Ng og Yang fram til at bufferdeling kan redusere det totale bufferbehovet med 20 prosent til 40 prosent avhengig av uttnyttelsen av disken.

En annen form for partisjonering av bufferet som kunne være mer interessant enn den som ble foreslått av Rotem og Zhao, er å la hver av mediatypene få hver sin partisjon (eller hvert sitt buffer) med en egen sideutbyttingsmekanisme. Som tidligere omtalt har de forskjellige mediene forskjellige referansestrukturer, og i mange tilfeller kan det være vanskelig å ta hensyn til alle disse i en og samme sideutbyttingsmekanisme. Ved å la hver medietype få sin egen partisjon med egen sideutbyttingsalgoritme, kan vi dermed skreddersy sidestørrelsene og sideutbyttingen til hvert av mediene.

## 6.5 Forhåndshenting versus "side-på-forespørsel"

Vi så i avsnitt 6.2.2 at forhåndshenting av data ved avspilling av kontinuerlige medier er mulige på grunn av en forutsigbar referansestruktur. I dette avsnittet undersøker vi dette spørsmålet nærmere, og viser at vi *må* ha en form for forhåndshenting av data fra disk til hovedminnet.

For å se hvorfor vi trenger forhåndshenting av data, har vi regnet på dataoverføringertider fra disken i tradisjonelle sideutbyttingsmekanismer som FIFO og LRU. Disse sideutbyttingsalgoritmene ber om datasider etterhvert som dataene trenges, det vil si at data hentes på forespørsel fra disken og får en sidefeil hver gang nye data må hentes fra disken. En slik buffermekanisme vil ikke kunne ha fungert i et DHS for det elektroniske klasserommet siden det ville ha tatt for lang tid å hente data.




Figur 6.3: En typisk referansestreng ved sider på 16 KBytes.

Til beregningen av dataoverføringstider bruker vi referansestrengen i figur 6.3[4]. Av denne strengen ser vi at en del av dataene brukes flere ganger slik at ikke alle dataene må byttes ut hver gang. Sider som inneholder data fra audio, dokumentkamera og foiler brukes over flere perioder[5].

| **Sidestørrelse** | **8 KBytes** | **16 KBytes** | **32 KBytes** |
|---|---|---|---|
| **Leveringstid (alle mediene)** | 45 | 23 | 12 |
| **Leveringstid (video/audio/kamera)** | 20 | 10 | 5 |
| **Leveringstid (video/audio)** | 12 | 6 | 3 |
| **Leveringstid (video)** | 8 | 4 | 2 |

Tabell 6.2: Antall sideutbyttinger ved forskjellige sidestørrelser i et "side-på-forespørsel"-system for 1/24 sekund.

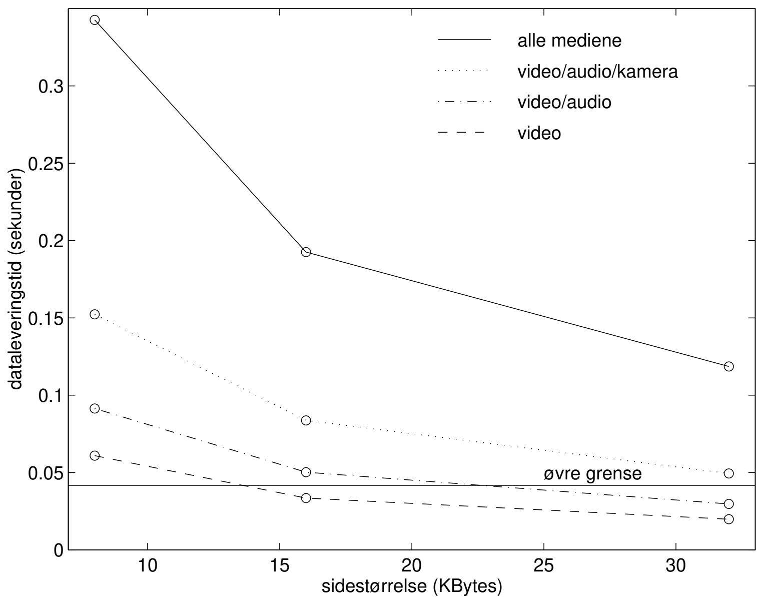
Tabell 6.2 viser hvor mange sideutbyttinger vi må foreta hvert 1/24 sekund avhengig av sidestørrelsen og hvilke av mediene som må hentes. Hvis vi tar for oss et enbrukersystem og bruker

[4]Figuren viser en typisk referansestreng ved bruk av 16 KBytes sider som brukes senere i testingen av buffermekanismen.

[5]Som vi vil se senere i oppgaven (avsnitt 6.7.1) må videobildet byttes ut for hvert 1/24 sekund, audiosampelen må byttes for hvert sekund, bildet fra dokumentkameraet for hvert 1/6 sekund og foilene byttes etter et par minutter.

| Sidestørrelse | 8 KBytes | 16 KBytes | 32 KBytes |
|---|---|---|---|
| Leveringstid (alle mediene) | 0,3427 | 0,1926 | 0,1186 |
| Leveringstid (video/audio/kamera) | 0,1523 | 0,0837 | 0,0494 |
| Leveringstid (video/audio) | 0,0914 | 0,0502 | 0,0297 |
| Leveringstid (video) | 0,0609 | 0,0335 | 0,0198 |

Tabell 6.3: Dataleveringtider i sekunder fra disken ved forskjellige sidestørrelser i et "side-på-forespørsel"-system.




Figur 6.4: Leveringstider for data ved forskjellige sidestørrelser i et "side-på-forespørsel"-system.

tallene fra tabell 6.2 og disken SEAGATE ELITE 23 fra tabell 6.1, får vi leveringstider på data slik som det er vist i tabell 6.3 og figur 6.4[6]. Tabell 6.3 viser overføringstiden for forskjellige kombinasjoner av medier med sidestørrelsene 8 KBytes, 16 KBytes og 32 KBytes når alle sidene hentes på forespørsel. Figur 6.4 viser viser det samme scenarioet, og i tillegg viser figuren den maksimale tiden (1/24 sekund) vi kan bruke før andre data igjen må hentes. Av dette ser vi at disken vi bruker ikke greier å levere data i tide ved henting av sider på forespørsel for hele forelesningen uansett sidestørrelse hvis data for alle mediene må hentes samtidig.

Flere brukere gir enda lengre dataoverføringstider, og en bufferhåndteringsmekanisme i et DHS for det elektroniske klasserommet må derfor bruke forhåndshenting av data for å kunne støtte kontinuerlig fremvisning av en forelesning.

## 6.6 Evaluering av forskjellige bufferhåndteringsmekanismer for det elektroniske klasseromscenarioet

Kravene og designparameterene til et bufferhåndteringssystem fra forrige avsnitt sier oss at bufferhåndteringsmekanismen i det elektroniske klasserommet må ha en form for **forhåndshenting** av sidene. Det er så strenge krav til å levere data i tide for de kontinuerlige, tidsavhengige mediene

[6]Merk at dette bare er overføringstider fra disken. Eksekveringstider, minneaksesser og så videre er ikke iberegnet her. Den faktiske hentetiden for dataene vil derfor i praksis være enda større.

at det ikke ville være mulig å støtte en kontinuerlig fremvisning av en forelesning hvis vi skulle få en sidefeil for hver forespurte side i bufferet. Videre vil presentasjon av diskrete medier også dra nytte av forhåndshenting. I stedet for å måtte begynne å hente en foil når den behøves, kan vi ved å bruke forhåndshenting kunne vise foilen med en gang det er ønskelig og unngå venting.

Videre bør **bufferet være så stort som mulig** og ha en forholdsvis **stor sidestørrelse**. Som vi ser av dataene over, vil små sider kunne gi mange flere sidefeil. Selv om vi bruker forhåndshentingsteknikker, vil vi likevel få sidefeil, og alle dataene vil heller ikke ligge etterhverandre på disken. Ved bruk av større og dermed færre sider, vil det totalt bli færre sidefeil og igjen mindre total overføringstid av dataene.

I tillegg må sideutbyttingsalgoritmen støtte QoS-parameterene. Dette gjøres ved å finne en fornuftig utbyttingsstrategi i forhold til sidenes referansestruktur i applikasjonen som skal kjøres.

Videre vurderer vi en del av de forskjellige sideutskiftningsalgoritmene fra avsnittene 5.4 og 5.6.1 mot kravene til en bufferhåndteringsmekanisme beskrevet over, og undersøker om noen av dem egner seg i en bufferhåndteringsmekanisme i vårt planlagte DHS for det elektroniske klasserommet.

### 6.6.1 Tradisjonelle sideutbyttingsalgoritmer

De tradisjonelle sideutbyttingsalgoritmene fra avsnitt 5.4 er alle beregnet for DHSer med små dataelementer. Et eksempel er et relasjons-DBS for en bank med informasjon om kunder og kontoer med saldoer, renter og ut- og innskudd. Det som er særlig viktig her, er å finne ut av hvilke sider som har en hyppig referansefrekvens og om det er sansynlig at de vil bli brukt igjen. I vårt DHS for det elektroniske klasserommet, vil vi i store deler av en avspilling av en forelesning ha en vanlig avspilling og bare vise data sekvensielt. Vi vil bruke dataene en gang, og deretter vil dataene være "unyttige", det vil si i tilfeller hvor vi ikke får interaksjoner fra brukeren. Likevel må sideutbyttingsalgoritmen støtte de interaksjonene som kommer fra brukeren på best mulig måte.

I avsnitt 5.6.1 så vi et (ekstremt) eksempel på hvordan LRU kan risikere å bytte ut alle sidene vi trenger i bufferet ved et hopp, for så senere å hente disse inn igjen. Samme eksempelet kan benyttes på de andre tradisjonelle algoritmene. Før vi får en interaksjon fra brukeren har vi hatt en vanlig avspilling, og vi ha brukt alle sidene en gang, det vil si at ingen av sidene er mer egnet for utbytting enn andre med hensyn på en hyppig referansefrekvens. Ved bruk av ren LFU, vil alle sidene være utbyttbare, mens alle de andre algoritmene vil bytte ut sidene på samme måte som LRU.

Ser vi på avspilling bakover (normal eller rask hastighet), vil igjen ren LFU risikere å bytte ut sider vi har brukt for. De andre algoritmene derimot, tar i tillegg hensyn til alderen av sidene og vil ikke bytte ut sider vi vil komme til å trenge igjen.

Kanskje det største problemet med disse algoritmene er at for hver nye side som må hentes, får vi en sidefeil. Det er ingen form for forhåndshenting. Dette vil, som vist i avsnitt 6.5, resultere i mye høyere overføringstid av dataene som igjen gir avbrudd og skjevheter i videopresentasjonen, forsinkelser i visning av foiler og lignende.

### 6.6.2 Sideutbyttingsalgoritmer i multimediasystemer

En annen algoritme som spesielt er utviklet for applikasjoner med kontinuerlige medier, er L/MRP som ble presentert i avsnitt 5.6.1. Denne algoritmen tar ingen hensyn til hverken alder eller referansefrekvensen til en side. Det denne algoritmen gjør er å se hvilke sider som til en hver tid er relevante for presentasjonen av videoen, og disse sidene blir forhåndshentet inn i bufferet.

Denne algoritmen er godt egnet for applikasjoner med kontinuerlige medier ved at den bruker forhåndshenting av sider til bufferet. I et VoD-scenario får man en forskjell i sidefeilraten mellom OPT og L/MRP på bare tre prosent uavhengig av bufferstørrelsen [Moser et al. 95]. Videre kan denne algoritmen støtte forskjellige interaksjoner ved å definere forkjellige relevansfunksjoner på sidene. På denne måten kan algoritmen også støtte operasjoner som bare å hente foiler ved å sette relevansverdien for andre datasider til null.

## 6.7 Valg av en sideutbyttingsalgoritme for vår eksempelapplikasjon

Det er to sentrale punkter i valget av en sideutbyttingsalgoritme i vårt DHS for det elektroniske klasserommet med tidsavhengige, kontinuerlige datatyper: *utskiftning* og *forhåndshenting* av sider.

Vi ønsker å bytte ut den siden som det er lengst tid til vil bli referert. I det elektroniske klasserommet trenger vi en algoritme som ved valg av sider som skal hentes og byttes ut, tar hensyn til hvor vi er i presentasjonen og ikke hvilke sider som er referert ofte og hvor lenge sidene har ligget i bufferet. Det er de sidene som er relevante for presentasjonen som er interessante.

Videre trenger vi en form for forhåndshenting av sider, slik at vi ikke må vente på at data skal hentes fra disken, men at de ligger klare for eksekvering i bufferet før det egentlig er behov for dem. En henting av sidene etter forespørsel vil ikke kunne garantere kontinuitet og vil kunne gi en "hakkete" presentasjon.

Til vår eksempelapplikasjon har vi valgt å prøve en sideutbyttingsalgoritme à la **L/MRP** [Moser et al. 95]. L/MRP støtter, som tidligere omtalt i avsnitt 5.6.1 og 6.6.2, både utbytting av urelevante sider og forhåndshenting av relevante sider i forhold til presentasjonen.

I avsnittene under beskrives vår valgte sideutbyttingsalgoritme for vår valgte eksempelapplikasjon.

### 6.7.1 Enheter for utbytting - presentasjonsenheter

Vi antar at alle dataene til en forelesning er delt inn i *presentasjonsobjekter* (POer), det vil si at et PO inneholder alle dataene til en spesiell medietype. POene er igjen delt opp i *presentasjonsenheter* (PEer) uavhengig av datamengder, sidestørrelser og kompresjonsteknikker. Disse PEene kan derfor være av ulik størrelse og inneholde et ulikt antall buffersider, men en PE er aldri mindere enn en bufferside. Sideutbyttingsalgoritmen bytter dermed ut PEer som ikke er relevante og forhåndshenter relevante PEer. Dette betyr at antall sider som byttes ut kan variere med størrelsen og typen av PE. En slik PE kan for eksempel inneholde en viss mengde med data fra et kontinuerlig medie, eller det kan være en foil. Ser vi på de kontinuerlige mediene, kan for eksempel et bilde i den JPEG-kodede videoen være en video-PE, mens for audio har vi kanskje et sekund med data i en audio-PE.

Oppdelingen av et PO i PEer er gjort slik:

- et bilde for video er en PE, det vil si at en video-PE inneholder $\frac{1}{24}$ sekund eller 41,67 ms med data.
- et sekund med data fra audio er en audio-PE.
- et bilde med video fra dokumentkameraet, det vil si at en dokumentkamera-PE inneholder $\frac{1}{6}$ sekund eller 166,67 ms med data.
- en foil regnes som en tavle-PE.

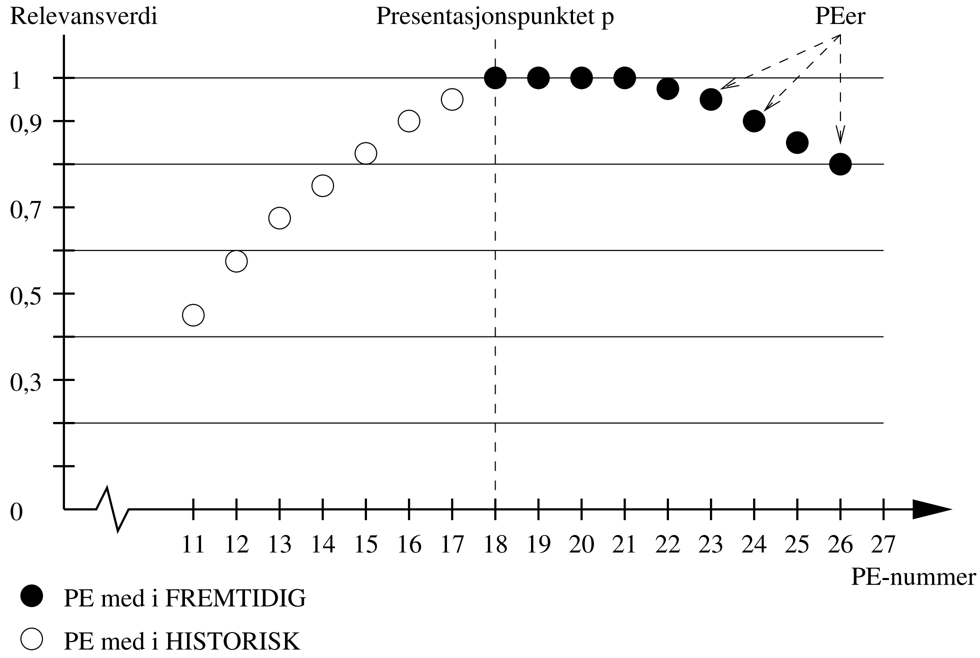
Selv om vi har delt inn POet i PEer av forskjellige størrelser, har vi i bufferet faste sidestørrelser. Vår algoritme vil derfor bytte ut PEer som beskrevet over hvor disse PEene automatisk mappes ned til de bestemte datasidene i algoritmen.

### 6.7.2 Generell modell

PEer er relaterte til såkalte *interaksjonssett*. Dette er forskjellige sett av PEer som setter relevansen til en PE avhengig av hva slags presentasjonsmodus vi har og PEens avstand fra presentasjonspunktet. Hovedsaklig har vi to typer interasjonssett:

- *FREMTIDIG:* Dette settet inneholder de PEene som vil bli referert i nærmeste fremtid.
- *HISTORISK:* Dette settet består av de PEene som allerede har blitt referert.

Figur 6.5 viser hvordan interaksjonssettene får relevansverdier. PEer for fremtidig bruk vil få relevansverdier fra en med synkende verdier etter hvor langt unna PEene er presentasjonspunktet (p) som viser hvor vi er i presentasjonen. Historiske PEer får en viss relevansverdi i tilfelle brukeren spiller av forelesningen bakover med bilde eller hopper et lite stykke tilbake for å høre hva foreleseren sa en gang til.




Figur 6.5: Eksempel på interaksjonssett med relevansverdier.

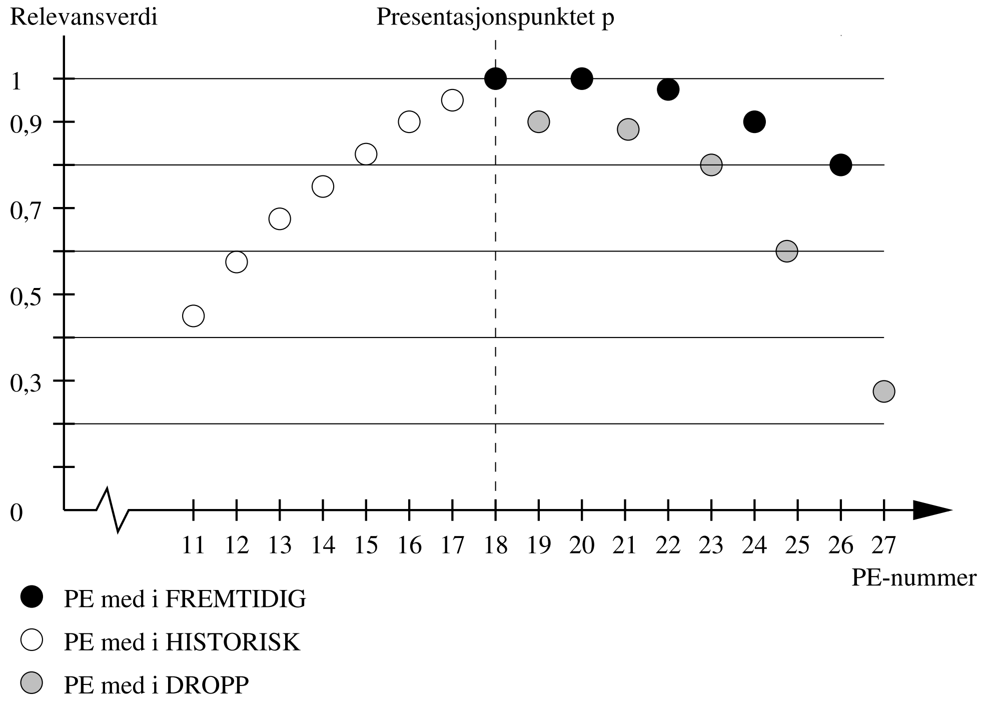
Ved en rask avspilling er det unødvendig å hente alle dataene fra disken. Videoen har for eksempel bare behov for å vise annethvert bilde, og audioen er ikke forståelig ved andre presentasjonsformer enn vanlig avspilling. Vi innfører derfor et ekstra interaksjonssett for å bedre kunne håndtere forskjellige presentasjonsmoduser:

- *DROPP:* Dette settet inneholder de PEene som vil bli hoppet over ved en rask avspilling.

Dette settet gir ved rask avspilling for eksempel annenhver PE i videoen lavere referanserelevans fordi bare annenhver PE brukes i presentasjonen. Figur 6.6 viser hvilke PEer som tilhører hvilke interaksjonssett og med hvilke relevansverdier ved presentasonspunktet p = 18 ved rask avspilling forover.

Disse interaksjonssettene beregnes avhengig av en presentasjonsstatus (s). Denne er gitt av et tuppel som sier hvor vi er i presentasjonen og hva slags droppfaktor (presentasjonmodus) vi har:

$$s \ = < p, dropp >, \quad \text{p} \in N_0 \text{ og dropp} \in \{\text{-2, -1, 1, 2}\}$$




Figur 6.6: Interaksjonssettene FREMTIDIG, HISTORISK og DROPP med relevansverdier.

Her er p presentasjonspunktet, og dropp forteller hva slags presentasjonsmodus vi har. Verdiene til dropparameteren betyr henholdsvis rask baklengs -, baklengs -, vanlig -, rask avspilling. Ved henting av data viser dropp hvilken PE som presenteres neste gang. Ved rask avspilling av video ser vi at droppverdien er 2, og dette betyr at bare annenhver PE skal presenteres. I et slikt tilfelle henter vi derfor bare inn annenhver PE. For audio vil vi ikke hente noe data hvis ikke droppverdien er 1, men audio-PEer i bufferet får fortsatt en relevans for utbytting.

En PE kan være med i forskjellige interaksjonssett ($\mathrm{A}_s$) med relevansverdier avhengige av s. Interaksjonssettene inneholder PEer med tilhørende relevanser. For eksempel ser vi av figur 6.6 at

$$FREMTIDIG_{<18,1>} = \{(PE_{18}, 1), (PE_{20}, 1), (PE_{22}, 0, 975), (PE_{24}, 0, 9), ...\}$$

Tilsvarende blir de andre interaksjonssettene, men med andre PEer med andre relevansverdier.

For å gi hver PE en relevansverdi i et gitt interaksjonssett ($\mathrm{A}_s$) definerer vi en *distanserelevans-funksjon* ($\mathrm{dr}_{\mathrm{A}_s}$) som gir en PE en relevansverdi:

$$dr_{A_s} : N_0 \mapsto [0, 1]$$

$\mathrm{dr}_{\mathrm{A}_s}$(i) gir relevansverdien til en PE med distanse i til presentasjontpunktet p, og denne verdien beskriver hvor relevant denne PEen er for presentasjon.

Ved hjelp av $\mathrm{dr}_{\mathrm{A}_s}$ kan vi nå gi følgende definisjon av et interaksjonssett $\mathrm{A}_s$:

$$A_s = \{(c_j, dr_{A_s}(i)) \mid c_j \in PO, i \in N_0,\ j\ =\ g(i, s)\} \tag{6.1}$$

hvor hvilke $\mathrm{c}_j$ (PEer) som skal være med i settet bestemmes av en funksjon g som avhenger av distansen til presentasjonspunktet (i) og presentasjonsstatusen s. Relevansverdien til $\mathrm{c}_j$ gis av $\mathrm{dr}_{\mathrm{A}_s}$(i).

For så å sammenligne relevansene til PEer for hele POet definerer vi en *relevansfunksjon* ($r_{A_s}$) for et interaksjonssett ($A_s$):

$$\begin{aligned} r_{A_s} &: PO \mapsto [0,1] \\ r_{A_s}(c) &= \begin{cases} v, & (c, v) \in A_s \\ 0, & \text{ellers} \end{cases} \end{aligned} \tag{6.2}$$

Denne funksjonen gir en hvilken som helst c (en PE) i POet en relevans i forhold til et interaksjonssett ($A_s$). Hvis $c \in A_s$ får c relevansen den har i dette interaksjonssettet. Ved så å bruke relevansfinksjonen kan vi introdusere en funksjon ($r_{PO}$) som regner ut relevansverdiene til alle PEene i hele POet:

$$\begin{aligned} r_{PO,s} &: PO \mapsto [0,1] \\ r_{PO,s}(c) &= \max_{k=A_{1,s},\cdots,A_{n,s}} (r_k(c)) \end{aligned} \tag{6.3}$$

hvor $A_{1,s}, \cdots, A_{n,s}$ er de forskjellige interaksjonsettene. Funksjonen $r_{PO}$ beskriver relevansverdiene til alle PEene i POet ved å ta maksimum av relevansverdiene PEene har i de forskjellige interaksjonssettene.

### 6.7.3 Definisjon av interaksjonssettene

Selve definisjonen av interaksjonssettene varierer avhengig av applikasjonen og de forskjellige mediene. I de følgende underavsnittene definerer vi de forskjellige interaksjonssettene. Relevansfunksjonene for de forskjellige interaksjonssettene følger den generelle definisjonen gitt i (6.2).

#### 6.7.3.1 Potensielle fremtidig refererte presentasjonsenheter

PEene i interaksjonssettet FREMTIDIG kan hentes rett fra presentasjonsstatusen s, og relevansverdiene er bestemt av distanserelevansfunksjonen:

$$FREMTIDIG_s = \{(c_j, dr_{FREMTIDIG}(i)) \mid c_j \in PO, i \in N_0, j = p + i \times dropp\} \tag{6.4}$$

Her har vi bare satt inn verdier for funksjonen g i den generelle definisjonen i (6.1). Indeksen j til en PE bestemmes av presentasjonspunktet p, distansen til presentasjonspuntet i og verdien i dropparameteren dropp.

For å garantere kontinuitet i presentasjonen av en forelesning ønsker vi å sette relevansverdien til PEer "nær" presentasjonspunktet til 1, det vil si at disse PEene er høyt aktuelle for presentasjon og skal ikke byttes ut. Hva som igjen menes med nær presentasjonspunktet avhenger av hvor mye data vi ønsker (har mulighet til) å forhåndshente. En maksimum grense vil være det minste av POs lengde og den tilgjengelige bufferplassen. Distanserelevansfunksjonen for interaksjonssettet FREMTIDIG, $dr_{FREMTIDIG}$, defineres som følger:

$$dr_{FREMTIDIG}(i) = \begin{cases} 1, & 0 \leq \text{i} \leq \text{f} \\ 1 - \alpha \times i, & \text{f} + 1 \leq \text{i} \end{cases} \tag{6.5}$$

hvor f bestemer hva nær presentasjonspunktet p betyr, og $\alpha$ sier hvor fort relevansen skal avta når distansen til p blir større enn f. PEene i FREMTIDIG får verdien 1 hvis de ikke er lengre unna p enn f. Hvis distansen i til p er større enn f får PEene en relevansverdi som er avtagende etter hvor stor distansen er.

#### 6.7.3.2 Potensielle refererte presentasjonsenheter ved rettningsforandring av avspillingen

Ved forandring av retningen i avspillingen, trenger vi de PEer som akkurat har blitt presentert. PEen $c_j$ som kommer etter $c_p$ er med i FREMTIDIG. For å bedere støtte retningsforandringer gis derfor også refererte PEer en avtagende relevansverdi etter hvor stor distansen i er til presentasjonspunktet p:

$$HISTORISK_s = \{(c_j, dr_{HISTORISK}(i)) \mid c_j \in PO, i \in N_0, j = p - (i+1) \times \frac{dropp}{\mid dropp \mid}\} \quad (6.6)$$

hvor j telles opp eller ned avhengig av om avspillingen går frem- eller bakover.

Distanserelevansfunksjonen for interaksjonssettet HISTORISK, $dr_{HISTORISK}$, er en avtagende funksjon som defineres slik:

$$dr_{HISTORISK}(i) = 1 - \beta \times i \quad (6.7)$$

hvor $\beta$ sier hvor fort relevansen skal avta. For å støtte retningsforandringer enda bedere kan $dr_{HISTORISK}$ defineres noe à la $dr_{FREMTIDIG}$ med en relevansverdi på de nærmeste PEene lik en.

#### 6.7.3.3 Potensielle refererte presentasjonsenheter ved hastighetsforandring

Når vi har hurtige avspillinger i en av retningene, vil vi ha en droppverdi større enn en. I et slikt tilfelle vil ikke alle PEer være med i FREMTIDIG[7]. Likevel kan disse PEene være aktuelle for presentasjon ved en forandring i presentasjonsmodus, og de gis en relevansverdi i interaksjonssettet DROPP:

$$DROPP_s = \{(c_j, dr_{DROPP}(i)) \mid c_j \in PO, \mid dropp \mid > 1, i \in N_0, j = p + \frac{dropp}{\mid dropp \mid} \times (1 + i + \lfloor \frac{i}{\mid dropp \mid -1} \rfloor)\} \quad (6.8)$$

$DROPP_s$ inneholder alle PEer i presentasjonsretningen som ikke er med i $FREMTIDIGE_s$. Dette er en generell definisjon som gjelder for droppverdier større enn 2 og mindre en -2.

Distanserelevansfunksjonen for DROPP, $dr_{DROPP}$, lar vi være avtagende med $0 \leq dr_{DROPP}(i) \leq dr_{FREMTIDIG}(i)$ for alle $i \in N_0$:

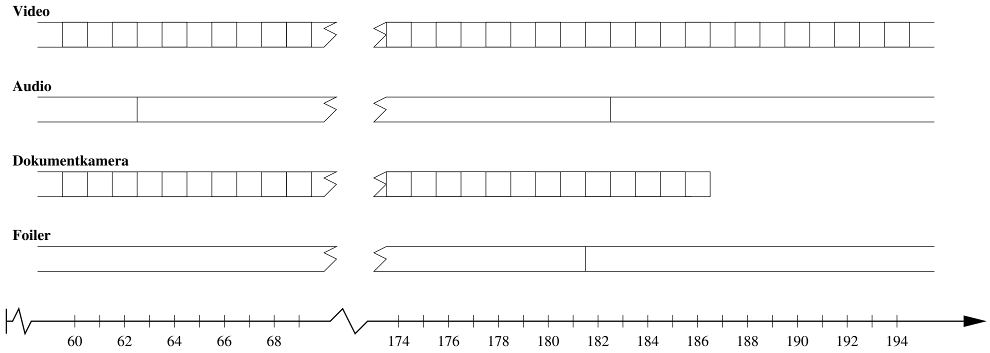
$$dr_{DROPP}(i) = 1 - \gamma \times i \quad (6.9)$$

hvor $\gamma$ sier hvor fort relevansen skal avta.

### 6.7.4 Sideutbyttings- og forhåndshentingsstrategi

Selve sideutbyttings- og forhåndshentingsstrategien baserer seg som sagt på å forhåndshente de PEer som er mest relevant for presentasjon. Hvis det ikke er ledig plass i bufferet byttes disse ut

[7] Dette gjelder ikke for tavlen da alle dataene her vises uansett.




Figur 6.7: De forskjellige mediene og tidsaksen delt inn i presentasjonspunkter.

med de PEer som har minst relevansverdi. Algoritmen kan beskrives slik:

Hent neste PE som skal presenteres(s)
**BEGIN**
  **FOR ALL** $c \in PO\ hvor\ r_{PO,s}(c) == 1$ **DO**
  // FORHÅNDSHENT DE MEST RELEVANTE PEENE
    **BEGIN**
      **IF** $c \notin BUFFER$ **THEN**
        **BEGIN**
        // BYTT UT DEN MINST RELEVANTE PE V MED RELEVANTE PE C
          $v \in \{c_j \mid c_j \in BUFFER, \min_{alle\ c_j\ i\ bufferet}(r_s(c_j))\}$
          $bytt\ ut\ v\ ved\ \ hente\ c\ inn\ i\ bufferet$
        **END**
    **END**
  **RETURN** $bufferadressen\ til\ PE$
**END**

Dette er en generell beskrivelse som brukes for alle typer POer. For video og dokumentamera fungerer den akkurat slik algoritmen over beskriver. For audio hentes bare nye PEer inn i bufferet hvis droppverdien er 1. Ved andre verdier settes bare relevansverdiene for en eventuell utbytting av PEene.

Når neste PE som skal presenteres er c, er $r_{\text{FREMTIDIGE}}(c)$= 1, det vil si at c er mest relevant for presentasjon og må forhåndshentes. Hvis vi ser bort i fra overføringstider fra disken og eksekveringstiden for selve algoritmen, garanterer derfor denne algoritmen at neste PE som skal presenteres ligger i bufferet. Dette kan vi imidlertid ikke gjøre, og dette vil bli tatt i betraktning under simuleringen i kapittel 7.

### 6.7.5 Presentasjonspunktet

Vi bruker bilderaten for video for oppdeling av sekundet, det vil siden vi trenger 24 bilder per sekund si at et sekund deles opp i 24. For verdiene til presentasjonspunktet p, vil dette bety at verdiene til p øker 24 ganger i sekundet eller hvert 41,67 ms. Ved å dele inn presentasjonspunktene slik, kan vi også velge størrelsene på en video-PE slik at vi best mulig kan utnytte algoritmens støtte for forskjellige presentasjonsformer. Figur 6.7 viser hvordan de forskjellige mediene er oppdelt etter presentasjonspunktene.

# Kapittel 7

# Realisering

Bufferhåndteringsmekanismen modellert i kapittel 6 er blitt "implementert" og testet i **MATLAB**[1] (matrix laboratory) [MathWorks 92] for å se om den vil fungere i vårt DHS for det elektroniske klasserommet. Vi har valgt å teste bufferhåndteringsmekanismen ved en simulering på grunn av at det slik er lettere å se på ytelsen til denne komponenten uten påvirkning av andre faktorer, for eksempel overføringer over nettverk og så videre, som ikke har noe å gjøre med selve håndteringen av bufferet.

Et ønske med dette arbeidet er å komme frem til et reelt system hvor vi ender opp med en realisering av et multimedia-DHS for det elektroniske klasserommet slik det er beskrevet i kapittel 3. For en endelig realisering av et slikt system, har vi sett på OSet CHORUS som er et mikrokjernebasert OS som er godt egnet for distribuerte systemer, og vi beskriver derfor hvordan det ville være å realisere vår bufferhåndteringsmekanisme i dette OSet.

I dette kapittelet beskriver vi først enkelte aspekter ved en implementasjon og realisering av algoritmen (avsnitt 7.1), og videre ser vi på simuleringsomgivelsene hvor vi beskriver systemet med lagringsstrukturer og oppbyggning i hardware (avsnitt 7.2). Deretter kommer vi inn på hvordan vi har beregnet eksekveringstider i algoritmen (avsnitt 7.3) og hvordan vi har konstruert testdataene til simuleringen (avsnitt 7.4). De to siste avsnittene som beskriver simuleringen tar for seg parametere til selve algoritmen (avsnitt 7.6) og parametere som forandres under simuleringen (avsnitt 7.4).

Til slutt i dette kapittelet beskriver vi de viktigste egenskapene i OSet CHORUS og hvilke muligheter det ville være for å realisere et multimedia-DHS for det elektroniske klasserommet og bufferhåndteringsmekanismen (avsnitt 7.7).

## 7.1 Implementering av bufferhåndteringsmekanismen

Algoritmen slik den er beskrevet i avsnitt 6.7.4 vil ha en total overhead på O(bufferstørrelse) siden algoritmen må gå igjennom alle elementene i bufferet for hver av PEene som skal forhåndshentes. Siden det er en forholdsmessig kostbar algoritme, går vi ut i fra at algoritmen kjøres etter en bestemt tidsperiode slik at vi rekker å hente nye data i tide for presentasjon eller hver gang vi får en sidefeil.

Til denne simuleringen implementeres mekanismen slik at vi kjører forhåndshentingsalgoritmen hvert halve sekund[2] eller hver gang vi har en sidefeil. Dette gjør at data for de neste to sekundene fra der hvor vi er i presentasjonen hentes hvert halve sekund, det vil si at vi har et og

[1]MATLAB er et interaktivt system og programeringsspråk for vitenskaplige og tekniske numeriske beregninger, det vil si et matematisk modelleringsverktøy for simuleringer.

[2]Hvis det å hente dataene tar over et halvt sekund, begynner den bare å hente nye data med en gang den er ferdig. Hvis det derimot tar over et sekund, vil vi etterhvert få en sidefeil.

et halvt sekund på å hente disse dataene. Hele bufferhåndteringsmekanismen fungerer dermed slik at data forhåndshentes etter et bestemt tidsrom slik at dataene allerede finnes i bufferet når det er behov for dem. Hvordan algoritmen er realisert i matlab-kode vises i appendiks B.

## 7.2 Simuleringsomgivelser

I denne simuleringen antar vi at vi har et multimedia-DBS liggende på en sentralisert server hvor alle de lagrede dataene fra forelesningene ligger. Brukere (klienter) kobler seg så opp mot denne serveren og ber om å få avspilt forelesninger, det vil si at vi får en klient/server-arkitektur. Det undersøker i denne oppgaven er bufferet mellom disken og prosessoren på serveren hvor DBSet ligger, det vil si at vi sjekker om data kan leveres raskt nok videre til neste komponent i systemet slik at ikke bufferhåndteringen blir en flaskehals for støtte av kontinuerlige fremvisninger.

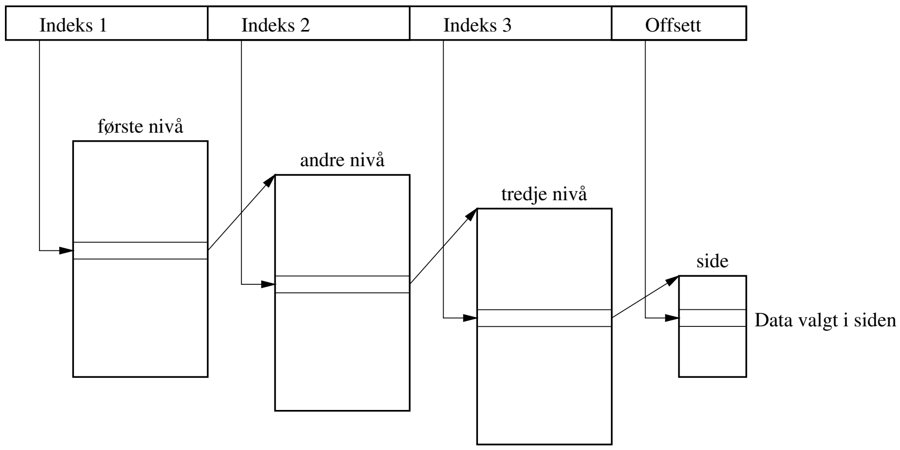
I denne simuleringen antar vi at den reele serveren er satt opp på en UltraSPARC-maskin med en 200 MHz prosessor. Videre antar vi at DRAM-brikkene har en hastighet på 60 ns, og som sekundær lagringsenhet tenker vi oss disken SEAGATE ELITE 23 (tabell 6.1).

### 7.2.1 Lagringsstruktur

I vår simulering av en bufferhåndteringsmekanisme i et DHS for det elektroniske klasseromscenarioet, antar vi at alle medietypene ligge på samme disken. Videre antar vi at de enkelte mediene ligger lagret sammen slik at for eksempel all videodata er lagret kontinuerlig på disken. Bufferet er delt av samtidige brukere og av forskjellige medier.

### 7.2.2 Oppbyggingen av sidetabellen i hardware

Til vårt DHS for det elektroniske klasserommet antar vi at vi har en sidetabell à la den SPARC har i sin sideutbyttingsmekanisme [Tanenbaum 92]. Dette er en tre-nivås sidetabell som vist i figur 7.1 hvor vi som tidligere nevnt har et felles, delt buffer for alle medier og brukere. Her har vi tre nivåer med sidetabeller hvor man ved å bruke *indeks 1* finner ut hvilken sidetabell som skal brukes i det andre nivået. *Indeks 2* gir oss sidetabellen i det tredje nivået, og *indeks 3* gir oss hvilken side vi skal ha i bufferet.




Figur 7.1: SPARCs sidetabell på tre nivåer.

Ved å bruke en slik struktur får vi et mye mindre plassforbruk til selve sidetabellene, men vi får flere minneaksesser ved at vi må aksessere minnet en gang får hvert nivå med sidetabeller.

## 7.3 Eksekveringstider i algoritmen

Som vi så i avsnitt 7.2.2, antar vi at vi har en tre-nivås struktur på sidetabellen uten noe bruk av assossiativ minne. Dette gjør at hver virtuelle adresse må mappes ned til forskjellige sidetabeller på hvert nivå. Vi får med dette tre minnereferanser, en for hvert nivå, for hvert søk etter en side i sidetabellen. Hvis vi igjen bruker aksesstiden vi har her for DRAM får vi en aksesstid til minnet på 60 ns, det vil si at vi får en søketid i sidetabellene på $3 \times 60$ ns = 180 ns. I tillegg får vi eksekveringstiden for utregningen av adressene til de forskjellige sideoppslagene i tabellene.

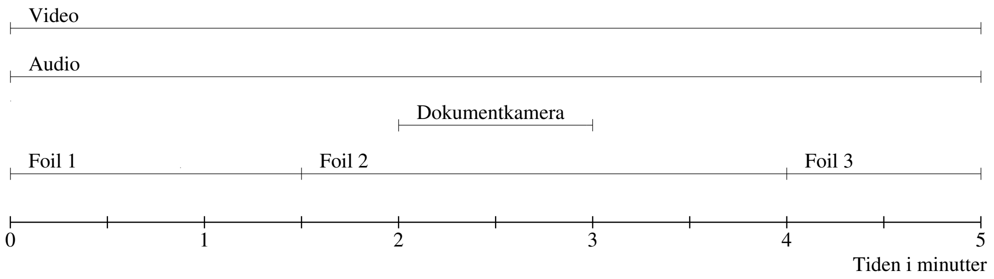
Tidene for diskaksesser og overføringstider for data fra disken beregnes etter disken SEAGATE ELITE 23 som vist i tabell 6.1. Når vi beregner overføringstider, antar vi at vi har lagret dataene på disken kontinuerlig etter hverandre ("clustering") slik at når vi henter dataene får vi bare en søketid. En del av systemet som vi ikke har tatt med i beregningene er at vi ikke har sett på kapasiteten til en eventuell buss mellom disken og bufferet. Vi antar i denne simuleringen at en slik buss har stor nok kapasitet og ikke vil gi noen forsinkelser i dataleveringene.

Eksekveringstiden for selve algoritmekoden er beregnet ved hjelp av programmet "Quantify". "Quantify" er en software-overvåker som estimerer eksekveringstid i blant annet antall klokkesykler brukt i prosessoren. "Quantify" gjør om koden til tilsvarende assemblerkode og beregner antall sykler etter hver assemblerinstruksjon. Siden antall sykler beregnes etter at en bestemt intruksjon tar et visst antall sykler, blir ikke dette helt nøyaktig, men kan likevel brukes som et godt estimat for eksekveringstidene. Videre blir selve eksekveringstiden beregnet etter antall klokkesykler og prosessorens hastighet.

Eksekveringstidene som beregnes slik vi har beskrevet over puttes så inn i simuleringskoden i matlab. Her "måler" vi tiden det tar å bruke bufferhåndteringsmekanismen hver gang den eksekveres. Videre ser vi så etter om dataene for presentasjonen i den tiden vi eksekverer algoritmen ligger i bufferet, slik at vi ikke får et avbrudd i fremvisningen av forelesningen.

## 7.4 Simuleringsdata

Bufferhåndteringsmekanismen slik den er presentert over er som tidligere nevnt blitt implementert i MATLAB hvor vi har konstruert et typisk forelesningsklipp og enkle, typiske referansestrenger til denne forelesningen.




Figur 7.2: En modell av den simulerte forelesningen.

### 7.4.1 Forelesningsklipp

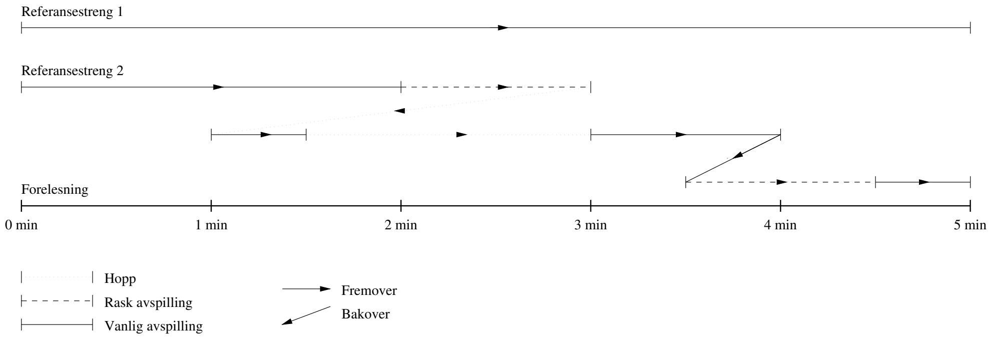
For simuleringen av vår bufferhåndteringsmeanisme, har vi, som vist i figur 7.2, generert et fem minutters forelesningsklipp[3] bestående av data fra video, audio, dokumentkamera og tavlen. Fo-

[3] Vi har ikke konstruert noe lenger forelesningsklipp siden referansestrukturen og datamengdene er av samme type og størrelse under en hel forelesning, det vil si at den referansestrengen vi har konstruert på fem minutter er representativ for en hel forelesing.

relesningen består av kontinuerlig presentasjon av video og audio, samt at tavlen viser en foil hele tiden. Dokumentkameraet starter etter to minutter og sender data i et minutt fremover.

### 7.4.2 Referansestrenger

Vi har, som vist i figur 7.3, generert to forskjellige referansestrenger til vår simulering. *Referansestreng 1* er en helt vanlig avspilling av forelesningen fra start til stopp som et VoD-scenario. *Referansestreng 2* inneholder alle de operasjonene vi beskrev i avsnitt 6.1 som vi antar at en student har behov for ved en fremvisning av en forelesning.




Figur 7.3: De forskjellige referansestrengene i simuleringen av forelesningen.

## 7.5 Parametere til sideutbyttingsalgoritmen

Relevansfunksjonene definert i avsnitt 6.7.2 er generelle uten bestemte verdier for hvordan relevansen til en PE settes. For de kontinuerlige mediene tar vi utgangspunkt i visning av en video i PAL-kvalitet hvor ventetiden ved begynnelsen av presentasjonen og etter en interaksjon er maksimum to sekunder [Moser et al. 95]. Vi prøver derfor å forhåndshente to sekunder med data hver gang vi henter data fra disken, det vil si at alle PEene som ikke ligger lenger unna en to sekunder fra presentasjonspunktet får relevansverdi lik en. PEer som ligger lengre frem i tid enn to sekunder lar vi få en monotont avtagende relevansverdi hvor relevansverdien null settes på PEer som er ti minutter eller lengere unna presentasjonspunktet.

Videre kom det frem i avsnitt 6.7.4 at algoritmen vi modellerte i forrige kapittel, er ganske kostbar hvis den kjøres for ofte. I denne simuleringen vil vi derfor prøve å kjøre den hvert halve sekund[4] og eventuelt ved hver sidefeil.

For å sette relevanser for historiske PEer og PEer som droppes ved hurtige avspillingsformer, lar vi relevansen avtar monotont ettersom avstanden blir lengere til presentasjonspunktet, og når avstanden i tid blir et halvt minutt settes relevansverdien til null.

For tavlen setter vi data som ligger to sekunder frem i tid i forhold til presentasjonspunktet som høyt relevant for presentasjon slik at vi begynner å hente foilene to sekunder før de skal brukes. Relevansen til foilene som tilhører HISTORISK eller som ligger mer enn to sekunder frem i fremvisningen avtar monotont etter hvor mange foiler det er mellom foilen som presenteres og foilen som ligger i bufferet. Når det er mer enn to foiler mellom blir relevansen satt til null.

[4]Siden det da skal ligge data for et og et halvt sekund i bufferet, har vi denne tiden på å hente de neste to sekundene med data.

## 7.6 Parametere til simuleringen

I denne simuleringen prøver vi forskjellige side- og bufferstørrelser, samt både enbruker- og flerbrukermodus til våre testdata. Hvordan vi varierer disse parameterene i simuleringene beskives i avsnittene under.

### 7.6.1 Buffer- og sidestørrelser

Som det er beskrevet i avsnitt 6.3 bør bufferet være så stort som mulig. Videre vil vi som nevt i avsnitt 7.5 forhåndshente to sekunder med data. Dette gir oss et minstemål på bufferet. Hver bruker bør minst kunne ha circa fire sekunder med data i bufferet til enhver tid. Ut i fra de datatypene vi simulerer med fra det elektroniske klasserommet vil dette si at vi må kunne ha $4 \times 24 \times 60$ KBytes = 5,76 MBytes med videodata fra videokameraene, $4 \times 32$ KBytes = 0,128 MBytes audiodata, $4 \times 6 \times 60$ KBytes = 1,44 MBytes med videodata fra dokumentkameraet og $2 \times 200$ KBytes = 0,4 MBytes data for to foiler. Dette gir en samlet bufferstørrelse på 7,73 MBytes. Ved flere samtidige brukere kan dette kravet reduseres noe ved å anta at flere brukere bruker de samme dataene.

Fra avsnitt 6.3 kom det videre fram at tradisjonelle systemer har sidestørrelser fra 4 KBytes og oppover, og vi så at MMSer bør ha noe større sidestørrelser enn dette.

I denne simuleringen prøver vi forskjellige størrelser på sidene og bufferet, og vil prøve forskjellige variasjoner av sider på 8 KBytes, 16, KBytes og 32 KBytes og buffere på 16 Mbytes, 32 MBytes, 64 MBytes og 128 MBytes[5].

### 7.6.2 Samtidige brukere

Tallet på antall samtidige brukere vi kan ha i et slikt elektronisk klasserom-scenario er vanskelig å forutsi. Vi har derfor simulert to mulige scenarioer. Det ene er når vi bare har *en* bruker inne av gangen, og det andre er et scenario hvor vi har *tre* samtidige brukere hvor disse starter fremvisningen av forelesningen et og et halvt minutt etter hverandre.

## 7.7 Realisering i Chorus

Et mål med arbeidet rundt det elektroniske klasserommet er å virkeliggjøre bufferhåndteringsmekanismen og DHSet for det elektroniske klasserommet. CHORUS, som beskrives under, har allerede mange av de egenskapene som trengs for å støtte multimediaapplikasjoner.

Opprinnelig var målet å implementere bufferhåndteringsmekanismen i CHORUS, men på grunn av manglende dokumentasjon og for sene leveranser fra Chorus Systèmes hadde ikke riktig versjon av dette OSet blitt installert i tide. I avsnittene under beskrives de viktigste trekkene ved CHORUS og eventuelt hvilke mekanismer som ville være godt egnet å bruke i vårt DHS for det elektroniske klasserommet og spesielt for bufferhåndteringsmekanismen.

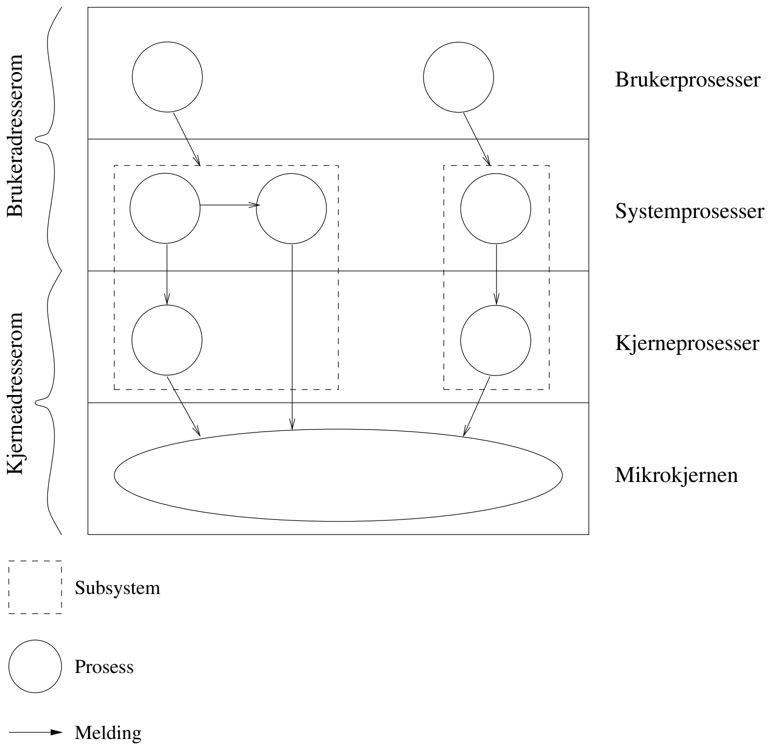
### 7.7.1 Operativsystemet Chorus

CHORUS [Rozier et al. 91, Tanenbaum 95] er et mikrokjerne-basert OS for bruk i distribuerte systemer. Det startet som et forsøksprosjekt i distribuerte OSer og har etter hvert utviklet seg gjennom flere nye versoner til et UNIX-emulerende OS. CHORUS er nå binær kompatibel med *System V* UNIX og har mulighet for sanntidsapplikasjoner og objektorientert programering. Dette avsnittet inneholder derfor en kort beskrivelse av CHORUS-V3.

[5] Som tidligere nevnt innvirker bufferstørrelsen, det vil si antall bufferplasser, på eksekveringstiden av vår bufferhåndteringsmekanisme. Store buffere bør derfor ha store sidestørrelser slik at ikke overheaden med eksekveringen av selve algoritmen blir for kostbar.

#### 7.7.1.1 Systemets oppbygning

CHORUS er som vist i figur 7.4 strukturert i fire lag: mikrokjernen, kjerneprosesser, systemprosesser og brukerprosesser.




Figur 7.4: Systemstrukturen til CHORUS er lagdelt

**Mikrokjernen** ligger i bunn og har minimal funksjonalitet for styring av prosesser (avsnitt 7.7.1.3), thread'er (avsnitt 7.7.1.4), minne (avsnitt 7.7.1.6) og kommunikasjon, og den er lik på alle maskiner i det distribuerte systemet. Håndteringen av disse ressursene gjøres ved systemkall til mikrokjernen. Det er 112 slike systemkall, og disse danner grensesnittet til kjernen av OSet. Det er gjennom systemkallene vi får tilgang til kjernens tjenester.

Siden mikrokjernen har minimal funksjonalitet, må resten av OSet ligge i høyere lag. Til dette har vi **kjerneprosesser**. Disse er bygd på toppen av mikrokjernen og er for håndtering av ting som for eksempel disker, mus og printere. Kjerneprosessene kan legges inn dynamisk, det vil si at når systemet startes sjekkes det hva som finnes av hardware og bare de kjerneprosessene som trengs legges inn. Kjerneprosessene kan også fjernes under kjøring.

Ved å bruke kjerneprosesser i tillegg til mikrokjernen, kan vi holde mikrokjernen liten samtidig som vi kan utvide funksjonaliteten til kjernen ved å legge til kjerneprosesser uten å øke størrelse og kompleksitet på kjernen permanent. Sammen med mikrokjernen, utgjør kjerneprosessene det vi kaller *kjernelaget*.

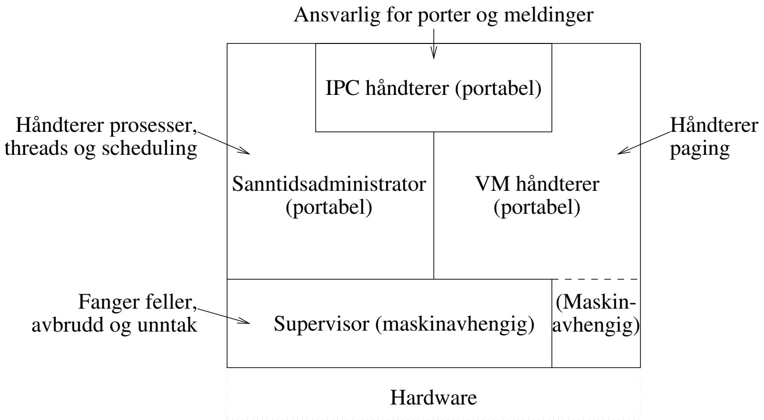
Figur 7.4 viser også hvor i adresserommet prosessene eksekveres. Mikrokjernen og kjerneprosessene kjører i kjernens adresserom mens system- og brukerprosesser kjører i brukeradresserommet. Kjerneprosessene kan kommunisere med mikrokjernen og andre kjerneprosesser. Systemprosessene kan kommunisere med andre systemprosesser, kjerneprosesser og i enkelte tilfeller sende kall til mikrokjernen. Brukerprosessene kommuniserer ofte med et helt subsystem som har et veldefinert grensesnitt til brukeren. De hele pilene i figuren viser kommunikasjonen i systemet.

| | |
|---|---|
| **Prosess** | Enhet for ressursallokering, samler og styrer ressurser |
| **Thread** | Enhet for sekvensiell eksekvering, de aktive delene i systemet og får fordelt prosessortid av kjernen som bruker en prioritetbasert fordelingsalgoritme |
| **Regioner** | Enhet for strukturering av en prosess' adresserom, sammenhengende områder av det virtuell adresserommet |
| **Meldinger** | Enhet for kommunikasjon |
| **Porter, portgrupper** | Enhet for adressering, buffer for innkommende meldinger |
| **Unike identifikatorer** | Et globalt navn, navngir ressurser |
| **Segment** | Enhet for datainnkapsling |
| **Capability** | Enhet for dataaksesskontroll, gir globle objekter et navn |
| **Beskyttlesesidentifikator** | Enhet for identifisering |

Tabell 7.1: Basiske abstraksjoner implementert og håndtert av kjernen (og i sammarbeid med subsystemer).

#### 7.7.1.2 Abstraksjoner

Mikrokjernen styrer prosesser, thread'er, regioner, meldinger, porter og unike identifikatorer (UI-er) som sammen danner basisen for CHORUS. Øverste del av tabell 7.1 viser disse seks abstraksjonene samt deres oppgave i systemet. Nederste del av tabellen viser tre andre abstraksjoner som håndteres av kjernen og subsystemene sammen.




Figur 7.5: Strukturen til mikrokjernen i CHORUS.

Mikrokjernen er som vist i figur 7.5 delt inn i fire deler. I bunnen har vi en **supervisor** som i grove trekk håndterer hardware'en og fanger opp feller (traps), unntak (exceptions) og avbrudd (interrupts). Den **virtuell minnemanageren** er ansvarlig for lav-level delen av sidehåndteringsystemet, det vil si at den manipulere hardware'en til det virtuelle minnet og lokale minneressurser. **Sanntidsadministratoren** styrer prosesser, prosessorallokering, thread'er, synkronisering og scheduling, mens **interprosess kommunikasjonsmanageren** holder orden på meldinger, porter og UIer.

Mens sanntidsadministratoren og interprosess kommunikasjonsmanageren er portable til andre maskinarkitekturer, er supervisor delen er maskinavhengig og må forandres når CHORUS legges inn på ny hardware. Den virtuelle minnemanageren er som vist av den stiplede linjen i figur 7.5 delvis portabel.

CHORUS har tre forskjellige "innganger" til kjernen. **Feller** er kall til kjernen for å be om å få utført diverse tjenester. Til dette brukes systemkall. **Unntak** er uventede, gale hendelser som skjer ved uhell. Dette kan være divisjon med null eller heltalls overflod ved for eksempel at vi bruker et tall større en $2^{15}$ (32767) for et *heltall* (integer) i C på 16 bits. Den tredje og siste inngangen til kjernen er **avbrudd**. Her er det asynkrone hendelser, for eksempel et ferdig I/O-kall, som ikke nødvendigvis har noe med den kjørende thread'en å gjøre som er årsaken.

#### 7.7.1.3 Prosesser

En prosess i CHORUS er ikke som prosesser i andre OSer som beskrevet i avsnitt 4.1.1. I CHORUS er prosess en samling av aktive og passive elementer som sammen skal utføre en beregning. De aktive elementene er thread'er, og de passive elementene er adresserom og porter. En prosess med *en* thread er som en tradisjonell UNIX-prosess. En prosess uten noen thread'er kan ikke gjøre noen ting.

CHORUS har som vist i tabell 7.2 tre typer prosesser med forskjellige privilegier for å utføre beskyttede instruksjoner som I/O og ulik tillit til å aksessere kjernen direkte. Den høyest prioriterte prosesstypen er **kjerneprosessene** som har tillatelse til å gjøre kjernekall og utføre beskyttede instruksjoner. De kjører i kjernemodus og deler adresserom med alle de andre kjerneprosessene. Kjerneprosessene kan sees på som en dynamisk utvidelse av mikrokjernen.

| Prosesstype | Priviligert | Tillit | Modus | Adresserom |
|---|---|---|---|---|
| *Bruker* | Nei | Nei | Bruker | Bruker |
| *System* | Nei | Ja | Bruker | Bruker |
| *Kjerne* | Ja | Ja | Kjerne | Kjerne |

Tabell 7.2: De tre prosesstypene med privilegier, tillit, modus og adresserom.

**Systemprosessene** får ikke eksekvere beskyttede instruksjoner, men kan gjøre kall direkte til kjernen slik at den har direkte tilgang til kjernens tjenester. De kjører i brukermodus og benytter brukerens adresserom.

Til slutt har vi **brukerprosessene** som er den lavest prioriterte prosesstypen. Disse er upriviligerte og har ikke tillit. Dette betyr at de ikke kan direkte aksessere kjernen og at de ikke kan utføre beskyttede instruksjoner. Som for systemprosessene, kjører brukerprosessene i brukermodus og benytter brukerens adresserom.

#### 7.7.1.4 Thread'er

En aktiv prosess i CHORUS må inneholde en eller flere thread'er. En **thread** er en slags liten miniprosess (ofte kalt "light-weight process") i prosessen og er det eksekverende elementet i prosessen. Hver thread er en uavhengig kontrollenhet som eksekveres i parallell som om den var en egen prosess.

Thread'ene i en prosess deler prosessens adresserom. Det vil si at de deler de samme globale variablene, men hver thread har sin private programteller, egen stakk for å holde orden på hvor den er i eksekveringen og egne registre. Disse lagres når en thread blokkeres og hentes frem igjen når thread'en blir aktiv.

Ved å la en prosess ha flere thread'er, kan en prosess fortsette å kjøre selv om en thread blir blokkert for eksempel av venting på lesing fra en disk. En annen thread kan overta kontrollen. Det blir mulig å opprettholde blokkerende systemkall (som gir enklere programmering) i sekvensielle prosesser og likevel beholde parallellitet. Parallellisme gir bedere ytelse og data gjennomstrømming enn for en prosess med en thread som ville ha blitt blokkert.

Thread'ene eksekveres i den modusen og i det adresserommet som tilhører sin egen prosess. Med andre ord kjører thread'ene til en kjerneprosess i kjernemodus, mens en brukerprosess' thread'er utføres i brukermodus. Thread'ene tilhører den prosessen de ble opprettet i og kan ikke flyttes til andre prosesser.

Det er kjernen som bestemmer hvilken thread som får kjøre på prosessoren, slik at når en thread blokkeres, kan en annen starte. Overheaden med dette er at vi må sende kall til kjernen for å opprette og terminere thread'ene, sammen med ekstra arbeid med å holde orden på dem.

I CHORUS kan en thread ha en eller flere av fire tilstander [Tanenbaum 95]. Disse er:

1. AKTIV. Thread'en er logisk klar til å kjøre. Den kjører, eller den venter bare på at prosessoren skal bli ledig slik at den kan få kjørt.
2. SUSPENDERT. Thread'en har blitt suspendert av en annen thread (eller seg selv).
3. STOPPET. Selve prosessen som threaden tilhører er blitt suspendert. Når dette skjer, blir alle thread'ene i prosessen stoppet.
4. VENTENDE. Thread'en venter på at noe, for eksempel lesing fra disk, skal skje og er blokkert til dette er ferdig.

Tilstand 1 er eksklusiv. Tilstandene 2 - 4 er avskilte, men ikke eksklusive, det vil si at vi kan ha flere tilstander av en thread samtidig.

Det er flere grunner til at CHORUS bruker thread'er [Rozier et al. 91]. Thread'er er et kraftig verktøy for programmering av I/O-drivere, servere kan multiprogrammeres slik at vi får samtidighet, og at vi kan bruker flere prossessorer til en enkelt prosess. Sist, men ikke minst er en "context switch" mellom thread'er mye mindre kostbar enn en "context switch" mellom prosesser på grunn av at thread'er deler data slik at vi ikke bruker tid på å skifte ut data i registere og så videre.Thread'er gir oss derfor en billig mekanisme for å få samtidighet innen et multiprogram.

#### 7.7.1.5 Scheduling

Som tidligere nevnt ordner kjernen fordelingen av prosessorressursene. Til dette brukes en avbrytbar, prioritetsbasert schedulingalgoritme.CHORUS støtter videre tidsdeling og prioritetsdegradering på per-thread basis, noe som sammen med schedulingalgoritmen tillater at samme kjerne støtter sanntidsapplikasjoner og interaktive flerbrukeromgivelser. Hver thread får en prioritet relatert til sin prosess og egen prioritet innen prosessen. Kjernen holder orden på hvilke thread'er som er AKTIVE, og lar de med høyest prioritet kjøre.

Thread'ene deles grovt inn i to grupper, og thread'ene i disse gruppene rangeres igjen etter prioritet. De som har **høy** prioritet fortsetter å kjøre til den enten blir stoppet av en thread med høyere prioritet, eller den avslutter av seg selv. Disse thread'ene tidsdeles ikke. Thread'ene som har **lav** prioritet bruker i tillegg til bare prioritetbasert scheduling en tidsdeling av prosessortiden slik at hver thread bare har en hvis tid av gangen med prosessoraksess. Det finnes imidlertid også systemkall som kan sette prioritet, slik at brukeren kan fortelle hvilke prosesser som må kjøres først.

#### 7.7.1.6 Minnehåndtering

Minnehåndteringen i CHORUS er basert på abstraksjonene **regioner** og **segmenter**. Adresserommet til en prosess er delt opp i regioner. En region består av en kontinuerlig rekke med virtuelle adresser og kan i teorien starte og slutte hvor som helst i adresserommet. Likevel bør en region være sideregulert og bestå av et visst antall hele sider. Alle dataene i regionen må ha samme beskyttelse (for eksempel bare lesing, ikke skriving), og region tilhører en prosess og kan leses av denne prosessens thread'er.

Et segment som er et sekundert lagringsobjekt, for eksempel kan være en fil, er en lineær sekvens av bytes identifisert av en *capability*. En capability i CHORUS består av en UI og en nøkkel. De kan bli lest og skrevet ved hjelp av tradisjonelle systemkall for I/O-operasjoner eller ved å mappe en del av segmentet inn i en region. Til denne mappingen brukes en **mapper** som er en ekstern sidehåndteringsmekanisme. Hver mapper kontrollerer et eller flere segmenter, og hvert segment kan være avbildet i flere regioner på samme eller forskjellige maskiner. En protokoll mellom kjernen og mapperen styrer strømmen av sider. Hvis en thread får en sidefeil, sjekker kjernen et eget hurtigbuffer for sider. Hvis siden ikke er der, gir kjernen en mapper en forespørsel om å få dette segmentet. Mapperen sjekker så sitt eget adresserom. Ved en eventuell miss her, gir mapperen en melding til diskkontrollen om å hente segmentet. Når mapperen har segmentet hos seg, gir den kjernen beskjed som igjen tilslutt starter opp thread'en som er suspendert.

CHORUS har også støtte for *sidebasert delt distribuert minne*, og også her er enheten for deling segmenter. Segmentene blir splittet opp i fragmenter på en eller flere sider. Hvert slikt fragment er enten bare lesbart og kan finnes på flere maskiner samtidig, eller det er både leslig og skrivbart og finnes da bare på en maskin. Hver prosessor kan direkte aksessere det globale adresserommet, slik at et segment enkelt kan hentes fra andre maskiner.

### 7.7.2 Realisering av bufferhåndteringsmekanismen i Chorus

I dette avsnittet beskriver vi hvordan vi kan implementere bufferhåndteringsmekanismen som er blitt simulert i MATLAB i CHORUS.

#### 7.7.2.1 Prosesser og thread'er

For det første kan CHORUS' prosesser og thread'er forbedre eksekveringstider av selve algoritmen i forhold til en implementasjon i et annet, "tradisjonelt" OS som for eksempel UNIX. En presentasjon av en forelesning består av flere datastrømmer, og slik vi har implementert mekanismen for simuleringen, går algoritmen sekvensielt gjennom hver datastrøm. Hver gang algoritmen sjekker om data er i bufferet eller vi skal forhåndshente data, sjekker algoritmen først video. Deretter finner den dataene henholdsvis for audio, dokumentkameraet og tavlen. Ved bruk av thread'er kan vi øke ytelsen til bufferkomponenten, for hver gang data hentes for video, blir algoritmen stående å vente til videodataene er hentet fra disken til bufferet. Har vi imidlertid implementert algoritmen med threads, kan vi opprette en thread for hver datastrøm. Når en thread suspenderes for å vente på I/O fra disken, kan thread'ene for de andre datastrømmene eksekveres, og vi får bedre parallellitet i eksekveringen av algoritmen.

#### 7.7.2.2 Scheduling

Vanlige scheduling algoritmer skal være like rettferdige mot alle prosesser. De lar alle prosesser få tildelt like mye prosessortid, vil ofte ikke kunne støtte eksekvering av prosesser med sanntidskrav slik som multimediaapplikasjoner har. Vi har i vår eksempelapplikasjon kontinuerlige datastrømmer med tidsfrister for levering av resultater, og vi trenger derfor en schedulingmekanisme som kan garantere at vi får nok prosessortid. Ettersom CHORUS støtter sanntidsprosessering og har en prioritetsbasert scheduling, kan en applikasjone med sanntidskrav gis en høy prioritet slik at vi kan få den prosessortiden vi trenger.

Dette med å gi prosessene en høy prioritet gjelder i høyeste grad vår bufferhåndteringsmekanisme. Hvis eksekveringen av denne skulle bli avbrutt, vil den reele leveringstiden ikke holde for å støtte de kravene som applikasjonen stiller til denne komponenten. Dette kan imidlertid unngås ved å gi prosessen en høy prioritet.

Videre kan også dette utnyttes til å gi prioriteter for hver datatype. Som tidligere nevnt er det viktigst å støtte kontinuerlig presentasjon av audio. Ved å parallellisere algoritmen som beskrevet i avsnitt 7.7.2.1, kan hver thread igjen gis forskjellig prioritet slik at de viktigste dataene hentes først.

#### 7.7.2.3 Minnehåndtering

Vi har delt opp forelesningen i POer som består av en medietype. CHORUS' segmenter som er et sekundært lagringsobjekt, kan brukes til lagringen av hver PO. På denne måten får vi en naturlig innkapsling av data som hører sammen.

Videre tenker vi oss at for hver presentasjon av en forelesning opprettes det en prosess og innen denne prosessen opprettes det en thread for hver bruker som ønsker å se på denne presentasjonen. Vi får da en felles, delt region for presentasjon av en forelesning, det vil si at alle brukere som ser på samme forelesning deler bufferplass og kan dele data slik at samme data ikke okkuperer mer enn en bufferplass.

For å legge inn vår sideutbyttingsalgoritme, benytter vi CHORUS' mapper-mekanisme som håndterer datahentingen fra en eller flere segmenter inn i regionene. Denne mapperen implementerer vi så til å bytte ut sider i regionen etter L/MRP.

# Kapittel 8

# Simuleringsresultater

I dette kapittelet beskrives resultatene fra simuleringen av bufferhåndteringsmekanismen som ble modellert i kapittel 6 og "implementert" i matlab som beskrevet i kapittel 7. Det første vi undersøkte var om en kontinuerlig presentasjon av forelesningen kunne støttes av bufferhåndteringsmekanismen slik at ikke denne komponenten ble en flaskehals i MMSet og kvaliteten på tjenesten ikke måtte reduseres på grunn av dårlig datagjennomstrømming og lange responstider i bufferet.

Først undersøker vi om forhåndshenting av data slik det er beskrevet i kapitlene før, kan støtte kontinuerlig visning av forelesningen (avsnitt 8.1). Videre sammenligner vi L/MRP med tradisjonelle sideutskiftningsstrategiene LRU og vilkårlig i antall sideutskiftninger i bufferet (avsnitt 8.2), og tilslutt tar vi for oss den totale tiden det tar å hente data fra disken til bufferet (avsnitt 8.3).

## 8.1 Forhåndshentingsmekanismer

Her undersøker vi om vår forhåndshentingsstrategi i L/MRP kan støtte en kontinuerlig presentasjon av en forelesning fra vårt DHS for det elektroniske klasserommet. Vi tester om vi får sidefeil i avspillingen, det vil si vi ser bort i fra de sidefeilene som oppstår ved en interaksjon fra brukeren hvor presentasjonsmoduser forandres eller vi har hopp. Interaksjoner fra brukeren vil alltid gi sidefeil, og sammenligner vi igjen på den europeiske TV-standarden PAL, godtar vi et opphold i presentasjonen ved interaksjoner på opptil to sekunder (avsnitt 7.5).

### 8.1.1 Avspilling uten interaksjoner

Her har vi simulert en situasjon hvor brukeren(e) vil se en forelesning i sin helhet uten andre interaksjoner enn det å starte avspillingen av forelesningen.

#### 8.1.1.1 Enbrukerscenario

For alle de forsjellige scenarioene med side- og bufferstørrelsekombinasjoner vi har simulert her, viser det seg at med forhåndshenting, kan data til forelesningen leveres raskt nok til å støtte en kontinuerlig presentasjon uten forstyrrelser og feil. Dette betyr at vi ikke fikk noen sidefeil utenom i starten av presentasjonen og at i dette scenarioet vil ikke bufferhåndteringsmekanismen være noen flaskehals i systemet som en enhet.

Videre viser tabell 8.1[1] responstidene i sekunder fra brukeren startet forelesningen til fremvisningen kunne begynne. Alle responstidene fra simuleringene i dette scenarioet var på under et

[1] "-" i tabellene i dette kapittelet betyr at vi ikke har simulert dette scenarioet. Dette er scenarioer hvor vi får unaturlig små sidestørrelser i forhold til bufferstørrelsen.

halvt sekund. I dette tilfellet vil altså buffermekanismen kunne levere data raskt nok til å støtte PALs krav til responstider.

| Buffer-/Sidestørrelse | 8 KBytes | 16 KBytes | 32 KBytes |
|---|---|---|---|
| **16 MBytes** | 0,38 | 0,36 | 0,35 |
| **32 MBytes** | 0,40 | 0,36 | 0,36 |
| **64 MBytes** | 0,45 | 0,38 | 0,36 |
| **128 MBytes** | - | 0,40 | 0,37 |

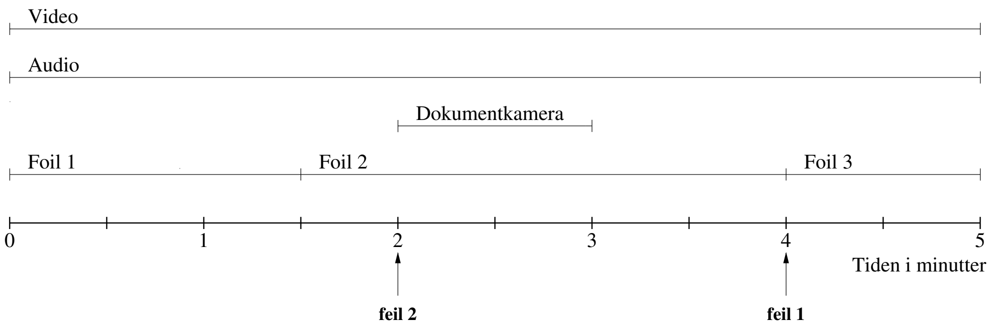
Tabell 8.1: Responstider i sekunder med en bruker i en vanlig avspilling.

#### 8.1.1.2 Flerbrukerscenario

Som tidligere nevnt har vi her hatt tre brukere som starter et og et halvt minutt etter hverandre. I de simuleringene vi har foretatt oss her viser det seg at den valgte bufferhåndteringmekanismen kan støtte kontinuerlig fremvisning av en forelesning for tre samtidige brukere. I tabell 8.2 presenteres det de resultatene vi fikk. Her ser vi at ved bruk av et buffer på 128 MBytes og en sidestørrelse på 32 KBytes, kan vi støtte en feilfri presesntasjon. For alle de andre kombinasjonene får vi *en* eller *to* sidefeil. I de scenarioene hvor vi har en sidefeil, ligger denne sidefeilen er på det samme punktet i presentasjonen for alle scenarioene. Sidefeilen oppstår når den første brukeren skal hente den tredje foilen som vist av pilen for "feil 1" i figur 8.1. I dette punktet får vi en forsinkelse i presentasjonen av foilen på circa 0,3 sekunder i alle scenarioene før presentasjonen fortsetter igjen, og dette vil ikke brukeren merke som dårligere kvalitet. I de scenarioene hvor vi har to sidefeil får vi i tillegg en sidefeil når dokumentkameraet starter for den første brukeren. Dette er i figur 8.1 markert med pilen "feil 2". Avbruddet i avspillingen av forelesningen tar også her circa 0,3 sekunder. Brukeren vil heller ikke her merke om vi utsetter starten av dokumentkameraet i 0,3 sekunder eller om vi bare hopper over den første rammen. Disse avbruddene ligger derfor under en akseptabel toleransegrense, særlig for henting av foiler, så vi kan si at L/MRP støtter kontinuerlig fremvisning av en forelesning ved tre samtidige brukere.

| Buffer-/Sidestørrelse | 8 KBytes | 16 KBytes | 32 KBytes |
|---|---|---|---|
| **16 MBytes** | 1 | 1 | 1 |
| **32 MBytes** | 1 | 1 | 1 |
| **64 MBytes** | 2 | 1 | 1 |
| **128 MBytes** | - | 2 | 0 |

Tabell 8.2: Antall sidefeil med tre brukere i en vanlig avspilling.




Figur 8.1: Hvor sidefeilene oppstod i referansestrengen til forelesningen.

Videre viser tabell 8.3 responstidene for de forkjellige brukerene i sekunder fra brukeren startet forelesningen til fremvisningen kan begynne. Det første vi kan si er at alle responstidene ligger under en akseptabel grense. De ligger for eksempel under PALs øvre grense. Videre viser tabellen at responstidene blir større etter hvor mange brukere som allerede spiller av en forelesning. Ved flere samtidige brukere, er det mer data å hente og responstidene stiger.

| ***Bruker 1*** | | | |
|---|---|---|---|
| **Buffer-/Sidestørrelse** | **8 KBytes** | **16 KBytes** | **32 KBytes** |
| **16 MBytes** | 0,38 | 0,36 | 0,35 |
| **32 MBytes** | 0,36 | 0,36 | 0,36 |
| **64 MBytes** | 0,45 | 0,38 | 0,36 |
| **128 MBytes** | - | 0,41 | 0,37 |
| ***Bruker 2*** | | | |
| **Buffer-/Sidestørrelse** | **8 KBytes** | **16 KBytes** | **32 KBytes** |
| **16 MBytes** | 0,49 | 0,47 | 0,46 |
| **32 MBytes** | 0,48 | 0,48 | 0,47 |
| **64 MBytes** | 0,59 | 0,50 | 0,47 |
| **128 MBytes** | - | 0,53 | 0,48 |
| ***Bruker 3*** | | | |
| **Buffer-/Sidestørrelse** | **8 KBytes** | **16 KBytes** | **32 KBytes** |
| **16 MBytes** | 0,59 | 0,56 | 0,55 |
| **32 MBytes** | 0,57 | 0,57 | 0,57 |
| **64 MBytes** | 0,70 | 0,59 | 0,56 |
| **128 MBytes** | - | 0,55 | 0,50 |

Tabell 8.3: Responstider i sekunder med tre bruker i en vanlig avspilling.

### 8.1.2 Avspilling med interaksjoner

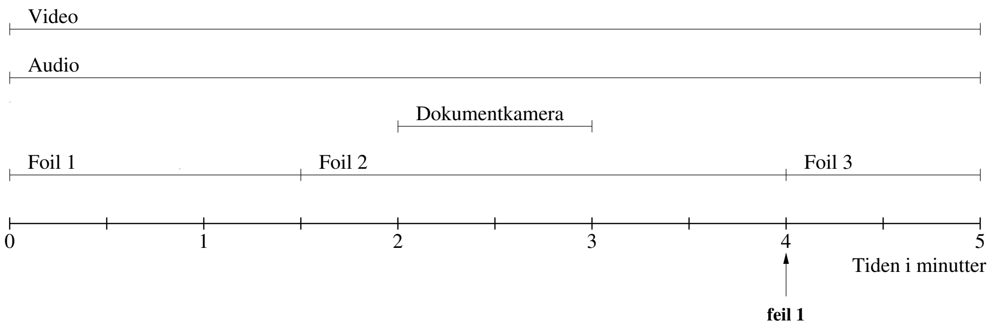
Her har vi simulert en situasjon hvor brukeren(e) vil se en forelesning i sin helhet med flere typer interaksjoner. Her har vi lagt inn hurtigspoling, hopp og avspilling bakover.

| **Buffer-/Sidestørrelse** | **8 KBytes** | **16 KBytes** | **32 KBytes** |
|---|---|---|---|
| **16 MBytes** | 1 | 1 | 1 |
| **32 MBytes** | 1 | 1 | 1 |
| **64 MBytes** | 1 | 1 | 1 |
| **128 MBytes** | - | 1 | 1 |

Tabell 8.4: Antall sidefeil med en bruker i en avspilling med interaksjoner.

#### 8.1.2.1 Enbrukerscenario

Her har vi undersøkt om L/MRP kan støtte en kontinuerlig fremvisning av forelesningen for en bruker i en avspilling med forskjellige interaksjoner, og resultatene fra simuleringene er positive. Tabell 8.4 viser antall sidefeil under presentasjonen. Vi får *en* sidefeil i samme punktet som i forsøket med tre brukere hvor forelesningen avspilles uten interaksjoner. Dette er som markert i figur 8.2 med “feil 1” hvor den første brukeren bytter foil siste gang. Forsinkelsen i avspillingen av forelesningen tar her circa 0,06 sekunder for alle kombinasjonene av buffer- og sidestørrelser vi har simulert. Dette er for det første en minimal forsinkelse, og for det andre kan man godt utsette visningen av en foil et par sekunder uten at det blir lagt merke til som dårlig kvalitet.

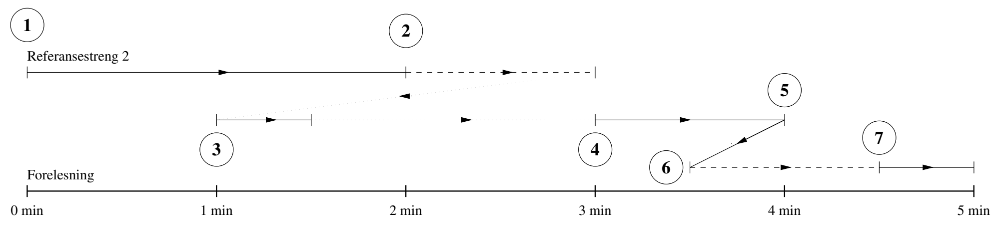



Figur 8.2: Hvor sidefeilen oppstod i referansestrengen til forelesningen.

Videre viser tabell 8.5 responstidene i de forskjellige punktene vi har forandringer i presentasjonsmodusen. De forskjellige punktene er vist i figur 8.3, og følger vi PALs standard for responstider etter en interaksjon, kan vi tillate oss et avbrudd i presentasjonen i hvert punkt på opptil to sekunder. Resultatene fra tabellen viser at alle responstidene ligger under PALs krav, og videre ser vi at flere av presentasjonsforandringene ikke førte til noe avbrudd i presentasjonen. Av forandringene i presentasjonsmodus var bare starten av forelesningen (punkt 1) og hoppet bakover (punkt 3) som gav gav et lite avbrudd i presentasjonen.

| **Buffer- og sidestørrelse/Punkt** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|
| **16 MBytes/8 KBytes** | 0,37 | 0 | 0,38 | 0 | 0 | 0 | 0 |
| **16 MBytes/16 KBytes** | 0,35 | 0 | 0,36 | 0 | 0 | 0 | 0 |
| **16 MBytes/32 KBytes** | 0,35 | 0 | 0,35 | 0 | 0 | 0 | 0 |
| **32 MBytes/8 KBytes** | 0,40 | 0 | 0,40 | 0 | 0 | 0 | 0 |
| **32 MBytes/16 KBytes** | 0,36 | 0 | 0,36 | 0 | 0 | 0 | 0 |
| **32 MBytes/32 KBytes** | 0,35 | 0 | 0,36 | 0 | 0 | 0 | 0 |
| **64 MBytes/8 KBytes** | 0,45 | 0 | 0,26 | 0 | 0 | 0 | 0 |
| **64 MBytes/16 KBytes** | 0,37 | 0 | 0,22 | 0 | 0 | 0 | 0 |
| **64 MBytes/32 KBytes** | 0,36 | 0 | 0,21 | 0 | 0 | 0 | 0 |
| **128 MBytes/16 KBytes** | 0,40 | 0 | 0,21 | 0 | 0 | 0 | 0 |
| **128 MBytes/32 KBytes** | 0,36 | 0 | 0,19 | 0 | 0 | 0 | 0 |

Tabell 8.5: Responstider i sekunder i de forkjellige punktene for forandring i presentasjonsmodus.



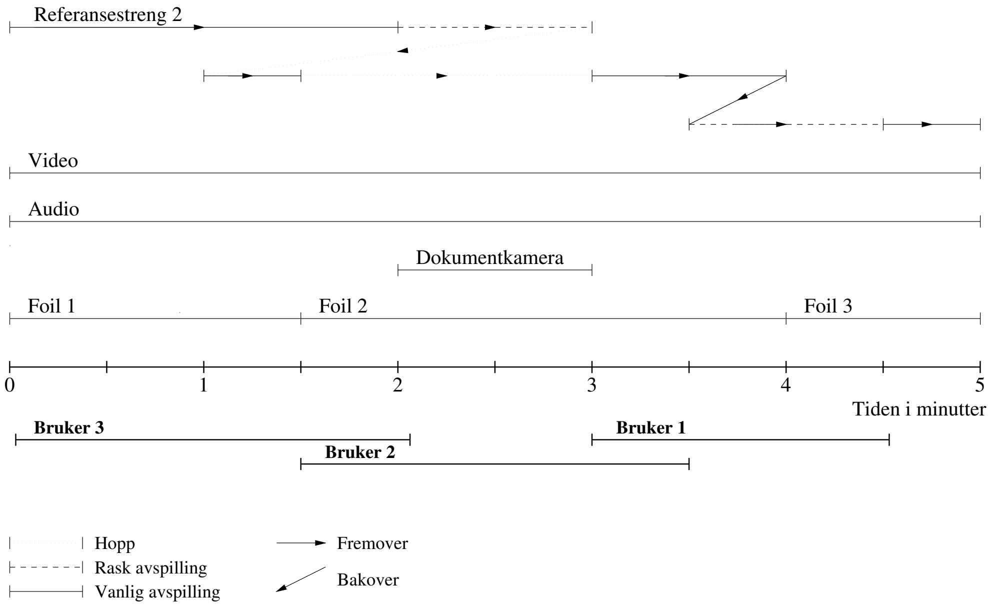

Figur 8.3: Punkter for de forskjellige forandringene i presentasjonsmodus.

| Buffer-/Sidestørrelse | 8 KBytes | 16 KBytes | 32 KBytes |
|---|---|---|---|
| **16 MBytes** | 173 | 174 | 174 |
| **32 MBytes** | 173 | 172 | 172 |
| **64 MBytes** | 174 | 171 | 172 |
| **128 MBytes** | - | 172 | 172 |

Tabell 8.6: Antall sidefeil med tre bruker i en avspilling med interaksjoner.

#### 8.1.2.2 Flerbrukerscenario

I dette forsøket har vi undersøkt om L/MRP kan støtte en kontinuerlig fremvisning av forelesningen for tre samtidige bruker i en avspilling med forskjellige interaksjoner. Tabell 8.6 viser hvor mange sidefeil vi fikk for de forskjellige kombinasjonene av side- og bufferstørrelser. Videre viser figur 8.4 hvor de forskjellige brukerene var i fremvisningen av forelesningen og i referansestrengen da sidefeilene oppstod. Tiden for å fortsette presentasjonen igjen etter sidefeilene ligger mellom 0,15 og 0,36 sekunder.




Figur 8.4: Hvor sidefeilene oppstod i referansestrengen til forelesningen.

Sammenligner man disse tallene med hvor mange sider som ble hentet fra disken, er prosentandelen med sidefeil minimal. Ser vi på scenarioene med 16 KBytes sider har vi en prosentandel av sidefeil på 0,18 prosent til 0,27 prosent avhengig av bufferstørrelsen. For scenarioene med 8 KBytes sider ligger prosentandelen med sidefeil på mellom 0,09 prosent til 0,10 prosent, og for sider på 32 KBytes ligger prosentandelen med sidefeil mellom 0,34 prosent og 0,50 prosent. Sammenligner vi igjen med sideutbyttingsmekanismer som henter sidene på forespørsel og får sidefeil hver gang disken aksesseres, ser vi av tabell 8.7 at vi får veldig store forbedringer. Denne tabellen viser hvor mange sidefeil LRU og vilkårlig sideutbytting gir i et 16 MBytes buffer og sidestørrelser på 8 KBytes, 16 KBytes og 32 KBytes. I tillegg viser tabellen hvor mange sidefeil L/MRP gir i forhold til LRU og vilkårlig sideutbytting i prosent. Videre kommer disse sidefeilene med et lite mellomrom, slik at vi for eksempel kan hoppe over den videorammen som kommer for sent og forsinke visningen av en foil uten at dette legges merke til. Resultatene fra disse forsøkene tilfredsstiller derfor våre krav med tanke på å støtte en kontinuerlig presentasjon av en forelesning.

| **Algoritme** | *LRU* | | *Vilkårlig* | |
|---|---|---|---|---|
| **Buffer- og sidestørrelse** | **antall sidefeil** | **Antall sidefeil L/MRP har i forhold (prosent)** | **antall sidefeil** | **Antall sidefeil L/MRP har i forhold (prosent)** |
| **16 MBytes/8 KBytes** | 234962 | 0,07 | 247879 | 0,07 |
| **16 MBytes/16 KBytes** | 117481 | 0,15 | 123874 | 0,14 |
| **16 MBytes/32 KBytes** | 58746 | 0,30 | 62039 | 0,28 |

Tabell 8.7: Antall sidefeil L/MRP har i prosent i forhold til tradisjonelle sideutbyttingsalgoritmer.

Tabell 8.8 viser responstidene ved de forskjellige punktene (se figur 8.3) for forandring i presentasjonsmodusen for de tre brukerene. Igjen ser vi at hvis vi sammenligner med PALs grense for responstider, ligger resultatene fra forsøket under dette.

Tabell 8.8: Responstider i sekunder i de forkjellige punktene for forandring i presentasjonsmodus for de tre brukerene.

| *Bruker 1* | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Buffer- og sidestørrelse/Punkt** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **16 MBytes/8 KBytes** | 0,37 | 0 | 0,15 | 0,52 | 0 | 0,45 | 0 |
| **16 MBytes/16 KBytes** | 0,35 | 0 | 0,14 | 0,50 | 0 | 0,42 | 0 |
| **16 MBytes/32 KBytes** | 0,35 | 0 | 0,14 | 0,50 | 0 | 0,41 | 0 |
| **32 MBytes/8 KBytes** | 0,39 | 0 | 0,16 | 0,47 | 0 | 0,34 | 0 |
| **32 MBytes/16 KBytes** | 0,36 | 0 | 0,15 | 0,43 | 0 | 0,31 | 0 |
| **32 MBytes/32 KBytes** | 0,35 | 0 | 0,15 | 0,50 | 0 | 0,31 | 0 |
| **64 MBytes/8 KBytes** | 0,45 | 0 | 0,08 | 0,53 | 0 | 0,33 | 0 |
| **64 MBytes/16 KBytes** | 0,37 | 0 | 0,08 | 0,44 | 0 | 0,28 | 0 |
| **64 MBytes/32 KBytes** | 0,36 | 0 | 0,07 | 0,42 | 0 | 0,27 | 0 |
| **128 MBytes/16 KBytes** | 0,40 | 0 | 0,08 | 0,47 | 0 | 0,29 | 0 |
| **128 MBytes/32 KBytes** | 0,36 | 0 | 0,08 | 0,44 | 0 | 0,27 | 0 |

| *Bruker 2* | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Buffer- og sidestørrelse/Punkt** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **16 MBytes/8 KBytes** | 0,49 | 0 | 0,23 | 0,45 | 0 | 0,48 | 0 |
| **16 MBytes/16 KBytes** | 0,47 | 0 | 0,23 | 0,42 | 0 | 0,38 | 0 |
| **16 MBytes/32 KBytes** | 0,46 | 0 | 0,22 | 0,41 | 0 | 0,37 | 0 |
| **32 MBytes/8 KBytes** | 0,52 | 0 | 0,25 | 0,34 | 0 | 0,50 | 0 |
| **32 MBytes/16 KBytes** | 0,48 | 0 | 0,23 | 0,32 | 0 | 0,39 | 0 |
| **32 MBytes/32 KBytes** | 0,46 | 0 | 0,23 | 0,31 | 0 | 0,38 | 0 |
| **64 MBytes/8 KBytes** | 0,59 | 0 | 0,28 | 0,33 | 0 | 0,56 | 0 |
| **64 MBytes/16 KBytes** | 0,49 | 0 | 0,24 | 0,28 | 0 | 0,47 | 0 |
| **64 MBytes/32 KBytes** | 0,47 | 0 | 0,23 | 0,27 | 0 | 0,38 | 0 |
| **128 MBytes/16 KBytes** | 0,53 | 0 | 0,19 | 0,29 | 0 | 0,29 | 0 |
| **128 MBytes/32 KBytes** | 0,48 | 0 | 0,21 | 0,27 | 0 | 0,22 | 0 |

*fortsetter på neste side*

*fortsettelse fra forrige side*

| ***Bruker 3*** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Buffer- og sidestørrelse/Punkt** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **16 MBytes/8 KBytes** | 0,52 | 0 | 0,51 | 0,48 | 0 | 0 | 0 |
| **16 MBytes/16 KBytes** | 0,50 | 0 | 0,48 | 0,38 | 0 | 0 | 0 |
| **16 MBytes/32 KBytes** | 0,49 | 0 | 0,48 | 0,37 | 0 | 0 | 0 |
| **32 MBytes/8 KBytes** | 0,57 | 0 | 0,54 | 0,51 | 0 | 0 | 0 |
| **32 MBytes/16 KBytes** | 0,51 | 0 | 0,50 | 0,39 | 0 | 0 | 0 |
| **32 MBytes/32 KBytes** | 0,49 | 0 | 0,49 | 0,38 | 0 | 0 | 0 |
| **64 MBytes/8 KBytes** | 0,63 | 0 | 0,61 | 0,56 | 0 | 0 | 0 |
| **64 MBytes/16 KBytes** | 0,53 | 0 | 0,51 | 0,47 | 0 | 0 | 0 |
| **64 MBytes/32 KBytes** | 0,50 | 0 | 0,49 | 0,38 | 0 | 0 | 0 |
| **128 MBytes/16 KBytes** | 0,48 | 0 | 0,51 | 0,29 | 0 | 0 | 0 |
| **128 MBytes/32 KBytes** | 0,44 | 0 | 0,46 | 0,18 | 0 | 0 | 0 |

### 8.1.3 Oppsummering

L/MRP er en betraktelig forbedring i forhold til tradisjonelle bufferhåndteringmekanismer som ikke støtter forhåndshenting og presentasjonsrelevant sideutbytting. I forsøkene vi har foretatt oss på en vanlig avspilling, viser det seg at avspilling av forelesningen kan støttes med en bruker uansett buffer- og sidestørrelser med de "implementasjons"-parameterne og de simuleringsomgivelsene (avsnitt 7.1) vi har brukt. Videre viser det seg at også tre brukere kan avspille en forelesning med et veldig godt resultat, det vil si at vi får fra null til to sidefeil avhengig av kombinasjonen side- og bufferstørrelse. I scenarioet med en sidestørrelse på 32 KBytes og et buffer på 128 MBytes gikk avspillingen feilfri. For de andre scenarioene fikk vi en eller to sidefeil hvor vi fikk et opphold i presentasjonen på circa 0,3 sekunder.

I forsøkene med avspillinger med forskjellige interaksjoner, viser det seg at L/MRP også her er langt bedre enn andre tradisjonelle bufferhåndteringmekanismer. I scenarioet med en bruker får vi en sidefeil utenom de som skyldes interaksjon fra brukeren for alle simulerte kombinasjoner av buffer- og sidestørrelser. Dette gir et avbrudd i fremvisningen av forelesningen på circa 0,06 sekunder. I scenarioet med tre brukere får vi en del flere sidefeil, det vil si mellom 171 og 174 avhengig av buffer- og sidestørrelse. Alle disse kommer mens det er tre brukere hvor vi får avbrudd på mellom 0,15 og 0,36 sekunder.

Betydningen av de sidefeilene vi har fått her er minimale, det vil si at brukeren ikke vil legge merke til de fleste av dem og i alle fall ikke registrere sidefeilene som dårligere kvalitet. Som tidligere nevnt kan en foil forsinkes i et par sekunder, og en videoramme kan droppes. Det er i tilfeller hvor sidefeilen skylles mangler på audiosampler brukeren vil legge merke til sidefeilene. Vi kan derfor tolke disse resultatene slik at L/MRP er et veldig godt alternativ til tradisjonelle bufferhåndteringsmekanismer for støtte av multimediaapplikasjoner med kontinuerlige datatyper.

I tillegg viser det seg at vår valgte bufferhåndteringmekanisme uansett referansestreng, antall brukere, buffer- og sidestørrelse har bedre responstider enn de gitte krav. L/MRP støtter blant annet PALs krav til responstider hvor resultatene viser responstider fra 0,35 sekunder til 0,63 sekunder avhengig av buffer- og sidestørrelse samt antall samtidige brukere.

## 8.2 Sideutbytting

I dette avsnittet undersøker vi om vår sideutbyttingsalgoritme vil kunne gi noen færre sideutbyttinger[2] enn tradisjonelle sideutbyttingsalgoritmer ved å bytte ut de minst relevante dataene. Vi simulerer her med et buffer på 32 MBytes og 64 MBytes og sidestørrelsene 8 KBytes, 16 KBytes og 32 KBytes, og vi sammenligner resultatene fra L/MRP med LRU og vilkårlig[3] sideutbytting.

Videre sjekker vi hvordan forskjellige bufferstørrelser påvirker sideutbyttingen ved at data som allerede er i bufferet ikke må hentes på nytt. Vi sammenligner antall sideutbyttinger i bufferet ut i fra L/MRPs sideutbytting ved forskjellige bufferstørrelser og en fast sidestørrelse.

### 8.2.1 Avspilling uten interaksjoner

I en avspilling uten interaksjoner har vi bare undersøkt antall sideutbyttinger i et flerbrukerscenario. Vanlig avspilling i enbrukermodus bruker bare dataene en gang slik at vi får samme mengde datautbytting uansett sideutbyttingsstrategi og bufferstørrelse.

#### 8.2.1.1 Sammenligninger av forskjellige sideutbyttingalgoritmer

Tabell 8.9 og figur 8.5 viser antall sideutbyttinger fra forskjellige sideutbyttingsalgoritmer i et 32 MBytes buffer med variabel sidestørselse. L/MRP gir færre sideutbyttinger enn både LRU - og vilkårlig sideutbytting. De samme resultatene får vi ved bruk av et 64 MBytes buffer som vist i tabell 8.10 og figur 8.6[4]. Disse resultatene viser vi at L/MRP gir færrest sideutbyttinger for begge bufferstørrelsene uansett hvor store sidene er, og LRU gir igjen noe bedre resultater en vilkårlig sideutbytting. Færre sideutbyttinger betyr igjen at det er mindre data som må hentes fra disken. Siden henting av data fra disken er den største tidkrevende faktoren (se avsnitt 8.3.1), vil derfor L/MRP kunne gi ytterligere besparelser i den totale tiden brukt på å hente data i forhold til de andre algoritmene ved at vi får en mer effektiv gjenbruk av dataene gjennom mer fornuftig sideutbytting.

| Algoritme/Sidestørrelse | 8 KBytes | 16 KBytes | 32 KBytes |
|---|---|---|---|
| **Vilkårlig** | 190124 | 95065 | 47576 |
| **LRU** | 185290 | 92645 | 46325 |
| **L/MRP** | 180086 | 90055 | 45032 |

Tabell 8.9: Antall sideutbyttinger for forskjellige sideutbyttingsalgoritmer i et 32 MBytes buffer.

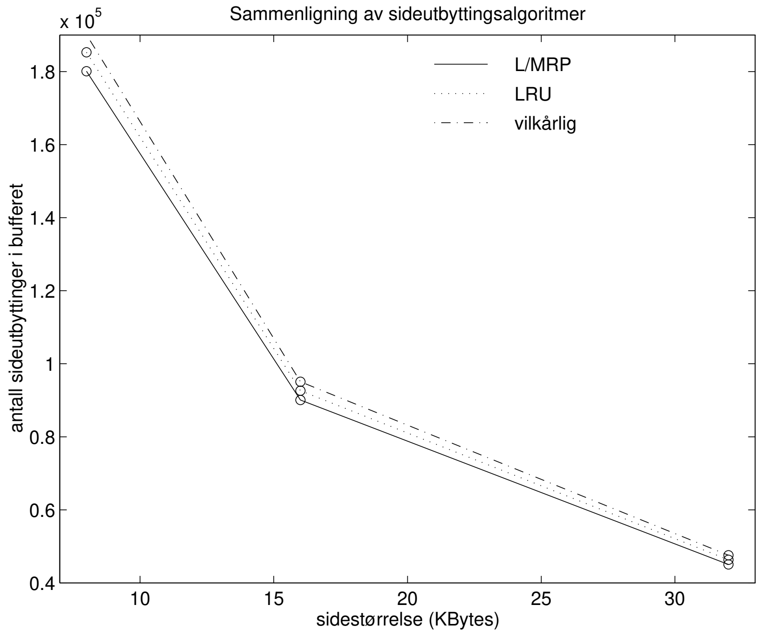
| Algoritme/Sidestørrelse | 8 KBytes | 16 KBytes | 32 KBytes |
|---|---|---|---|
| **Vilkårlig** | 185939 | 92916 | 46513 |
| **LRU** | 185240 | 92620 | 46312 |
| **L/MRP** | 165892 | 82954 | 41481 |

Tabell 8.10: Antall sideutbyttinger for forskjellige sideutbyttingsalgoritmer i et 64 MBytes buffer.

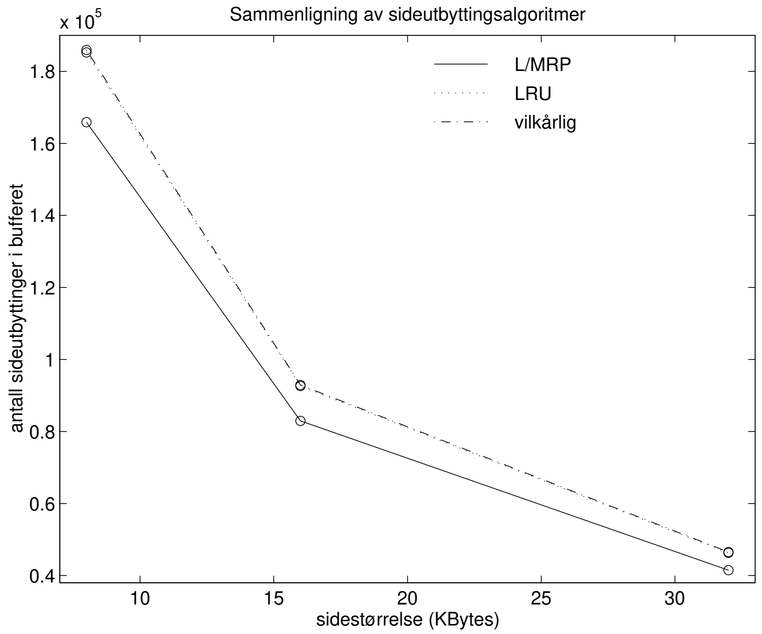
[2] Som tidligere nevnt vil L/MRP uansett gi færre diskaksesser ved at data forhåndshentes og mer data hentes for hver gang disken aksesseres.

[3] Her er det, som tidligere nevnt i avsnitt 6.5, verdt å merke seg at antall utbyttinger i bufferet for tradisjonelle sideutbyttingsalgoritmer er det samme som antall sidefeil disse algoritmene får.

[4] Kurvene for LRU og vilkårlig sideutbytting ligger over hverandre.




Figur 8.5: Sammenligning av sideutbyttingsalgoritmer i et 32 MBytes buffer.



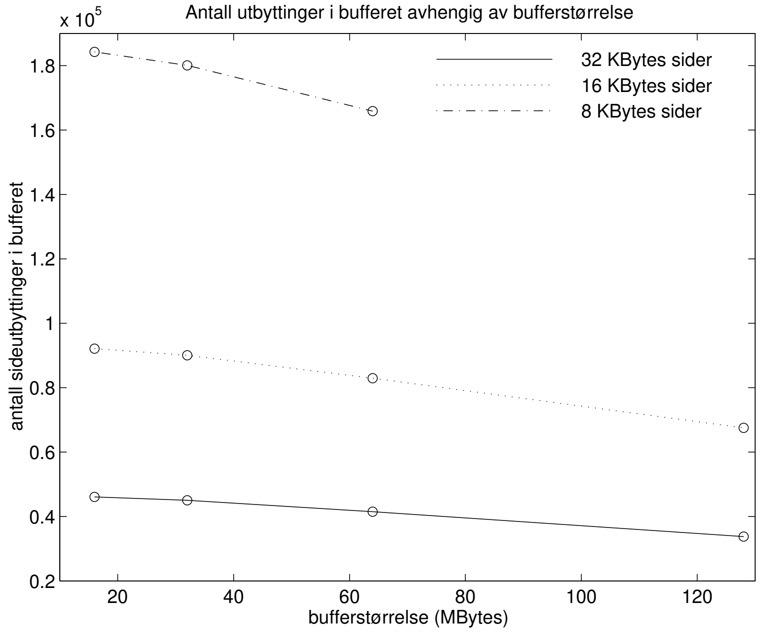

Figur 8.6: Sammenligning av sideutbyttingsalgoritmer i et 64 MBytes buffer.

#### 8.2.1.2 Sammenligning av sideutbytting med forskjellige bufferstørrelser

Videre har vi undersøkt hvordan forskjellige størrelser på bufferet påvirker sideutbyttingen ved bruk av L/MRP. Vi har her simulert forskjellige scenarioer hvor vi holder sidestørrelsen fast og varierer bufferstørrelsen. Resultatene er vist i tabell 8.11 og figur 8.7. Forsøkene viser at ved større buffere kan vi beholde mer av dataene i bufferet, slik at vi får en bedre "gjenbruk"[5], det vil si at større buffere gir færre sideutbyttinger. Et stort buffer kan dermed redusere den totale tiden for uthenting av data fra disken.

| **Bufferstørrelse/Sidestørrelse** | **8 KBytes** | **16 KBytes** | **32 KBytes** |
|---|---|---|---|
| **16 MBytes** | 184273 | 92139 | 46072 |
| **32 MBytes** | 180086 | 90055 | 45032 |
| **64 MBytes** | 165892 | 82954 | 41481 |
| **128 MBytes** | - | 67547 | 33780 |

Tabell 8.11: Antall sideutbyttinger for L/MRP med forskjellige bufferstørrelser.




Figur 8.7: Sammenligning av antall sideutbyttinger for L/MRP i bufferet avhengig av bufferstørrelsen.

### 8.2.2 Avspilling med interaksjoner

I dette forsøket har vi simulert et enbrukerscenario. I referansestrengen med forskjellige interaksjoner trenger vi samme data flere ganger, samt at vi kan se om L/MRP tjener noe på å hente mindre data ved hurtigspoling.

#### 8.2.2.1 Sammenligninger av forskjellige sideutbyttingalgoritmer

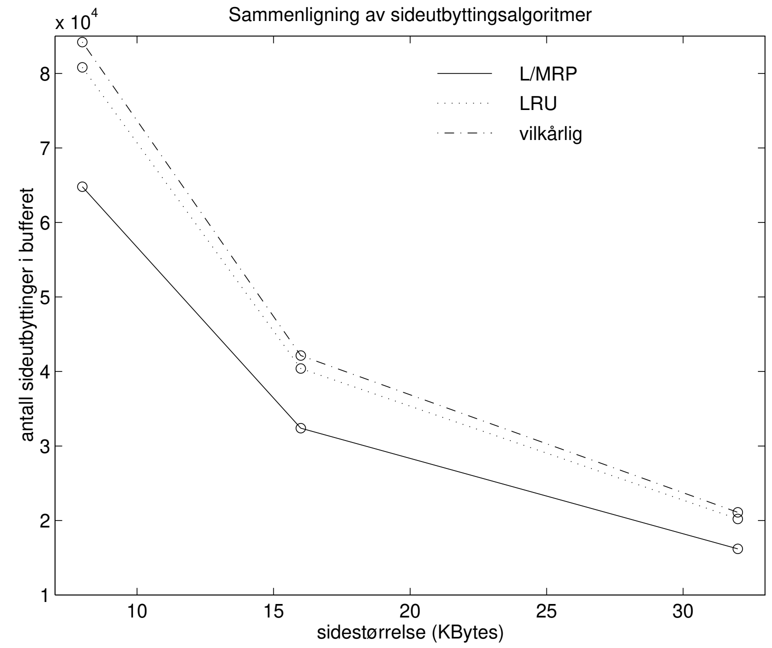
Tabell 8.12 og figur 8.8 viser forskjeller i antall sideutbyttinger i et enbrukerscenario hvor vi har en avspilling med flere interaksjoner i et buffer på 32 MBytes. Tabell 8.13 og figur 8.9 viser resultatene for det samme scenarioet i et 64 MBytes buffer. Resultatene viser igjen at L/MRP gir

[5] Med "gjenbruk" av data mener jeg her at dataene finnes allerede i bufferet slik at de ikke må hentes fra disken en gang til.

| **Algoritme/Sidestørrelse** | **8 KBytes** | **16 KBytes** | **32 KBytes** |
|---|---|---|---|
| **Vilkårlig** | 84204 | 42144 | 21098 |
| **LRU** | 80806 | 40403 | 20205 |
| **L/MRP** | 64787 | 32395 | 16199 |

Tabell 8.12: Antall sideutbyttinger for forskjellige sideutbyttingsalgoritmer i et 32 MBytes buffer.

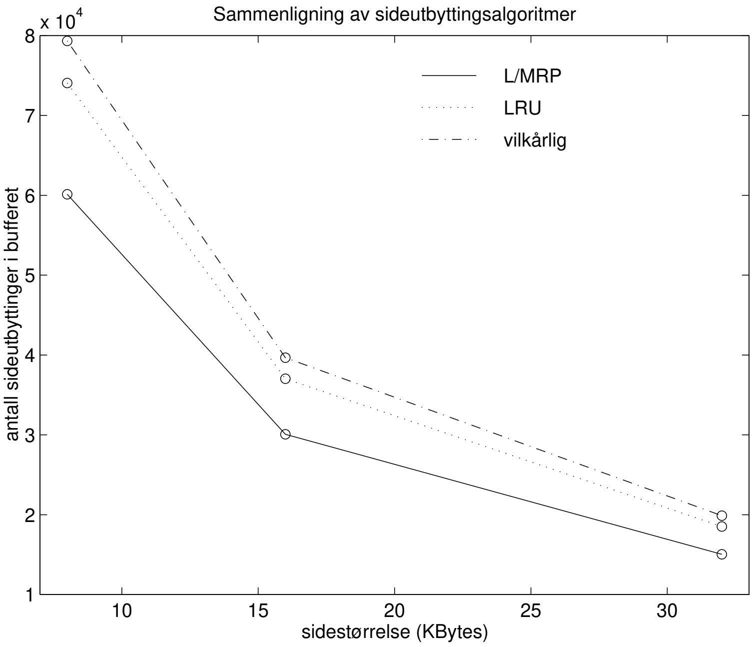
en del færre sideutbyttinger enn de to andre sideutbyttingsalgoritmene. Dette er resultatet av at en del data allerede finnes i bufferet og ikke må hentes igjen ved hopp tilbake og baklengs avspilling. I tillegg tjener L/MRP en del på å hente mindre data fra video ved hurtig avspilling (audio hentes ikke her uansett algoritme).




Figur 8.8: Sammenligning av sideutbyttingsalgoritmer i et 32 MBytes buffer.

| **Algoritme/Sidestørrelse** | **8 KBytes** | **16 KBytes** | **32 KBytes** |
|---|---|---|---|
| **Vilkårlig** | 79333 | 39670 | 19875 |
| **LRU** | 74064 | 37032 | 18518 |
| **L/MRP** | 60134 | 30070 | 15038 |

Tabell 8.13: Antall sideutbyttinger for forskjellige sideutbyttingsalgoritmer i et 64 MBytes buffer.




Figur 8.9: Sammenligning av sideutbyttingsalgoritmer i et 64 MBytes buffer.

#### 8.2.2.2 Sammenligning av sideutbytting med forskjellige bufferstørrelser

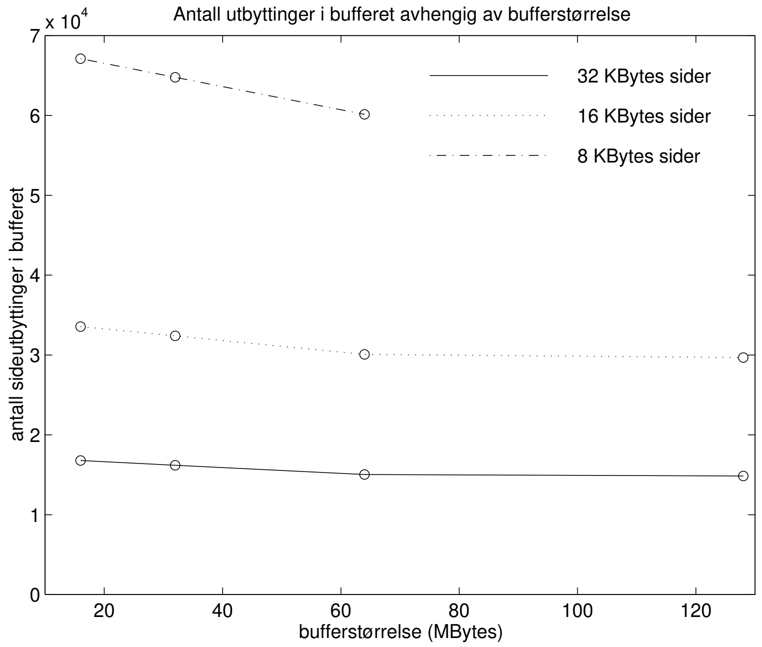
Vi har også undersøkt hvordan bufferstørrelsen påvirker antall sideutbyttinger i et enbrukerscenario i en avspilling med interaksjoner. Selv om det ikke er så store muligheter for bruk av data som allerede er i bufferet i et enbrukerscenario, har vi noen få interasjoner som gir bruk av samme data flere ganger. Resultatene er vist i tabell 8.14 og figur 8.10. Det viser seg, som i avsnitt 8.2.1.2, at et større buffer gir færre sideutbyttinger.

| **Bufferstørrelse/Sidestørrelse** | **8 KBytes** | **16 KBytes** | **32 KBytes** |
|---|---|---|---|
| **16 MBytes** | 67122 | 33564 | 16785 |
| **32 MBytes** | 64787 | 32395 | 16199 |
| **64 MBytes** | 60134 | 30070 | 15038 |
| **128 MBytes** | - | 29687 | 14845 |

Tabell 8.14: Antall sideutbyttinger for L/MRP med forskjellige bufferstørrelser.

### 8.2.3 Oppsummering

Disse forsøkene viser at L/MRP gir færre sideutbyttinger enn LRU og vilkårlig ved å ta i betraktning hvilke sider som er mest og minst egnet for utbytting. Videre gir et stort buffer færre sideutbyttinger avhengig av mulighetene for bruk av samme data flere ganger. Ved bruk av L/MRP i et stort buffer, noe som vil gi færre sideutbyttinger i bufferet, vil vi dermed kunne spare mye tid på den totale overføringen av data fra disk til buffer.




Figur 8.10: Sammenligning av antall sideutbyttinger for L/MRP i bufferet avhengig av bufferstørrelsen.

## 8.3 Hentetider for data

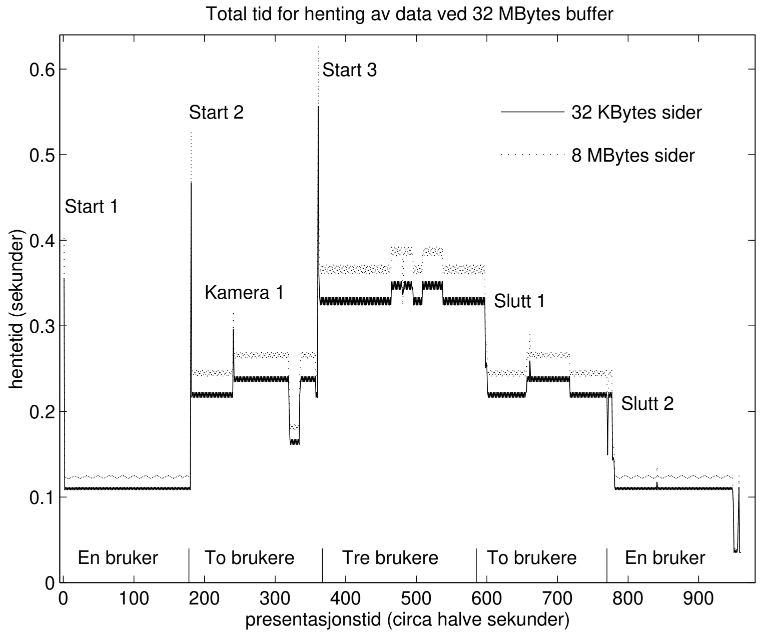
I dette avsnittet undersøker vi de totale tidene algoritmen bruker på å hente data til bufferet fra disken. Her har vi medberegnet overføringstider fra disken, minneaksesser og eksekveringstider i algoritmen. For disken bruker vi som tidligere nevnt data fra SEAGATE ELITE 23. Det er her vanskelig å vite når vi får minimale og maksimale utslag på diskens lesearm, så vi har beregnet overføringstidene etter den gjennomsnittlige søketiden. Videre har vi beregnet aksesstider i minnet ut i fra RAM-brikker med aksesstider på 60 ns, og algoritmekodens eksekveringstider er beregnet etter programmet "Quantify" som estimerer antall klokkesykler per instruksjon. Eksekveringstiden i sekunder regnes deretter ut ved bruk av en 200 MHz prosessor (UltraSPARC).

### 8.3.1 Forholdet mellom den totale overføringstiden og overføringstider fra disken

Tidene for overføringene av data fra disken utgjør den største faktoren i den totale tiden vi bruker på å hente dataene. Den totale tiden for henting av data hvor den første brukeren starter og alle dataene må hentes fra disken (ikke dokumentkameraet), blir 0,3558 sekunder i et buffer på 32 MBytes og en sidestørrelse på 32 KBytes. Tiden for selve overføringen fra disken utgjør her 0,3179 sekunder eller 89,34 prosent av den totale tiden. I et punkt i presentasjonen hvor avspillingen har begynt (ikke dokumentkameraet), og det ligger en del av dataene i bufferet, har vi typiske hentetider på 0,1083 sekunder totalt. Her vil bare overføringene fra disken ta 0,0848 sekunder eller 78,33 prosent av den totale hentetiden. Disse beregningene viser at det er det mye tid å spare ved å minske mengden av data som må hentes fra disken.

### 8.3.2 Fast bufferstørrelse

For å sammenligne eksekveringstider har vi først sammenlignet hvordan forskjellige sidestørrelser påvirker den totale overføringstiden av data fra disken til bufferet. Vi har her brukt et 32 MBytes buffer og brukt sidestørrelser på 8 KBytes og 32 KBytes. I dette scenarioet er det like store mengder med data som må byttes ut, men antallet sideutbyttinger blir flere for den minste sidestørrelsen ved at hver enkelt PE må mappes ned til flere sider. Resultatet er vist i figur 8.11 hvor det viser seg at 8 KBytes sider stort sett gir en høyere overføringstid enn sider på 32 KBytes. Dette er som sagt stort sett forskjeller i selve eksekveringstiden av algoritmen og flere minneaksesser ved at algoritmen må mappe PEer til sider. Det er de samme mengdene med data som hentes samtidig så dette gir ikke noen forskjeller her.



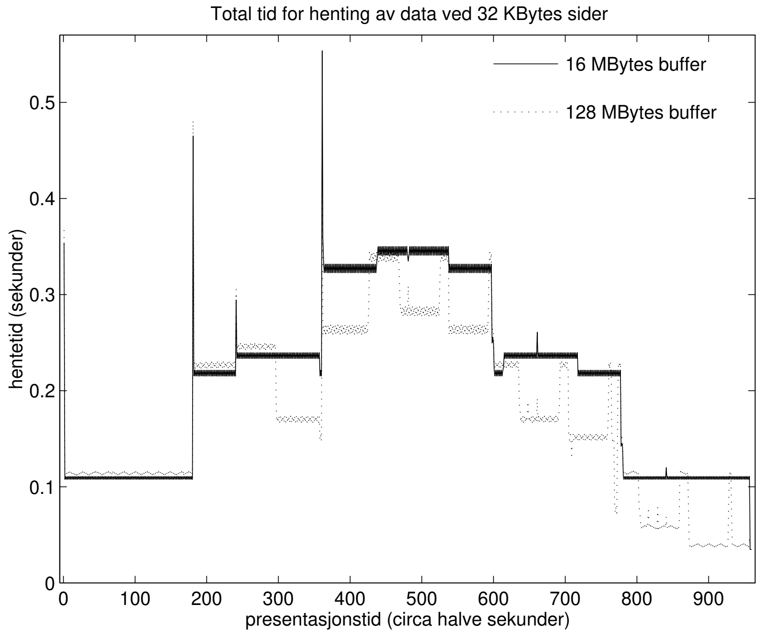

Figur 8.11: Sammenligninger av de totale tidene for henting av data i et 32 MBytes buffer.

Videre viser figur 8.11 enkelte høye hopp i kurven over den totale hentetiden. De største av disse er markert i figuren med hvorfor hentingen av data tar lengre (eventuelt kortere) tid på dette tidspunktet i presentasjonen. De tre høyeste toppene viser hvor de tre forskjellige brukerene starter sine presentasjoner av forelesningene. Den toppen mellom starten av bruker to og tre viser at dokumentkameraet for bruker en starter å sende data. De to største fallene i hentetidene viser hvor brukerene avslutter avspillingen av forelesningen.

I tillegg til markeringer av hva de største hoppene i kurven betyr har vi også markert hvor mange samtidige brukere det er i systemet. Som forventet ser vi at de totale tidene avhenger sterkt av antall brukere ettersom mer data må hentes og algoritmen må eksekveres flere ganger ved flere brukere.

### 8.3.3 Fast sidestørrelse

Den andre sammenligningen vi har gjort med tanke på totale hentetider av data, er at vi ser på hva bufferstørrelsen har å si. Vi har satt en fast sidestørrelse på 32 KBytes og sett på bufferstørrelsene 16 MBytes og 128 MBytes. Dette gir like mange sider å mappe ned til samt at antallet minneaksesser vil være like stort. De totale hentetidene for data er vist i figur 8.12 hvor hoppene betyr de samme som i figur 8.11. Fra starten av denne kurven hvor vi ikke har noe data i bufferet fra før viser det seg at hentetiden for data er høyere i et stort buffer enn i et lite. Dette kommer av at vi har flere bufferplasser og må derfor beregne relevansverdier for flere sider i bufferet. Vi får altså høyere eksekveringstider av selve algoritmen i et stort buffer.




Figur 8.12: Sammenligninger av de totale tidene for henting av data med 32 MBytes sider.

Videre viser det seg at lengre ut i presentasjonen blir den totale tiden for henting av data mindre i det store bufferet. Eksekveringstid av selve algoritmekoden vil fortsatt være lengre i et stort buffer, men siden bufferet er en del større og en del data allerede er hentet inn, vil vi kunne beholde dataene lengre i bufferet før de byttes ut. Dette gir brukerene som starter avspillingen av foreleningen senere en bedre mulighet til bruke data som allerede er i bufferet, og som vi så over, vil overføring av mindre datamengder fra disken gi betydelige reduksjoner i den totale overføringstiden.

### 8.3.4 Oppsummering

Forsøkene her viser at det meste av den totale tiden for henting av data fra disk til buffer består av selve overføringen av data fra disken. Videre ser vi at en mindre sidestørrelse vil gi lengre eksekveringstider av selve algoritmen, samt at vi får flere minneaksesser. Sammenligner vi forskjellige bufferstørrelser gir større buffere lengre eksekveringstider, men den totale overføringstiden vil kunne bli mindre ved at vi får bedre “gjenbruk” av data.

# Kapittel 9

# Konklusjon

I dette kapittelet oppsummerer vi de viktigste punktene i denne oppgaven (avsnitt 9.1). Deretter beskriver vi de viktigste resultatene vi har kommet frem til, samt at vi evaluerer og tolker simuleringsresultatene (avsnitt 9.2). Tilslutt diskuterer vi ubesvarte spørsmål og tar for oss fremtidig arbeid (avsnitt 9.3).

## 9.1 Oppsummering

Som vi har beskrevet tidligere i denne oppgaven har innføringen av multimediaapplikasjoner og multimediadata i DHSer innført mange nye krav til håndtering av data i et MMS. Målet med denne hovedfagsoppgaven er å finne en bufferhåndteringsmekanisme som kan støtte disse kravene slik at ikke bufferhåndteringen blir noen flaskehals.

På veien til å finne en egnet mekanisme til den eksempelapplikasjonen vi har valgt, et DHS for det elektroniske klasserommet, har vi gått igjennom de kravene denne applikasjonen stiller til et DHS. I slike tidsavhengige, kontinuerlige multimediaapplikasjoner trenger vi en stor båndbredde, god gjennomstrømming av data og en jevn, rask responstid. Videre har vi kommet frem til at tradisjonelle bufferhåndteringsmekanismer som FIFO, LRU, LFU og så videre ikke kan støtte slike applikasjoners behov. Ved å evaluere forskjellige bufferhåndteringsmekanismer mot multimediaapplikasjonenes krav, har vi kommet fram til at en bufferhåndteringsmekanisme for applikasjoner, som består av kontinuerlige datatyper, bør støtte en form for forhåndshenting av data, samt at den bør ha en form for presentasjonsrelevant sideutbytting.

Videre har vi brukt egenskaper og dataenes referansestrukturer fra DHSet for det elektroniske klasserommet til å finne en egnet bufferhåndteringsmekanisme for multimediaapplikasjoner. En bufferhåndteringmekanisme som egner seg for tidsavhengige, kontinuerlige multimediaapplikasjoner som vår eksempelapplikasjon er L/MRP.

## 9.2 Evaluering og tolkning av resultater

Her gis en kort evaluering av resultatene vi har kommet frem til i denne oppgaven. Vi beskriver de krav multimediaapplikasjoner stiller til bufferhåndteringen og presenterer kort resultatene fra forsøkene. Tilslutt har vi tolket resultatene vi har kommet frem til i denne oppgaven og kommer med et forslag til en bufferhåndteringsmekanisme for multimediaapplikasjoner.

### 9.2.1 Krav til bufferhåndteringen fra multimediaapplikasjoner

Innføringen av multimediaapplikasjoner har gitt mange nye krav til DHSene. Det er enorme mengder data som må lagres samtidig med at disse må synkroniseres og leveres til bestemte tidsfrister.

For at et MMS skal kunne støtte en viss QoS må alle funksjonelle enheter støtte samme QoS, og sett fra DHSenes side betyr dette at data må kunne hentes raskt for å kunne støtte en kontinuerlig presentasjon av dataene. Igjen betyr dette at vi må ha en stor båndbredde og en god gjennomstrømming av data mellom primær- og sekundærminnet, samt at vi trenger kontinuerlige, jevne responstider under en viss terskel.

Hvordan dette oppnåes avhenger igjen av hvilken delkomponent av DHSet som undersøkes. Sett i fra bufferhåndteringen viser det seg at tradisjonelle sideutbyttingsmekanismer ikke støtter multimediaapplikasjoners krav, og at nye mekanismer derfor er nødvendige. Ut i fra de kravene vi har beskrevet i denne oppgaven, har vi kommet frem til at en bufferhåndteringsmekanisme i et DHS for det elektroniske klasserommet trenger to viktige egenskaper: forhåndshenting av data fra disken til bufferet og presentasjonsrelevant sideutbytting. "Side-på-forespørsel"-systemer kan ikke levere data fort nok til bufferet ved at vi får en sidefeil hver gang vi trenger nye data. Data må derfor forhåndshentes slik at den totale tiden brukt på å hente en viss mengde med data minimeres. Videre kan antall diskaksesser, som utgjør den største delen av den totale hentetiden av data fra disken til bufferet, reduseres ytterligere ved å finne en sideutbyttingsalgoritme som gir minimalt med sideutbyttinger. Tradisjonelle sideutbyttingsalgoritmer tar bare i bruk kriterier som for eksempel alder og referansefrekvens. Det vi har kommet frem til er at vi trenger en sideutbyttingsstrategi som tar hensyn til hvilke data vi trenger når vi bytter ut sider i bufferet.

Videre har vi gjennom litteraturstudier kommet frem til at det er bedre å dele bufferet mellom samtidige brukere fremfor å måtte partisjonere det. Vi kan dermed minske det totale bufferbehovet ved at vi kan bruke hele bufferet. En partisjonering kan i perioder gi ubrukte bufferplasser, og det totale bufferbehovet vil bli større.

Ved en overføring av data fra disken til bufferet, går mye tid med til å søke etter riktig spor og å rotere til riktig sektor. Ved å hente mer data av gangen, kan den totale tiden reduseres. Data bør derfor lagres kontinuerlig på disken, og mest mulig bør hentes samtidig. Dette tilsier at vi i et MMS bør ha en stor bufferside. Videre bør vi ha et så stort buffer som mulig, slik at bufferet kan beholde mest mulig data og dermed slippe å hente samme data flere ganger fra disken.

### 9.2.2 Forsøkene

Simuleringene og deres resultater er tidligere presentert i kapittel 7 og kapittel 8. Her gis en kort evaluering av resultatene fra simuleringsforsøkene.

#### 9.2.2.1 Støtte for kontinuerlige avspillinger

I simuleringene vi har foretatt oss her, viser resultatene at L/MRP er en veldig stor forbedring i forhold til tradisjonelle bufferhåndteringsmekanismer. L/MRP kan støtte kontinuerlig avspilling av en forelesning med den lagringsstrukturen og simuleringsomgivelsene vi har brukt. Vi får i enkelte av scenarioene noen sidefeil, men betydningen av disse er relativt liten. Siden forsinkelsene bare er mellom 0,06 sekunder til 0,36 sekunder, vil brukeren i de aller fleste tilfellene ikke legge merke til sidefeilen. Videre kan en foil forsinkes med et par sekunder og en videoramme ofte bare droppes. Det er i tilfeller hvor sidefeilen skyldes mangler på audiosampler, at brukeren vil legge merke til sidefeilene. Vi kan derfor tolke disse resultatene slik at L/MRP er et veldig godt alternativ til tradisjonelle bufferhåndteringsmekanismer for støtte av multimediaapplikasjoner med kontinuerlige datatyper.

#### 9.2.2.2 Sideutbytting i bufferet

Sammenlignet med tradisjonelle sideutbyttingsalgoritmer som LRU og vilkårlig (random), gir L/MRP færre sideutbyttinger i bufferet. Dette gir igjen færre diskaksesser som vi tidligere har sett er en kostbar operasjon. Videre inneholder et stort buffer mer data av gangen slik at vi slipper å bytte ut sider så ofte som i mindre buffere. Dette fører til at mer data kan bli liggende i bufferet, og ved nye referanser av sider som er brukt tidligere, er det en større mulighet for at siden fortsatt er i bufferet. Ved å la bufferet inneholde mer data, kan vi igjen få færre diskaksesser.

#### 9.2.2.3 Hentetider for data

Når vi sammenligner hvordan bufferhåndteringsmekanismen fungerer med forskjellige buffer- og sidestørrelser, viser resultatene at selve eksekveringstiden av algoritmen er forskjellig. Små sidestørrelser gjør at det for hver PE må mappes flere sider slik at eksekveringskostnadene øker. Økte eksekveringstider for algoritmen og lange overføringstider fra disken med små sider taler derfor for en stor sidestørrelse[1].

Store buffere gir også større eksekveringskostnader som igjen varierer med sidestørrelsen, det vil si at vi flere bufferplasser (større sidetabeller) å søke igjennom. Likevel kan det være mange diskaksesser å spare ved at et stort buffer beholder mer data i bufferet. Den optimale løsningen er derfor et stort buffer med en stor sidestørrelse.

### 9.2.3 Forslag til bufferhåndteringsmekanisme

De resultatene vi har fått i denne hovedfagsoppgaven viser at L/MRP med forhåndshenting og presentasjonsrelevant sideutbytting egner seg bedre til kontinuerlige multimediaapplikasjoner enn tradisjonelle sideutbyttingsalgoritmer som LRU og vilkårlig. L/MRP kan med riktige buffer- og sidestørrelser støtte kontinuerlige avspillinger av forelesninger fra et DHS for det elektroniske klasserommet. Videre viser forsøkene at L/MRP gir en del færre sideutbyttinger i bufferet ved at data brukes om igjen mer effektivt. Dette gir færre diskaksesser som igjen gir mye spart tid.

Når det gjelder buffer- og sidestørrelser viser det seg at forsøkene med de største bufferene og de største sidene var mest effektive når det gjelder den totale hentetiden for data. Store sider gir kortere eksekveringstid av algoritmen og færre minneaksesser. Store buffere gir lengre eksekveringstid av algoritmen, men kan gi bedre gjenbruk av data og færre diskaksesser. Som vi så i avsnitt 8.3.1 dominerer selve overføringstiden for data fra disken i den totale hentetiden, så lengre eksekveringstid er å foretrekke fremfor mange diskaksesser.

Ut i fra forsøkene her bør derfor en bufferhåndteringsmekanisme i et DHS for det elektroniske klasserommet benytte forhåndshenting og sideutbytting à la L/MRP med så store buffere og sider som mulig.

## 9.3 Ubeskrevene punkter og fremtidig arbeid

Det er mange aspekter ved et distribuert system som vil kunne påvirke ytelsen til bufferhåndteringsmekanismen vi har modellert og simulert for et DHS for det elektroniske klasserommet. I dette avsnittet beskriver vi slike punkter.

### 9.3.1 Buss mellom disken og bufferet

Som nevnt i avsnitt 7.3 har vi her sett bort i fra bussen mellom disken og bufferet. Ser vi på SCSI (small computer system interface) sin nyeste buss, UltraSCSI, har denne en maksimal overførngskapasitet på 40 MBytes/s [Seagate 96]. Dette tilsvarer en betjening på circa 20 samtidige brukere,

[1] Problemet med å få mye ubrukt plass i store datasider vil ikke være fremtredene i applikasjoner som vi har beskrevet her. Datamengdene er så store i multimediaapplikasjoner med kontinuerlige datatyper at de aller fleste sidene vil være fulle av data.

men i praksis vil flere disker konkurrere om å overføre data på bussen, slik at den reele overføringen kan bli noe mindre. Likevel vil en slik buss støtte de scenarioene vi har simulert i denne oppgaven, og heller ikke bussen ville ha blitt noen flaskehals i systemet.

### 9.3.2 Implementasjon i et eksisterende system

Et annet viktig punkt for fremtidig arbeid er det å gå fra et simulert system til en implementasjonen av et reelt, kjørbart system. Vi har sett på hvordan vi kan bruke OSet CHORUS til å implementere deler av vårt system, men vi har bare sett på noen få komponenter. Som vi blant annet så i kapittel 1 og av figur 1.1, må systemet som enhet støtte de samme kravene. Hvis noen av komponentene blir en flaskehals i systemet, vil ytelsen til systemet som enhet bli dårlig. Det er derfor viktig å undersøke alle delene i lenken av komponenter ved en realisering til et kjørbart system.

### 9.3.3 Forslag til mulige, fremtidige forbedringer

Et av målene med bufferhåndteringsmekanismen i DHS for det elektroniske klasserommet er å kunne tilby en kontinuerlig presentasjon av en forelesning. I forsøkene vi har foretatt oss viser det seg at L/MRP er en stor forbedring fra andre tradisjonelle bufferhåndteringsmekanismer, men at det fortsatt er rom for forbedringer.

#### 9.3.3.1 Lagringsstruktur

I utprøvingene av bufferhåndteringsmekanismen har vi brukt en lagringsstruktur med en disk og et delt buffer. For det første kan dette forbedres ved bruk av flere disker slik at vi for eksempel kan bruke en disk per medietype. I slike tilfeller kan vi parallellisere hentingen av data til de forskjellige medietypene fra disken slik at den totale overføringstiden reduseres.

Ved å ha egne disker til hver medietype kan vi også ha egne buffere. Dette vil gi rom for mulige optimeringer av sidestørrelser og bufferhåndteringsmekanismer avhengig av medietype. Som vi har sett avhenger blant annet eksekveringstiden til algoritmekoden av antall sider som skal hentes. Ved å sette sidestørrelsen lik PE-størrelsen, får vi bare en side å hente for hver PE slik at den totale tiden for dataoverføringen reduseres.

Videre kan antall minneaksesser reduseres ved bruk av et assosiativt minne. Dette er særlig nyttig for medietyper hvor hver PE benyttes flere ganger, og adressen til de samme sidene i bufferet må hentes gjentatte ganger fra sidetabellen.

Lagringsstrukturen vi har brukt i denne oppgaven begrenser antall samtidige brukere. Disken vi har brukt (tabell 6.1) har en dataoverføringsrate på 10,75 - 15,5 MBytes/s. Teoretisk kan dermed bare fem samtidige brukere tilbys tjenester med de totale båndbreddebehovene på circa 2 MBytes/s som vi har beregnet per bruker. I praksis vil antall samtidige brukere være mindre, for vi vil få søketider og rotasjonsforsinkelser for hver medietype og hver samtidige bruker. Av de gjennomførte forsøkene viser resultatene at tre brukere er akkurat på grensen til hva et system med en disk kan tjene.

Andre forsøk på samme disk [Lund 97] viser også at maksimalt tre samtidige brukere kan tjenes samtidig, og at antall nødvendige disker er avhengig av antall samtidige brukere og hvordan data er lagret på forskjellige disker (alle medietyper på alle disker, separate disker for hver medietype hvor igjen hver medietype deles opp på flere disker). Videre viser samme forsøk at det kan lønne seg å forhåndshente mer data av gangen. Dette kan gi muligheter for å tjene flere samtidige brukere, men sett i fra bufferhåndteringen må dette vurderes mot en god gjennomstrømming av data. Det å forhåndshente mer data vil føre til mindre ledig bufferplass. Vi får mindre data som til en hver tid er tilgjengelig for utbytting noe som kan gi lavere gjennomstrømming av data.

Antall samtidige brukere kan dermed igjen reduseres. Her gjelder det derfor å finne en "gylden middelvei" slik at systemet som helhet kan tjene flest mulige samtidige brukere.

#### 9.3.3.2 Kompresjons- og lagringsteknikker

I vår simulering har vi gått ut i fra de kompresjons- og lagringsteknikkene som gir en konstant, høy datarate. I et fremtidig system, vil det være mye å tjene på å finne bedre alternativer, slik at enda flere samtidige brukere kan tjenes. For eksempel vil det å bruke MPEG i steden for JPEG gi en stor besparelse i båndbreddebehovet. JPEG har et båndbreddebehov på circa 1,5 MBytes/s, og bytter vi til MPEG, trenger vi en båndbredde på circa 225 KBytes (avsnitt 3.5.1).

#### 9.3.3.3 Adapsjon av tjenestekvaliteten

I noen av forsøkene fikk vi enkelte punkter hvor vi fikk en liten forsinkelse på grunn av sidefeil. Slike sidefeil er det ønskelig å unngå. Dette kan gjøres ved å reservere nok ressurser for hver forespørsel (noe som kan gi ubrukte ressurser og en begrensning av antall samtidige brukere) eller ved å redusere mengden av data som overføres ved å overvåke arbeidsmengden til systemet.

Den siste muligheten, som vil gi en best mulig QoS avhengig av systemets arbeidsmengde, er prøvd ut ved "German National Research Center for Information Technology" (GMD) sammen med L/MRP i et multimedia-DBS for håndtering og lagring av video [Hollfelder et al. 96]. For å kunne oppdage potensielle flaskehalser i dataleveringen fra serveren, overvåkes utnyttelsen av bufferet på klientsiden. Utnyttelsen måles her i hvor mange PEer som er i bufferet noe som er en indikator på ytelses variasjoner i presentasjonen. Hvis ytelsen i dataleveringen synker, vil utnyttelsen av bufferet også senkes. For å forhindre overbelastning brukes en adapsjonsteknikk for å minske mengden med data som overføres ved å senke kvaliteten på overføringen. I motsatt fall, når ressurser frigis og en bedre QoS kan tilbys, brukes samme tilpassningsmekanisme for igjen å bedre kvaliteten på tjenesten.

Avhengig av graden av bufferutnyttelse benyttes forskjellige kombinasjoner av to adapsjonsstrategier for å tilpasse mengden av data som overføres. Det vil si endre antall dataenheter og kvaliteten på hver enkelt dataenhet:

**Temporal adapsjon** vil si å redusere presentasjonshastigheten. Ved GMD gjøres dette ved å senke rammeraten, det vil si å vise færre bilder per sekund slik at overføringsmengden av data minskes.

**Romlig adapsjon** vil si å senke oppløsningen eller størrelsen på hvert bilde. Ved GMD har de lagret flere versjoner av samme video, men med forkjellige kvaliteter (og dermed forskjellige datamengder).

Hvordan disse to metodene kombineres avhenger av bufferutnyttelsen. Ved full bufferutnyttelse, noe som indikerer at hentingen av data ikke er noe problem, tilbys den beste videokvaliteten. Ved lavere bufferutnyttelser brukes lavere kvaliteter på bildet, og dette kombineres igjen med liten eller stor grad av rammedropping.

Ved å bruke en tilsvarende overvåkningsteknikk av tilgjengelige ressurser som ved GMD, kan de oppståtte sidefeilene i noen av forsøkene vært unngått.

En annen form for tilpassning av dataleveringene som også kan passe i vårt scenario er å bare droppe eller forsinke fremvisningen av den rammen som kommer for sent i fremvisningen. I en avspilling av video vil det omtrent ikke kunne merkes om vi hopper over et bilde, og for foilene kan byttet av foil forsinkes med et par sekunder uten at dette blir problematisk.

# Del III

# Appendiks

# Tillegg A

# Ordliste og forkortelser

## A.1 Ordliste

Nesten all litteraturen vi har brukt har vært på engelsk, og det har vært vanskelig å finne en god oversettelse til mange av ordene. I noen tilfeller har vi beholdt det engelske ordet. Under viser vi en ordliste over de fleste oversatte ordene. I oversettingsarbeidet har vi hovedsaklig brukt følgende ordbøker:

- Berulfsen, B., Berulfsen, T., "Engelsk-Norsk ordbok", femte utgave, Kunnskapsforlagets blå ordbøker, Aschehoug-Gyldendal, 1989
- Reiersen, O., "Engelsk-Norsk teknisk ordbok", 4. reviderte og omarbeidede utgave, F. Bruns bokhandels forlag, Trondheim, 1990
- Svenkerud, H., "Cappelens store Engelsk-Norske ordbok", 2. reviderte og utvidede utgave, J. W. Cappelens forlag AS, Oslo, 1988

**ORDLISTE BRUKT I DENNE OPPGAVEN**

| **Engelsk** | **Norsk** |
|---|---|
| access path | aksessti |
| allocation | allokering, tildelelse av plass |
| | |
| belateral negotiation | tosidig forhandling |
| best-effort qos | best mulig qos |
| bit | bit |
| bound | begrense |
| broadcast | kringkasting |
| buffer frames | bufferrammer |
| byte | byte |
| | |
| capability | evne, egenskap, mulighet |
| cache memory | hurtigbuffer |
| compulsory qos | tvungen qos |
| concurrency (control) | samtidighet (-skontroll) |
| congestion | sperring, opphoping, flaskehals, overflyt, overbelastning |
| constrained | begrenset, ufrihet |
| constraint | restriksjon |

*fortsetter på neste side*

*fortsettelse fra forrige side*

| **Engelsk** | **Norsk** |
|---|---|
| continuity | kontinuitet |
| contigous | tilstøtende |
| conversational | konverserbar |
| | |
| data consistency | data konsistens |
| data definition language | datadefinisjonsspråk |
| data integrity | data integritet |
| data manipulation language | datamanipulasjonsspråk |
| data path | datasti |
| data recovery | (data) gjennoppretting |
| database management system | databasehåndteringsystem |
| deadline | tidsfrist |
| degradation | forringelse, degradering |
| disk failure recovery | gjennoppretting etter en diskfeil |
| disk layout | disk arkitektur/organisering, hardware og software |
| dispatch | avslutte |
| display unit | displayenhet, fremviser |
| drums | tromler |
| | |
| earliest deadline first | tidligste tidsfrist først |
| eligible | verd å velge, kvalifisert, egnet |
| encapsulate | kapsle inn |
| encapsulation | innkapsling |
| | |
| entity | entitet, objekt |
| execption | unntak |
| | |
| frame | ramme |
| | |
| glitch | feil, svikt, (data) uforklarlig elektronisk fenomen |
| granularity | kornethet, oppdeling |
| granule | (liten) partikkel/korn/del |
| grouping | gruppering |
| guaranteed qos | garantert qos |
| | |
| hardware | hardware |
| hit rate | treffrate |
| | |
| inherent | iboende, som hører naturlig til |
| interleaving | innskytning, innfelling (blanding) |
| interrupt | avbrudd |
| invalidate | gjøre ugyldig, forkaste |
| invalidation (-rate) | ugyldiggjøring (-srate) |
| inversion | inverson, omvendt |
| item | objekt |

*fortsetter på neste side*

*fortsettelse fra forrige side*

| **Engelsk** | **Norsk** |
|---|---|
| jitter | forsinkelse |
| | |
| latency | ventetid |
| layout | layout |
| laxity | [(deadline-nåtid)-gjenstående eksekveringstid] |
| least/most relevant presentation | minst/mest relevant for presentasjon |
| least frequently used | lavest referansefrekvens |
| least laxity first | minst laxity først |
| least reference density | minst referansetetthet |
| least recently used | sist brukt |
| log | (1) logg (journal), (2) logge (føre en journal) |
| log sequence number | logg sekvensnummer |
| | |
| mainframe | prossesorenhet |
| management | håndtering |
| mapping | mapping, avbilding |
| memory | minne (internminne, hovedminne) |
| metascheduler | metadatafordeler |
| | |
| negotiation | forhandling |
| non-preemptable process | prosess ikke som kan avbrytes av andre prosesser med høyere prioritet |
| | |
| on-demand | på forespørsel |
| overhead | overhead, ekstra arbeid |
| | |
| page fault | sidefeil |
| page | side |
| pager | sidehåndteringsmekanisme |
| paging | sideutbytting, sidehåndtering |
| path | sti |
| performance | ytelse |
| persistence | persistens |
| pinning | sette fast (i minnet) |
| playback | avspilling |
| precedence | prioritet, rangfølge |
| preempt | okkupere, dytte bort |
| preemptable prosess | prosess som kan avbrytes av andre prosesser med høyere prioritet |
| presentational | presenterbar |
| printer | skriver |
| | |
| quality-of-service | tjenestekvalitet |
| query | spørsmål |
| query language | spørrespråk |
| query processing | spørsmålsprosessering |

*fortsetter på neste side*

*fortsettelse fra forrige side*

| Engelsk | Norsk |
|---|---|
| rate monotonic | ratemonoton |
| realtime | sanntid |
| record | post |
| redundancy | overflødighet |
| reliability | pålitelighet |
| remote access | fjerntilgang, -aksess |
| remote procedure call | fjernprosedyrekall |
| response time | responstid |
| retrive | hente ut |
| rollback | rulle tilbake |
| runtime | kjøretid |
| sample (audio) | sampel |
| scanner | skanner |
| schedulatibility test | ressursfordelings test |
| scheduler | prosessmanager |
| scheduling | fordeling/planlegging av ressursene |
| service provider | tjenesteyter |
| service user (called -) (calling -) | tjenestebruker (kallende -) (kalt -) |
| sink | datamottaker |
| software | software |
| spatial | romlig |
| switch | bryter |
| threads | miniprosess, tråder |
| threshold qos | terskel qos |
| throughput | gjennomstrømning |
| time garbled | tids forvirret, tids uriktig |
| timeliness | aktualitet |
| timely | tidsriktig |
| time slicing | tidsdeling |
| track | spor |
| trade-off | kompromis |
| transparency | transparens |
| transparent | transparent |
| trap | felle |
| triangular negotiation | treangulær forhandling |
| unilateral negotiation | ensidig forhandling |
| video/audio streams | video/audio datastrømmer |
| virtuel memory | virtuellt minne |
| volume | (1) disk, (2) årgang |
| worstcase | verste mulige tilfelle |

## A.2 Forkortelser

I dette avsnittet har vi samlet alle forkortelsene vi har brukt i denne oppgaven.

**FORKORTELSER BRUKT I DENNE OPPGAVEN**

| **Forkortelser** | |
|---|---|
| ACID | atomicity, consistency, isolation, durability |
| ATM | asynchronous transfer mode |
| | |
| bits/s | bits per sekund |
| Bytes/s | Bytes per sekund |
| | |
| CD | compact disc |
| | |
| DB | database |
| DBMS | database management system |
| DBS | databasesystem |
| DDL | data definition language |
| DHS | datahåndteringssystemer |
| DML | data manipulation language |
| DRAM | dynamic RAM |
| | |
| EDF | earliest deadline first |
| EOF | end of file |
| | |
| FAT | filallokeringstabell |
| FIFO | først inn, først ut |
| | |
| GCLOCK | generalisert CLOCK |
| GMD | german national research center for information technology |
| GIF | graphics image format |
| | |
| HDTV | high definition television |
| HTML | hypertext markup language |
| Hz | hertz |
| I/O | input/output |
| IP | intelligent preloading |
| ISO | international standards organization |
| | |
| JPEG | joint photographic expert group |
| | |
| L/MRP | least/most relevant for presentation |
| LFU | least frequently used |
| LLF | least laxity first |
| LRD | least reference density |
| LRU | least recently used |
| LSN | log sequence number |
| | |
| MMS | multimediasystem |

*fortsetter på neste side*

*fortsettelse fra forrige side*

| **Forkortelser** | |
|---|---|
| MoD | media-on-demand |
| MPEG | moving picture expert group |
| | |
| NoD | news-on-demand |
| | |
| OODBS | objektorientert databasesystem |
| OS | operativsystem |
| OSI | open systems interconnection reference model |
| | |
| PAL | phase alternating line |
| PC | personal computer |
| PCM | pulse coded modulation |
| PE | presentasjonsenhet |
| PO | presentasjonsobjekt |
| | |
| QoS | quality-of-service |
| | |
| RAM | random access memory |
| RM | rate monotonic |
| RR | round robin |
| | |
| SCSI | small computer system interface |
| SRAM | static RAM |
| | |
| TLB | translation-lookaside buffer |
| TV | televisjon |
| | |
| UI | unik identifikator |
| UNIK | universitetsstudiene på Kjeller |
| USIT | universitetets senter for informasjonsteknologi |
| | |
| VoD | video-on-demand |
| | |
| WAL | write ahead log |
| WORM | write once, read many |
| WWW | world wide webb |

## A.3 Notasjoner i algoritmen

Her gir vi en oversikt over de forskjellige notasjonene vi har brukt under modelleringen av algoritmen.

**NOTASJONER BRUKT I MODELLERINGEN AV ALGORITMEN**

| **Notasjoner** | |
|---|---|
| $\mathrm{A}_s$ | interaksjonssettet A for presentasjonsstatusen s |
| PE | presentasjonsenhet |
| PO | presentasjonsobjekt |
| | |
| c, $\mathrm{c}_j$ | notasjone for PE'er |
| $\mathrm{dr}_{\mathrm{A}_s}$ | distanserelevansfunksjonen for $\mathrm{A}_s$ |
| dropp | presentasjonmodus |
| i | distansen til presentasjonspunktet |
| p | presentasjonspunktet |
| $\mathrm{r}_{\mathrm{A}_s}$ | relevansfunksjonen for $\mathrm{A}_s$ |
| s | presentasjonsstatus |

# Tillegg B

# Kildekoden

Her vises "kildekoden" for implementasjonen av bufferhåndteringsmekanismen for simuleringsforsøkene vi har gjort i matlab [MathWorks 92]. Denne koden er for simuleringene med tre brukere. Koden for en bruker er helt tilsvarende, men noen parametere er satt forskjellig.

## B.1 Simuleringen

Her er koden for selve simuleringen. Sidestørrelsene, bufferstørelsene, referansestrenfene og antall brukere er blitt variert.

```
% ##########################################################
% Simulering av en avspilling av en forelesning
% ##########################################################

format long;

load forelsn;                    % Legg inn den lagrede forelesningen
load s3X; refStreng = string3X;  % Legg inn referansestrengen

sidestrl         = 32            % Sidestørrelsen i KBytes
bufferstorrelse  = 16000         % Bufferstørrelse i KBytes

CPUhastighet     = 2e8;          % CPU hastigheten i Hz

videoPE          = 60;           % Størrelser på de forskjellige PEene i KBytes
audioPE          = 32;
dokumentkameraPE = 60;
foilPE           = 200;

antV = ceil(videoPE / sidestrl);  % Antall buffersider per PE
antA = ceil(audioPE / sidestrl);
antD = ceil(dokumentkameraPE / sidestrl);
antF = ceil(foilPE / sidestrl);

antallsidefeil  = 0; antallutbyttinger = 0;

bufferplasser = bufferstorrelse / sidestrl;

PPtid = 1/24;                    % Tid for et presentasjonspunkt

feilpunkter = [];                % For å se hvor en sidefeil oppstår
hentetid    = [];                % For å se hvor lang tid det tar å hente data

% ##########################################################
% Opprett et tomt buffer med plass til "bufferplasser" PEer.

buffer = [];

for teller = 1:bufferplasser
   buffer = [buffer; 0 0];
end

% ##########################################################
% Utfør simuleringen.

pPunkt = [0 0 0];
i = 1;
ovregrense = size(refStreng, 1);
forrigeHenting = 0;
antIbrukt = 0;
nestehent = 0;

while i <= ovregrense
```

```
   tid = 0;

   if  antIbrukt < 12
      nestehent = 12;
   else
      nestehent = antIbrukt;
   end;

   [pPunkt, dropp] = presentasjonsP(i, pPunkt, refStreng)

   tid = tid + 815/CPUhastighet;          % Eksekveringstid for start av mekanismen = 815 sykler

   % ############################################################
   % Selve algoritmen
   % ############################################################

   if i >= forrigeHenting + nestehent
      % Forhåndshent data -> gått et halvt sekund eller mer
      forrigeHenting = i;
      [brukttid, buffer, antBytt] =
          LMRPhent(pPunkt, dropp, buffer, bufferplasser, sidestrl, antV, antA, antD, antF, forelesning);
      tid = tid + brukttid;
      hentetid = [hentetid; tid pPunkt];
      antIbrukt = ceil(tid / PPtid);
      antallutbyttinger = antallutbyttinger + antBytt;
   else
      % Sjekk om data i bufferet. Ikke -> hent.
      [brukttid, feil] =
          LMRPfinn(pPunkt, dropp, buffer, bufferplasser, sidestrl, antV, antA, antD, antF, forelesning);
      tid = tid + brukttid;

      if feil == 1
         [brukttid, buffer, antBytt] =
             LMRPhent(pPunkt, dropp, buffer, bufferplasser, sidestrl, antV, antA, antD, antF, forelesning);
         tid = tid + brukttid;
         hentetid = [hentetid; tid pPunkt];
         antIbrukt = ceil(tid / PPtid);
         antallutbyttinger = antallutbyttinger + antBytt;
         forrigeHenting = i;
         antallsidefeil = antallsidefeil + 1;
         feilpunkter = [feilpunkter; pPunkt tid];
      end;
   end;

   % ############################################################

   i = i + 1;
end;

% ############################################################
% Skriv ut resultater.

................
```

## B.2 Beregning av presentasjonspunktet

Hvordan beregningen av presentasjonspunktet ble gjort etter en referansestreng bestående av presentasjonsmodus i hvert punkt, det vil si verdiene -2, -1, 1 eller 2.

```
function [pPunkt, dropp] = presentasjonsP(i, forrigePP, refStreng)

% ##########################################################
% Sjekk om siden er i bufferet
% ##########################################################

if refStreng(i, 1) == 0             % Ingen presentasjon
   pPunkt(1) = 0;
   dropp(1)  = 0;
elseif refStreng(i, 1) == 5         % Hopp
   pPunkt(1) = refStreng(i, 2);
   dropp(1)  = refStreng(i + 1, 1);
else                                % Rask eller vanlig avspilling
   pPunkt(1) = forrigePP(1) + refStreng(i, 1);
   dropp(1)  = refStreng(i, 1);
end;


if refStreng(i, 3) == 0             % Ingen presentasjon
   pPunkt(2) = 0;
   dropp(2)  = 0;
elseif refStreng(i, 3) == 5         % Hopp
   pPunkt(2) = refStreng(i, 4);
   dropp(2)  = refStreng(i + 1, 3);
else                                % Rask eller vanlig avspilling
   pPunkt(2) = forrigePP(2) + refStreng(i, 3);
   dropp(2)  = refStreng(i, 3);
end;
```

```
if refStreng(i, 5) == 0              % Ingen presentasjon
   pPunkt(3) = 0;
   dropp(3)  = 0;
elseif refStreng(i, 5) == 5          % Hopp
   pPunkt(3) = refStreng(i, 6);
   dropp(3)  = refStreng(i + 1, 5);
else                                 % Rask eller vanlig avspilling
   pPunkt(3) = forrigePP(3) + refStreng(i, 5);
   dropp(3)  = refStreng(i, 5);
end;
```

## B.3 Finn data i bufferet

Her sjekker vi bare om dataene som trenges for presentasjonen ligger i bufferet. Hvis de ikke gjøre det gis det en sidefeil.

```
function [tidBrukt, dataIBufferet] =
    LMRPfinn(p, dropp, buffer, bufferstrl, sidestrl, antV, antA, antD, antF, forelesning)

CPUhastighet     = 2e8;      % CPU hastigheten i Hz

% ##########################################################
% Sjekk om siden er i bufferet
% ##########################################################

bufferfeil = 0; utbyttinger = 0;

feil = 0;

bruktid = 0; antallSykler = 0;


antBrukere = 3;

for teller = 1:antBrukere
if p(teller) ~= 0
   % ##############################################################
   % Video

   videoPE = forelesning(p(teller), 2);

   vteller = 1;

   antallSykler = antallSykler + 5;             % Eksekveringstid = 5 sykler

   while feil == 0 & vteller <= antV
      antallSykler = antallSykler + 5;          % Eksekveringstid = 5 sykler

      % Videosider starter på side 1
      sidenummer = ((videoPE - 1) * antV) + vteller;

      antallSykler = antallSykler + 5;          % Eksekveringstid = 5 sykler

      [tid, funnetIBufferet, indeks] = finn(sidenummer, buffer, bufferstrl);
      brukttid = brukttid + tid;

      if funnetIBufferet == 0
         feil = 1;
         antallSykler = antallSykler + 2;       % Eksekveringstid = 2 sykler
      end;

      vteller = vteller + 1;
   end;

   if feil == 1                                 % Testen i "while" utføres uansett
      antallSykler = antallSykler + 5;          % Eksekveringstid = 5 sykler
   end;

   % ##############################################################
   % Audio

   audioPE = forelesning(p(teller), 3);

   ateller = 1;

   antallSykler = antallSykler + 5;             % Eksekveringstid = 5 sykler

   while feil == 0 & ateller <= antA & dropp(teller) == 1
      antallSykler = antallSykler + 5;          % Eksekveringstid = 5 sykler

      % Audiosider starter på side 200001
      sidenummer = ((audioPE - 1) * antA) + ateller + 200000;

      antallSykler = antallSykler + 6;          % Eksekveringstid = 6 sykler

      [tid, funnetIBufferet, indeks] = finn(sidenummer, buffer, bufferstrl);
      brukttid = brukttid + tid;

      if funnetIBufferet == 0
         feil = 1;
```

```
        antallSykler = antallSykler + 2;       % Eksekveringstid = 2 sykler
     end;

     ateller = ateller + 1;
  end;

  if feil == 1                                 % Testen i "while" utføres uansett
     antallSykler = antallSykler + 5;          % Eksekveringstid = 5 sykler
  end;

  % ############################################################
  % Dokumentkamera

  dokumentkameraPE = forelesning(p(teller), 4);

  dteller = 1;

  antallSykler = antallSykler + 5;             % Eksekveringstid = 5 sykler

  while feil == 0 & dteller <= antD & dokumentkameraPE ~= 0
     antallSykler = antallSykler + 5;          % Eksekveringstid = 5 sykler

     % Dokumentkamerasider starter på side 400001
     sidenummer = ((dokumentkameraPE - 1) * antD) + dteller + 400000;

     antallSykler = antallSykler + 6;          % Eksekveringstid = 6 sykler

     [tid, funnetIBufferet, indeks] = finn(sidenummer, buffer, bufferstrl);
     brukttid = brukttid + tid;

     if funnetIBufferet == 0
        feil = 1;
        antallSykler = antallSykler + 2;       % Eksekveringstid = 2 sykler
     end;

     dteller = dteller + 1;
  end;

  if feil == 1                                 % Testen i "while" utføres uansett
     antallSykler = antallSykler + 5;          % Eksekveringstid = 5 sykler
  end;

  % ############################################################
  % Foil

  foilPE = forelesning(p(teller), 5);

  fteller = 1;

  antallSykler = antallSykler + 5;             % Eksekveringstid = 5 sykler

  while feil == 0 & fteller <= antF
     antallSykler = antallSykler + 5;          % Eksekveringstid = 5 sykler

     % Foilsider starter på side 600001
     sidenummer = ((foilPE - 1) * antF) + fteller + 600000;

     antallSykler = antallSykler + 6;          % Eksekveringstid = 6 sykler

     [tid, funnetIBufferet, indeks] = finn(sidenummer, buffer, bufferstrl);
     brukttid = brukttid + tid;

     if funnetIBufferet == 0
        feil = 1;
        antallSykler = antallSykler + 2;       % Eksekveringstid = 2 sykler
     end;

     fteller = fteller + 1;
  end;

  if feil == 1                                 % Testen i "while" utføres uansett
     antallSykler = antallSykler + 5;          % Eksekveringstid = 5 sykler
  end;
end;
end;
% ############################################################
% Returner

tidBrukt = tid  + antallSykler/CPUhastighet;;
dataIBufferet = feil;
```

## B.4 Henting av data til bufferet

Her forhåndshenter vi to sekunder med data.

```
function [tidBrukt, buf, antallByttinger] =
    LMRPhent(p, dropp, buffer, bufferstrl, sidestrl, antV, antA, antD, antF, forelesning)

% ############################################################
% Sideutbytting og forhåndshenting av data etter L/MRP.
% ############################################################
```

```
maxVideoPE = 7200;
minVideoPE = 1;

maxAudioPE = 7200;
minAudioPE = 1;

maxDokumentameraPE = 4320;
minDokumentameraPE = 2881;

maxFoilPE = 3;
minFoilPE = 1;


Vf = 47;                    % Forhåndshenter f + 1 PEer (2 sekunder med data).
Af = 1;
Df = 11;

CPUhastighet     = 2e8;     % CPU hastigheten i Hz

brukttid = 0;
antallSykler = 0;

% ##########################################################
% Sett inn nye relevansverdier i bufferet
% ##########################################################

[tid, buffer] = settRelVerdier(p, dropp, buffer, bufferstrl, antV, antA, antD, antF, forelesning);

brukttid = brukttid + tid;

% ##########################################################
% Bytt ut de minst relevante PEer med de mest relevante PEer
% Forutsetter bufferstørrelse > f.
% ##########################################################

byttingsteller = 0;
byttinger = 0;

antBrukere = 3;

for teller = 1:antBrukere
if p(teller) ~= 0
   % ##########################################################
   % Video

   i = 0; j = 0;

   while j <= Vf & i + p(teller) <= maxVideoPE & minVideoPE <= i + p(teller)
      antallSykler = antallSykler + 8;           % Eksekveringstid = 8 sykler

      videoPE = forelesning(i + p(teller), 2);

      antallSykler = antallSykler + 3;           % Eksekveringstid = 3 sykler

      for vteller=1:antV
         % Videosider starter på side 1
         sidenummer = ((videoPE - 1) * antV) + vteller;

         [tid, buffer, antallutbyttinger] = LMRPfinnBytt(sidenummer, buffer, bufferstrl);
         brukttid = brukttid + tid;
         byttingsteller = byttingsteller + antallutbyttinger;
         byttinger =  byttinger + antallutbyttinger;

         antallSykler = antallSykler + 10;       % Eksekveringstid = 10 sykler
      end;

      i = i + dropp(teller); j = j + 1;
      antallSykler = antallSykler + 6;           % Eksekveringstid = 6 sykler
   end;

   % Tiden for selve uthentingen av data ved uthenting av alle dataene samtidig.

   brukttid = brukttid + hent(2 * sidestrl * byttinger);
   byttinger = 0;
   % ##########################################################
   % Audio

   i = 0; j = 0;

   while j <= Af & i + p(teller) <= maxAudioPE & minAudioPE <= i + p(teller) & dropp(teller) == 1
      antallSykler = antallSykler + 9;           % Eksekveringstid = 9 sykler

      audioPE = forelesning(i + p(teller), 3);

      antallSykler = antallSykler + 3;           % Eksekveringstid = 3 sykler

      for ateller=1:antA
         % Audiosider starter på side 200001
         sidenummer = ((audioPE - 1) * antA) + ateller + 200000;

         [tid, buffer, antallutbyttinger] = LMRPfinnBytt(sidenummer, buffer, bufferstrl);
         brukttid = brukttid + tid;
         byttingsteller = byttingsteller + antallutbyttinger;
         byttinger =  byttinger + antallutbyttinger;
```

```
      antallSykler = antallSykler + 10;        % Eksekveringstid = 10 sykler
   end;

   i = i + dropp(teller); j = j + 1;
   antallSykler = antallSykler + 6;            % Eksekveringstid = 6 sykler
end;

% Tiden for selve uthentingen av data ved uthenting av alle dataene samtidig.

brukttid = brukttid + hent(2 * sidestrl * byttinger);
byttinger = 0;

% ############################################################
% Dokumentkamera

i = 0; j = 0;

while j <= Df & i + p(teller) <= maxDokumentameraPE & minDokumentameraPE <= i + p(teller)
   antallSykler = antallSykler + 8;            % Eksekveringstid = 8 sykler

   dokumentkameraPE = forelesning(i + p(teller), 4);

   antallSykler = antallSykler + 3;            % Eksekveringstid = 3 sykler

   if dokumentkameraPE ~= 0
   for dteller=1:antD
      % Dokumentkamerasider starter på side 400001
      sidenummer = ((dokumentkameraPE - 1) * antD) + dteller + 400000;

      [tid, buffer, antallutbyttinger] = LMRPfinnBytt(sidenummer, buffer, bufferstrl);
      brukttid = brukttid + tid;
      byttingsteller = byttingsteller + antallutbyttinger;
      byttinger = byttinger + antallutbyttinger;

      antallSykler = antallSykler + 10;        % Eksekveringstid = 10 sykler
   end;
   end;

   i = i + dropp(teller); j = j + 1;
   antallSykler = antallSykler + 6;            % Eksekveringstid = 6 sykler
end;

% Tiden for selve uthentingen av data ved uthenting av alle dataene samtidig.

brukttid = brukttid + hent(2 * sidestrl * byttinger);
byttinger = 0;

% ############################################################
% Foil

foilPE = forelesning(i + p(teller), 5);

antallSykler = antallSykler + 3;               % Eksekveringstid = 3 sykler

for fteller=1:antF      % Hent denne foilen
   % Foilsider starter på side 600001
   sidenummer = ((foilPE - 1) * antF) + fteller + 600000;


   [tid, buffer, antallutbyttinger] = LMRPfinnBytt(sidenummer, buffer, bufferstrl);
   brukttid = brukttid + tid;
   byttingsteller = byttingsteller + antallutbyttinger;
   byttinger = byttinger + antallutbyttinger;

   antallSykler = antallSykler + 10;           % Eksekveringstid = 10 sykler
end;

% Tiden for selve uthentingen av data ved uthenting av alle dataene samtidig.

brukttid = brukttid + hent(2 * sidestrl * byttinger);
byttinger = 0;

pPunktTOsek = p(teller) + (dropp(teller) * 48);

if pPunktTOsek > 0 & pPunktTOsek < 7200
FoilomTOsek = forelesning(pPunktTOsek, 5);
else
   FoilomTOsek = foilPE;
end;

antallSykler = antallSykler + 16;              % Eksekveringstid = 16 sykler

if FoilomTOsek ~= foilPE   % Neste foil skal vises om 2 sekunder hent denne også
   for fteller=1:antF
      % Foilsider starter på side 600001
      sidenummer = ((foilPE - 1) * antF) + fteller + 600000;

      [tid, buffer, antallutbyttinger] = LMRPfinnBytt(sidenummer, buffer, bufferstrl);
      brukttid = brukttid + tid;
      byttingsteller = byttingsteller + antallutbyttinger;
      byttinger =  byttinger + antallutbyttinger;

      antallSykler = antallSykler + 10;        % Eksekveringstid = 10 sykler
   end;
```

```
   end;

   % Tiden for selve uthentingen av data ved uthenting av alle dataene samtidig.

   brukttid = brukttid + hent(2 * sidestrl * byttinger);
   byttinger = 0;
end;
end;
% ##############################################################
% Returner tiden vi har brukt og bufferet

tidBrukt = brukttid + antallSykler/CPUhastighet;
buf = buffer;
antallByttinger = byttingsteller;
```

## B.5 "Oppslag" i sidetabellen

Vi har ikke implementert noen sidetabell, men simulerer en ved å søke direkte i bufferet. Eksekveringstiden er imidlertid beregnet etter et oppslag i SPARCs tre-nivås sidetabell.

```
function [tidBrukt, funnetIBufferet, indeks] = finn(sidenummer, buffer, bufferstrl)

CPUhastighet     = 2e8;      % CPU hastigheten i Hz

% ##############################################################
% I simuleringen har vi ikke implementert noen sidetabell. Vi søker
% her etter siden i selve bufferet.
% ##############################################################

% ##############################################################
% Finner ut om en side er i bufferet.

tabellsokeTid = 3 * 90e-9;  % Tid for tre minneaksesser i sidetabellen i bufferet

eksTid = 6;                 % Eksekveringstid for utregning av 3 indekser gitt i klokkesykler

bufferindeks = 1; funnet = 0;

while funnet == 0 & bufferindeks <= bufferstrl    % Søk etter siden i bufferet
   if buffer(bufferindeks,1) == sidenummer
      funnet = 1;
   else
      bufferindeks = bufferindeks + 1;
   end;
end;

% ##############################################################
% Returner tiden brukt, om siden var i bufferet og indeksen til siden i bufferet

tidBrukt = tabellsokeTid + eksTid/CPUhastighet;
funnetIBufferet = funnet;
indeks = bufferindeks;
```

## B.6 Hent inn og bytt ut data i bufferet

Når data skal hentess finner denne funksjonen ut om dataene ligger i bufferet eller må hentes fra disken. I det sidet tilfellet, beregnes også hvilden side som må byttes ut.

```
function [tidBrukt, buf, antallutbyttinger] = LMRPfinnBytt(sidenummer, buffer, bufferstrl)

CPUhastighet     = 2e8;      % CPU hastigheten i Hz

% ##############################################################
% Finner ut om en side er i bufferet. Hvis ikke byttes denne inn
% ved at en annen byttes ut etter LMRP
% ##############################################################

byttinger = 0; brukttid = 0; antallSykler = 0;

% ##############################################################
% Søk etter siden i bufferet

[tid, funnet, indeks] = finn(sidenummer, buffer, bufferstrl);

brukttid = brukttid + tid;

% ##############################################################
% Hvis ikke siden er i bufferet -> bytt ut siden i bufferet med minst relevansverdi.
% Hvis siden er i bufferet -> sett relevansverdien til 1

if funnet == 1
   buffer(indeks,2) = 1;
   antallSykler = antallSykler + 2;          % Eksekveringstid = 2 sykler
```

```
else
   % Finn siden i bufferet med minst relevansverdi
   bufferindeks = 1; verdi = buffer(bufferindeks, 2);
   antallSykler = antallSykler + 8;           % Eksekveringstid = 8 sykler

   for teller = 2 : bufferstrl
      if buffer(teller,2) < verdi
         verdi = buffer(teller,2);
         bufferindeks = teller;
         antallSykler = antallSykler + 13;    % Eksekveringstid = 13 sykler
      else
         antallSykler = antallSykler + 6;     % Eksekveringstid = 6 sykler
      end;
   end;

   % Bytt ut siden i bufferet med minst relevans med side "sidenummer"
   buffer(bufferindeks,1) = sidenummer;
   buffer(bufferindeks,2) = 1;

   byttinger = byttinger + 1;
   antallSykler = antallSykler + 10;          % Eksekveringstid = 10 sykler
end;

% ##############################################################
% Returner

tidBrukt = brukttid + antallSykler/CPUhastighet;
antallutbyttinger = byttinger;
buf = buffer;
```

## B.7 Setting av relevansverdier i bufferet

For hver gang data byttes ut i bufferet, beregnes byttingen ut i fra hvilken side som har lavest relevansverdi. Denne funksjonen beregner verdiene ut i fra de forskjellige brukerenes presentasjonspunkt.

```
function [tidBrukt, buf] = settRelVerdier(pp, dropp, buffer, bufferstrl, antV, antA, antD, antF, forelesning)

% ##########################################################
% Funksjon som setter relevansverdier i bufferet
% ##########################################################

Valfa  = 1/14400; % "Synke"-parameterene for relevansverdiberegning.
Vbeta  = 1/720;
Vgamma = Vbeta;

Aalfa  = 1/10;
Abeta  = 1/30;
Agamma = Abeta;

Dalfa  = 1/3600;
Dbeta  = 1/180;
Dgamma = Dbeta;

Falfa  = 1/2;
Fbeta  = 1/2;

Vf = 47;  % Forhåndshenter f + 1 PEer (2 sekunder med data).
Af = 1;
Df = 11;

CPUhastighet     = 2e8;  % CPU hastigheten i Hz

antBrukere = 3;

brukttid = 0;
antallSykler = 0;
minneaksessTid = 90e-9;

% ##########################################################
% Sett inn relevansverdier

for bufferindeks = 1:bufferstrl
   sidenummer = buffer(bufferindeks,1);
   antallSykler = antallSykler + 3;                    % Eksekveringstid = 3 sykler

   if sidenummer == 0                                  % Dette er en tom bufferplass
      buffer(bufferindeks,2) = 0;
      antallSykler = antallSykler + 8;                 % Eksekveringstid = 8 sykler

   elseif sidenummer > 600000                          % Siden er en foil-side
      PEnummer = ceil((sidenummer - 600000)/antF);
      buffer(bufferindeks,2) = 0;

      antallSykler = antallSykler + 8;                 % Ekeskveringstid = 8 sykler

      for teller = 1:antBrukere
         if pp(teller) ~= 0
            presentasjonsFoil = forelesning(pp(teller), 5);
```

```
            pPunktTOsek = pp(teller) + (dropp(teller) * 48);

            antallSykler = antallSykler + 10;          % Ekeskveringstid = 10 sykler

            if pPunktTOsek > 0 & pPunktTOsek < 7200
               FoilomTOsek = forelesning(pPunktTOsek, 5);
            else
               FoilomTOsek = presentasjonsFoil;
            end;

            presentasjonsavstand = PEnummer - presentasjonsFoil;

            antallSykler = antallSykler + 10;          % Ekeskveringstid = 10 sykler

            if (presentasjonsavstand < 0 & dropp(teller) > 0) | (presentasjonsavstand > 0 & dropp(teller) < 0)
               % PE tilhører HISTORISKE
               verdi = drH(abs(presentasjonsavstand), Fbeta);
               antallSykler = antallSykler + 26;       % Eksekveringstid = 26 sykler
            elseif PEnummer == presentasjonsFoil | PEnummer == FoilomTOsek
               % PE tilhører FREMTIDIG
               verdi = 1;
               antallSykler = antallSykler + 28;       % Eksekveringstid = 28 sykler
            else
               verdi = 1 - Falfa * abs(presentasjonsavstand);
               antallSykler = antallSykler + 27;       % Eksekveringstid = 27 sykler
            end;
         end;

         buffer(bufferindeks,2) = max(buffer(bufferindeks,2), verdi);
         antallSykler = antallSykler + 2;              % Ekeskveringstid = 2 sykler
      end;

   elseif sidenummer > 400000                          % Siden er en dokumentkamera-side
      PEnummer = ceil((sidenummer - 400000)/antD);
      buffer(bufferindeks,2) = 0;

      antallSykler = antallSykler + 8;                 % Ekeskveringstid = 8 sykler

      for teller = 1:antBrukere
         if pp(teller) ~= 0
            DpresentPunkt = forelesning(pp(teller),4);
            presentasjonsavstand = PEnummer - DpresentPunkt;

            antallSykler = antallSykler + 10;           % Ekeskveringstid = 10 sykler

            if (presentasjonsavstand < 0 & dropp(teller) > 0) | (presentasjonsavstand > 0 & dropp(teller) < 0)
               % PE tilhører HISTORISKE
               verdi = drH(abs(presentasjonsavstand), Dbeta);
               antallSykler = antallSykler + 26;        % Eksekveringstid = 26 sykler
            elseif rem(DpresentPunkt, dropp(teller)) == rem(DpresentPunkt + presentasjonsavstand, dropp(teller))
               % PE tilhører FREMTIDIG
               verdi = drF(abs(presentasjonsavstand), Df, Dalfa);
               antallSykler = antallSykler + 29;        % Eksekveringstid = 29 sykler
            else
               % PE tilhører DROPP
               verdi = drD(abs(presentasjonsavstand), Dgamma);
               antallSykler = antallSykler + 28;        % Eksekveringstid = 28 sykler
            end;

            buffer(bufferindeks,2) = max(buffer(bufferindeks,2), verdi);
            antallSykler = antallSykler + 2;           % Ekeskveringstid = 2 sykler
         end;
      end;

   elseif sidenummer > 200000                           % Siden er en audio-side
      PEnummer = ceil((sidenummer - 200000)/antA);
      buffer(bufferindeks,2) = 0;

      antallSykler = antallSykler + 8;                  % Ekeskveringstid = 8 sykler

      for teller = 1:antBrukere
         if pp(teller) ~= 0
            ApresentPunkt = forelesning(pp(teller),3);
            presentasjonsavstand = PEnummer - ApresentPunkt;

            antallSykler = antallSykler + 10;           % Ekeskveringstid = 10 sykler

            if (presentasjonsavstand < 0 & dropp(teller) > 0) | (presentasjonsavstand > 0 & dropp(teller) < 0)
               % PE tilhører HISTORISKE
               verdi = drH(abs(presentasjonsavstand), Abeta);
               antallSykler = antallSykler + 26;        % Eksekveringstid = 26 sykler
            elseif dropp(teller) ~= 1
               % PE tilhører DROPP
               verdi = 1 - Agamma * abs(presentasjonsavstand);
               antallSykler = antallSykler + 28;        % Eksekveringstid = 28 sykler
            else
               % PE tilhører FREMTIDIG
               verdi = drF(abs(presentasjonsavstand), Af, Aalfa);
               antallSykler = antallSykler + 31;        % Eksekveringstid = 31 sykler
            end;

            buffer(bufferindeks,2) = max(buffer(bufferindeks,2), verdi);
            antallSykler = antallSykler + 2;           % Ekeskveringstid = 2 sykler
         end;
      end;
```

```
   else                                              % Siden er en video-side
      PEnummer = ceil(sidenummer/antV);
      buffer(bufferindeks,2) = 0;

      antallSykler = antallSykler + 8;               % Ekeskveringstid = 8 sykler

      for teller = 1:antBrukere
         if pp(teller) ~= 0
            VpresentPunkt = forelesning(pp(teller),2);
            presentasjonsavstand = PEnummer - VpresentPunkt;

            antallSykler = antallSykler + 10;        % Ekeskveringstid = 10 sykler

            if (presentasjonsavstand < 0 & dropp(teller) > 0) | (presentasjonsavstand > 0 & dropp(teller) < 0)
               % PE tilhører HISTORISKE
               verdi = drH(abs(presentasjonsavstand), Vbeta);
               antallSykler = antallSykler + 26;     % Ekeskveringstid = 26 sykler
            elseif rem(VpresentPunkt, dropp(teller)) == rem(VpresentPunkt + presentasjonsavstand, dropp(teller))
               % PE tilhører FREMTIDIG
               verdi = drF(abs(presentasjonsavstand), Vf, Valfa);
               antallSykler = antallSykler + 29;     % Eksekveringstid = 29 sykler
            else
               % PE tilhører DROPP
               verdi = drD(abs(presentasjonsavstand), Vgamma);
               antallSykler = antallSykler + 28;     % Eksekveringstid = 28 sykler
            end;

            buffer(bufferindeks,2) = max(buffer(bufferindeks,2), verdi);
            antallSykler = antallSykler + 2;         % Ekeskveringstid = 2 sykler
         end;
      end;
   end;
end;

% Tid for å sjekke sidenummer + sette inne ny relevansverdi, dvs to minnereferanser

brukttid = brukttid + (minneaksessTid * bufferstrl * 2 * antBrukere);


% ############################################################
% Returner

tidBrukt = brukttid + antallSykler/CPUhastighet;
buf = buffer;
```

## B.8 Distanserelevansfunksjonene

Distanserelevansfunksjonene beregner selve relevansverdiene ut i fra en “synkefaktor” (alfa, beta, gamma) og distansen til presentasjonspunktet.

### B.8.1 Interansjonssettet FREMTIDIG

```
function dr = drF(i,f, alfa)

% distanserelevansfunksjonen til FREMTIDIG

if i <= f
   dr = 1;
else
   dr = 1 - alfa * i;
end
```

### B.8.2 Interansjonssettet HISTORISK

```
function dr = drH(i, beta)

% distanserelevansfunksjonen til HISTORISK

dr = 1 - beta * i;
```

### B.8.3 Interansjonssettet DROPP

```
function dr = drD(i, gamma)

% distanserelevansfunksjonen til DROPP

dr = 1 - gamma * i;
```

## B.9 Beregning av overføringstid fra disken

Beregning av overføringstidene for å hente data fra disken SEAGATE ELITE 23 [Seagate 97]. Vi antar her at dataene som hentes samtidig ligger etter hverandre på disken.

```
function t = hent(R)

% Beregner hentetiden til R blokker på 512 Bytes fra disken
% t = gjennomsnittlig søketid + rotasjonsforsinkelse + selve overføringstiden

t1rot = 0.011111111111111;
B = 235;

trans = (t1rot ./ B) .* R;

t = 0.0013 + 0.00556 + trans;

return
```

# Bibliografi


[Abrossimov et al. 89a] Abrossimov, V., Rozier, M., Gien, M., “Virtual Memory Management in Chorus”, Workshop on Proceedings of the Progress in Distributed Operating Systems and Distributed Systems Management, Berlin, LNCS, Springer, april 1989, også som Teknisk rapport CS/TR-89-30.1, Chorus Systèmes, mai 1989

[Abrossimov et al. 89b] Abrossimov, V., Rozier, M., Shapiro, M., “Generic Virtual Memory Management for Operating System Kernels”, Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP ’89), Litchfield Park, Arizona, desember 1989, s. 123 - 136

[Annevelink et al. 95] Annevelink, J., Ahad, R., Carlson, A., Fishman, D., Heytens, M., Kent, W., “Object SQL - A language for the Design and Implementation of Object Databases”, i [Kim 95a], s. 42 - 68

[Bach 86] Bach, M., J., “The Design of The UNIX Operating System”, Prentice Hall, 1986

[Bakke et al. 94] Bakke, J., W., Hestnes, B., Martinsen, H., “Distance education in the electronic classroom”, Televerkets Forskningsinstitutt, rapport TF R 20/94, 1994

[Blakeley 95] Blakeley, J., A., “OQL[C++]: Extending C++ with ab Object Query Capability”, i [Kim 95a], s. 69 - 88

[Bringsrud et al. 94] Bringsrud, K., Å., Pedersen, G., “Distributed electronic classrooms with large electronic whiteboards”,The MUNIN project, USIT, 1994

[Buford 94] Buford, J., F., K. (Editor), “Multimedia Systems”, Addison-Wesley, 1994

[Campbell et al. 95] Campbell, A., Aurrecoechea, C., Hauw, L., “Architectural Perspectives on QoS Management in Distributed Multimedia Systems”, Second Workshop on Protocols for Multimedia Systems, PROMS’95, Saltzburg, Østerrike, oktober 1995, s. 329 - 354

[Carey et al. 94] Carey M., J., DeWitt, D., J., Franklin, M., J., Hall, N., E., McAuliffe M., L., Naughton, J., F., Schuh, D., T., Solomon, M., H., Tan, C., K., Tsatalos, O., G., White, S., J., Zwilling, M., J., “Shoring Up Persistent Applications”, Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data, Minneapolis, Minnesota, mai 1994, s. 383 - 394

[Chen et al. 93] Chen J., B., Bershad, B., N., “The Impact of Operating System Structure on Memory System Performance”, ACM Operating Systems Review (SIGOPS), årg. 27, nr. 5, desember 1993, s. 120 - 133

[Christodoulakis et al. 95] Christodoulakis, S., Koveos, L., “Multimedia Information Systems: Issues and Approaches”, i [Kim 95a], s. 318 - 337

[Coulson et al. 92] Coulson, G., Blair, G., S., Stefani, J., B., Horn, F., Hazard, L., "Supporting the Real-Time Requirements of Continuous Media in Open Distributed Processing", Teknisk rapport MPG-92-35, Universitetet i Landcaster, 1992

[Coulson et al. 94] Coulson, G., Blair, G., S., Robin, F., Shepherd, D., "Supporting Continuos Media Applications in a Micro-Kernel Enviroment", Teknisk rapport MPG-94-16, Universitetet i Landcaster, 1994

[Coulson et al. 95] Coulson, G., Blair, G., "Arcitectural Principles and Techniques for Distributed Multimedia Application Support in Operating Systems", ACM Operating Systems Review, årg. 29, nr. 4, oktober 1995, s. 17 - 24

[Dan et al. 90] Dan, A., Dias, D., M., Yu, P., S., "Database Buffer Model for the Data Sharing Enviroment", IEEE Data Engineering, 6th International Conference on Data Engineering, Los Angeles, California, USA, februar 1990, s. 538 - 544

[Danthine et al. 93] Danthine, A., Bonaventure, O., "From Best Effort to Enhanced QoS", Teknisk rapport, Universitetet i Liege, R2060/ULg/CIO/DS/P/004/b1, juli 1993

[Date 95] Date, C., J., "An Introduction to Database Systems", 6th edition, Addison-Wesley, 1995

[DeWitt et al. 96] DeWitt, D., J., Naughton, J., F., Shafer, J., C., Venkataraman, S., "Parallelizing OODBMS traversals: a performance evaluation", The VLDB Journal, Springer Verlag, årg. 5, nr. 1, 1996, s. 3 - 18

[Effelsberg et al. 84] Effelsberg, W., Haerder, T., "Principles of Database Buffer Management" ACM Transactions on Database Systems, årg. 9, nr. 4, desember 1984, s. 560 - 595

[Elmasri et al. 94] Elmasri, R., Navathe, S., B., "Fundamentals of Database Systems", 2nd Edition, Addison-Wesley, 1994

[Gemmell et al. 95] Gemmell, D., J., Vin, H., M., Kandlur, D., D., Rangan P., V., Rowe, L., A., "Multimedia Storage Servers: A Tutorial", IEEE Computer, årg. 28, nr. 5, mai 1995, s. 40 - 49

[Ghafoor 95] Ghafoor, A., "Multimedia Database Management: Perspectives and Challenges", 13th British National Conference on Databases, BNCOD'13, Manchester, United Kingdom, juli 1995, LNCS 940, Springer, 1995, s. 12 - 23

[Goebel et al. 95] Goebel, V., Johansen, B., H., Løchsen, H., C., Plagemann, T., "Next Generation Database Technologies for Advanced Communication Services", 3rd International Conference on Intelligence in Broadband Networks and Services (IS&N 95), Kreta, Hellas, oktober 1995, LNCS 998, Springer, 1995, s. 320-333

[Goebel et al. 96a] Goebel, V., Plagemann, T., "Data Management and QoS in Distributed Multimedia Systems: Towards an Integrated Framework", Teknisk rapport, Universitetet i Oslo, UNIK, januar 1996

[Goebel et al. 96b] Goebel, V., Plagemann, T., Berre, A.-J., Nygård, M., "OMODIS - Object-Oriented Modeling and Database Support for Distributed Systems", Norsk Informatikk Konferanse (NIK'96), Alta, Norge, november 1996, s. 7 - 18

[Goebel et al. 97] Goebel, V., Plagemann, T., Eini, I., Halvorsen, P., Lund, K., Løchsen, H., C., Rønning, L., "Aktiviteter innen distribuerte multimediasystemer på UNIK - det elektroniske klasseromscenarioet -", Teknisk rapport, UNIK, kommer i 1997

[Gopalakrishnan et al. 95] Gopalakrishnan, R., Parulkar, G., M., "A Framework for QoS Guarantees for Multimedia Applications within an Endsystem", German and Swiss Computer Science Society Conference, GISI'95, Zurich, Sveits, september 1995

[Gray et al. 93] Gray, J., Reuter, A., "Transaction Processing: Concepts and Techniques", Morgan Kaufmann Publishers, San Mateo, California, 1993

[Guillemont 91] Guillemont, M., "Microkernel Design Yields Real Time in a Distributed Environment", Computer Technology Review, vinter 1990, s. 13 - 19

[Hollfelder et al. 96] Hollfelder, S., Kraiß, A., Rakow, T., C., "A Buffer-Triggered Smooth Adaption Technique for Time-Dependent Media", Teknisk rapport nr. 1002, German National Research Center for Information Technology (GMD), juni 1996

[ISO/IEC 95] ISO/IEC, "Information Technology - Quality of Service - Framework - Final CD", ISO/IEC JTC 1/SC 21 N9680, CD 13236.2, juli 1995

[Johnson et al. 94] Johnson, T., Shasha, D., "2Q: A Low Overhead High Performance Buffer Management Replacement algoritm", Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994, s. 439 - 450

[Kemper et al. 94] Kemper, A., Kossmann, D., "Dual-Buffering Strategies in Object Bases", Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994, s. 427 - 438

[Kim 95a] Kim, W. (Editor), "Modern Database Systems", Addison-Wesley, 1995

[Kim 95b] Kim, W., "Introduction to Part 1: Next-Generation Database Technology", i [Kim 95a], s. 5 - 17

[Korth et al. 91] Korth, H., F., Silberschatz, A., "Database System Consepts", 2nd edition, McGraw-Hill, 1991

[Krieger et al. 95] Krieger, D., Andrews, T., "C++ Bindings to an Object Database", i [Kim 95a], s. 89 - 107

[Lau et al. 95] Lau, S., W., Lui, J., C., S., Wong, P., C., "A Cost-effective Near-line Storage Server for Multimedia System", Proceedings of the 11'th International Conference on Data Engineering, Tapei, Taiwan, mars 1995, s. 449 - 456

[Lockemann et al. 87] Lockemann, P., C., Schmidt, J., W. (Editorer), "Datenbank-Handbuch", Springer Verlag, Berlin, 1987

[Lund 97] Lund, K., "Teknikker for dataplassering i multimedia databasesystemer - En analytisk undersøkelse", Hovedfagsoppgave, UNIK, mai 1997

[MathWorks 92] MATLAB (Versjon 4.2c), High-Performance Numeric Computation and Visualization Software, User's Guide og Reference Guide, The MATH WORKS Inc, 1992

[Moser et al. 95] Moser, F., Kraiß, A., Klas, W., "L/MRP: A Buffer Management Strategi for Interactive Continuous Data Flows in a Multimedia DBMS", Proceedings of the 21th VLDB Conference, Zurich, Sveits, 1995, s. 275 - 286

[Moss 86] Moss, J., E., B.,"Getting the Operating System Out of the Way", IEEE Database Engineering, årg. 9, nr. 3, september 1986, s. 35 - 42

[Nahrstedt et al. 95] Nahrstedt, K., Smith, J., M., “The QoS Broker”, IEEE Multimedia, årg. 2, nr. 1, vår 1995, s. 53 - 67

[Ng et al. 94] Ng, R., T., Yang, J., “Maximizing Buffer and Disk Utilization for News-On-Demand”, Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994, s. 451 - 462

[O’Neil et al. 93] O’Neil, E., J., O’Neil, P., E., Weikum, G., “The LRU-K Page Replacement Algorithm For Database Disk Buffering”, Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., USA, mai 1993, s. 297 - 306

[OSI 88] “Open Systems Interconnection (OSI) - Model and Notation, Service Definition, Recommendations X.200 - X.219”, ITU International Telecommunication Union, CCITT, Blue Book, Volume VIII - Facsimile VIII.4, Melbourne, Australia, november 1988

[Oyang et al. 95] Oyang, Y.-J., Lee, M.-H., Wen, C.-H., Cheng, C.-Y., “Design of Multimedia Storage Systemes for On-Demand Playback”, Proceedings of the 11’th International Conference on Data Engineering, Tapei, Taiwan, mars 1995, s. 457 - 465

[Patterson et al. 94] Patterson, D., A., Hennessy, J., L., “Computer Organization and Design: The Hardware/Software Interface”, Morgan Kaufmann Publishers, San Mateo, Caifornia, 1994

[Plagemann et al. 95] Plagemann, T., Sæthre, K., A., Goebel, V., “ Application Requirements and QoS negotiation in Multimedia Systems”, Second Workshop on Protocols for Multimedia Systems, PROMS’95, Salzburg, Østerrike, oktober 1995

[Rakow et al. 95] Rakow, T., C., Neuhold, E., J., Löhr, M., “Multimedia Database Systems - The Notion and the Issues”, Datenbanksysteme in Büro, Technik und Wissenschaft, BTW’95, GI-Fachtagung, Dresden, Tyskland, mars 1995, Springer, 1995, s. 1 - 19

[Reddy et al. 94] Reddy, A., L., N., Wyllie, J., C., “I/O Issues in a Multimedia System”, IEEE Computer, årg. 27, nr. 3, mars 1994, s. 69 - 74

[Rodriguez et al. 95] Rodriguez, A., A., Rowe, L., A., “Multimedia Systems and Applications”, IEEE Computer, årg. 28, nr. 5, mai 1995, s. 20 - 22

[Rozier et al. 91] Rozier, M., Abrossimov, V., Armand, F., Boule, I, Gien, M., Gulliemont, M., Herrmann, F., Kaiser, C., Langlois, S., Léonard, P., Neuhauser, W, “Overview of the CHORUS Distributed Operating System”, Teknisk rapport CS/TR-90-25.1, Chorus Systèmes, februar 1991

[Rotem et al. 95] Rotem, D., Zhao, J., L., “Buffer Management for Video Database Systems”, Proceedings of the 11’th International Conference on Data Engineering, Tapei, Taiwan, mars 1995, s. 439 - 448

[Seagate 96] Seagate, http://www.seagate.com/new/sep96/ultrascsi.shtml, 1996

[Seagate 97] Seagate, http://www.seagate.com/disc/elite/elite.shtml, 1997

[Silberschatz et al. 97] Silberschatz, A., Korth, H., F., Sudarshan, S., “Database System Consepts”, 3rd edition, McGraw-Hill, 1997

[Soley et al. 95] Soley, R., M., Kent, W., “The OMG Object Model”, i [Kim 95a], s. 18 - 41

[Steinmetz 95] Steinmetz, R., “Analyzing the Multimedia Operating System”, IEEE Multimedia, årg. 2, nr. 1, vår 1995, s. 68 - 84

[Steinmetz et al. 95] Steinmetz, R., Nahrstedt, K., “Multimedia: Computing, Communications and Appications”, Prentice Hall, 1995

[Stonebraker 81] Stonebraker, M., “Operating System Support for Database Management”, Communications of the ACM, årg. 24, nr. 7, juli 1981, s. 412 - 418

[Stonebraker 84] Stonebraker, M., “Virtual Memory Transaction Management”, ACM Operating Systems Review, årg. 18, nr. 2, 1984, s. 8 - 16

[Sæthre 96] Sæthre, K., A., “Distribuerte multimedia applikasjoner og tjenestekvalitet”, Hovedfagsoppgave, UNIK, 1996

[Tanenbaum 87] Tanenbaum, A., S., “Operating Systems: Design and Implementation”, Prentice Hall, 1987

[Tanenbaum 92] Tanenbaum, A., S., “Modern Operating Systems”, Prentice Hall, 1992

[Tanenbaum 95] Tanenbaum, A., S., “Distributed Operating Systems”, Prentice Hall, 1995

[Tokuda 94] Tokuda, H., “Operating System Support for Continuous Media Applications”, i [Buford 94], s. 201 - 220

[Traiger 82] Traiger, I., L., “Virtual Memory Management for Database Systems”, ACM Operating Systems Review, årg. 16, nr. 4, oktober 1982, s. 26 - 48

[Vogel et al. 95] Vogel, A., Kerhervé, B., Gecsei, J., von Bochmann, G., “Distributed Multimedia and QoS: A Survey”, IEEE Multimedia, årg. 2, nr. 2, sommer 1995, s. 10 - 19

[Özsoyoğlu et al. 95] Özsoyoğlu, G., Snodgrass, R., T., “Temporal and Real-Time Databases: A Survey”, IEEE Transactions on Knowledge and Data Engineering, årg. 7, nr. 4, august 1995, s. 513 - 532

[Özsu et al. 95] Özsu, M., T., Szafron, D., El-Medani, G., Vittal, C., “An Object-Oriented Multimedia Database System for a News-on-Demand Application”, Multimedia Systems, årg. 3, nr. 4 & 5, 1995, s. 182 - 203