Oppsummering i TDT4186 Operativsystemer

Av en eller annen

http://folk.ntnu.no/askeland

KAPITTEL 1 - DATAMASKINER, INTRODUKSJON

Grunnleggende elementer

En datamaskin består av en prosessor, minne, og I/O moduler. Alle dissemodulene er koblet sammen.

Prosessor: Ansvarlig for å prosessere data og styre datamaskinen.

Primærminne: Flyktig minne, hvor data og programmer er lagret.

I/O moduler: Flytter data mellom datamaskinen og dens eksterne komponenter.

System buss: Tilbyr kommunikasjon mellom prosessorene, primærminnet og I/O modulene.

Prosessorens registre

En prosessor har et sett registre som er en type minne, som er kjappere og mindre enn primærminnet. Registrene kan deles inn i to typer:

Bruker-registre: Gjør det mulig for maskin- eller assembly-programmereren til å minske primærminne referansene ved å optimalisere registerforbruket. En kompiler vil typisk prøve å optimalisere et kompilert program ved å gjøre intelligente valg på hvilke variable som skal ligge i bruker-registrene og hvilke som skal i primær minnet. Typiske registre er: Data og adresse registre.

Kontroll og status registre: Brukt av prosessoren og styrt av operativsystemet for å kontrollere utførselen av programmer. Typiske registre: MAR, MBR, I/OAR, I/OBR, PC og IR.

Utførelse av instruksjoner

I sin enkleste form består utførelse av instruksjoner i tre steg: *fetch, decode, execute*. Prosessoren henter, *fetch*, instruksjoner, tyder dem, *decode*, og utfører dem, *execute*. En instruksjonssykel er den prosesseringen en enkelt instruksjon krever. I en typisk prosessor inneholder program telleren, *PC,* adressen til den neste instruksjonen som skal hentes, mens instruksjonsregisteret, *IR*, inneholder den gjeldende instruksjonen.

Avbrudd

Avbrudd er en mekanisme som brukes for å forbedre utnyttelsen av prosessorkraften. Med avbrudd kan prosessoren utføre andre instruksjoner mens en I/O operasjon arbeider. Et typisk avbrudd blir behandlet slik:

1. En modul sender et avbrudd til prosessoren.
2. Prosessoren gjør ferdig behandlingen av den gjeldende instruksjonen før den behandler avbruddet.
3. Prosessoren sier i fra til modulen at den har mottatt avbruddssignalet.

4. Prosessoren forbereder seg på å gi kontrollen til avbrudds rutinene og lagrer all informasjon om det gjeldende programmet, som blir dyttet på systemkontrollstakken.
5. Prosessoren laster inn begynnelsen av plasseringen til avbruddsbehandleren inn i programtelleren. Kontrollen er nå hos avbruddsbehandleren.
6. Avbruddsbehandleren lagrer all informasjon om registrene på programstakken.
7. Avbruddsbehandleren begynner nå å behandle det gjeldende avbruddet.
8. Når denne behandlingen er ferdig, lagrer den informasjon om registrene, og laster inn den gamle informasjonen.
9. Programmet som ble avbrutt av avbruddet kan nå kjøre igjen.

Minnehierarkiet

For å utnytte prosessorkraften må minne holde tempoet til prosessoren. For at det skal være mulig å ha kjappaksesstid, lav kostnad og stort minne, trenger vi et hierarki av minnekomponenter. Dette hierarkiet består av registre, cache, primærminne og harddisker. Når man går fra toppen og ned skjer følgende:

1. Kostnaden per bit minsker,
2. Kapasiteten øker,
3. Aksesstiden øker.

Cache-minne

Ideelt sett, skulle primærminnet hatt samme teknologi og aksesstid som registrene, men på grunn av kostnaden er ikke dette mulig. Løsningen er et lite, kjapt mellomlager mellom primærminnet og CPU'en.

I/O-kommunikasjon

Det er tre muligheter for I/O kommunikasjon:

1. Programmert IO,
2. Avbruddsdreven IO,
3. Direkte minne aksess (DMA).

Programmert I/O: Når en prosessor utfører et program og oppdager en instruksjon relatert til I/O, utfører den instruksjonen ved å sende et signal til den spesifikke I/O modulen. I/O modulen gjør så de etterspurte handlingene, men sier ikke i fra til prosessoren når den er ferdig. Derfor, etter at en I/O instruksjon er kjørt, må prosessoren periodisk spørre I/O modulen om den er ferdig. Prosessoren er ansvarlig for lesing og skriving til primærminnet.

Avbruddsdreven I/O: Et alternativ til programmert I/O er avbruddsdreven I/O. Prosessoren sier da i fra til modulen at den har fått en I/O instruksjon, og går så å gjør noe annet, mens modulen tar seg av I/O instruksjonene. I/O modulen sier fra til prosessoren ved hjelp av et avbrudd når den er ferdig. Prosessoren overfører da dataene og fortsetter der den slapp.

Direkte Minne Aksess: Når prosessoren vil lese eller skrive data til en modul sender den informasjon til en DMA modul og fortsetter sitt arbeid. Den har da delegert ansvaret for I/O operasjonene til DMA modulen. Modulen overfører så data til og fra primærminnet uten at den må gå gjennom prosessoren. Når den er ferdig sender den et avbrudd til prosessoren.

KAPITTEL 2 - OPERATIVSYSTEMET

Operativsystemets oppgaver og funksjoner

Et operativsystem er et program som kontrollerer utførelsen avapplikasjoner og fungerer som et brukergrensesnitt mellom brukerprogrammer og maskinvaren i en datamaskin. Et operativsystem har tre mål:

1. Gjøre det enklere for brukeren å bruke datamaskinen.
2. Fordele datamaskinens ressurser på en effektiv måte.
3. Mulighet for videreutvikling.

Et operativsystem yter som vanlig følgende service:

Programutvikling: Tilbyr editorer og debuggere for programmerere.

Programkjøring: Tar seg av stegene i utførelsen av et program.

Aksess til I/O: Tilbyr et interface til I/O moduler.

Aksess til filer: Tar seg av filbehandling og håndtering.

System aksess: Kontroll av aksess til systemet.

Feilbehandling: Må ta seg av feil som skjer i program eller maskinvare.

Bokføring: Samle inn statistikk for brukeren.

En liten del av operativsystemet, kalt kernelen, ligger alltid i primærminnet. Denne tar seg av de mest brukte operasjonene og den delen av operativsystemet som er i bruk ved en gitt tid. Resten av primærminnet inneholder brukerprogrammer og data.

Operativsystemets evolusjon

Seriell Prosessering: De første datamaskinene på 1940 og til midten av 1950-tallet hadde ikke operativsystemer. Programmereren måtte forholde seg til maskinvaren direkte.

Enkle Batch Systemer: Den sentrale delen i dette systemet ble kalt en monitor. Med denne typen operativsystemer trengte ikke brukeren lenger å forholde seg til maskinvaren, men i stedet la de inn programmene sine på tape eller hullkort til en datamaskinoperatør, som samlet programmene og gav dem sekvensielt til monitoren.

Multiprogrammerte Batch Systemer: Sekvensiell kjøring av programmer førte til mye I/O venting. I stedet for å kjøre programmene ferdige, en etter en, kunne man kjøre flere samtidig, og veksle mellom disse når et program måtte vente på I/O. Dette førte til at alle programmene som skulle kjøre måtte ligge i minnet, og man trengte en form for minnehåndtering.

Time-Sharing Systemer: Akkurat som multiprogrammerte systemer håndterer flere prosesser kunne man flere brukere dele en datamaskin interaktivt. Dette kalles time-sharing. Her ble responstid et viktig stikkord.

De største nyvinningene

Man regner med at det har vært fem teoretiske steg som er nådd i operativsystemets historie:

1. Prosess
2. Minnehåndtering
3. Informasjonsbeskyttelse og sikkerhet
4. Planlegging og ressurshåndtering
5. System struktur

Utvikling som har ledet til moderne operativsystemer

Mange designelementer og framgangsmåter har blitt prøvd ut både i komersielle og eksprementelle operativsystemer, men det meste kan deles inn slik:

1. Mikrokernell arkitektur
2. Multithreading
3. Symmetrisk multiprosessering
4. Distribuerte operativsystemer
5. Objektorientert design

En *mikrokernell arkitektur* er en arkitektur der kun et fåtall av operativsystemets mest essensielle funksjoner kjører i kernellen. De andre prosessene som operativsystemet styrer kjører i brukermodus, slik som alle andre programmer. *Multithreading* er en teknikk der hver prosess kan deles opp i tråder som kjøres synkront. Dette er nyttig når vi har programmer som utfører flere uavhengige operasjoner. En *symmetrisk multiprosessor* er et eget datasystem med flere homogene prosessorer, delt minne og I/O mulgiheter. Et *distribuert operativsystem* gir illusjonen om at det bare finnes et primærminne og et sekundærminne. *Objektorientert design* gjør prosessen med å utvikle moduler til operativsystemer enklere.

KAPITTEL 3 - PROSESSER

Hva er en prosess?

En prosess kan defineres på mange måter, blant annet som et program under utførelse, et instans av et program som kjører på en datamaskin eller en del av et program som utføres av prosessoren. En prosess består av programkode og et sett med data. En prosess kan bli unikt definert ved følgende:

1. **Identifikator**: En prosess har en unik identifikator som skiller den fra andre prosesser.
2. **Tilstand**: En prosess kan ha flere tilstander, når den kjøres er den i *running state.*
3. **Programteller**: Adressen til neste instruksjon som skal kjøres.
4. **Minnepekere**: Pekere til programkode og data assosiert med prosessen.
5. **Kontekstdata**: Data som ligger i prosessorens registre.
6. **I/O informasjon**: Inkluderer utestående I/O forespørsler, medier osv.
7. **Bokføringsinformasjon**: Statistikk om forbruk av prosessortid osv.

Den nevnte informasjonen inkluderes i prosessens kontrollblokk, som behandles av operativsystemet. Ved et avbrudd, lagers kontrollblokken, slik at den kan hentes fram igjen ved senere kjøring.

Prosessens tilstander

En dispatcher er et program som bytter mellom kjøring av prosesser. I en modell med fem tilstander har vi følgende tilstander:

1. **Running**: Prosessen som blir utført av prosessoren akkurat nå.
2. **Ready**: En prosess som er klar for utførelse.
3. **Blocked**: En prosess som venter på f.eks. en I/O operasjon.
4. **New**: En prosess som enda ikke har blitt flyttet til primærminnet, men har fått en kontrollblokk tilknyttet seg.
5. **Exit**: En prosess som har blitt fjernet fra settet med kjørbare prosesser av operativsystemet, enten fordi den er ferdig med kjøring eller avsluttet.

En prosess kan også suspanderes. Det kan være ulike grunner til dette, blant annet:

1. Prosessen ikke er helt klar for kjøring.
2. Prosessen ble suspandert av en annen prosess eller operativsystemet for å forhindre kjøring.

Prosessbeskrivelse

Når et operativsystem skal behandle prosesser må det ha tilgang til informasjon om prosessene og tilgjengelige ressurser. En generell metode er å behandle tabeller for hver entitet den bahandler. Man har da fire tyer tabeller: en for minne, I/O, fil og prosess. Minnetabellene brukes for å holde styr på primær og sekundærminne. Disse inneholder informasjon om allokasjonen av minne til prosessene, informasjon om beskyttet minne og hva slags tilgang de forskjellige prosessene har. I/O tabellene inneholder informasjon om hvilke I/O medier som er i bruk av hvilke prosesser. Operativsystemet har også tabeller for filer, som inneholder informasjon om filenes innhold, lokasjon

på sekundærminnet, status og attributter. Til slutt må operativsystemet også ha en tabell med oversikt over alle prosessene.

En kontrollblokk er tilknyttet hver prosess, denne inneholder en del attributter som beskriver prosessen. Denne inneholder blandt annet:

1. Prosessidentifikasjon
2. Prosessens tilstandsinformasjon
   - Brukerregistere
   - Kontroll og status registere
   - Stakkpekere
3. Prosessens kontrollinformasjon
   - Planlegging og tilstandsinformasjon
   - Datastruktur
   - Interprosesskommunikasjon
   - Prosessens privilegier
   - Minnehåndtering
   - Eierskap

Prosesskontroll

De fleste prosessorer støtter to eller flere kjøringsmodi. I et priveligert modus kan man endre kotrollregistere, kjøre I/O instruksjoner og minnehåndtering. I et mindre priveligert modus kan man kjøre brukerprogrammer. Vi skiller altså mellom kernel og bruker modus. Grunnen for dette er at operativsystemets kjøring og dets tabeller for prosesser må beskyttes for brukerprogrammene.

Når en ny prosess genereres, skjer følgende steg:

1. Generer et nytt unikt id for prosessen og legg det til i prosesstabellen.
2. Allokker minne til prosessen.
3. Initialiser en ny kontrollblokk til prsessen.

KAPITTEL 4 - TRÅDER, SMP OG MIKROKERNELL

Prosesser og tråder

Når et operativsystem kan behandle flere tråder samtidig, kaller vi det *Multithreading*. I et multithreaded miljø er en ressurs defienert som en del av ressurs allokasjonen og en del av beskyttelsen. Følgende er assossiert med en prosess:

1. En virituell adresse som holder prosess bildet (dette er en samling av program, data, stakk og attributter som er definert i prosessens kontrollblokk).
2. Beskyttet aksess til prosessorer, ondre prosesser, filer og I/O ressurser.

I en prosess kan de være en eller flere tråder, hver med følgende:

1. En trådtilstand (Running, ready osv.)
2. Et lagret trådkontekst
3. En stakk
4. Statisk lagringsplass
5. Et delt lager

Fordeler med å bruke tråder:

1. Det tar mye kortere tid å lage en ny tråd, enn det det tar å lage en ny prosess.
2. Det tar mye kortere tid å avslutte en tråd, enn en prosess.
3. Det tar kortere tid å bytte mellom tråder i samme prosess enn å bytte prosesser.
4. Tråder kommuniserer mer effektivt med hverandre.

Det er fire grunnleggende trådtilstander:

1. **Spawn**: Når en ny prosess blir dannet, blir som regel også en ny tråd dannet. En tråd i en prosess kan lage en ny tråd i samme prosess, gi den en instruksjons peker og argumenter for den nye tråden. Den nye tråden får eget registerkontekst og stakkplass, og blir plassert i Ready køen.
2. **Block**: Når en tråd må vente på en hendelse blir den blokkert. Prosessoren kan nå utføre en annen tråd.
3. **Unblock**: Når en tråd unblockes, flyttes den til Ready køen.
4. **Finish**: Når en tråd har kjørt ferdig blir kontekstregistrene og stakkplassen deallokert.

Det finnes to nivåer der tråder kjører, brukernivå og kernelnivå. I en brukernivåtråd, er all trådbehandling gjort av applikasjonen og kernellen er ikke klar over eksistensen til trådene. I en kernellnivåtråd er alt behandling av tråder gjort av kernellen. Fordelene med brukernivåtråder er:

1. Trådbytting krever ikke kernellnivå privilegier.
2. Planlegging kan være applikasjonsspesifikt.
3. Trådene kan kjøre på et hvert operativsystem.

Symmetrisk multiprosessering

I en symmetrisk multiprosessor kan kernellen kjøres på en hvilken prosessor og alle prosessorene gjør typisk planleggingen selv og henter ut prosesser eller tråder fra et felles sett. Kernellen kan konstrueres som flere prosesser og kan utføres i parallell.

Alle prosessorene har tilgang til et felles delt primærminne og I/O moduler gjennom en delt buss. Prosessorene kan kommunisere med hverandre gjennom primærminnet. I moderne datamaskiner har også prosessorene private cache minner.

Det er mye som må tas hensyn til når vi skal designe et symmetrisk multiprosseserende system:

1. Simultan parallellprossesering av tråder
2. Planlegging av prosesser
3. Synkronisering
4. Minnehåndtering
5. Pålitelighet

Mikrokerneller

En mikrokernell er en liten del av operativsystemets kjerne som tilbyr grunnlaget for modulært design. Den bakomliggende filosofien er at en mikrokernell bare skal inneholde de mest essensielle operativsystem funksjonene, og mindre essensielle oppgaver skal flyttes til brukermodus. Det er flere fordeler med å designe systemet slikt:

1. Uniforme grensesnitt
2. Utvidbarhet
3. Fleksibilitet
4. Flyttbarhet
5. Pålitelighet
6. Support for distribuerte systemer
7. Support for objektorientert design

Mikrokernellen må inkludere de funksjonene som direkte avhenger av maskinvaren og de funksjonene som trengs for støtte til servere og applikasjoner i brukermodus.

KAPITTEL 5 - GJENSIDIG UTELUKKELSE

Parallellitetsprinsipper

En sentral del av operativsystemet er behandlingen av prosesser og tråder:

Multiprogrammering:	Behandling av flere prosesser i en enkjerneprosessor.
Multiprosessering:	Behandling av flere prosesser i en flerkjerneprosessor.
Distribuert prosessering:	Behandling av flere prosesser som kjører på flere distribuerte systemer.

Selv om ekte parallellitet ikke oppnås, og selvom det er en del overhead forbundet med å bytte mellom prosesser, så er det ofte mye mer effektivt å kjøre programmer i parallell. Når programmer kjører i parallell får vi et problem med delte ressurser. Disse må beskyttes slik at bare en prosess kan ha skriverettigheter om gangen. Enhver enhet som skal ha støtte for gjensidig utelukkelse må møte følgende krav:

1. Bare en prosess av gangen kan slippes inn i en kritisk region.
2. En prosess som stopper i en ikke-kritisk region må ikke være i veien for andre prosesser.
3. Det må ikke være mulighet for deadlock eller utsulting i køen inn i kritisk region.
4. Når ingen prosess er i kritisk region kan enhver prosess slippe inn uten forsinkelse.
5. Ingen antagelser gjøres om prosessorhastigheter eller antall prosessorer.
6. En prosess er i kritisk region for en avsluttende tidsperiode.

Gjensidig utelukkelse

For å garantere gjensidig utelukkelse i en uniprosessor er det nok å beskytte en prosess fra avbrudd. Dette kan gjøres på følgende måte:

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

Fordi den kritiske regionen ikke kan bli avbrutt, kan vi garantere gjensidig utelukkelse.

I en multiprosessor deler flere prosessorer et delt minne. Det er da heller ikke noen avbruddsmekanisme som kan garantere gjensidig utelukkelse. På maskinvarenivået vil en aksess til en delt minnecelle ekskludere andre aksesser til de samme minnecellen. Med dette som et grunnlag har prosessordesignere foreslått flere maskininstruksjoner som kan kjøre to atomiske instruksjoner samtidig på en minnecelle. Det finnes flere måter, den ene er en atomisk testset instruksjon:

```
boolean testset (int i)
{
	if(i==0)
	{
		i=1;
		return true;
	}
	else
	{
		return false;
	}
}
```

Å implementere maskinvareinstruksjoner for gjensidig utelukkelse har flere fordeler:

1. Det kan gjøres på flere prosesser på enten en prosessor eller på en multiprosessor med delt minne
2. Algoritmene er enkle og dermed enkle å bevise.
3. Det kan brukes på flere kritiske regioner, hver med sin variabel.

Det er også noen bakdeler med bruk av maskinvareinstruksjoner som gjensidig utelukkelse:

1. Når en prosessor venter, bruker den prosessortid (Busy Waiting).
2. Det er mulighet for utsulting.
3. Det er mulighet for deadlock.

Semaforer

Vi kan se på en semafor som en variabel med en verdi og følgende operasjoner:

1. En semafor kan bli initialisert til en ikke-negativ verdi.
2. Operasjonen *SemWait* teller ned.
3. Operasjonen *SemSignal* teller opp.

Med semaforer kan vi løse problemet med gjensidig utelukkelse:

```
/* program mutalexclusion */
const int n = /* number of processes */
semaphore s = 1;
void P(int i)
{
	while(true)
	{
		semwait(s);
		/* critical section */
		semsignal(s);
		/* remainder */
	}
}
void main()
{
	parbegin (p(1), p(2), … , p(n));
}
```

Normal kjøring kan nå foregå parallellt, mens kjøring av kritiske regioner må skje seriellt. Operasjonene semWait og semSignal må være atomiske.

Monitorer

En monitor er et programmeringsspråk-onjekt som gir ekvivalent funksjonalitet som semaforer, men som er enklere å kontrollere. En monitor er en programvaremodul som består av en eller flere produsenter, en initialiseringssekvens og lokale data. Karakteristikken av en monitor er den følgende:

1. De lokale dataene er aksessbare kun av monitoren.
2. En prosess kan bruke monitoren ved å bruke en av dens prosedyrer.
3. Bare en prosess kan bruke monitoren av gangen, andre prosesser som vil bruke den samtidig, må vente.

En monitor støtter synkronisering ved hjelp av kondisjonsvariable som bare kan brukes i monitoren:

1. **cwait(c):** Suspenderer kjøringen av den prosessen med kondisjon c. Monitoren blir da ledig for andre prosesser.
2. **csignal(c):** Gjenopptar kjøringen av en suspendert prosess på samme kondisjon. Hvis flere kaller denne, blir kun en valgt.

Meldinger

Når to prosesser skal kommunisere sammen må to krav være tilfredsstilt: synkronisering og kommunikasjon. En metode er da meldingssending, som har følgende primitiver:

1. send(destination, message)
2. recieve(source, message)

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

KAPITTEL 6 - VRANGLÅS OG UTSULTING

Vranglås, introduksjon

Et sett av prosesser er i vranglås om hvis settet er blokkert mens det venter på noe. En typisk vranglås sekvens er:

Prossess P	**Prosess Q**
Get A	Get B
Get B	Get A
Realase A	Release B
Release B	Release A

Tre betingelser må være tilstede for at en vranglås skal kunne skje:

1. Gjensidig utelukkelse: Bare en prosess må kunne bruke en ressurs av gangen.
2. Hold og vent: En ressurs kan holde en ressurs og vente på en annen.
3. Ingen preempting: Ingen ressurs kan fjernes med makt fra en som har den.

Umuliggjøre vranglås

Strategien bak vranglåsbeskyttelse er å designe et system slik at en vranglås ikke kan oppstå. Det går ikke an å kvitte seg med gjensidig utelukkelse, da denne trengs ved parallellitetskjøring av prosesser. Hold og vent kan man kvitte seg med, men da må en prosess blokkes til den kan få alle ressursene samtidig. Dette er ikke veldig effektivt. Å kunne preempte en prosess kan løse problemet. Hvis en prosess som holder en ressurs blir nektet en annen, kan den bli tvungen til å gi fra seg den første, helt til den kan få den neste.

Unngå vranglås

Strategien bak å unngå vranglås, tillater alle de tre betingelsene, men gjør enkelte valg, slik at en vranglås aldri oppstår. Det finnes to strategier;

1. Å ikke starte en prosess, når muligheten for vranglås er der.
2. Å ikke gi en prosess en ressurs om det kan lede til vranglås.

Oppdage og rette opp en vranglås

Å oppdage og rette opp en vranglås kan gjøres ofte eller sjeldent. Om det gjøres ofte blir vranglåsen tidlig oppdaget og tidlig rettet opp i, noe som gjør at vi sparer en del venting. Om vi gjør det sjeldent sparer vi en del prosessortid ved at algoritmen for å finne en vranglås ikke må kjøres så ofte. Vi kan finne en vranglås ved å lete etter sykluser i en ventegraf.

KAPITTEL 7 - MINNEHÅNDTERING

Minnehåndtering

Når vi skal se på de forskjellige mekanismene som er assosiert med minnehåndtering, er det viktig å tenke på hva minnehåndteringen skal tilfredsstille:

1. Reallokering
2. Beskyttelse
3. Deling
4. Logisk organisering
5. Fysisk organisering

Partisjonering av minnet

Hovedoppgaven til minnehåndteringen er å bringe en prosess inn i primærminnet slik at den kan kjøres av prosessoren. Vi kan gå ut i fra at operativsystemet bruker en fast del av primærminnet. Resten av minnet går til brukerprosesser. Dette ledige minne kan deles inn i like store eller ha ulik størrelse på partisjonene. Hvis disse størrelsene er faste, kaller vi dette fast partisjonering. Med like store partisjoner er det enkelt å finne plass til en ny prosess, det er bare å finne en ledig partisjon. Med ulike partisjonsstørrelser kan vi for eksempel bruke den minste, men passende partisjonen.

For å overkomme noen av vanskelighetene med fast partisjonering, kan vi bruke dynamisk partisjonering. Her er partisjonene av variabel lengde og antall. Når en prosess skal inn i primærminnet blir det allokert akkurat så mye plass som en prosess trenger. Denne situasjonen starter bra, men vi ender opp med veldig mange små hull i minnet etter hvert som plass reallokeres. Dette kalles ekstern fragmentering. Dette kan unngås ved kompaktering av minnet. Når en ny prosess skal inn i primærminnet brukes tre metoder for å finne en plass; best-fit, first-fit og next-fit.

Fordi fast partisjonering kan bruke minnet lite effektivt og dynamisk minne er veldig komplekst å vedlikeholde er buddy-systemet et interessant kompromiss. Her starter vi først med en hel minneblokk, og deler denne opp etter hvert som prosesser blir allokert minne. Vi deler først i to, om dette er for mye, deler vi i to igjen, osv og vica versa ved deallokering.

En logisk adresse er en referanse til en minnecelle uavhengig av den gitte oppgaven av dataet i minnet. En oversettelse til en fysisk adresse må gjøres før en eventuell minneaksess. En relativ adresse er et eksempel på en logisk adresse som er relativt til et gitt punkt. En fysisk adresse er et faktisk sted i primærminnet.

Sidedeling

Ved sidedeling deles minnet opp i like store, men små biter, og hver prosess er delt inn i like små biter. En prosess kan da oppta et endelig nummer med minnebiter, altså en sideramme. Operativsystemet har en sidetabell for hver prosess som viser lokasjonen til hver side.

Segmentering

Ethvert program kan deles inn til ett eller flere segmenter av ulik lengde. Som med sidedeling består en logisk adresse av to deler, i dette tilfelle, segmentnummer og et offset. Sidedeling er ikke synlig for programmereren, mens segmentering er synlig, slik at data og programmer kan organiseres.

KAPITTEL 8 - VIRITUELT MINNE

Maskinvare og kontrollstrukturer

Hvis alle referanser til minnet til en prosess er logiske adresser som er dynamisk oversatt til fysiske adresser under kjøretid og alle prosesser kan brytes ned til sider eller segmenter, trenger ikke alle sider eller segmenter i en prosess være i primærminnet samtidig. Den delen av en prosess som faktisk er i primærminnet kalles for et *resident set*. Fordi en prosess bare kan kjøres i primærminnet kalles dette ofte for ekte minne, mens den delen av en prosess som ligger for eksempel på disk, kalles virituelt minne.

Operativsystemets programvare

Designet av minnehåndteringen i et operativsystem avhenger av tre fundamentale valg:

1. Om virtuelt minne skal brukes eller ikke.
2. Om segmentering, sidedeling eller begge skal brukes.
3. Algoritmene som skal brukes.

Det finnes en del algoritmer som kan brukes til replassering av prosesser i virtuelt minne:

1. Optimal
2. Sist brukt (LRU)
3. Først inn, Først ut (FIFO)
4. Klokke

Den optimale algoritmen bytter ut den prosessen som det er lengst til skal brukes. Dette fungerer ikke i praksis, da operativsystemet ikke kan ha denne kunnskapen. Sist brukt, velger den prosessen som har ligget lengst og ikke vært brukt for utbytting, mens FIFO ser på minnet som et sirkulær buffer og bytter ut i round-robin stil. Klokkealgoritmen, ser på minnet som en klokke med en pil, og prosessene har et Use-bit, som kan være 1 eller 0. Om prosessen har vært i bruk på denne runden, settes bit'et til 1, ellers 0. Den bytter så ut første prosess som har 0.

KAPITTEL 9 - UNIPROSESSOR-PLANLEGGING

Typer av prosessorplanlegging

Målet med prosessorplanlegging er å velge hvilke prosesser som skal kjøres av prosessoren eller prosessorene til enhver tid, slik at responstid, flyt og effektiviteten maksimeres. I mange systemer brytes dette ned til lang, medium og korttids planlegging. Langtids-planlegging brukes når en ny prosess dannes, mens mediumtids-planlegging er en del av minnehåndteringen (hvilke prosesser skal være i primærminnet og klare for kjøring). Korttids-planleggingen består av å velge ut en prosess for kjøring i prosessoren.

Planleggingsalgoritmer

Hovedoppgaven til korttids-planleggeren er å allkoere prosessortid til prosessene, slik at effektiviteten i systemet maksimeres. Mange systemer bruker prioriteter, slik at hver prosess får en prioritet, og planleggeren plukker ut en prosess med høy prioritet. Dette kan føre til utsulting av prosesser med lav prioritet. Vi kan også bruke en utvelgelsesfunksjon, som velger prosess etter følgende kriterier:

1. (W) Tid brukt i systemet, venting og kjøring.
2. (e) Tid brukt på kjøring.
3. (s) Total kjøretid etterspurt av prosessen.

Vi må også skille mellom to kategorier; nonpreemptive og preemptive.

Vi har da følgende algoritmer:

1. Først inn, først ut (FCFS).
2. Kontinuerlig rundgang (Round Robin).
3. Korteste totaltid først (SPN).
4. Korteste gjenværende tid først (SRT).
5. Høyeste responstid først (HRRT).
6. Tilbakekopling (FB).

Først inn, Først ut

Dette er en enkel FIFO k, der alle prosesser har lik prioritet og har ingen avbrytingsmekanisme. FCFS plukker ut neste prosess etter hvem som har høys W; maks(W). Lange prosesser har en tendens til å ta overhånd i et slikt system, da det er skjeldent de korte får komme til.

Round-Robin

Dette er en enkel algoritme for å løse problemet med at lange prosesser blir favorisert. Alle prosesser får en gitt tidskvante og alle prosessene kjøres på rundgang. Vi bruker her altså avbryting og ingen prioritering. Prosessen som blir valgt som neste er den som har ventet lengst denne runden; Maks(W –per runde). I denne algoritmen byttes det en del mellom prosessene, og dette tar prosessortid. Korte prosesser vil gå kjappt gjennom systemet, men lange vil måtte kjøres lenge. Vi kan forbedre dette ved å stille på størrelsen på tidskvantene.

Korteste totaltid først

Dette er en algoritme, der de korte prosessene blir favorisert. Den bruker ikke avbrytning, så prosessen som er valgt, vil kjøre ferdig. Denne algoritmen er mer kompleks, da den må vite, eller ha et estimat, over ønsket kjøretid for alle prosesser. Her har vi en risiko for utsulting av lange prosesser.

Korteste gjenværende tid først

Dette er en avbrytbar versjon av SPN. Planleggeren vil velge den prosessen med den korteste gjenværende tiden. En prosess som kjører kan bli avbrutt når det kommer en ny prosess og korte prosesser blir fortsatt favorisert.

Høyeste responstid først

Dette er en algoritme uten avbrytbarhet, og som prioriterer både lange og korte prosesser. Den lager et estimat for prioriteringen til en prosess:

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

R – prioritering
w – total ventetid
s – forventet kjøretid

Tilbakekopling

Formålet bak denne algoritmen er å vurdere hvor lang tid en prosess har brukt i systemet. Vi trenger derfor ikke noe estimat på hvor lang tid den tror den skal bruke, vi ser heller på kjørehistorien. Algoritmen bruker tidskvanter og er avbrytbar. Når en prosess får kjøre, plasseres den i kø R0, når den avbrytes plasseres den i kø R1 med lavere prioritet. En kort prosess vil bli kjørt fort, mens en lang prosess gradvis vil kjøres og havne nedover i køsystemet. På hver kø er en FCFS planlegger brukt. En svakhet er at lange prosesser kan utsultes.

KAPITTEL 10 - MULTIPROSESSOR-PLANLEGGING

Multiprosessorplanlegging

Prosesshåndtering på multiprossessorer må ta hensyn til:

1. Fordeling av prosesser til prossessorer.
2. Multiprogrammering på de individuelle prossessorene.
3. Valg av prosess for kjøring.

Ved en statisk utvelgelse av prosesser, slik at en prosess velges en gang og kjøres ferdig på en CPU, blir det mindre overhead i planleggeren, siden utvelgelsen er gjort en gang for alle. En bakside med dette er at en prosessor kan ha en tom kø, mens en annen har full kø. Vi må også vurdere om en prosessor skal være multiprogrammert.

Med en multiprosessor kan vi kjøre tråder i parallell. Det finnes flere måter å planlegge dette på:

1. Load sharing: Prosessene har ikke en dedikert prosessor, men en global kø av tråder som er klare for kjøring holdes og en prosess som er klar for kjøring velger ut en tråd og kjører denne. Her blir trådene effektivt fordelt og alle prosessorer har like mye arbeid å gjøre. Køen må ha gjensidig utelukkelse, og det kan være en flaskehals om det er mange prosesser.
2. Gang Scheduling: Et sett tråder kjører på et sett prosessorer på samme tid, et en til en forhold. Hvis relaterte prosesser kjører samtidig vil nødvendigheten av prosesskifte minke og systemet blir mer effektivt.
3. Dedicatet Processor Assignment: Programmene fåår tildelt et antall prosessorer, like mange som de har tråder for kjøring.
4. Dynamic Scheduling: Trådene kan byttes ut under kjøring av en prosessor.

Sanntidsplanlegging

En hard sanntidsoppgave er en oppgave som må være ferdig kjørt på en deadline. I et sanntidssytem er det flere faktorer som er viktig:

1. Forutsigbarhet
2. Responsitivitet
3. Brukerkontroll
4. Pålitelighet
5. Feiltilpassning

Det finnes flere måter å planlegge et slikt system på. To vanlige måter er å gi enten en spesiell oppgave høy prioritet, eller prioritere den oppggaven som har den korteste avstanden til deadlinen.

KAPITTEL 11 - I/O OG DISKHÅNDTERING

I/O medier

Vi kan dele I/O mediene inn i tre grupper:

1. Interaksjonsmedier
2. Maskinmedier
3. Kommuniksjonsmedier

Organisasjon av I/O funksjonen

Tre typer I/O kommunikasjon:

1. Programmert I/O
2. Avbruddsdreven I/O
3. DMA

En DMA modul virker på følgende måte. Når prosessoren ønsker å lese eller skrive en blokk med data sender den en kommando til DMA modulen, ved å sende følgende informasjon:

1. Om den skal lese eller skrive
2. Adressen til I/O modulen
3. Startlokasjonen i minnet den vil lese fra eller til
4. Antall ord som skal leses eller skrives

Prosessoren fortsetter så med sitt arbeid, alt annet er dedikert til DMA modulen, som kommuniserer direkte med minnet, og sender over eller leser en blokk av gangen, ord for ord, uten hjelp fra prosessoren.

Designobjektiver for operativsystemet

Effektivitet er et viktig aspekt, da I/O operasjoner ofte kan være en flaskehals i et system. Et annet aspekt er generalitet, slik at alle I/O moduler kan behandles likt. I/O kommunikasjonen kan framstilles som en lagmodell, bestående av blant annet:

1. Logisk I/O
2. Modul I/O
3. Planlegging og kontroll
4. Mappehåndtering
5. Filsystem
6. Fysisk organisering

I/O Buffering

Å flytte data direkte fra I/O modulen og rett til primærminnet kan være dumt da prosessen som skal bruke dataene uansett må vente på I/O overføringen. En måte å unngå dette er å tilføre et mellom-lagring av data. Vi skiller mellom to typer overføringer:

1. **Blokkorientert**: Lagrer informasjon i blokker og overfører en av gangen
2. **Strømorientert**: Sender dataene som en strøm.

Det finnes flere varianter av buffering:

1. **Single buffring**: I/O modulen overfører noe data og prosessen jobber med disse, mens I/O modulen overfører flere data.
2. **Dobbel buffring**: En forbedring over singlebuffering der det er to buffere, en som prosessen leser eller skriver til, og en som operativsystemet leser og skriver til.
3. **Sirkulær buffring**: Enda flere buffere. (Bounded-buffer modellen).

Diskplanlegging

Det finnes flere måter å prioritere disk I/O på, her er et par:

1. **FIFO**: Henter ut data fra disken og leser sekvensielt. Dette er en rettferdig algoritme.
2. **Prioritert**: Behandler lesingen for proiriterte jobber først.
3. **LIFO**: Baehandler den sist innkommne lesingen først. Denne er ofte nær den forrige, nåe som gir bedre utnyttelse.
4. **Korteste søk først**: Prioriterer det søket som er nærmest disk armen.
5. **Scan**: Armen beveger seg i en rettning til den når siste sektor og fortsetter i andre rettning.
6. **C-Scan**: Skanner i en retning, når armen er i siste sektor, flyttes den tilbake og starter på nytt.

KAPITTEL 12 - FILHÅNDTERING

Introduksjon

Filsystemet er en abstraksjon for typisk sekundærlagring. Filsystemet tillater brukeren å lage datasamlinger, kalt filer med følgende karakteristikk:

1. Langsiktig eksistens
2. Delbare mellom prosesser
3. Struktur

Operativsystemets filarkitektur er typisk en lagmodell, med moduldrivere i nederste lag. Disse driverene er ansvarlige for kommunikasjon med kontrollere og lagringsmedier. Det neste nivået er det grunnleggende filsystemet, som tar seg av kommunikasjon via utveksling av datablokker. Det grunnleggende I/O laget tar seg av initialisering av filer, kontrollstrukturer og filstatus. Det logiske I/O laget gir brukerne mulighet til å aksessere poster.

Filorganisasjon

Når vi skal velge en filorganisasjon, må vi se på følgende kriterier:

1. Kort aksesstid
2. Enkle oppdateringsmuligheter
3. Økonomisk lagring
4. Enkelt vedlikehold
5. Trygg lagring

Vi har flere organiseringer:

1. Haugstruktur
2. Sekvensiell fil
3. Indeksert sekvensiell fil
4. Indeksert fil
5. Direkte eller hashet fil

Den minst kompliserte filstrukturen er en haugfil. Data blir samlet i den rekkefølgen de kommer. Siden det ikke er noe struktur på filen, blir data hentet frem ved søk. Den mest vanlige filstrukturen er en sekvensiell fil. Her er det et bestemt format med poster, og et bestemt felt brukes som nøkkel. Aksess krever et sekvensielt søk på nøkkelfeltet. En indeksert sekvensiell fil kan være kjappere enn en enkel sekvensiell fil. Her er filen indeksert på nøkler.

Filkataloger

En katalog er i seg selv en egen fil. Katalogen inneholder informasjon om filene, inkludert attributter, lokasjon og eierskap. Vi bruker gjerne hierarkiske strukturer for å representere kataloger.

Fildeling

Et filsystem bør yte en fleksibel måte å dele filer mellom brukere. Systemet bør ha en rekke muligheter for kontrollering av hvem som har tilgang til hvilke filer.

Plassallokering

Poster må organiseres som blokker når de skal legges på sekundærlageret. Det er i hovedsak tre metoder som brukes:

1. Fast lengde
2. Variabel-lengde med spenn
3. Variabel-lengde uten spenn

Sekundærlaget

Når vi skal allokere plass til en fil på sekundærlageret må vi ta hensyn til følgende:

1. Når skal plassen allokeres?
2. Hvor mye plass skal allokeres?

Vi har flere typer diskallokering:

1. Sammenhengende
2. Kjedet –fast
3. Kjedet – variabel
4. Indeksert –fast
5. Indeksert –variabel

KAPITTEL 14 - DISTRIBUERT PROSESSERING

Client/Server modellen

En klient er som regel en enkel PC med grafisk brukergrensesnitt, som brukes av en bruker. En server yter en tjeneste til klientene. I tillegg til klienter og servere kreves også et nettverk. Det er dette nettverket som lar klientene og serverne kommunisere. Det finnes flere grunner til å bruke distribuerte systemer til for eksempel databasedrift:

1. Det er en stor jobb å sortere og søke gjennom en database. Dette krever mye ressurser og det er som regel for dyrt å implementere dette i alle klientene.
2. Det ville blitt for mye trafikk på nettet om alt bare skulle lastes fra serveren til klienten, disse må derfor også kunne søke.

Det finnes flere metoder å ha klient-tjener kommunikasjon:

1. Hot-based: Brukes sjeldent, all prosessering er gjort på en sentral tjener. Klientene er som regel bare terminal emulatorer.
2. Server-based: Klienten er ansvarlig for brukergrensesnittet, mens all prosessering er gjort på tjeneren.
3. Client-based: Nesten all prosessering er gjort hos klienten, med unntak av databaseprosessering som er gjort på tjeneren.
4. Cooperative: Applikasjonsprosessering er optimert, og man tar fordelen av å bruke både klient og tjener til distribuering av data og prosessering.

For å minke belastningen på linjen holder klientene midlertidige kopier av data liggende lokalt. Dette kalles cashede filer. Når de cashede filene er eksakte kopier av filene på serveren, sier vi at de er konsistente. Problemet med å holde lokale kopier konsistente med tjener-kopiene kalles cachekonsistens-problemet.

Både servere og klienter har et lag i sin modell som kalles middelware. Denne modellen sørger for at alle klientapplikasjoner kan støttes av servere, og man slipper å bekymre seg for versjoner og utgaver av operativsystemene. Vi får altså uniform aksess.

Remote Procedure Call, RPC

Meningen med denne teknikken er at forskjellige maskiner skal kunne kommunisere med enkle prosedyrekall, med samme semantikk som om de skulle vært på samme maskin. Dette har følgende fordeler:

1. Allment kjent og brukt abstraksjon
2. Dokumentert grensesnitt med kjente operasjoner
3. Bred støtte for et standardisert grensesnitt

Klustere

Klustering er et alternativ til symmetrisk multiprosessering, som gir bedre ytelse og tilgang, noe som gjøre det spesielt attraktivt for servere. Det er en del fordeler ved klustering:

1. Absolutt skalerbarhet: Det er mulig å lage store klustere som har mye høyere ytelse en selv den største enkeltmaskin.
2. Inkremental skalerbarhet: Det er mulig å legge til nye deler av systemet i små inkrementer.
3. Høy tilgang: Fordi hver node er en enkelt maskin som kan kjøres alene, betyr det ikke noe for systemet om en maskin går ned.
4. Overlegen pris på ressurser: Det er som regel rimeligere å bygge klustere en å kjøre store og kraftige enkeltpcer.

Det finnes flere måter å konfigurere en server:

1. Separate servere: Hver datamaskin er en separat server, med egne disker, og uten delt minne. Dette gir høy ytelse og tilgang, men det kan bli problemer om en maskin feiler.
2. Ingen deling: Alle servere deler felles disker, men bruker hver sin partisjon.
3. Delte disker: Alle servere deler felles disker, og alle har et felles område. Her må låsing implementeres, slik at kun en maskin kan hente ut samme data av gangen.