DTK notater eksamen høst 2018

# Cache

**Mapping-funksjon**

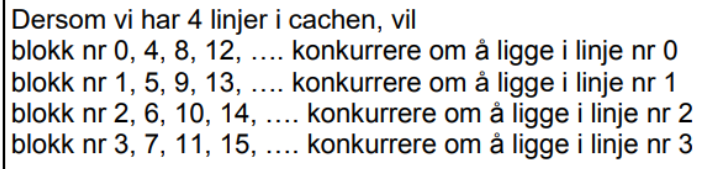
En mappingfunksjon bestemmer hvilken linje i cachen blokka skal ligge. Det er viktig å merke seg at CPU har kanskje bare noen nanosekunder på å finne ut om dataen er i cachen eller ikke, derfor kan dette valget kan ha stor betydning for effektiviteten til cachen. Det finnes tre ulike mapping-funksjoner:

Full-assosiativ

En blokk kan legges i hvilken som helst linje. Derfor må CPU sjekke hver eneste tag i cachen når den skal hente data. Det er tidskrevende å sjekke hver linje etter tur, selv med den hurtigste elektronikken som finnes. Denne mapping-funksjonen er komplisert å implementere, derfor brukes den ikke i praksis.

Ikke-assosiativ (direkte mapping)

En blokk har en fast plass i cachen. Hver blokk «tilhører» en bestemt linje, men det er mange slike blokker som «tilhører» hver linje. NB! Det kan kun ligge en blokk på hver linje. Når CPU skal hente data vet den hvor blokken kan ligge. F.eks. hvis blokk nr.14 finnes, så vil den ligge i linje 2. Da trenger ikke CPU å sjekke alle linjene, kun linjen som blokken kan ligge i.



*Legg merke til at i virkeligheten så har en cachen mye mer enn fire linjer, kun for illustrasjon.*

Trashing

Ulempen med ikke-assosiativ cache, er at to ulike ord som brukes kan havna i samme linje. Hvis to data-verdier har samme tilhørlighet, altså skal ligge på samme linje, vil de tvinge hverandre ut av cache hele tiden. Da gitt at begge data-verdiene brukes i kommende instruksjoner.

Sett-assosiativ

For å unngå trashing, kan man organisere linjene i sett. Blokker sloss om samme linje får flere linjer å dele på, derfor er det plass til flere av dem. Man kan se på sett-assosiativ som løsningen på trashing, til en viss grad. Det f.eks. 2 eller 4 linjer som slås sammen til et sett. En cache som kalles 2-veis, vil kunne ha to konkurrerende blokker inne samtidig. I praksis brukes sett-assosiativ nesten alltid, siden den er nok enkel, men også forhindrer trashing og treghet i cache.

**Minneadministrasjon**

Baseregister

Prosessdeskriptoren som angir status til hver prosess inneholder et eller flere baseregister. Verdien på baseregisteret vil angi startpunktet i det fysiske minnet hvor prosessen ligger. Altså hvis baseregisteret er 500, vil 500 være den første adressen til denne prosessen i det fysiske minnet. Den virtuelle adressen som prosessoren bruker derimot, vil være 0. Alle de virtuelle adressene kan dermed enkelt bli konvertert til fysiske adresser ved å plusse på baseregisteret.

Dette gjør at man enkelt flytte på programmer i det fysiske minnet uten å måtte endre på annet en baseregistrene til programmene for at de skal fungere som tidligere. Systemer i dag har flere enn ett baseregister per prosess, siden minnet deles opp i sider, derfor har man sidetabeller. Ved å bruke sidetabellene kan man holde styr på hvor de ulike blokkene til prosessen ligger. Hvis man vet størrelsen på blokkene, som er samme for alle, kan man regne ut adresserommet til blokkene.

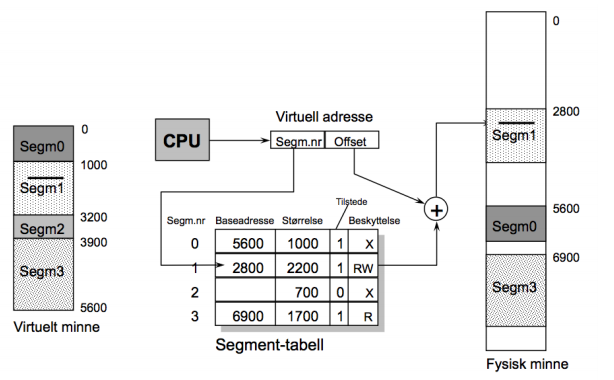
Sidedelt system

**Ulemper med sidedelt system**: Gjør det bl.a. vanskelig å dele programdeler og dataområder med andre prosesser, det er mer problematisk å sette aksessrettigheter på f.eks. dataområder fordi side-grensene ikke stemmer overens med det dataområdet vi ønsker å dele på.

**Fordelen med sidedelt system**: Kun aktive deler av programmet trenger å være minnet under kjøring. Dette fører til bedre utnyttelse av minnet i forhold til andre metoder, derfor ønsker man å beholde fordelene med sidedelt system og samtidig fikse ulempene med det.

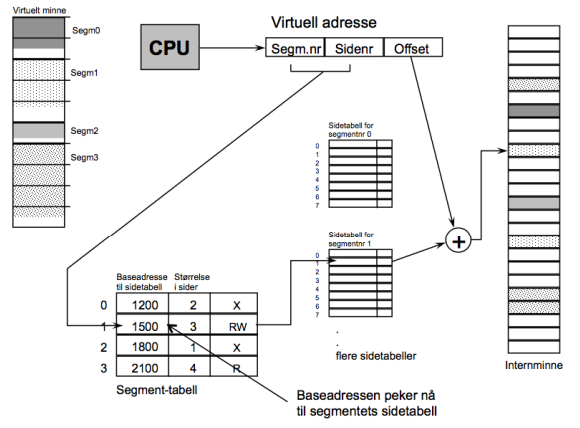
Segmentdelt system

Segmenter kan i forhold til sider ha variabel størrelse. I den aller enkleste formen snakker man om kodesegment, datasegment og stakksegment, men som oftest finnes det mange flere segmenter. Ulempen med dette er at etter hvert som segmenter med ulik størrelse blir lagt inn i minnet, vil det oppstå hull, man sier at minnet er fragmentert.



Kombinert segmentdelt sidedelt system

Man trekke fordelene fra både sidedelt- og segmentdelt system i et kombinert system. Dette systemet består av segmenter som er delt inn i sider. Da vil ikke hele segmenter ligge i minnet, kun deler av segmentet, altså flere sider innad i segmentet, vil ligge i minnet. For å oppnå dette må man ha en egen sidetabell til hvert segment. Segmenttabellen vil derfor referere til flere sidetabeller.



**Oppbygging og virkemåte til en datamaskin**

En datamaskin består blant annet av disse viktigste hovedkomponentene: CPU (ALU, kontrollenhet, Cache(r), registre, MMU og mer), minnebuss, primærminne, IO (lagring, tastatur, utvidelsekort).

Alle kretser består av transistorer som man kan lage logiske operatorer med; AND, OR og NOT. Hvis man igjen går opp et abstraksjonsnivå vil man utvikle ulike komponenter med disse operatorene; lagring (registre, RAM), ALU (den som kalkulerer).

**Instruksjonstyper**

Det finnes tre typer instruksjoner en CPU kan utføre.

1. **Dataprosessering**: utføring av aritmetiske eller logiske operasjon.
2. **Dataoverføring**: Overføring av data mellom CPU-register og minnelokasjon, begge veier.
3. **Programkontroll**: Endre rekkefølgen på instruksjonene som kjøres.

**Lagringstyper**

**ROM** (Read Only Memory): informasjon lagres hvis strømmen forsvinner. Brukes i BIOS, som inneholder oppstarts rutiner. Kan egentlig ikke endre på innholdet i ROM, men utvikling har ført til den nyeste ROM-teknologien, Flash-memory. Dette brukes i minnepenner og SSD-disker.

**RAM** (Random Access Memory): informasjonen blir ikke lagret hvis strømmen forsvinner. Navnet sier at man kan lese og skrive hvor som helst i minnet ulikt ROM.

Finnes to hovedtyper RAM:

1. **DRAM**(Dynamic RAM, 35-70ns aksesstid): kondensatorer brukes, derfor «husker» den informasjonen en stund før den må oppfriskes. Brukes i RAM.
2. **SRAM**(Static RAM, aksesstid 1-10ns): Ren transistorbasert lagring, husker så lenge det er spenning. Brukes til cache. Er mye dyrere og mindre i størrelse enn DRAM.

**Avbrudd**

Avbruddsmekanismen finnes slik at man kan gi varsler til CPU. Dette er veldig viktig i forbindelse med IO-utstyr. En harddisk må kunne sende signal tilbake til CPU når den har funnet frem rett data, siden CPU vil holde på med andre instruksjoner i mellomtiden.

Uten avbrudd: Polling

Hvis man ikke har avbrudd kalles det polling, eller stadig spørring. CPU må hele tiden spørre om den er ferdig, siden IO ikke kan sende noen avbruddssignaler tilbake til CPU. Ulempen med dette er at tiden det tar å utføre IO kunne ha blitt brukt til å utføre andre instruksjoner i CPU.

Med avbrudd: avbruddsdrevet IO

Når en prosess skal utføre IO operasjoner vil CPU-en gå over til en annen prosess i mellomtiden. Når IO er ferdig vil den sende et avbruddssignal til CPU slik at den fortsetter den den slapp med den originale prosessen.

*Avbruddsrutine ISR (Interrupt Service Routine)*: rutiner som kjøres ved avbrudd. Denne ligger i minnet

**Virkemåte til avbruddsmekanismen**

På kontrollbussen til minnebussen finnes det en egen linje for avbrudd. Over denne bussen kan IO sende avbruddsignal til CPU som vil da kjøre avbruddsrutine ISR. CPU vil derimot ikke kjøre avbruddsrutinen med en gang den får avbruddet, den vil gjøre seg ferdig med instruksjonen den holder på med i det øyeblikket.

Instruksjonssyklusen får derfor en delsyklus til; *avbruddssyklus*. Den består av følgende handlinger:

* *Lagre innholdet av PC-registeret*: lagre adressen til instruksjon i kjørende program.
* *Sett PC til startadressen for avbruddrutinen:* for å kjøre avbruddsrutinen.

**Avbruddstyper**

* **IO** **utstyr**: slik som ovenfor
* **Klokkeavbrudd (timer interrupt):** Avbrudd med svært høy prioritet, skjer ofte mellom 15 og 50 ganger i sekundet. OS benytter gjerne anledningen til å utføre administrasjon ved -avbruddet.
* **Utstyrsfeil**: CPU og IO har utstyr for å detektere feil som f.eks. bortfall av spenning.
* **Programfeil (exceptions?):** f.eks. divisjon med 0, ulovlige instruksjoner, ulovlig operand, forsøk på å lese eller skrive til/fra beskyttet del av minnet.
* **Systemkall (programmert avbrudd):** Når programmer skal utføre tjenester den ikke har privilegert tilgang til, vil den spørre OS å utføre det for seg. Det vil da sendes et avbrudd på lik måte som før, bare at CPU sender det selv. Dette er fordi vanlige funksjoner vil programmet kunne endre på og kjøre på egne vilkår. Når systemkall avbruddet blir tolket i CPU, vil CPU-en settes i privilegert modus, slik at den har tilgang og rettigheter til alt. Det er vanlig at IO instruksjoner blir gjort med systemkall.

**Direkte minneaksess DMA**

IO instruksjoner går vanligvis alt igjennom CPU-en, det er den som utfører IO. Derimot med DMA så kan IO moduler utføre noe arbeid selv hvis de har fått nok informasjon av CPU. Kommunikasjonen foregår da mellom IO modul og primærminnet, altså IO har fått direkte minneaksess, som navnet tilsier.

1. I/o-enheten sender et avbruddssignal til cpu-en

2. Cpu gjør ferdig instruksjonen den holder på med akkurat nå, og svarer deretter på avbruddssignalet

3. Cpu sender et signal tilbake til i/o-enheten for å bekrefte at avbruddet er mottatt.

4. Cpu gjør forberedelser for å kjøre avbruddsprogrammet til dette avbruddet. Først må det lagre unna statusinformasjon om kjørende program, bl.a innhold i cpu-registre, adresse til neste instruksjon som skal kjøres, etc.

5. Cpu laster inn en verdi i IP-registeret som peker på første instruksjon i avbruddsprogrammet. Det betyr at nå hentes instruksjonene fra avbruddsprogrammet sine minneadresser.

6. Avbruddsprogrammet kjører på cpu-en. Det sjekker bl.a status til i/o-enheten, finner ut hvorfor avbruddet kom, og sørger deretter for at nødvendige operasjoner for å betjene avbruddet blir utført.

7. Når avbruddsprogrammet er ferdig vil cpu hente inn den lagrede informasjonen om det avbrutte programmet, og fortsette kjøringen av dette på samme sted som det ble avbrutt. Dette er mulig fordi både registerinnhold og innholdet i instruksjonspekeren (IP) ble gjemt unna.