
NOTATER UKE 19 –

Internminne

SLIDE: INTERNMINNE OG CACHE

Internminnet, også kjent som hovedminne eller ram (Random Access Memory), er en type datalagring som brukes av datamaskiner til å midlertidig lagre data og programvare som kjører på en datamaskin. Hovedminne kan aksesseres raskt og direkte av prosessoren, og dataene blir slettet når datamaskinen slås av.

Navnet RAM kommer av at det tar like lang tid å hente en byte uavhengig av hvor det ligger, om det er 1 000 eller 100 000.

Cache er en type minne som brukes for å redusere tiden det tar å aksessere data fra langsommere lagringsenheter som har disker eller SSD-er. Cache lagrer kopier av data som ofte brukes i et mindre og raskere minneområde som ligger mellom prosessoren og den langsommere lagringsenheten. Dette kan redusere tiden det tar å aksessere data og forbedre datamaskinens ytelse.

Vi skal se hva som skjer med ram når store programmer bruker mye ram, med store array og sånt. Og ikke minst hvordan organiseringen av RAM skjer i Windows. Igjen skjer det i nært samarbeid med hardware. Det er mange hardware spesifiserte instruksjoner og konstruksjoner, ikke minst MMU, som er lagd av hardware slik at det å bruke RAM skal gå fort. Veldig mye av dette skyldes i utgangspunktet at ram er veldig tregt. Cache er mye raskere enn SRAM er mye raskere enn DRAM.

Som vi har sett på tidligere er CPU-registere og cache laget av SRAM (Static RAM). SRAM består av 6 transistorer som er meget hurtig og statisk (trenger ikke å oppfriskes).

RAM og Internminnet er laget av DRAM, altså dynamic ram, som består av en transistor og en kondensator. Den må oppfriskes 10 ganger per sekund, så da må vi ha en liten loop som oppfrisker alle verdiene i ram hele tiden. Når vi skrur av datamaskiner slettes jo alt av lagring, så hvis vi vil lagre noe så må det lagres på harddisken, fordi der ligger det permanent. Hvis

den har ladning er det 1, hvis ikke er det 0. tidligere var det ikke vanlig å synkronisere DRAM, men for omtrent 15-20 år siden (se bildet under):

SDRAM:

- Synchronous DRAM (SDRAM), er DRAM hvor lesing og skriving er synkronisert med en ekstern klokke.

Slik som det alltid har vært i CPU-en, der har vi en klokke, men det er også da en klokke ut mot databussen, slik at man synkroniserer lesing og skriving. Og da finnes det forskjellige versjoner av RAM, hvor den aller nyeste er DDR5 (double data rate), hvor alt dette er SDRAM. Det som gjør det bedre for hver gang er at det er høyere klokkefrekvens på databussen sånn at de raskere kan overføre data.

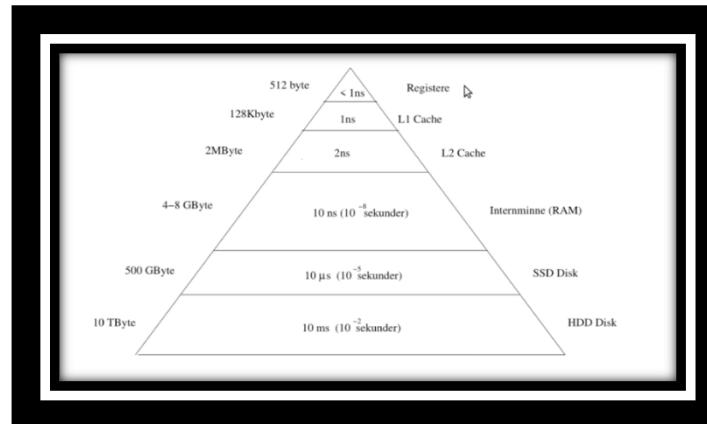
I praksis vil det ta forskjellig tid å hente en gitt byte. For eksempel en variabel ‘svar’ som man har i et C-program, hvis man skal gå ut i ram og hente den. Så vil det være forskjellig hvor lang tid det tar, og det er først og fremst på grunn av cache. Hvis du nylig har hentet den verdien vil den ligge i cache, og cache er hurtig minne, da går det raskere å hente. Sånn sett er det ikke random access.

Et annet tilfelle vi har hvor det er forskjell på tiden er med Newman noder. Hvis du har servere som har mange CPU-er, altså sånne ti-talls-CPU-er, slik som noen av de vi har sett på med 48 CPU-er. Eller sånn serveren docker containerne våre kjører i, Linux VM-ene, som vi kaller de. Den har 60 CPU-er eller noe sånt. Den har et system av Newman noder, og det betyr at det er slags kløstere av CPU-er, kanskje 6 eller 8 av gangen som tilhører spesielle noder, og disse nodene har RAM som ligger nærmere de nodene. Fysisk raskere slik at de har raskere tilgang til minnebuss. Så en hver CPU kan hente bytes fra RAM, men i noen tilfeller tar det lengre tid fordi de ikke sitter på den Newman noden. Så dermed vil det ikke alltid være random access.

SRAM, standard random access, det tar like lang tid å hente hver byte, og det er veldig forskjellig fra tradisjonelle spinnende harddisker hvor det avhenger av hvor hodet leser til harddisken står osv.

MINNEPYRAMIDEN, INTERNMINNET

REGISTERE: Vi starter helt innerst her med registere, som er ekstremt hurtige. Registrene ligger inni CPU-en, og de kan gå inn og kopiere mellom registrene på mindre enn ett sekund, typisk på en eller to klokkesykluser.



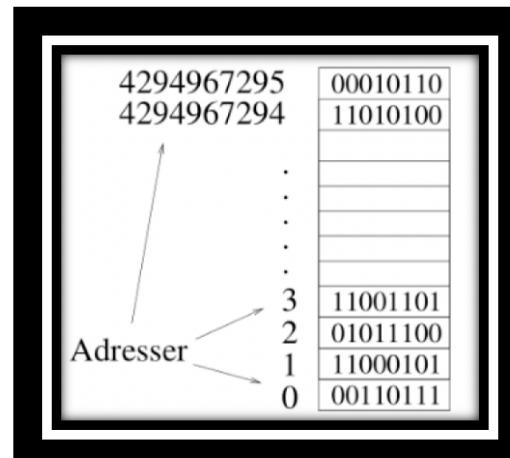
L1, L2 & L3: L1 cache er fortsatt SRAM, og det er på en måte samme teknologi, men det ligger litt lengre ut, og tar derfor litt lengre tid. Også kommer man enda lengre ut med L2 cache, og L3 cache er et hakk utenfor der igjen, med noen flere nanosekunder.

INTERNMINNET: Så kommer man ut i RAM eller internminnet og der ser vi at det er en faktor på 10 inn til registrene. Og det er denne forskjellen som gjør at cache er så viktig, man klarer ikke å få data fort nok til CPU-en uten å bruke cache.

DISKER: De har random access på samme måte som RAM, at det kan ta hvilke som helst tider på å hente en byte, det går mye saktere med en faktor på minst 1000, i forhold til RAM. Også har du kanskje grovt sett en faktor på 1000 igjen til fysiske harddisker som spinner rundt.

Internminnet/RAM/arbeidsminnet er et stort array av bytes med hver sin adresse. Vi ser på adresse null befinner det seg en 8-bit som tilsvarer en byte. Øverst ser vi $42 = 2^{32}$, så dette er 4 giga med adresser. Dermed har du 4 gigabyte med RAM i dette arrayet.

Vi har jo også sett hvordan vi har lagret et 64 bit integer som trenger 4 byte. Og da setter man av 4 byte etter hverandre.



ADRESSEROMMET

Vi trenger å kunne adressere det som lagres. Det har vi for eksempel sett på tidligere med move for å flytte noe fra RAM inn i de interne registrene i CPU-en. For eksempel med %rsp

som inneholder en adresse som peker til toppen av stacken, og toppen av stacken er en spesifikk adresse.

IP-ADRESSER: er et annet eksempel på adresserom. Alle IPv4 adresser må ligge innenfor dette rommet.

- Adresserommet: Alle mulige adresser (f. eks. IPv4 adresserommet 0.0.0.0 - 255.255.255.255)

Adresserommet for internminnet. Man trenger et register for å lagre en adresse. Til å begynne med hadde man 16-bits CPU-er, og da hadde man register størrelse på 16.

- Adresserom for internminnet: 0 - Maks

Antall bit man bruker bestemmer hvor stort adresserommet blir.

- Antall bit man bruker bestemmer hvor stort adresserommet blir

Registerstørrelse (i bit)	antall adresser
16	$2^{16} = 64 \text{ K}$ (Kilo, 10^3)
32	4 G (Giga, 10^9)
48	256 T (Tera, 10^{12})
64	20 E (Exa, 10^{18})

VIRTUELLE ADRESSEROM

Vi har et fysisk adresserom, det er alle adressene fra 0 til maks. Generelt er det ikke plass til alle adressene til internminnet på en gang. Hvis du har mange programmer, må de til en viss grad lastes inn og ut. I tillegg er det veldig viktig å ha et virtuelt adresserom, sånn at man dynamisk kan velge hvilke biter av et program som skal være med.

Måten man organiserer det: Hvert enkelt program får en adresse fra 0 til maks, f eks til 4G (hvis det er 32 bit adresser). Hver prosess tror den har tilgang til alt dette minnet. I

virkeligheten ligger noe av dette i RAM og noe i harddisken. Disse logiske adressene brukes overalt hvor programmet refererer til seg selv.

EKS:

```
mov (1023), %al
```

her er 1023 den virtuelle adressen, ikke den faktiske fysiske adressen.. Det sier flytt det som ligger i minneadresset 1023 til %al. Når et program lastes inn i internminnet og kjøres vil det variere hvor i det fysiske minnet programmet legges. CPU må oversette ekstremt hurtig mellom virtuelle og fysiske adresser. Gjøres av **MMU** (memory management unit).

MMU hardware som oversetter fra virtuelle adresser til fysiske.

Q = HVA ER GDDR? GRAPHICS DDR SOM BRUKES SAMMEN MED GPU-ER

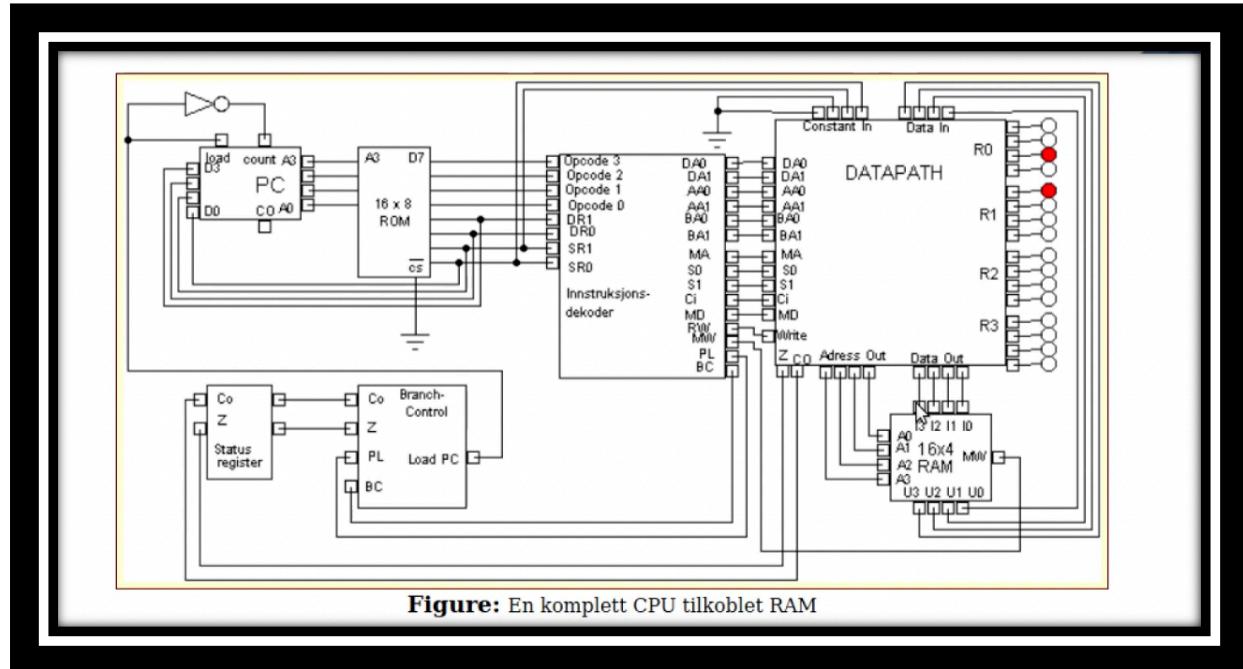
Graphic DDR. Brukes i GPU-er. Prosessoren som prosesserer grafikk. I GPU har man veldig mange uavhengige kjerner. Små kjerner med relativt lite RAM til hver, men kan ha flere uavhengige kjerner som kan jobbe i parallell, noe som er opplagt viktig når man jobber på grafikkenheter.

Q = HVA SLAGS REGISTERE LAGRER ADRESSER? VANLIGE CPU-REGISTERE. DEMO AV HVORDAN DET SER UT I SIMULERINGS-CPUEN

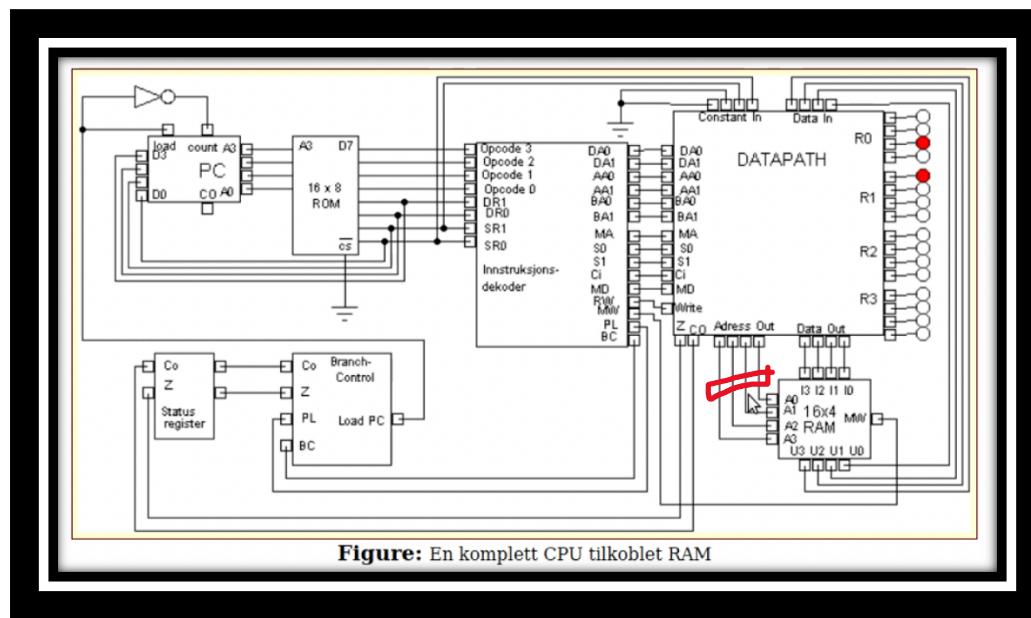
Bildet under viser hvordan RAM er koblet med CPU-en. Data-out, adresse-out tilsvarer databussen. I vår veldig enkle CPU hadde vi registrere med 4-bit, men det er det samme om du har 64-bit, da vil det være 64-bit registrere, med 64 såne koblinger mellom RAM og datapath. Vi ser at når vi skal snakke med RAM, så må vi spesifisere adressen og da er det 4 bit her som spesifiserer adressen. Og da er det vanlig at man kobler et register inn i bussen på adresselinjene. Og hvis man her kobler R1 til adresse-out betyr det skriv til byte nummer 1 i RAM. Står det 8 skriver man til 8. på den måten trenger man et register som kobles til adressen og et til datapath.

Hvis man f eks skal sende tallet 8 til adresse 4 her: så må det være et register med tallet 8, også må det være koblet med data-out, også må et annet register med tallet 4 være koblet til adresse-out. Også trykker du på skriv knappen og da skrives det. Alle tall som kommer fra

CPU-en på en eller annen måte, må lagres i registeret. Når Hårek skulle programmet dette skrev vi tallet som pekte til der det skulle lagres.



VIRTUELLE (LOGISKE) ADRESSER, INTERNMINNET OG KOMPILERING, LINKING OG LOADING

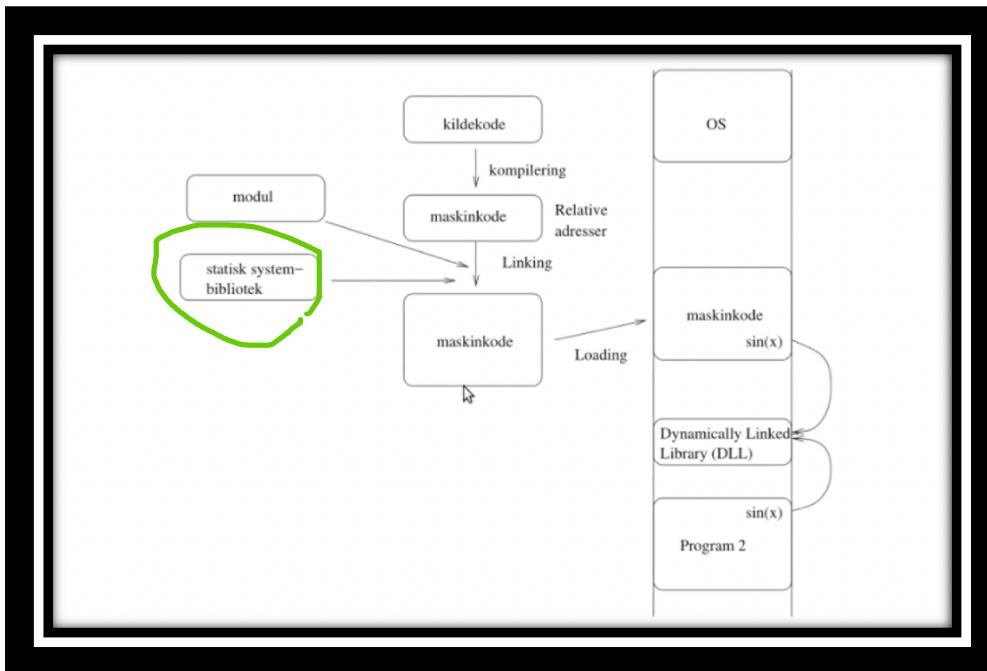


MMU-en vil tilsvare den røde delen markert over som ville ha sagt 'nei de mener ikke adressen 8 som er det virtuelle, de mener f eks 108 i det fysiske rommet'. Den enheten må være rask.

HVA ER DET SOM SKJER NÅR VI KOMPILERER PROGRAMMER

OG DE LASTER?

Man har kode også må det kompileres, så får man kjørbar kode, også må det legges inn i internminne og kjøres.



DDL = dynamic linked library er en samling av mindre programmer som større programmer kan laste etter bruk for å utføre spesifikke oppgaver. Kan brukes av flere samtidig.

DEMO: GENERERING OG BRUK AV STATISK OG DYNAMISK BIBLIOTEK I C++

BIBLIOTEK I C++

C++ library

Et C++ prosjekt kan ha metoder man ofte bruker i en egen library-fil som man kobler sammen med hovedprogrammet når man skal bruke det.

```

g++ -c calcTools.cpp          # Lager maskinkode calcTools.o
g++ -c randTools.cpp          # Lager maskinkode randTools.o
ar rcv libTools.a calcTools.o randTools.o # Lager lib-filen libTools.a

```

Senere kan dette biblioteket brukes i et prosjekt:

```

g++ -c -I../Tools mainsim.cpp      # Lager maskinkode mainsim.o
g++ -c -I../Tools simulation.cpp   # Lager maskinkode simulation.o
g++ -c -I../Tools user.cpp         # Lager maskinkode user.o
g++ -o sim mainsim.o simulation.o user.o -lm -L../Tools -lTools
sim # Kjører programmet

```

Over er det en simulering Hårek lagde, så det er et C++ prosjekt. Han lagde et bibliotek som gjorde noen beregninger, blant annet noe random generering av tall og regning ut av standardavvik osv. Det han gjorde var å lage et eget C++-library. Over ser vi masse ‘g++’ som er en C++ kompilator. Det han gjorde først var å kompilere kildekoden til biblioteket. ‘ar’ er en kommando for å lage et bibliotek:

```
g++ -c calcTools.cpp                      # Lager maskinkode calcTools.o
g++ -c randTools.cpp                      # Lager maskinkode randTools.o
ar rcv libTools.a calcTools.o  randTools.o # Lager lib-filen libTools.a
```

Så lagde han et bibliotek med libtools.a i den siste linje over i bildet.

Ved siden av den røde linja er det en kodelinje som lager en eksekverbare fil som heter sim ved å lime sammen alle programmene. Så i praksis lagde Hårek sin egen fil for det statiske-system-biblioteket (se ikke forrige, men bildet før det). Her prøves det i praksis:

calcTools.cpp er en metode som regner ut varians:

```
haugerud@lap:~/fag/simulerering/Tools$ cat calcTools.cpp
#include "calcTools.h"

void calcTools::CalculateVariance(float *aver, float *vari,int N)
{
    *aver = *aver/(1.0*N);
    *vari = *vari/(1.0*N);
    *vari = sqrt(*vari - (*aver)*(*aver));
}

void calcTools::init()
{
    int i;
    for(i= 0;i<1000000;i++)
    {
        big[i] = 1.0;
    }
}
```

Et lite skript som kompilerer. Først kompiles de to programmene, også lages det et bibliotek

```
haugerud@lap:~/fag/simulerering/Tools$ cat comp.sh
g++ -c calcTools.cpp                      # Lager maskinkode calcTools.o
g++ -c randTools.cpp                      # Lager maskinkode randTools.o
ar rcv libTools.a calcTools.o  randTools.o # Lager lib-filen libTools.a
```

Her kjøres det, og da har han lagd et bibliotek.

```
haugerud@lap:~/fag/simuleringsTools$ bash -x comp.sh
+ g++ -c calcTools.cpp
+ g++ -c randTools.cpp
+ ar rcv libTools.a calcTools.o randTools.o
r - calcTools.o
r - randTools.o
```

LS -LAT. libtools.a er selve biblioteket-filen som jeg nå skal bruke når jeg kjører simuleringen:

```
haugerud@lap:~/fag/simuleringsTools$ ls -lat
total 72
drwx----- 2 haugerud haugerud 4096 april 13 09:53 .
-rw-rw-r-- 1 haugerud haugerud 8094 april 13 09:53 libTools.a
-rw-rw-r-- 1 haugerud haugerud 5192 april 13 09:53 randTools.o
-rw-rw-r-- 1 haugerud haugerud 2272 april 13 09:53 calcTools.o
drwx----- 8 haugerud haugerud 4096 april 12 22:05 ..
-rwx----- 1 haugerud haugerud 156 april 12 22:05 calcTools.h
-rwx----- 1 haugerud haugerud 293 april 12 22:05 calcTools.cpp
-rwx----- 1 haugerud haugerud 134 april 12 22:05 calcTools.h~
-rwx----- 1 haugerud haugerud 184 april 12 22:05 calcTools.cpp~
-rwx----- 1 haugerud haugerud 227 april 12 22:05 comp.sh
-rwx----- 1 haugerud haugerud 303 april 12 22:05 recipesInC.h
-rwx----- 1 haugerud haugerud 584 april 12 22:05 randTools.h
-rwx----- 1 haugerud haugerud 1619 april 12 22:05 recipesInC.cpp
-rwx----- 1 haugerud haugerud 513 april 12 22:05 readwrite.cpp
-rwx----- 1 haugerud haugerud 1210 april 12 22:05 randTools.cpp
-rwx----- 1 haugerud haugerud 691 april 12 22:05 Makefile
```

Enda et lite program som kopierer selve simuleringen og det linkes sammen til en simulering til slutt.

```
haugerud@lap:~/fag/simuleringssysadm$ cat comp.sh
g++ -c -I../Tools mainsim.cpp      # Lager maskinkode mainsim.o
g++ -c -I../Tools -w simulation.cpp # Lager maskinkode simulation.o
g++ -c -I../Tools user.cpp         # Lager maskinkode user.o
g++ -o sim mainsim.o simulation.o user.o -lm -L../Tools -lTools
```

Kjører sim. Dets som skjer er at det lastes inn i minnet, inkludert biblioteks-filene.

```
haugerud@lap:~/fag/simuleringssysadm$ ./sim
```

```
week 0
```

```
^C
```

Dynamiske bibliotek i Linux heter .so og i Windows heter det .dll (dynamical linked library).

```

haugerud@lap:~/fag/simuleringsTools$ cat comp.sh
g++ -fPIC -c randTools.cpp
g++ -fPIC -c calcTools.cpp
g++ -shared -o libsTools.so randTools.o calcTools.o
haugerud@lap:~/fag/simuleringsTools$ bash -x comp.sh
+ g++ -fPIC -c randTools.cpp
+ g++ -fPIC -c calcTools.cpp
+ g++ -shared -o libsTools.so randTools.o calcTools.o

```

Jeg gadd ikke se mer av forelesningsvideoen her fordi jeg mener det er nok at jeg researcher om disse bibliotekene på nett.

Kjøringen hans med et dynamisk bibliotek:

```

haugerud@lap:~/fag/simuleringsTools$ ./sim
week 0
week 1
week 2
week 3
week 4
week 5
week 6
week 7
week 8
week 9
^C

```

Biblioteket lastes inn når han kjører det, så da kan han lage andre simuleringer og andre programmer som bruker akkurat det samme biblioteket.

Under ser vi at det er en forskjell på en kilobyte (venstre statisk høyre dynamisk). Den kilobyten er rett og slett at i tilfellet med statisk lenket → all koden fra biblioteket ligger fysisk inne i den kjørbare filen. Men til høyre så lastes biblioteket dynamisk og dermed så ser vi at koden er litt mindre.

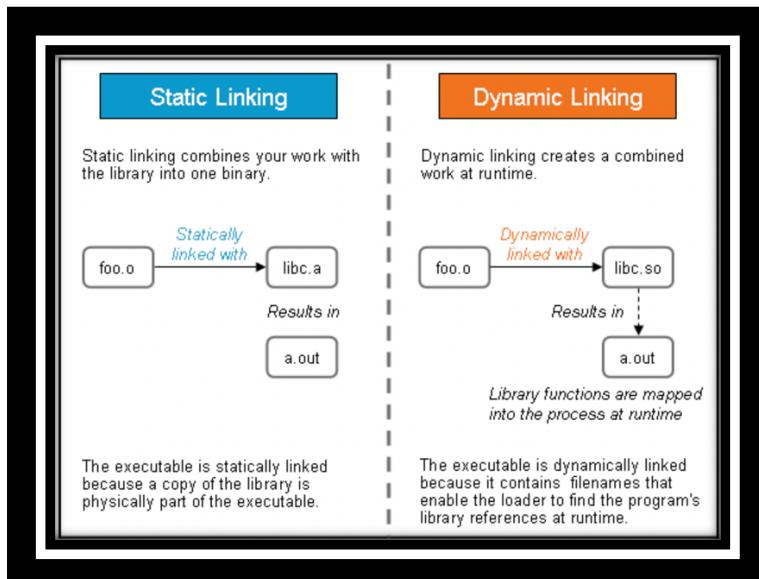
<pre> haugerud@lap:~/fag/simuleringsTools\$ bash -x comp.sh + g++ -fPIC -c randTools.cpp + g++ -fPIC -c calcTools.cpp + g++ -shared -o libsTools.so randTools.o calcTools.o haugerud@lap:~/fag/simuleringsTools\$ cd .. haugerud@lap:~/fag/simuleringsTools\$ ls -l sim haugerud@lap:~/fag/simuleringsTools\$ ls -l sim </pre>	<pre> haugerud@lap:~/fag/simuleringsTools\$ ls -l sim -rwxrwxr-x 1 haugerud haugerud 26688 apr 13 09:56 sim haugerud@lap:~/fag/simuleringsTools\$ </pre>
--	---

BIBLIOTEKER I C

Et bibliotek i C er en samling objektfiler som er eksponert for bruk og bygge andre programmer, så i stedet for å omskrive deler av kode henter vi tilbake informasjon som allerede eksisterer, det er her det kommer til konseptet med et bibliotek. I C finnes det to typer

biblioteker, de første er de statiske som lar oss koble til programmet og er ikke relevante under kjøringen, og de andre kalles dynamiske biblioteker som er å foretrekke å bruke når du kjører mange programmer på samme tid som bruker det samme biblioteket og du ønsker å være mer effektiv.

Selv om statiske biblioteker kan gjenbrukes i flere programmer, er de låst til et program på kompileringstidspunktet. Dynamiske eller delte biblioteker eksisterer på den annen side som separate filer utenfor den kjorbare filen.



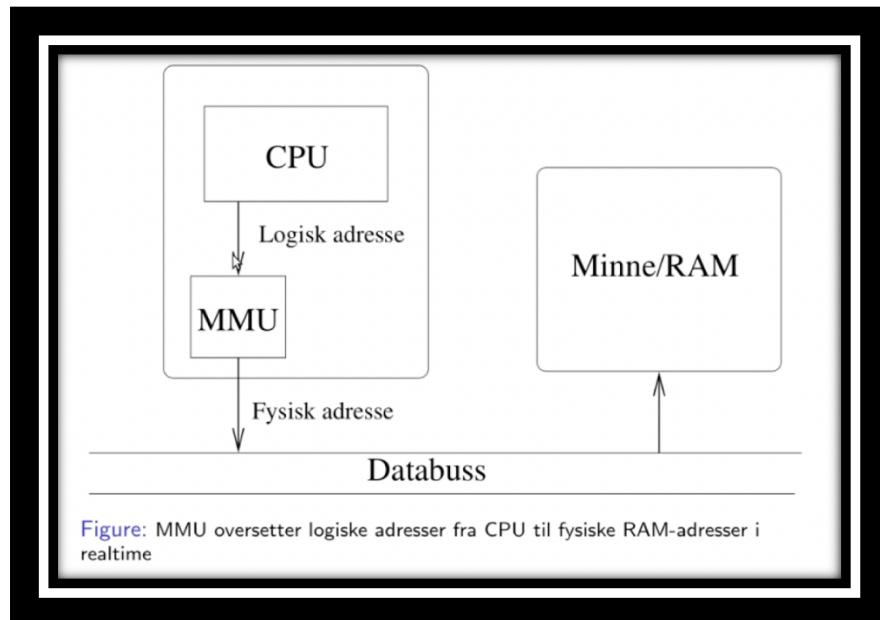
LINUX-PROSESS SEGMENTATION

Dette er slik Linux velger å gjøre det når det når de legger opp minnet i RAM. Stack brukes til lokale variabler. Nede har vi tekst (eller koden), typisk maskinkoden som programmet bruker å kjøre i. Heap: her lagres det globale variabler og data som genereres dynamisk mens programmet kjører. Og den vokser oppover, som betyr mens programmet kjører kan vi legge på flere globale variabler (eller array). MMAP, som er minneavbildninger av filer (og enheter) på disk direkte i det virtuelle minnet).

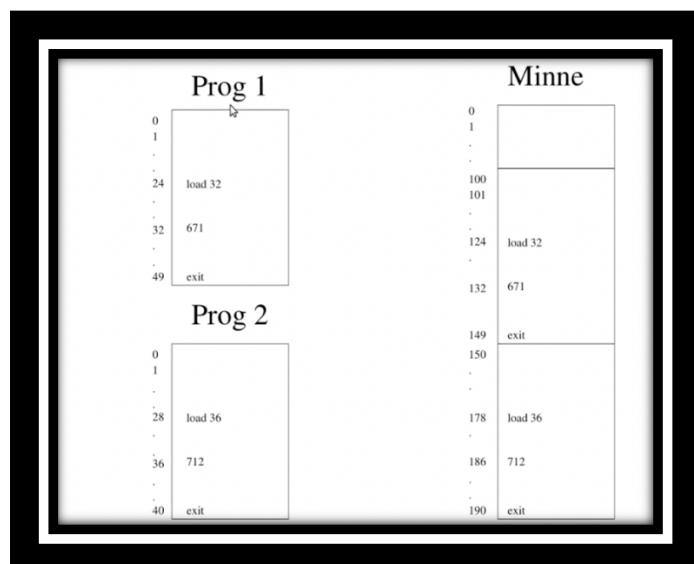
MINNEADRESSERING OG MMU MED LITE EKSEMPEL

Alle de virtuelle adressene må kunne knyttes til fysiske adresser. Dette kunne ha skjedd ved loading, altså lasting, men det er tidskrevende og tungvint, så i moderne OS gjøres dette dynamisk mens programmene kjører. Dette gjør at programmer og biblioteker kan flyttes til og fra harddisk og bare loades når det er behov for dem. Det kunne ha vært et alternativ å la

os oversette fra fysiske til virtuelle adresser. Men det tar for lang tid, og det tar for mye belastning av CPU. Det må skje på brøkdelen av et nanosekund. Dermed er det ikke tid til en add-instruksjon, for eksempel. Derfor har vi en egen enhet inne i CPU-en, MMU.

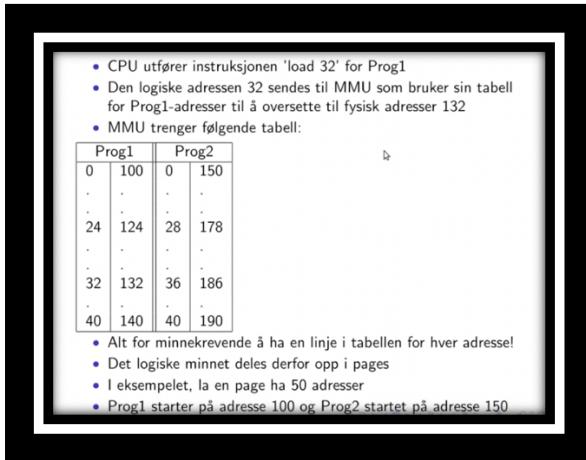


Anta at de 2 programmene program 1 og program 2 skal kjøres. Etter at man har kompilert programmene, så er adressene logiske. Da må man ha MMU enheten eller en slags tabell som oversetter de adressene til de riktige.



Vagt forklart over sier vi at i program én så sier linje kode 24 at vi skal laste inn verdien som ligger i registeret adressen 32, og det er 671. I program 2 så står det at vi skal laste verdien som ligger i adresse 36 i RAM, og det ser vi er 712. I minne, altså i ram til Høyre, så ser vi at i registeret adressen 32 så ligger tallet. 671 men det er egentlig ikke 32 som er register adressen. Det er 132. Tilsvarende for program 2, så er ikke den ekte adressen 36. Det er 186. Hovedpoenget er at de logiske eller virtuelle adressene må oversettes til fysiske adresser.

Vi kunne ha sagt at adressen 24 er 124 i fysisk. Adresse 28 er 178 i fysisk. Men dette er for minnekrevende. Da vil en linje tabellene eksistere for hver eneste adresse. I stedet har vi pages. Vi sier et program en starter på adresse 100 og program 2 starter på adresse 150. Dette kan vi se i bildet over.

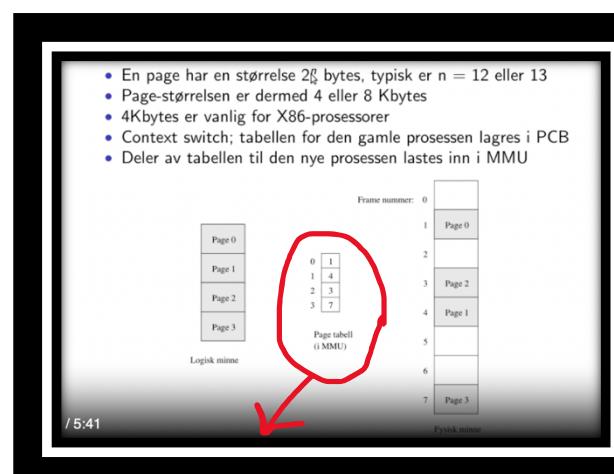


PAGING, PAGES OG PAGE TABLE ENTRY

Dette med å bruke et virtuelt minnerom kalles paging. Det er fordi man deler hver prosess sitt minnerom i et antall sider. La oss si at de forrige prosessene hadde 50 byte hver side. Så hvis de bestod av 500 bytes, så ville de ha fått 10 sider. Og i MMU ville det bare ha stått hvor de 10 sidene ligger. Os kan da effektivt laste disse sidene inn og ut av ram og samtidig holde oversikt over hvor hver side er i page tabellen. Det gjør at dynamisk flytting av deler av prosesser til og fra disk blir mulig. Det gir full kontroll for os over prosessers minnebruk. Det er muliggjør å bruke diskplass til å utvide minnet.

En side har en størrelse på 2 opphøyd i n-te byte.

Vanligvis er $n = 12$ eller 13. Da får sidestørrelsen 4 eller 8 kbytes. 4 kbyte er vanlig for x86 prosessorer. Hver prosess har en egen tabell. I en prosess er det bare en CPU som kjører av gangen. I MMU da trenger vi bare en tabell for den prosessen som kjører av gangen. Når det skjer en kontekst switch, så lagres tabellen for den gamle prosessen i PCB. I bildet under kan vi se for oss til Venstre at vi har et bilde av det logiske minnet med side 0 1 2 3. Vi ser også at MMU-en vil vise hvor de fysiske adressene i minne ligger.

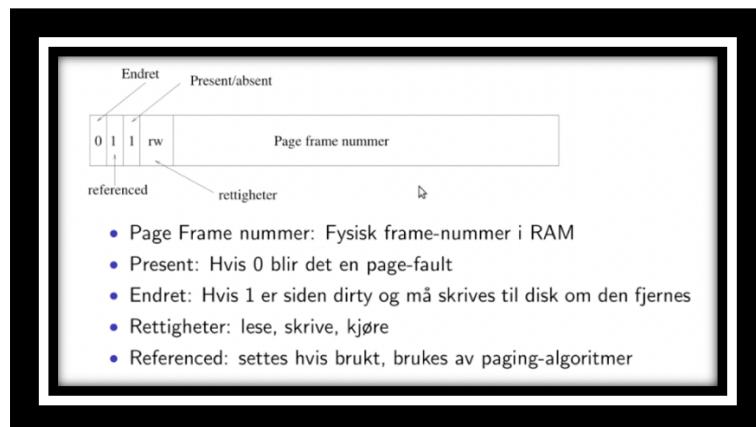


I tabellen har vi en page tabell entry for hver side i tabellen. Denne sammenhenger med midterste delen av bildet over. Man kan sette rettigheter på minne også, per prosess og per page.

Present/absent sier om denne siden ligger i det fysiske minnet. Hvis det står 0 betyr det at den ligger på disken. Hvis programmet ønsker noe da må det hentes fra disken og det kalles page-fault. Den siden mangler og det må ut og hentes, noe som tar veldig lang tid.

Endret: det er også 0 eller 1, hvis den er 1 så er siden dirty, eller har blitt endret, det betyr at hvis den siden skal ut av RAM så må verdien skrives til disk.

Reference: brukes av paging-algoritmer som bestemmer til hvilken tid hvilke sider skal ha plass i RAM.



TLB - TRANSLATION LOOKASIDE BUFFER, YTELSE OG TLB-CACHE

Vi har sett de CPU-en at vi har L1 og L2 cash for RAM. Da snakker vi hele tiden om kode og variabler. Men vi har det samme for MMU. Dette er fordi MMU må gå for raskt. Da har man TLB som er en cache bit spesielt for MMU adressene. EKS:

- En prosess som bruker 100 Mbyte minne vil med 4 Kbyte page størrelse bestå av omrent 25.000 sider

- En 4 GByte prosess gir en million sider i MMU
- Ikke plass til å lagre adressen til alle disse sidene i MMU
- Den fullstendige tabellen ligger selv i internminnet
- MMU bruker en Translation Lookaside Buffer (TLB) som er hurtig cache minne

$$4\text{kbyte} = 2^n = 2^{12}$$

Med eksempelet over kan vi se at en slik tabell blir fort veldig stor.

MMU bruker en Translation Lookaside Buffer (TLB) som er hurtig cache minne. Den vil bare holde en liten del av page-tabellen.

Ved oppslag på adresser til sider som ikke ligger i TLB, hentes de fra RAM. Det kalles TLB-miss eller soft-miss og tar lengre tid enn om de er i TLB fordi man må lenge ut i RAM.

TLB-YTELSE

- størrelse: 16 - 4096 linjer (1 - 256 kBytes)
- En cache linje er vanligvis 64 bytes
- oppslagstid: 0.5 - 1 klokke-sykkel
- ekstra tid ved TLB-miss: 10-100 klokke-sykler
- TLB-miss frekvens: 0.01 - 1%

Noe sånt under har vi sett. Vi ser at i prosessoren kan vi ha L1 og L2 cash. For Intel og AMD nyere, så har det fått plass til L3 cash også. Vi har sett at L1 cache er for data og instruksjoner. Dette er fordi det går raskere fremfor å hente det fra RAM hver gang. Helt det samme er det for MMU. Og da har vi en L1 cache inne i den røde sirkelen under. 99% av tilfellene så treffer man på TLB, og da går oversettelsen fra virtuelt minne til fysisk minne veldig raskt.

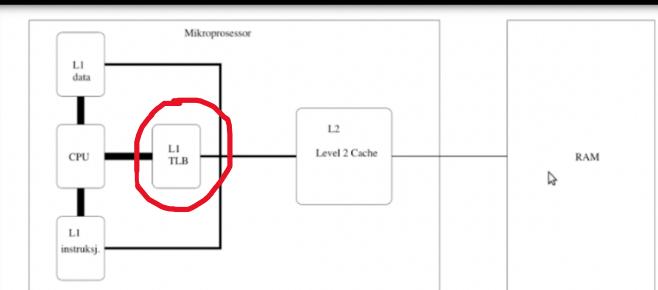


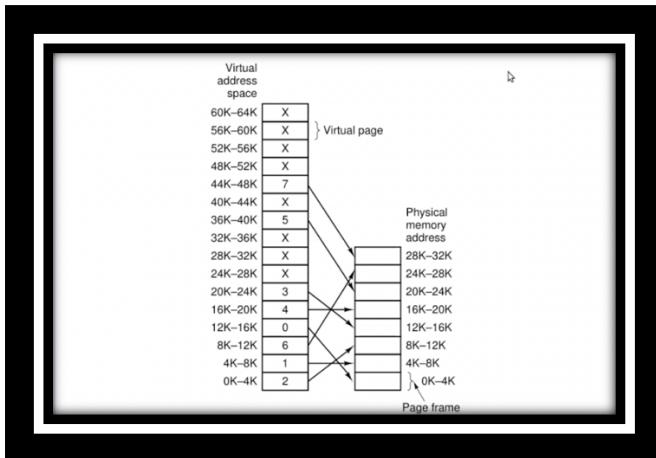
Figure: Level 1 cache (L1) bestående av tre deler. I AMD Athlon 64 er TLB i tillegg delt i to deler, en for adresser til instruksjoner og en for adresser til data.

For Intel Core i7 og AMD Opteron K10 har også L3 cache fått plass på prosessor-chip'en.

KONKRET EKSEMPEL MED MMU-OVERSETTELSE MED 64K

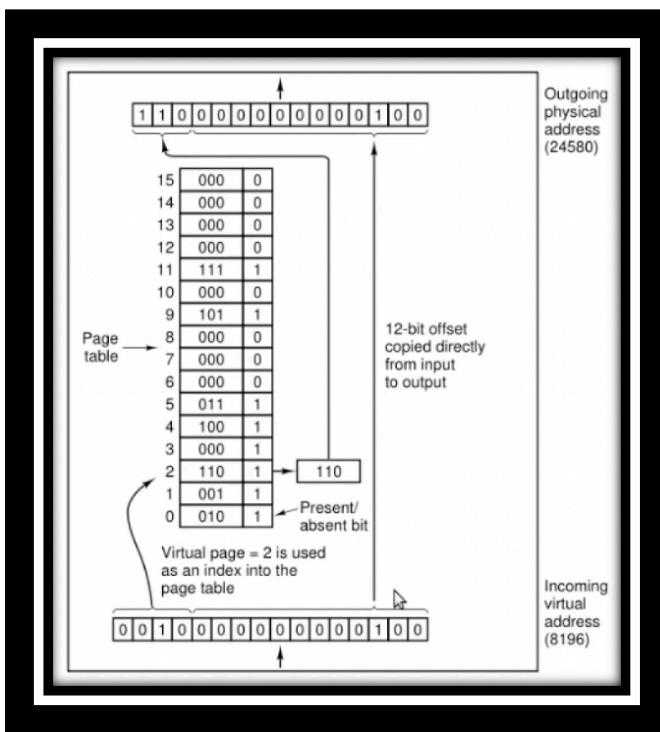
VIRTUELT OG 32K FYSISK INTERNMINNE

Under er det en figur hentet fra Tanenbaum. Det viser oversettelse fra det virtuelle adresserommet til det fysiske adresserommet. Her er det veldig små størrelser. Vi viser 32K fysisk minne, med 64K virtuelt minne. Nederst til venstre er side nummer 0 som er mappet til fysiske minnet nummer 2. Når det kommer en minnedresse fra venstre inn til MMU-en, må MMU lynraskt oversette adressen til fysisk minne.



HVORDAN OVERSETTE?

Det regnes ikke ved CPU-instruksjoner, det er direkte kabling:



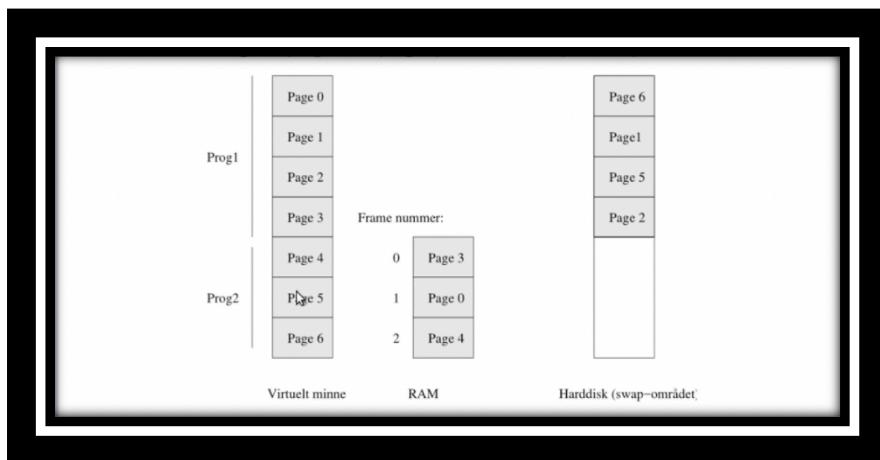
Vi har 12 bit under som sendes inn, og det vil si at hver side har 2^{12} adresser, og det 4096. den 12 bit-en som går inn utgjør 4096 adresser. I den innkommende virtuelle adressen så har vi i tillegg på starten en 4 bit som utgjør hvilken side de virtuelle adressene ligger på. 0010 utgjør 2 i desimal og vi ser den peker på side virtuelle side 2. de neste 12 bit-ene er hvor i adresse-

rommet det er. Verdien som ligger i side 2, 110 limes på starten av den utgående fysiske adressen.

Resten av delen har jeg lært, så jeg ser bort fra dette.

PAGING OG SWAPPING, PAGING-ALGORITMER

Paging gjør da at man deler inn programmer til sider (se bildet under). Og så har man fysisk RAM som ligger i midten på bildet under. Det er ikke alle sidene fra program én og program 2, som ligger i ram. De som ikke ligger i ram, ligger i harddisk på swap-området. Dette er et tilfelle hvor man ikke får plass til alt i RAM, da sliter man. Man kan høre fysisk disken fordi den kjører hele tiden ettersom ting flyttes inn og ut.



PAGING-ALGORITMER

Algoritmer som bestemmer hvilke sider som skal ligge i RAM, og styres av RAM. En page-fault er når man adresserer noe som ikke ligger i RAM, men ligger ute i disk. Tar veldig lang tid fordi man fysisk går ut i RAM for å hente noe, typisk vil en prosess scheduleres ut mens man venter på det.

SEGMENTATION FAULT

Hvis et program snakker med en adresse som ikke er sin, utenfor sine sider. Det er noe os sørger for.

HVORFOR LAGER MAN IKKE STØRRE L1, F EKS 1GB?

1. kostnad. 2. latens: jo større cache jo lengre tid å hente ut data fra den. 3. strømbruk: økes og er uønsket side effekt for mobiler og andre batteridrevne enheter.

NY FORELESNING!

SIST OPPSUMMERING: VIRTUELLE ADRESSER, MMU,

PAGETABLES, TLB

Det aller viktigste å få med seg er at internminnet eller ram ligger etter hverandre. Ram heter random access memory fordi det skal ta like lang tid å hente enn hva man trenger. Vi sa at dette ikke er helt sant grunnet cache. Vi så på MMU som er en enhet på harddisken som oversetter fra virtuelle til fysiske adresser, fordi hver en prosess har et eget adresserom, fra 0 og opp. Dette gjør at OS dynamisk kan laste inn og ut sider av ram og ikke minst laste inn prosesser. I tillegg så vi på TLB som er cache for minneadressering. Heldigvis ligger det meste i TLB, så det trengs ikke å gå til disken for å hente minne.

DYNAMISK ALLOKERING

Et program kan be om at det settes av ram til sine variabler før det starter. Men det kan også be om at minne allokeres dynamisk. For eksempel i java kan vi gjøre dette ved å deklarer arrays. Ellers slik som under:

PCB = new process;

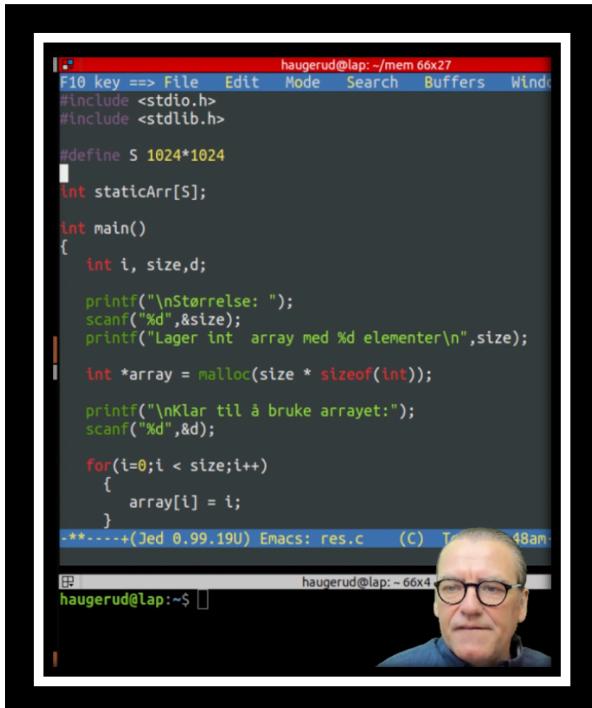
og da settes det av flere sider til programmene. Vi skal senere se på at i C og C++ må man eksplisitt slette objekter som ikke er i bruk lenger for å frigjøre minnene. Hvis man hele tiden allokerer mer minne uten å frigjøre det kan man få det som heter minnelekkasje. Det er et vanlig problem i programmer når det brukes mer og mer. En programmerer må hele tiden sørge for at minne hele tiden frigjøres.

JVM og C++ og relativt nyere språk sørger for dette automatisk og da kalles det garbage collection.

Den delen av RAM som dynamisk kan øke og minke i størrelse kalles heap, her legges alt av er som array (øker og minker) i størrelse. Mens i stacken inneholder den alt som hopper til å fra med metoder å gjøre. Dette er typer organiseringen som os driver med.

DEMO: KJØRING AV C-PROGRAM MED STATISK OG DYNAMISK ALLOKERING AV MINNE OG MONTORERING AV RAM-BRUK MED TOP; VIRT, RES OG SHR

Vi skal se i detalj på hvordan dette C-programmet virker. Det vi først ser er at vi definerer et stort int array. Da har jeg definert et tall som er $S = 1024 * 1024$. en int er 4 byte og dette arrayet vil dermed ta denne størrelsen.



```

F10 key ==> File Edit Mode Search Buffers Windows
haugerud@lap:~/mem 66x27
#include <stdio.h>
#include <stdlib.h>

#define S 1024*1024
int staticArr[S];

int main()
{
    int i, size,d;
    printf("\nStørrelse: ");
    scanf("%d",&size);
    printf("Lager int array med %d elementer\n",size);

    int *array = malloc(size * sizeof(int));
    printf("\nKlar til å bruke arrayet:");
    scanf("%d",&d);

    for(i=0;i < size;i++)
    {
        array[i] = i;
    }
}
***--+(Jed 0.99.19U) Emacs: res.c (C) Tom Hårek 48am
haugerud@lap:~$ 
```

Det første vi skal se på er ‘hva skjer når vi deklarerer dette arrayet?’. Nedover i koden ser vi at det stoppes for å skrive ut størrelse også ;. og scan, fordi den leser inn den variabelen. Etterpå skal vi se hvordan vi kan lese inn størrelsen på arrayet også allokkere det. Hårek endrer koden med at inni int staticArr[S] skriver han inn 1.



vi skal se hvordan det ser ut når vi kjører det programmet.



```

haugerud@lap:~/mem 66x15
haugerud@lap:~/mem$ gcc res.c
haugerud@lap:~/mem$ ./a.out
Størrelse: 
```

I et annet vindu ser vi at det kompileres og kjøres, og da spørres det om størrelse. Vi åpner et nytt vindu med top. Denne kommandoen under viser bare top for det programmet a.out. -p oppsjonen lar oss spesifisere hvilken prosessid vi vil se på. Under ser vi at hvis vi spør om pid for a.out så får vi det ut.

```
haugerud@Lap:~$ top -p^C(pidof a.out)
haugerud@Lap:~$ pidof a.out
15173
```

Vi skal se på minne her:

```
top - 08:50:54 up 12:06, 1 user, load average: 2,09, 1,77, 1,50
Tasks: 1 total, 0 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 8,7 us, 1,8 sy, 0,0 ni, 89,4 id, 0,0 wa, 0,0 hi, 0,0 si, 0
KiB Mem : 32857340 total, 25107444 free, 3590572 used, 4159324 buff/cach
KiB Swap: 33431548 total, 33431548 free, 0 used. 28610612 avail Mem

      PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+
 15173 haugerud  20   0    4516   748   684 S  0,0  0,0   0:00.00
```

VIRT er det virtuelle adresserommet, det er så mye [RAM] som er definert for denne prosessen. RAM her er definert i K. Det vil si det 4516K med virtuelt adresserom.

RES(resident) altså det som er i RAM nå, og som er i bruk. I virt ligger alt, men det er ikke alt som er i bruk, og det er ikke nødvendigvis at det kommer i bruk også, det spørs hvordan programmet kjøres. **Hvis mer tas inn i RAM, så skal vi se at RES øker.**

For å lage arrayet med 1 byte får vi 4516K

I programkoden endres det nå tilbake til 4byte:

```
#define S 1024*1024
int staticArr[S];
```

Etter at vi har gjort den endringen med int staticArr[4] burde det i virt bli lagt til så mye rom:

```
haugerud@lap:~$ re 4516 + 4*1024
8612
```

Da kompillerer vi igjen med endringene og kjører:

```
haugerud@lap:~ 74x16
top - 08:54:46 up 12:10, 1 user, load average: 2,04, 1,82, 1,58
Tasks: 1 total, 0 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 17,7 us, 2,5 sy, 0,0 ni, 79,7 id, 0,0 wa, 0,0 hi, 0,0 si, 0
KiB Mem : 32857340 total, 25115920 free, 3552904 used, 4188516 buff/cach
KiB Swap: 33431548 total, 33431548 free, 0 used. 28648364 avail Mem

 PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+
15385 haugerud 20 0 8412 748 684 S 0,0 0,0 0:00.00
```

Vi ser at RES er fremdeles 748 fordi vi ennå ikke har begynt å bruke static array greiene osv.

LIKE STORT ARRAY DYNAMISK

Da ser koden slik ut:

```
haugerud@lap:~/mem 66x27
F10 key ==> File Edit Mode Search Buffers Windows
#include <stdio.h>
#include <stdlib.h>

#define S 1024*1024

/*int staticArr[S]; En int er 4 byte: 4*1024*1024 = 4096*/

int main()
{
    int i, size,d;
    printf("\nStørrelse: ");
    scanf("%d",&size);
    printf("Lager int array med %d elementer\n",size);
    int *array = malloc(size * sizeof(int));
    printf("\nKlar til å bruke arrayet:");
    scanf("%d",&d);
    for(i=0;i < size;i++)
    {
        array[i] = i;
    }
}
```

Kompilerer og starter programmet igjen og skikker på top:

```
haugerud@lap:~ 74x16
top - 08:58:37 up 12:14, 1 user, load average: 1,51, 1,66, 1,56
Tasks: 1 total, 0 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 8,8 us, 1,8 sy, 0,0 ni, 89,3 id, 0,0 wa, 0,0 hi, 0,0 si, 0
KiB Mem : 32857340 total, 25078252 free, 3560848 used, 4218240 buff/cach
KiB Swap: 33431548 total, 33431548 free, 0 used. 28640408 avail Mem

 PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+
15551 haugerud 20 0 4516 720 656 S 0,0 0,0 0:00.00
```

Da har vi igjen det vi starte med i utgangspunktet: 4516K i virt.

Hvis vi ser i koden med scan %d, leses det inn ett tall, og lages et array med d elementer.

Og under er det en kommando malloc som gjør et kall til os kjernen for å be om å allokkere minne. Det som sendes med er size og størrelsen vi ønsker å allokkere til minne i byte i () .

Vi skriver inn dette tallet for å lage et så stort array:

```
haugerud@lap:~$ re 1024*1024
1048576
```

Da brukes det:

```
haugerud@lap:~/mem$ gcc res.c
haugerud@lap:~/mem$ ./a.out

Størrelse: 1048576
Lager int array med 1048576 elementer

Klar til å bruke arrayet:[]
```

Og vi ser i top.

```
haugerud@lap:~ 74x16
top - 09:01:11 up 12:16, 1 user, load average: 1,58, 1,62, 1,56
Tasks: 1 total, 0 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 17,3 us, 1,8 sy, 0,0 ni, 80,9 id, 0,0 wa, 0,0 hi, 0,0 si, 0
KiB Mem : 32857340 total, 25056900 free, 3563052 used, 4237388 buff/cach
KiB Swap: 33431548 total, 33431548 free, 0 used. 28638160 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+
15551 haugerud 20 0 8616 720 656 S 0,0 0,0 0:00.00
```

Det endret seg til 8616. det ble litt mer enn forrige gang hvor det ble det samme som under:

```
haugerud@lap:~$ re 4516 + 4*1024
8612 I
```

I terminalen står det ‘klar til å bruke arrayet:’ (se to bilder over) og der skriver Hårek 1, også enter. Da hopper RES:

```
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+
15551 haugerud 20 0 8616 5384 1416 S 0,0 0,0 0:00.00
```

Med engang array kom til bruk ved kodebiten under, kom det i RAM og MMU.

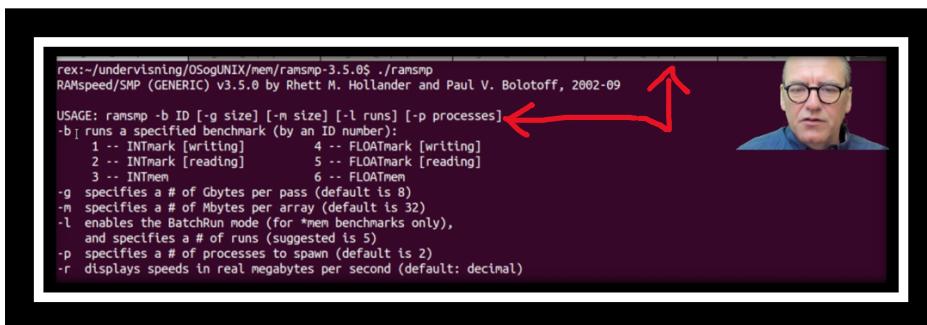
```
for(i=0;i < size;i++)
{
    array[i] = i;
}
```

Når man kjører programmer og kanskje har minneproblemer. Du kan ha kjempe stor virt men så lenge det ikke all er i bruk på res så er det ikke tungt for systemet.

SHR (shared) minne som deles med andre programmer, som oftest dynamiske bibliotek. F eks C bibliotek som vi så på sist at vi kan lage statiske bibliotek som blir kompilert inni programmet, og vil jo da være en del av virt og res, men dynamiske bibliotek vil dukke opp som SHR.

DEMO: RAMSMP, TEST AV RAM-HASTIGHET VED LESING OG SKRIVING AV BLOKKER AV FORSKJELLIG STØRRELSE

Vi skal kjøre RAM test. Hvis programmet kjøres uten argumenter, får vi info om programmet og vi trenger en -b id. Dette ser vi fordi opsjoner utenfor parenteser er obligatorisk å ha med.



```
rex:~/undervisning/OSogUNIX/mem/ramsmmp-3.5.0$ ./ramsmmp
RAMSpeed/SMP (GENERIC) v3.5.0 by Rhett M. Hollander and Paul V. Bolotoff, 2002-09

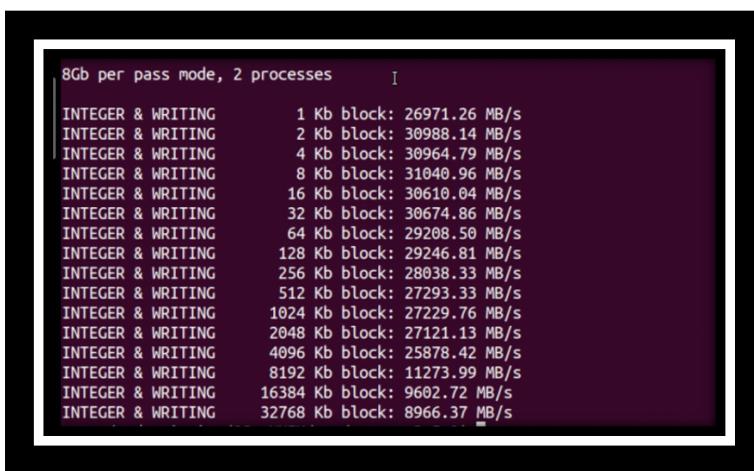
USAGE: ramsmp [-b ID [-g size] [-m size] [-l runs] [-p processes]
-b] runs a specified benchmark (by an ID number):
  1 -- INTmark [writing]      4 -- FLOATmark [writing]
  2 -- INTmark [reading]     5 -- FLOATmark [reading]
  3 -- INTmem                6 -- FLOATmem
-g specifies a # of Gbytes per pass (default is 8)
-m specifies a # of Mbytes per array (default is 32)
-l enables the BatchRun mode (for *mem benchmarks only),
  and specifies a # of runs (suggested is 5)
-p specifies a # of processes to spawn (default is 2)
-r displays speeds in real megabytes per second (default: decimal)
```

Opsjonen -b 1 gjør da en test slik at det skrives til RAM.



Det ramspeed gjør er å skrive til RAM, akkurat slik som det forrige programmet gjorde.

Under ser at det er noe med 8GB som skrives, men forskjellen er størrelsen på blokkene som skrives. I første linje skriver vi bare 1KB av gangen. Altså man har en løkke som går gjennom 250 bytes (1 KB totalt) som skrives til RAM. Og det går veldig fort, som vi kan lese på første setningen under er det 26 tusen MB/s.

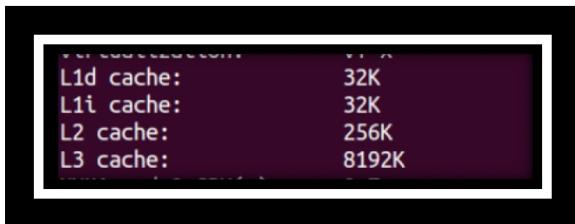


Block Size	Speed (MB/s)
1 Kb	26971.26
2 Kb	30988.14
4 Kb	30964.79
8 Kb	31040.96
16 Kb	30610.04
32 Kb	30674.86
64 Kb	29208.50
128 Kb	29246.81
256 Kb	28038.33
512 Kb	27293.33
1024 Kb	27229.76
2048 Kb	27121.13
4096 Kb	25878.42
8192 Kb	11273.99
16384 Kb	9682.72
32768 Kb	8966.37

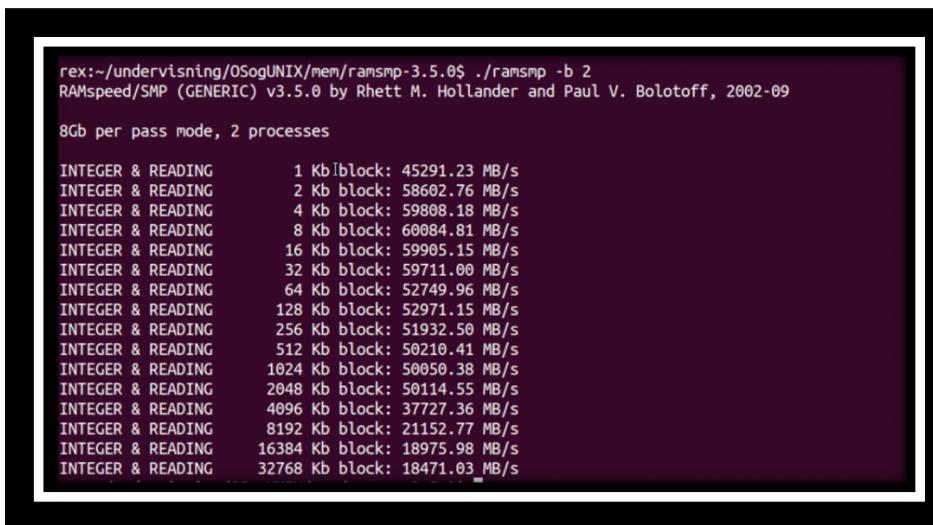
Vi ser at det minker jo lengre ned i raden. Dette er da pga cache. Fordi hvis vi skriver en liten blokk så får den plass i L1 cache, hvis vi skriver litt større får den plass i L2, og L3 cache på det siste nede (bildet under).



Skriv lscpu for å se cache størrelsene på maskinen. L1 cache har 32K for data og instruksjoner. D er data og i er instruksjoner. 236K i L2, og 8Mega i L3.



Vi kan prøve å lese også. Dette går enda forttere. Vi ser samme effekt som istad. Det faller litt etter L1 cache-en. Også faller det veldig mye fra 30K til 20K.



Å teste RAM-hastigheten med forskjellige blokkstørrelser kan hjelpe deg med å bestemme hvordan RAM-en fungerer under forskjellige arbeidsbelastninger og applikasjoner. En større blokkstørrelse kan gi deg en bedre idé om RAM-hastigheten når du arbeider med større filer eller databaser, mens en mindre blokkstørrelse kan gi deg en bedre idé om RAM-hastigheten når du arbeider med mindre data.

LINUX KOMMANDOEN FREE, FIL-CAHCE

Hvis vi bare skriver free, får vi ut en del minneinfo. Dette kan også sees på i top, men her fokuseres det mer på minne.

```
haugerud@lap:~/mem$ free
total        used        free      shared  buff/cache   available
Mem:    32857340     3611708    24914524    196428    4331108    28586764
Swap:  33431548          0    33431548
```

I utgangspunktet kommer minnet ut i K, men vi kan også spørre om det i M med opsjonen -m.

```
haugerud@lap:~/mem$ free -m
total        used        free      shared  buff/cache   available
Mem:      32087      3527     24327       191      4232      27916
Swap:    32647          0     32647
```

Under for å se i gigabyte:

```
haugerud@Lap:~/mem$ free -g
total        used        free      shared  buff/cache   available
Mem:      31          3         23          0          4         27
Swap:    31          0         31
```

Vi ser under total at det er 31 eller 32 gigabyte totalminne. Og til høyre for totalt så ser vi hvor mye som er brukt under megabyte screenshoten, og det var 3500. Til Høyre for det igjen kan vi se at det er veldig mye som er ledig. Under shared altså delt, så er det det som deles mellom flere prosesser. Vi ser at under buff/cache er det omtrent 4G, og kan være større. Det har intenting med L1 og L2 cache osv å gjøre, det er filcache. Når Linux ser her at det er masse G byte med minne ledig, som ikke brukes av noen prosesser, da tar Linux os-kjernen og cacher filer fra filsystemet. Så når man da leser inn filer så lagres det i RAM i et område som er satt av fra os, så når man senere skal lese en fil, istedet for å bruke veldig lang tid på å lese fra disken, så leser man direkte fra RAM, og det går hundretusen så fort. Og dette er en effektiv måte å bruke RAM på for å få systemet til å gå forttere. hvis programmer bruker opp alt eller mye plass så går buff/cache ned.

LINUX KOMMANDOEN TOP OG RAM, VIRT, RES OG SHR

Når man vanligvis taster top så gir den resultater i G. Hvis man taster E på tastaturet, kan man bytte på verdier. Obs sørger for store ting som faner osv. Det viktigste er å huske at det er RES som er i bruk.

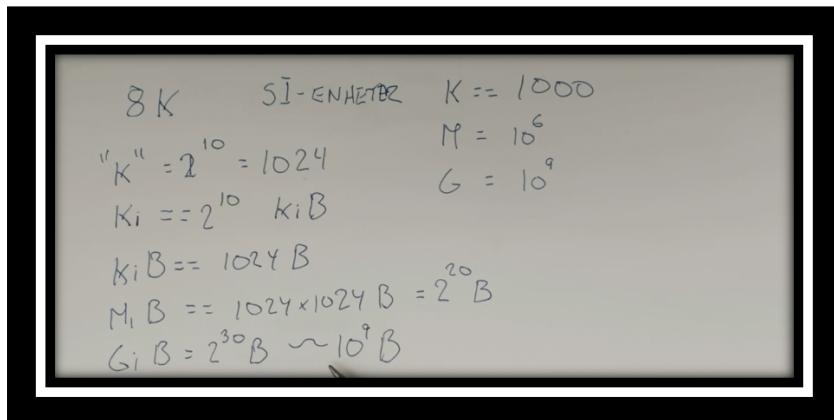
SPØRSMÅL: hvordan kan det før et program kjøres settes av mye plass?

Vi ser først på programmet vi kjørte med array størrelsene. Vi ser at det er et stort program og linker mange biblioteker. Det kreves nok plass til å sette hele infrastrukturen på plass for å kunne kjøre og endre programmet.

```
haugerud@lap:~/mem$ ls -lh ./a.out
-rwxrwxr-x 1 haugerud haugerud 8,3K april 20 08:58 ./a.out
```

TEGNING OG FORKLARING: SI-ENHETER: K, M OG GBYTE, BINÆRE ENHETER: Ki, Mi OG Gi

Når Hårek sier K snakker han om 1024byte.



C-PROGRAM MED ET ARRAY PÅ 4 GBYTE, SMÅ OG STORE HOPP I RAM GIR SVÆRT FORSKJELLIG TIDSBRUK

Starter med programmet under. Det er definert en milliard, eller en giga i [] til int array-et under. Det som prøves å definere her er et array som bruker en giga RAM. Også er det en enkel for lenke som kun går til 1/100 av det store arrayet. Vi setter $j = i * 100$, så den går helt opp til det store tallet som ble definert i arrayet. I første omgang setter vi hvert array[i] = i.

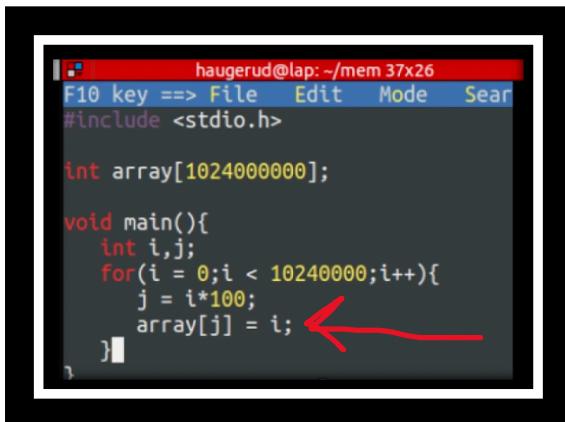
```
haugerud@lap:~/mem$ xxd
F10 key ==> File Edit Mode Search
#include <stdio.h>

int array[1024000000];
void main(){
    int i,j;
    for(i = 0;i < 10240000;i++){
        j = i*100;
        array[i] = i;
    }
}
```

Vi kompilerer og tar tiden på kjøring. Vi ser at til tross på at det skrives til 10 million array elementer så går programmet på 0,02 sekunder. Selvom vi sier at RAM er tregt så går det veldig fort, mye raskere enn til disk, det er bare det at registeret er enda raskere.

```
haugerud@lap:~/mem$ gcc -O m.c
haugerud@lap:~/mem$ time ./a.out
Real:0,019 User:0,005 System:0,014 96,84%
```

Så skal vi endre i til j.



```
haugerud@lap:~/mem 37x26
F10 key ==> File Edit Mode Sear
#include <stdio.h>

int array[1024000000];

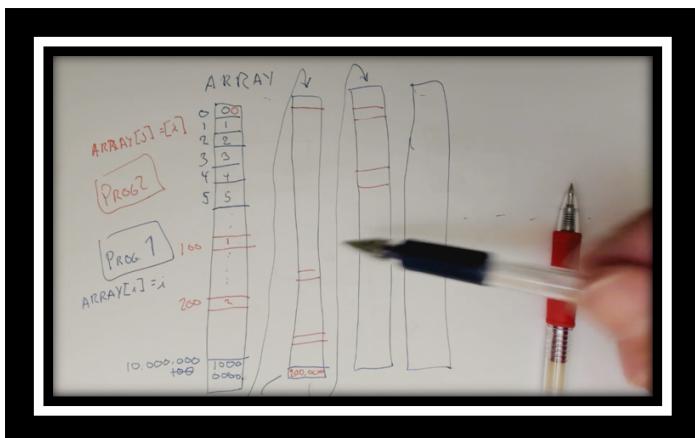
void main(){
    int i,j;
    for(i = 0;i < 10240000;i++){
        j = i*100;
        array[j] = i;
    }
}
```

Vi kompilerer igjen og kjører:

```
haugerud@lap:~/mem$ gcc -O m.c
haugerud@lap:~/mem$ time ./a.out
Real:1,546 User:0,469 System:1,077 100,00%
```

Det tar en mye lengre tid. Dette er noe som absolutt har med cache å gjøre.

Det er et stort array med mange elementer, og det ligger i RAM og setter seg på adresser etter hverandre, og det vil være logiske adresser 1, 2, 3, 4 osv. vi ser for oss at vi har mange kolonner ved siden av hverandre. Vi ser for oss at den første raden er til program 1.



Vi ser at med den blå pennen bare la vi til en og en verdi og det krevde ikke så mye plass utenfor i de andre kolonnene. Men med det andre programmet hoppet koden veldig mye og det krevde rask utvidelse av lagring på RAM. J økte med 100 for hver gang i program 2. man kan ta hele biter ut og legge det ut i L1 cache og dermed kan det gå fortare. Det andre programmet er spredt lengre ut i RAM med 100 byte i mellom, så kanskje det bare er få byte som treffer med L1 hver gang cache brukes.

MONITORERING AV KJØRING AV ARRAY-PROGRAMMET MED PROGRAMMET PERF, CACHE-REFERENCES, CACHE-MISSES OG MINOR FAULTS

Program som heter ‘perf’ brukes for å se nøyaktig hva som skjer når programmet brukes/kjøres.

```

hauerud@lap:~/mem$ gcc -O m.c
hauerud@lap:~/mem$ time ./a.out
Real:0,018 User:0,007 System:0,011 99,78%

```

Vi ser at vi starter med program 1 fra istad som skrives til ram med tallene rett etter hverandre.

Perf gir statistikk og info om et program og her er det bedt om eksplisitt om å skrive ut antall sykluser, antall instruksjoner, antall cache-referanser, cache-misses, major-faults er når man må ut på disken for å hente noe, minor-fault er når man ikke har lagd en side ute i MMU.

```

hauerud@lap:~/mem$ sudo perf stat -B -e cycles,instructions,cache-references,cache-misses,major-faults,minor-faults,context-switches ./a.out
[sudo] password for hauerud:

Performance counter stats for './a.out':

 63 888 751      cycles
 71 556 737      instructions      #   1,12  insn per cycle
 965 231      cache-references
 801 868      cache-misses      #  83,075 % of all cache refs
      0      major-faults
 10 046      minor-faults
      0      context-switches

 0,020898466 seconds time elapsed

```

En minor-fault oppstår når man ikke har referert til denne delen av minne og da lages det en adressering i MMU, og det gjøres 10K ganger.

Kompilerer igjen med endringen i programmet og kjører:

```

haugerud@lap:~/mem 37x26
F10 key ==> File Edit Mode Search
#include <stdio.h>

int array[1024000000];

void main(){
    int i,j;
    for(i = 0;i < 10240000;i++){
        j = i*100;
        array[j] = i;
    }
}

```

Det første vi ser er at det tar mye lengre tid å kjøre. Vi har mange flere cache-referanser og vi ser at nesten alle bommer. Vi får 75 millioner cache-misses i stedet for 8 hundretusen. Det er først og fremst det som gjør at det tar mye lengre tid, men det tar også tid å bygge opp MMU. Arrayet har 4 gigabyte og vi ser at i det første programmet trenger vi 10 tusen sider for å adresse lista mens i det andre programmet kreves det 1 million sider. Og det tar også lang tid, altså å bygge opp MMU.

```

haugerud@lap:~/mem$ sudo perf stat -B -e cycles,instructions,cache-references,cache-misses,major-faults,minor-faults,context-switches ./a.out

Performance counter stats for './a.out':
      4 957 439 408      cycles
      3 067 816 200    instructions          #   0,62  insn per cycle
      83 087 961  cache-references
      75 954 541  cache-misses          #   91,415 % of all cache refs
          0  major-faults
      1 000 045  minor-faults
          3  context-switches

 1,636458184 seconds time elapsed

```

Ved time ser vi at ved system tar det også veldig lang tid å bygge opp disse sidene, at det ikke bare er cache som bruker mye tid.

```

haugerud@lap:~/mem$ time ./a.out
Real:1,596 User:0,457 System:1,089 96,84%

```

Men hvorfor er det så mange cache misses i program 1? dette er fordi selv om det legges etter hverandre så går man veldig langt. Selv om man treffer veldig ofte, så er det med engang du kommer utenfor L1 cachen, så vil L1 cache kunne refreshes, og den vil refreshes 800 tusen ganger.

C-PROGRAM MED EN 20K*20K MATRISE, OMBYTTE AV INDEKS

GIR STOR FORSKJELL I TIDSBRUK, PERF

Dette arrayet som skriver til et array. Det som er viktig er å vite hvordan et todimensjonalt array, eller en matrise, lagres i RAM. Inni i for-løkken ser du hvordan man vanligvis bruker å lage en matrise, for å lagre data. Når man har to indeks hvordan skal det da ligges i RAM? Under i det kommenterte feltet er det skrevet hvordan dette gjøres i C.

```

haugerud@lap:~/mem 65x26
F10 key ==> File Edit Mode Search Buffers Windows Sys
#include <stdio.h>

int array[20000][20000];
int main(){
    int a,b;
    for(a = 0;a < 20000;a++){
        for(b = 0;b < 20000;b++){
            array[b][a] = 5;
        }
    }
}

// array[0][0] ligger rett før array[0][1] og array[0][2]
// array[0][0] ligger langt unna array[1][0] og array[2][0]

```

Vi ser med engang til høyre at det kan bli noe trøbbel eller problemer med hensyn til cache. Hvis vi skriver det på denne måten med $A[0]$ først og løper gjennom 0, 1, 2, 3 sånn så hele den biten sendes ut med cache.

Koden under endres til at b og a bytter plass her slik:

```

for(b = 0,b < 20000,b+
    array[a][b] = 5;
}

```

Under kompileres og kjøres det:

```

haugerud@lap:~/mem$ gcc -O mat.c
haugerud@lap:~/mem$ time ./a.out
Real:0,759 User:0,324 System:0,435 99,96%

```

Nå bytter vi tilbake i koden til at b kommer først.

```

for(b = 0,b < 20000,b+
    array[b][a] = 5;
}

```

Kompilerer og kjører:

```
haugerud@lap:~/mem$ gcc -O mat.c
haugerud@lap:~/mem$ time ./a.out
Real:5,306 User:4,890 System:0,412 99,92%
```

Vi ser på tegningen at i det første programmet skrev vi til arrayet slik at hvert element skrives etter hverandre, pent og ryddig, og det kan sendes i hele bolker i cache og sendes av gårde. I neste forsøk gjorde vi noen enorme hopp, vi hoppet fra a[0] [0] også til a[1] [0], og da gjorde vi et svært hopp i RAM, som var på 20 tusen. Og det kan vi se hvis vi kjører perf på det andre forsøket (Array [b] [a]):

```
haugerud@lap:~/mem$ sudo perf stat -B -e cycles,instructions,cache-references,cache-misses,major-faults,minor-faults,context-switches ./a.out
Performance counter stats for './a.out':
      16 232 033 823      cycles
      2 781 069 662      instructions      #   0,17  insn per cycle
      2 364 359 035      cache-references
      696 445 374      cache-misses      # 29,456 % of all cache refs
          0      major-faults
      390 670      minor-faults
          8      context-switches
                                         I
      5,384235924 seconds time elapsed
```

Array [a] [b], som vi trodde var det beste forsøket:

```
haugerud@lap:~/mem$ sudo perf stat -B -e cycles,instructions,cache-references,cache-misses,major-faults,minor-faults,context-switches ./a.out
Performance counter stats for './a.out':
      2 317 578 644      cycles
      2 777 653 530      instructions      #   1,20  insn per cycle
      33 684 909      cache-references
      29 926 414      cache-misses      # 88,842 % of all cache refs
          0      major-faults
      390 673      minor-faults
          2      context-switches
                                         I
      0,767158488 seconds time elapsed
```

Vi ser at minor-faults ble det samme fordi i dette tilfellet bruker vi jo like stor del av RAM, antall pages. Cache-misses er veldig ulikt. I det ene tilfellet måtte vi hoppe i RAM som tok lengre tid og i det andre tilfellet var alt ryddig og fint så cache ble benyttet bedre. Her handler det om å ta feil indeks i matrisen som kan ta lang tid.

NOEN VIKTIGE MINNE-BEGREPER

Soft miss = page referanse er ikke i TLB; må hentes fra internminnet, også kalt TLB-miss

Hard miss = page fault. En page mangler i minnet (og i TLB); må hentes fra disk

Major fault = page fault. En page mangler i minnet (og TLB); må hentes fra disk

Minor fault = en page mangler i page-tabellen i RAM og må lages. Må ikke hentes fra disk.

Dirty page = en side som har blitt endret slik at den må skrives til disk om den må ut av minnet

Working set (Windows) = det sett av sider som en prosess har brukt nylig. Samme som RES i Linux.

Segment = en logisk del av et programs minne, data programtekst, stack-segmenter

Buffer cache = del av minnet som brukes som filsystem-cache

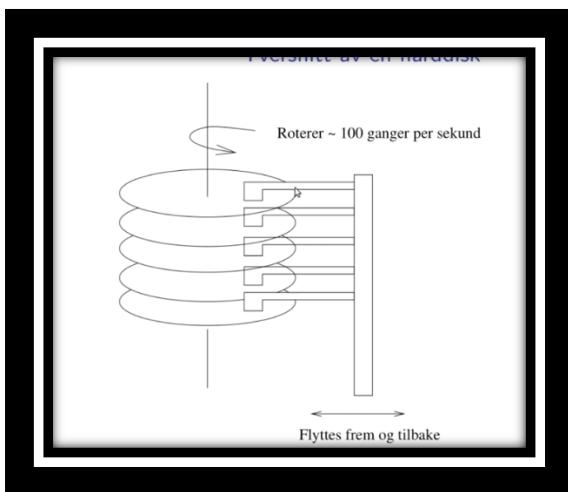
HVA ER FILSYSTEM-CACHE?

Filsystemet er en type cache som brukes av operativsystemet for å lagre nylige brukte data fra harddisken eller andre lagringsenheter midlertidig i RAM. Formålet er å redusere antall lesinger og skrivninger som må gjøres til lagringsenheten, som forbedrer ytelse.

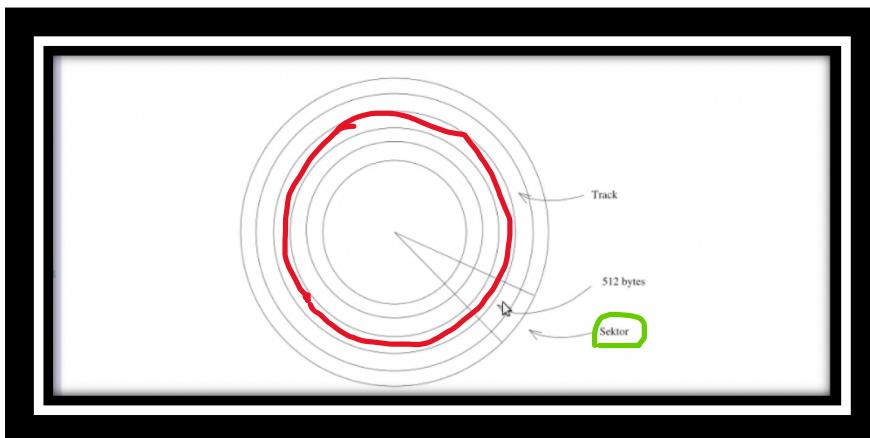
LINUX

MAGNETISKE DISKER

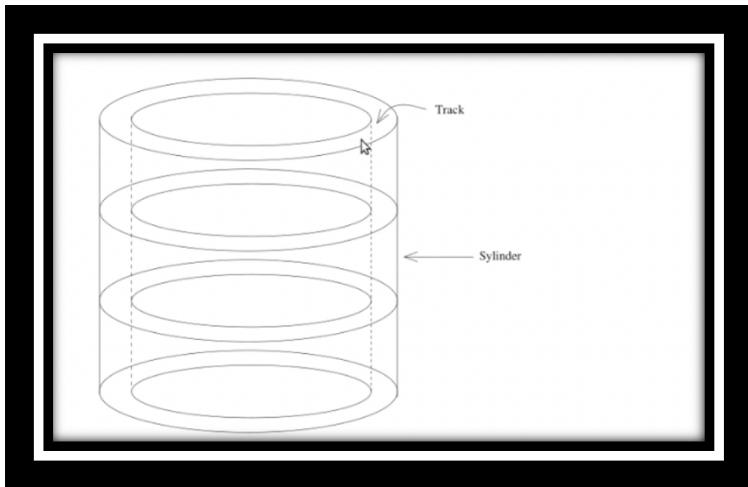
Ser ut som små snurreplater og snurrer veldig fort rundt, omtrent tusen ganger i sekundet. Det ligger lesehoder som er veldig tett intill diskene. Prinsippet er bygd på magnetisering hvor man har magnetisert områder på disken, hvor typisk et område som er magnetisert er en 1, og det som ikke er vil være 0. dette er en varig lagring, så uten strømtilførsel ligger det fremdeles der. Men for å lese av dette trenger man de lesehodene veldig tett inntil. De ligger både over og under sånn at man lagrer på begge sider av diskene, og typisk så har man mange lag med disker:



Sektor er den minste grunnenheten for en disk. Hvis man ser på bildet er det en sektor på 512 byte, og når man leser på disk leser man ikke en byte, men 512 byte av gangen da. En track er en av de sirklene (rød).



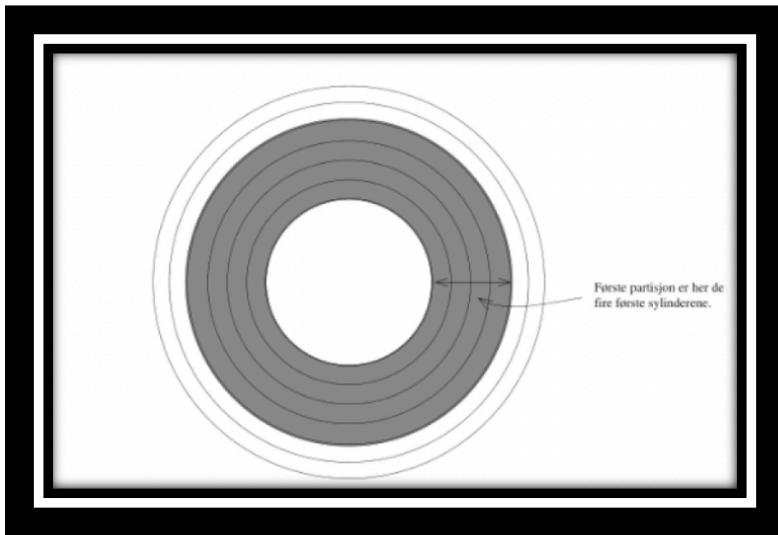
SSD disker er som RAM, de snurrer ikke rundt. En sylinder er samlingen av alle tracks fra alle platene i disken som ligger i samme avstand fra sentrum.



Generelt når man skal be om en liten sektor som grønn i to bilder opp, kan man gjøre det ved å si hvilke [lesehode, track, sektornummer] det er. Man kan slik be om alle sektorene en fil utgjør. Når os/diskcontroller vil lese noe fra disk, sendes en forespørsel med disse tre tallene.

PARTISJONER

En partisjon består av et antall sylinder som ligger etter hverandre.

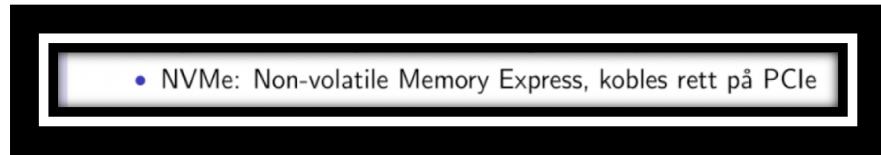


SOLID STATE DRIVE (SSD)

Helt annen type teknologi enn disker som snurrer med magnetisk lagrer. Det er basert på flash-minne som i minnepinner og har ingen bevegelige deler. Det likner mest på en liten lagringsenhet for RAM. Og i RAM hadde vi en liten transistor som holder ladning (med ladning = 1, uten = 0). den må refreshes, altså fylles opp på nytt (10 ganger i sekunder). Disse har en god del flere fordeler enn en vanlig harddisk. Med tanke på bærbarhet spesielt. I tillegg er disse mye raskere enn vanlige snurrende harddisker. De har random access tid mot 0,1

millisekunder omrent i forhold til roterende harddisker som har 1-5 ms. SSD er dyrere enn tradisjonelle disker å ha mindre plass.

NVMe:



PCI:



DEMO:

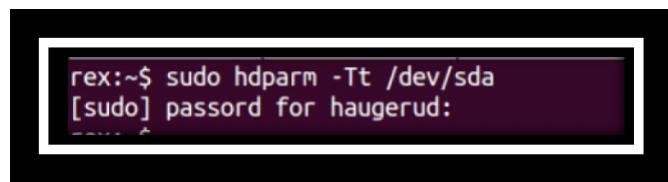
Kommandoen df -h gir en oversikt over disken og diskbruk. Det er en del /dev/loop her som typisk er lagret i RAM. Det som er selve harddisken, er feltet markert i rødt. Det er bare en partisjon, hadde det vært flere hadde det da vært /dev/sda2 og /dev/sda3 osv.

```

rex:~$ df -h
Filesystem      Størrelse Brukt Tilgj. Bruk% Montert på
udev            1,6G    0   1,6G  0% /dev
tmpfs           1,6G  178M  1,4G  12% /run
/dev/sda1        278G  43G  222G  17% /
tmpfs           7,9G  588K  7,9G  1% /dev/shm
tmpfs           5,8M  4,8K  5,8M  1% /run/lock
tmpfs           7,9G    0   7,9G  0% /sys/fs/cgroup
/dev/loop0       202M  202M    0  100% /snap/hiru/53
/dev/loop3       21M   21M    0  100% /snap/ubuntu-make/778
/dev/loop1       357M  357M    0  100% /snap/pycharm-educational/28
/dev/loop2       21M   21M    0  100% /snap/ubuntu-make/796
/dev/loop7       357M  357M    0  100% /snap/pycharm-educational/30
/dev/loop4       201M  201M    0  100% /snap/hiru/55
/dev/loop9       202M  202M    0  100% /snap/hiru/56
nexus.cs.hloa.no:/iu/nexus/ua/haugerud 493G  400G  78G  86% /home/haugerud
tmpfs           1,6G  180K  1,6G  1% /run/user/285
tmpfs           1,6G  32K  1,6G  1% /run/user/188
/dev/loop12      56M   56M    0  100% /snap/core18/1988
/dev/loop8       100M  100M    0  100% /snap/core/10908
/dev/loop5       56M   56M    0  100% /snap/core18/1997
/dev/loop10      100M  100M    0  100% /snap/core/10958

```

Starter med å kjøre en kommando HDPARM på en Linux desktop (med standard harddisk).



Swap er stedet på disken som brukes til virtuelt minne. Swapping går sakte, så på nye moderne systemer har man så mye RAM, at man aldri bruker swap. Bildet under viser swap som er sda5.

```
Command (m for help): p
Disk /dev/sda: 298,1 GiB, 320072933376 bytes, 625142448 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xcb46d2fa

Device      Boot   Start     End   Sectors  Size Id Type
/dev/sda1    *       2048 591679487 591677440 282,1G 83 Linux
/dev/sda2          591681534 625141759 33468226   16G  5 Extended
/dev/sda5          591681536 625141759 33468224   16G 82 Linux swap / Solaris
```

Hårek viser så bare hvor lang tid det tar å lese på disk.

Vi skal under se på hastigheten og da brukes det kommando hdparm på /dev/sda for å teste hvor lang tid det tar å lese disken. Det står først **timing cached reads**, og i praksis er det å lese fra disk cache. Og vi ser det er 13 tusen MG per sekund, 13 GB per sekund. Det går ekstremt fort og likner mer på RAM hastighet. Timing buffered disk reads som er disk hastigheten, og den leser direkte fra disk, og der er det nedi 100 MG per sekund. Det er ganske fort det også men ikke i nærheten av RAM.

```
rex:~$ sudo hdparm -Tt /dev/sda

/dev/sda:
Timing cached reads: 26186 MB in 1.99 seconds = 13155.83 MB/sec
Timing buffered disk reads: 238 MB in 3.01 seconds = 79.18 MB/sec
```

Så skal vi se det tilsvarende som er på Hårek sin laptop. Vi starter med df, blar opp og ser at selve disken ligger i den markerte linja på bildet under. Det står at det er nvme som er raskere enn en standard harddisk.

```
haugerud@lap:~$ df
Filesystem      1K-blocks    Used Available Use% Mounted on
udev             16405984      0  16405984   0% /dev
tmpfs            3285736    2176  3283560   1% /run
/dev/nvme0n1p6  201906120 159128664 32498188  84% /
tmpfs            16428668     628  16428040   1% /dev/shm
```

Vi ser det går litt tregere og det kan skyldes at pc-en kjører et os og zoom. Men vi ser at lesing fra disk går mye forttere, 1800 MB per sekund.

```
haugerud@lap:~$ sudo hdparm -Tt /dev/nvme0n1
[sudo] password for haugerud:
I

/dev/nvme0n1:
Timing cached reads: 20448 MB in 1.99 seconds = 10284.20 MB/sec
Timing buffered disk reads: 5474 MB in 3.00 seconds = 1824.29 MB/sec
```

alle under ‘devices’ er partisjoner. Under ser vi også Linux filsystemet,

```
haugerud@lap:~$ sudo fdisk /dev/nvme0n1
Welcome to fdisk (util-linux 2.31.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

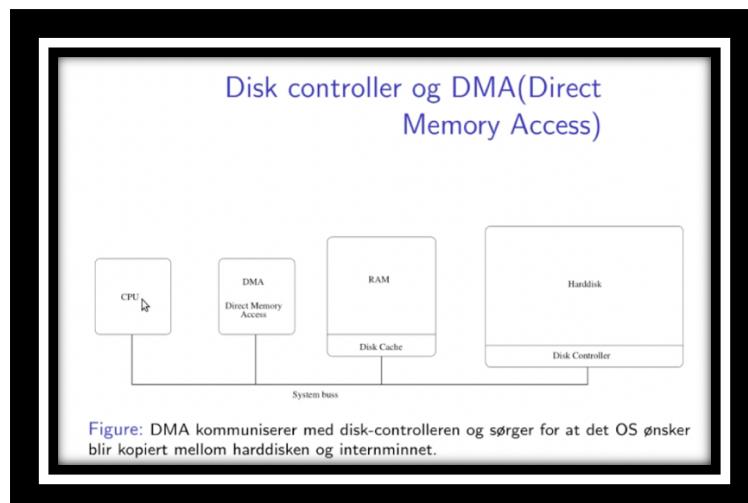
Command (m for help): p
Disk /dev/nvme0n1: 477 GiB, 512110190592 bytes, 1000215216 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 2AA4955E-5AD7-49DA-AEAB-5F8CB2BF4C52

Device      Start    End  Sectors  Size Type
/dev/nvme0n1p1       0  739327   739327  360M EFI System
/dev/nvme0n1p2  739328 1001471  262144 128M Microsoft reserved
/dev/nvme0n1p3 1001472 483962879 482961408 230,3G Microsoft basic data
/dev/nvme0n1p4 961341440 963348479 2007040 980M Windows recovery environment
/dev/nvme0n1p5 963348480 1000204287 36855808 17,6G Microsoft basic data
/dev/nvme0n1p6 483962880 894478335 410515456 195,8G Linux filesystem
/dev/nvme0n1p7 894478336 961341439 66863104 31,9G Linux swap

Partition table entries are not in disk order.
```

Hårek nevnte i starten at man har en diskcontroller som styrer harddisk. Hvis vi med bilder under tenker oss at CPU-en er os som kjører og gjør instruksjoner, så var det tidligere sånn at helt fra CPU-en fra os så kom det beskjed ‘hent sektor nr noe, track og plate nr 4’. men den detaljstyringen styres nå av diskkontroller som er en egen liten del.

DMA (direct memory access) gjør at os kan overlate del av oppgavene med å styre diskkontrolleren til DMA.



Tidligere detaljstyrte os harddisken. På den måten at den ba eksplisitt om lesehode, track og sektornummer. Os styrte algoritmene som leser fra disken. Det finnes flere typet algoritmer og en av er heisalgoritmen: den fungerer sånn at typisk trenger os liste med sektorer. Selv om man har en fil, kan den ligge rundt omkring i flere sektorer rundt omkring på disken. Da vil man trenge en algoritme som på smartest mulig måte kan hente ut det. En måte kan være at da lesehode flytter seg innover og utover så plukker den opp alt samtidig. Tidligere var alt dette styrt av os, men nå er det disk controlleren. La oss si at os vil ha en viss del fra harddisken, i stedet for å direkte snakke med harddisken så ber den DMA ta seg av det og sette det i RAM. Og når det er ferdig i RAM, så sendes det et interrupt som sier at 'den I/O jobben, er ferdig', i dette tilfellet disklesing. Så kan da os komme inn og fortsette med det den skulle.

FILSYSTEMER

I utgangspunktet når en ny disk kommer fra fabrikken så er den lavnivå-formatert. Dette betyr at disken er delt inn i 512 bytes, det kalles da en sektor, som er den minste lese og skriveenheten. Så når en fil har skrevet i en bokstav i seg, så hadde den tatt 512 bytes. Man snakker ofte om å formatere disker: og da snakker vi om fra scratch da ingen av sektorene er i bruk.

Et filsystem er et system som holder orden på hvor alle de sektorene ligger på disken. Før disken kan brukes må det lages et filsystem på den. FAT, NTFS, ReFS er for Windows. FAT er det gamle som fortsatt brukes i minnepinner, og FAT er bare en tabell som forteller hvilke sektorer som utgjør hvilken fil. ReFS (resiliant fil-system) er det nyeste. På Linux bruker vi ext3, hvor ext4 er den nyeste av de nye. ISO 9660 er filsystem som brukes i CD-er og DVD-er. Filsystemet fordeler mapper og filer på diskens sektorer, og holder orden på hvor alt ligger. Så man kan f eks si

Cat fil.txt

Også kommer den pent ut i riktig rekkefølge.

Vi skal se at filsystemer deles inn i større blokker enn sektorer, blocks, clustere eller fragments som det brukes i autopsy.

- Deler disken inn i større blokker (Linux: blocks, Windows: clustere).

Størrelsen på blokkene må bestemmes når filsystemet lages.

FORDELER OG ULEMPER MED STORE OG SMÅ BLOKKER

- Store blokker
 - Lese og skrive går hurtig, større sammenhengende områder
 - En liten fil vil bruke unødvendig mye plass
 - Bra til store filer, bilder og video
- Små blokker
 - Små filer bruker mindre diskplass
 - Større filer kan risikere å bli spredt rundt på disken
 - Lese og skrive store filer går da saktere
 - Bra hvis filsystemet skal inneholde mange små filer

Under er det forsøkt å illustrere et filsystem. I HDD vil det være mer random hvor i SSD er det rett etter hverandre. Når man sletter filer så kan fortsatt data ligge der, men pilene eller linkene som peker til der data ligger, de forsvinner. Data slettes først når det blir overskrevet, da er det tapt for godt. Disse blir markert som ledige så når det kommer mer data så kan de skrives inn i de hullene. Når det er masse sågne hull kalles det fragmentering. På Windows har man defragmentering som gjør at alle filene legges etter hverandre. Det er fra FAT infoen slettes. Da slettes all info om hvor blokkene ligger.

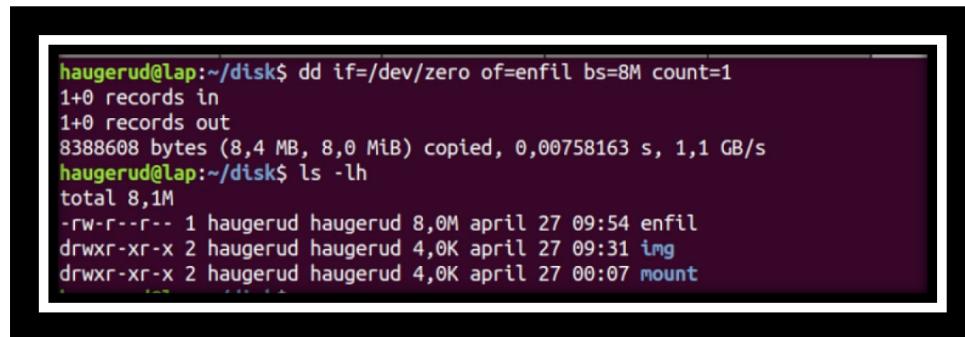
Tabell over filenes blokker								
Blokke størrelse 2 KByte			1	2	3	4	5	6
FIL	BLOKKER							
Fil1 6.1 KByte		1,2,3						
Fil2 3.9 KByte		4, 5						
Fil3 5KByte		6,7,8						
3KB ekstra til Fil 1		9,10						
Fil4 15KByte		11–18						
5KB ekstra til Fil 1		19,20						
12KB ekstra til Fil 2		21–26						
20KB ekstra til Fil 3		27–36						

Figure: Filsystemet holder oversikt over hvilke blokker en fil består av.
 Blokkstørrelsen er 2KByte i dette eksempelet. Bare hele blokker kan allokeres til en fil, slik at all plassen ikke utnyttes når filstørrelsen ikke eksakt går opp når man deler på blokkstørrelsen.

Kommando for hvordan man lager en fil med data. En måte å gjøre det på er ved å bruke kommando if, da sier man hvilke data man skal bruke, men andre ulike opsjoner:

```
dd if=/dev/zero of=enfil bs=8M count=1
```

if forteller hvilken data vi skal bruke, vi bruker /dev/zero, da bruker vi 0-ere. Of forteller hva vi kaller filen, og med bs sier vi hvor stor filen er. Count 1 sier vi har en fil som har 8M hvor resten er fylt med 0.



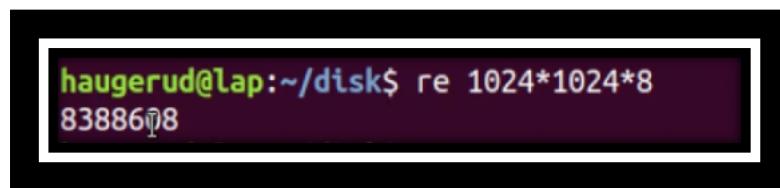
```
haugerud@lap:~/disk$ dd if=/dev/zero of=enfil bs=8M count=1
1+0 records in
1+0 records out
8388608 bytes (8,4 MB, 8,0 MiB) copied, 0,00758163 s, 1,1 GB/s
haugerud@lap:~/disk$ ls -lh
total 8,1M
-rw-r--r-- 1 haugerud haugerud 8,0M april 27 09:54 enfil
drwxr-xr-x 2 haugerud haugerud 4,0K april 27 09:31 img
drwxr-xr-x 2 haugerud haugerud 4,0K april 27 00:07 mount
```

Vi ser under at det er 8 mega.



```
haugerud@lap:~/disk$ ls -l enfil
-rw-r--r-- 1 haugerud haugerud 8388608 april 27 09:54 enfil
```

En mega er 1024×1024 , og hvis vi ganger det med 8, så ser vi akkurat det som ble satt av til det over.



```
haugerud@lap:~/disk$ re 1024*1024*8
8388608
```

I utgangspunktet så er det en fil med som kan se ut til å se tom ut grunnet alle 0-erene.



```
haugerud@lap:~/disk$ head enfil
```

Men dette kan gjøres til en image ved hjelp av en kommando. Vanligvis bruker man mkfs (make file-system) direkte på harddisker og lager da filsystemet på en disk. Ved hjelp av -t opsjonen burde man også spesifisere hvilket filsystem man vil ha og her skriver Hårek ext3, og han vil lage det på enfil.

Her lages filsystemet med default 1k blokker. Og da er det 1892 blokker som er 8×1024 .

```
haugerud@lap:~/disk$ mkfs -t ext3 enfil
mke2fs 1.44.1 (24-Mar-2018)
Discarding device blocks: done
Creating filesystem with 8192 1k blocks and 2048 inodes
I
Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
```

Så skal vi prøve å montere eller plassere denne disken/imaget. Kommandoen for det er mount på enfil, og det kan være i hvilken som helst mappe på systemet, men det finnes en mappe på systemet /mnt. Kommandoen df viser hva som er montert.

```
haugerud@lap:~/disk$ sudo mount enfil /mnt
[sudo] password for haugerud:
haugerud@lap:~/disk$ df
```

Selve disken er her, dette er en fysisk SSD disk

```
haugerud@lap:~/disk$ df
Filesystem      1K-blocks    Used Available Use% Mounted on
udev            16405984      0  16405984   0% /dev
tmpfs           3285736   2188  3283548   1% /run
/dev/nvme0n1p6  201906120 159420596 32206256  84% /
```

Loop brukes for å montere image:

```
tmpfs           3285732      0  3285732   0% /run/user/0
/dev/loop16     6907       50    6448   1% /mnt
haugerud@lap:~/disk$
```

Nå som han har montert disken så kan man gå inn i den med cd mnt:

```
haugerud@lap:/mnt$ ls -l
total 12
drwx----- 2 root root 12288 april 27 09:56 lost+found
haugerud@lap:/mnt$
```

Under har han nå to filer hvor enfil.txt har enfil inni seg og tofil.txt har tofil inni seg.

```
root@lap:/mnt# ls -l
total 14
-rw-r--r-- 1 root root    6 april 27 09:59 enfil.txt
drwx----- 2 root root 12288 april 27 09:56 lost+found
-rw-r--r-- 1 root root    6 april 27 09:59 tofil.txt
```

Hver av disse to bruker bare en hel blokk. Og hvis størrelsen på filen økes vil det fortsettes å skrive utover blokka.

Her går han ut av disken og skal sletter den. Hvis man prøver å gå inn nå så ligger det ingenting der.

```
root@lap:/mnt# exit
haugerud@lap:/mnt$ cd
haugerud@lap:~$ sudo su
root@lap:/home/haugerud# umount /mnt/
```

Sync synkroniserer filsystemets skrivebuffer med lagringsenheten (disk). Dette sikrer at eventuelle endringer i skrivebufferen blir permanent lagret på disken. Når man skriver filer eller gjør endringer i filene på datamaskinen sin, blir disse endringene lagret først i en midlertidig skrivebuffer i RAM (minne).

Nå skal Hårek vise oss at når vi tar rm (remove) randomFil.txt så slettes ikke fila. Han går tilbake til brukeren sin og endrer til root:

```
root@lap:/mnt# exit
haugerud@lap:~$ sudo su
root@lap:/home/haugerud# u
```

Vi ser at det fremdeles eksisterer enfil-disk.

```
root@lap:/home/haugerud# umount /mnt
root@lap:/home/haugerud# cd disk
root@lap:/home/haugerud/disk# ls -l
total 1100
-rw-r--r-- 1 haugerud haugerud 8388608 april 27 10:02 enfil
drwxr-xr-x 2 haugerud haugerud    4096 april 27 09:31 img
drwxr-xr-x 2 haugerud haugerud    4096 april 27 00:07 mount
```

Videre bruker han autopsy verktøyet for å kunne gå og se at det som ble slettet er slettet.

NTFS

Windows NT File System er Windows NT/XP/7/8/10 sitt eget filsystem, men også FAT16 og FAT32 støttes. NTFS (New Technology File System) er et filsystem som brukes av operativsystemer som Windows. Det er utviklet for å håndtere lagring, organisering og tilgang til filer og mapper på datalagringsenheter som harddisker. NTFS støtter avanserte funksjoner som sikkerhetsrettigheter, kryptering, kompresjon, feiltoleranse og journaling. Det gir også mulighet for filsystemfragmentering, filattributter og metadata. NTFS er kjent for sin pålitelighet, ytelse og evne til å håndtere store datavolumer.

Windows NT File System er Windows NT/XP/7/8/10 sitt eget filsystem men også FAT16 og FAT32 støttes.

- Deler inn diskens i clustere
- Clusterstørrelse på 512 bytes, 1 KiB, 2 KiB, 4 KiB og opp til maks 64 KiB
- 4 KiB clustere er default for diskene på 2GiB eller mer
- Clusterne adresseres med 64 bits pekere
- Komprimering
- Clusterstørrelse på mer enn 4 KiB kan ikke komprimeres og brukes vanligvis ikke
- Kryptering
- Alle endringer i filsystemet logges (men ikke endringer av data)
- Raskt å rekonstruere filsystemet ved disk-crash

RAID (Redundant array of independent disks)

Redundant betyr overflødig eller i overflod, generelt kan det her bety at man lagrer mer enn et sted, og det er typisk for å lagre det i parallel, så man kan lese det hurtigere. Da er det mange ulike nivåer, og RAID er først og fremst for å øke hastigheten, fordi diskene er trege.

RAID 0 = er egentlig ikke raid, fordi det er ikke redundans. Man stripere diskene, som betyr å ta en fil og skrive den over to diskene. Det dobler hastigheten.

RAID 1 = ekte raids. Da har man to diskene og dupliserer man dataene, så det skrives på begge diskene. Det tar like lang tid å skrive, men det går forttere å lese, fordi da kan du lese en setning fra hver.

RAID 3 OG 4 = der bruker man i tillegg paritet. De er ikke så mye brukt.

RAID 5 = der har man minst tre diskene og pariteten lagres fordelt på diskene. Paritet er om han har odde eller likt antall enere. Om man har tre enere så har man paritet lik 1, om man har 4 enere har man paritet 0. Og det brukes også i mange andre sammenhenger også for å sjekke om bit har endret seg underveis. Man har fordelt dataene på de tre diskene, og pariteten lagres også fordelt på de tre diskene. Og det betyr at hvis du har en disk som ryker så vil man kunne gjenopprette alt som var på de tre diskene ut ifra de to andre diskene.

RAID (Redundant Array of Independent Disks)	
RAID 0	Minst to diskene. Stripet diskene. Ingen redundans. Hurtigere å lese.
RAID 1	Minst to diskene. Dupliserer dataene. Hurtigere å lese. Kan fortsatt lese alt om en disk ryker.
RAID 3	Minst tre diskene. Parallel aksess, veldig små stripere, ned til en byte. Paritet lagres på en ekstra disk. Om en disk ryker kan informasjonen hentes ut fra de som er igjen. Optimalt høy overføringshastighet, men kun en forespørsel kan behandles av gangen.
RAID 4	Minst tre diskene. Paritet lagres på en ekstra disk. Store stripere, sektor eller blocks. Om en disk ryker kan informasjonen hentes ut fra de som er igjen. Kan behandle flere forespørsler samtidig. Bra for servere som får mange forespørsler.
RAID 5	Minst tre diskene. Paritet lagres fordelt på diskene. Store stripere, sektor eller blocks. Om en disk ryker kan informasjonen hentes ut fra de som er igjen.

RAID 3 OG PARITET

Nå brukes det raid tre fordi så lagrer den all paritet på en disk (se feltet helt til høyre). Vi ser for oss at det til venstre er kolonner med data som vi ønsker å lagre, også er det spredt ute på diskene. Også går det like fort å lese det som om alt lå på disk en. Vi ser at paritetsdisken er enkel, i første linje er det like mange enere så pariteten er 0, i neste linje er det oddetall en-ere så det er paritet 1. Det er ofte hardware som gjør dette her. Det kule igjen er at uansett hvilke diskene som ryker så kan man hente igjen all info, eller tilbake den disken som røk. Hvis paritetsdisken ryker så går det også bra, da kan man bare regne den på nytt.

disk 1	disk 2	disk 3	disk4	paritets-disk
0	1	0	1	0
1	0	1	1	1
0	0	1	1	0
1	1	0	0	0
0	0	1	0	1
1	1	1	1	0

PARITET OG TROLLDOM

Under skal vi se for oss at disk 2 ble ødelagt.

disk 1	disk 2	disk 3	disk4	paritets-disk
0	1	0	1	0
1	0	1	1	1
0	0	1	1	0
1	1	0	0	0
0	0	1	0	1
1	1	1	1	0

Hvordan kan man nå trytte frem igjen dataene på den ødelagte disk2 og legge dem inn på en ny disk?

