Contents

1 Forelesning 15/1-24(2 timer). Introduksjon til operativsystemer 10

1.1 Forelesningsvideoer 10
1.2 Om kurset 11
1.3 Vesentlige mål for kurset 11
1.4 Pensumlitteratur 11
1.5 Eksamen 12
1.6 Obligatoriske gruppe-innleveringer 12
1.7 Obligatoriske individuelle innleveringer 12
1.8 Nyttige personer 12
1.9 Hva er et operativsystem (OS)? 12
1.10 Hvor stort er et operativsystem? 13
1.11 OS-definisjon 13
1.12 Prinsippskisse av Linux 13
1.13 Sentralt begrep: prosess 14
1.13.1 Linux prosesser 14
1.13.2 Windows XP prosesser 15
1.13.3 Definisjon av prosess 15
1.14 Abstraksjon og hierarkier 16
1.14.1 Linux-eksempel på hierarki 16
1.15 Datamaskinarkitektur 17
1.16 Digitalteknikk og konstruksjon av kretser 17
1.17 Logiske porter og binær logikk 18
1.17.1 AND 18
1.17.2 OR 19
1.17.3 NOT 20

2 Forelesning 22/1-24(2 timer). Transistorer, porter og krets som adderer tall 21

2.1 Sist 21
2.2 Forelesningsvideoer 21

2.3 Porter og transistorer 22
2.4 CMOS 23
2.5 Boolsk algebra 25
2.6 Fra sannhetstabell til logisk krets. 25
2.7 Forenkling av logiske uttrykk. 27
2.8 Hvordan kan man få en logisk krets til å addere? 28

3 Forelesning 29/1-24(2 timer). Vipper og registre, CPU-arkitektur 32

3.1 Forelesningsvideoer 32
3.2 Sist 33
3.3 ALU 33
3.4 Lagring av data: vipper og registre 34
3.4.1 Lagringsenhet for en bit, D-lås(D-latch) 35
3.4.2 Simulering av shift-register med studenter 37
3.4.3 Vipper(flip flops) 38
3.5 Tellere 40
3.6 CPU-arkitektur 41
3.7 Beregningsenheter 41
3.8 En simulering av en datamaskin 42
3.9 Fra høynivåkode til CPU-beregning 44
3.10 Løkker og branch control 45

4 Forelesning 5/2-24(2 timer). C, maskinkode og assembly 46

4.1 Forelesningsvideoer 46
4.2 Simulerings-CPU og RAM 47
4.3 C-programmering 48
4.3.1 hello.c 48
4.3.2 Et C-program som summerer 49
4.3.3 Kompilering av C-funksjoner 50
4.4 Assembly 51
4.4.1 Summerings-funksjonen skrevet i Assembly 51

4.5 Assembly-kode generert av en kompilator 53

5 Forelesning 12/2-24(2 timer). C, maskinkode og pipelining 55

5.1 Maskinkode optimalisert for å kjøre hurtigst mulig 55

5.2 En linje høynivåkode kan gi flere linjer maskininstruksjoner 57

5.3 If-test 59

5.4 Forenklinger ved CPU Simuleringen 60

5.4.1 CPU-løkke (hardware-nivå) 60

5.5 Pipelining 61

5.6 Pipelining 61

5.7 Intel mikroarkitekturer 61

5.8 Superscalar arkitektur 62

5.9 Superscalar arkitektur 62

5.10 Intel Core 2 63

6 Forelesning 26/2-24(2 timer). Branch prediction, Multitasking 63

6.1 Forelesningsvideoer 63

6.2 Sist 64

6.3 branch prediction 64

6.4 Meltdown 66

6.5 Viktig å huske fra datamaskinarkitektur 66

6.6 OS historie 67

6.6.1 Microsoft Desktop-OS 67

6.6.2 Microsoft Server-OS 67

6.6.3 Unix operativsystemer 68

6.6.4 Interrupts (avbrytelser) 68

6.7 Singletasking OS 68

6.7.1 Internminne-kart 69

6.8 Multitasking-OS 69

6.9 Multitasking 69

6.10 PCB -Process Control Block 70

6.11 Timesharing og Context Switch 70
6.12 Multitasking i praksis, CPU-intensive programmer 71
6.13 Multitasking eksempel 72
6.14 CPU-intensiv prosess på system med èn CPU 72

7 Forelesning 5/3-24(2 timer). Multitasking, cache, hyperthreading 75

7.1 CPU-intensiv prosess på Mac med to CPU'er 76
7.2 Fem CPU-intensive prosesser på host med 4 CPUer 76
7.3 Internminnet og Cache 78
7.4 Multitasking og Multiprocessing 79
7.5 Intel Core og AMD K10 79
7.6 Mikroarkitektur 80
7.7 Hyperthreading 83
7.7.1 Kjører en CPU med hyperthreading to prosesser reelt sett samtidig? 84
7.8 Hyperthreading med prosess som bruker mye RAM 85
7.9 Deaktivering av hyperthreading 86
7.10 RAM-prosess uten hyperthreading 88
7.11 Taskset 89

8 Forelesning 12/3-24(2 timer). Systemkall, Scheduling og vaffelrøre 91

8.1 Slides og opptak 91
8.1.1 Slides brukt i forelesningen 91
8.1.2 Vaffel-video 92
8.2 Sist 92
8.3 Hvorfor kan ikke en prosess bruke to CPU-er? 92
8.4 Samtidige prosesser 93
8.5 Prosessor modus 93
8.6 Hvordan kan OS effektivt kontrollere brukerprosesser? 94
8.7 Bruker/Priviligert minne-kart 94
8.8 Systemkall 94
8.9 Prioritet i Linux-scheduling 95

8.10 need resched 96

8.11 Simulering av hvordan man ved hjelp av et operativsystem kan holde forelesning og lage vaffelrøre samtidig 99

8.12 Dagens faktum: Linux 101

9 Forelesning 19/3-24(2 timer). Prosesser, OS-arkitektur 103

9.1 Video av forelesningen 103

9.1.1 Slides brukt i forelesningen 103

9.2 Sist 104

9.3 Systemkall og timer ticks 104

9.4 Prioritet 105

9.4.1 Scheduling-algoritmer 106

9.4.2 Linux-eksempel: nice 106

9.5 Prosess-prioritet i Windows 106

9.6 Prosessforløp 106

9.6.1 Sentrale schedulingbegreper 107

9.6.2 Prosessforløp-demo 107

9.7 Lage en ny prosess 107

9.7.1 Linux: fork() 107

9.7.2 Windows: CreateProcess 108

9.8 Avslutte prosesser 109

9.8.1 Signaler 109

9.8.2 Signaler og trap i bash-script 109

9.9 OS arkitektur 110

9.9.1 Linux arkitektur 110

9.9.2 Windows arkitektur 110

9.10 Design av systemkommandoer 110

9.10.1 Linux-eksempel: setuid-bit 111

9.11 Utbredelse av Operativsystemer 111

9.11.1 Desktop OS 112

9.11.2 Server OS 114

10 Forelesning 26/3-24(2 timer). Plattformavhengighet og Threads 116

10.1 Video av forelesningen . . . 116
10.2 Sist . . . 116
10.3 Å kjøre Java, C og bash-programmer under forskjellige OS . . . 117
10.3.1 Hello.java . . . 117
10.3.2 hello.c . . . 117
10.3.3 hello.bash . . . 123
10.4 Test av C, Java, Python og bash på 5 plattformer . . . 123
10.5 Threads (tråder) . . . 123
10.6 Definisjoner av threads . . . 124
10.7 Fordeler med threads . . . 125
10.8 Java-threads . . . 125
10.8.1 Prioritet . . . 125
10.8.2 Java på Linux . . . 125
10.8.3 Variabler . . . 126
10.9 Java thread eksempel: Calc.java . . . 126

11 Forelesning 2/4-24(2 timer). Java threads og synkronisering 128

11.1 Forelesningsvideoer . . . 128
11.2 Sist . . . 128
11.3 Mange samtidige Java-tråder . . . 129
11.4 Java threads-eksempel: Prioritet . . . 131
11.5 Prior.java kjørt på Linux . . . 132
11.6 Prior.java kjørt på Windows 10 . . . 133
11.7 Blokkerende systemkall . . . 134
11.8 Thread-modeller . . . 135
11.9 Synkronisering . . . 136
11.10Serialisering . . . 136
11.10.1Eksempel: To web-prosesser som skriver ut billetter . . . 136
11.10.2Eksempel: to prosesser som oppdaterer en felles variabel . . . 137

11.11 Kritisk avsnitt 138

11.12 Kritisk avsnitt: Java-eksempel 138

11.12.1 Årsaken: race conditions 140

11.13 Race condition med C, to pthreads og én instruksjon 140

12 Forelesning 9/4-24(2 timer). Mutex, Semaforer, Deadlock 143

12.1 Forelesningsvideoer 143

12.2 Sist 144

12.3 Mulige måter å takle kritiske avsnitt 145

12.3.1 Linux-eksempel 145

12.3.2 Windows-eksempel 145

12.4 Softwareløsning for P1/P2 med MUTEX 145

12.4.1 Software-mutex, forsøk 1 146

12.5 Hardware-støttet mutex 146

12.6 X86-instruksjonen lock 146

12.7 Semaforer 146

12.7.1 Implementasjon av semafor i OS 147

12.8 Bruk av semafor i kritisk avsnitt 147

12.9 Bruk av semafor til å synkronisere to prosesser 148

12.10 Tanenbaums bruk av semaforer 149

12.11 Låse-mekanismer brukt i Linux-kjernen 149

12.12 Monitorer og Java synkronisering 149

12.13 Message passing 151

12.14 Dining Philosophers Problem 151

12.15 Deadlock 151

12.16 Kriterier for at deadlock kan oppstå 152

12.17 Tråder i Python 153

13 Forelesning 9/4-24. Internminne 155

13.1 Forelesningsvideoer 155

13.2 Internminne 156

13.3 Virtuelt adresserom 157
13.4 Internminnet/RAM 157
13.5 C++ library 157
13.6 Layout av en prosess sitt adresserom/segmentation 158
13.7 Minneadressering og MMU 159
13.8 Eksempel på MMU-tabell 159
13.9 Paging 160
13.10 Pages 161
13.11 MMU eksempel med 4k page-størrelse 161
13.12 Paging og swapping 163
13.13 Page Table entry 163
13.14 TLB - Translation Lookaside Buffer 164
13.15 Typisk TLB ytelse 164
13.16 Internminnet og Cache 164
13.17 Paging-algoritmer 165

14 Forelesning 23/4-24. Internminne i praksis **166**

14.1 Forelesningsvideoer 166
14.2 Dynamisk allokering 167
14.3 VIRT, RES og SHR i top 167
14.4 Noen minne-begreper 169
14.5 RAM-test 169
14.6 free 170
14.7 top 170
14.8 Eksempler på minnebruk 170

15 Forelesning 23/4-24. Disker og filsystemer **171**

15.1 Forelesningsvideoer 171
15.2 Disker 171
15.2.1 Sektor 171
15.2.2 Sylinder 172

15.3 Partisjoner 172
15.4 SSD (Solid State Drive) 173
15.5 Filsystemer 174
15.5.1 Tabell over filenes blokker 174
15.5.2 Fragmentering 174
15.5.3 Sletting av filer 175
15.6 Lage et Linux ext3 filsystem 175
15.7 NTFS 176
15.7.1 Volum 177
15.7.2 Master File Table(MFT) 177
15.7.3 Linux-partisjoner 178
15.8 Windows-partisjoner 179
15.9 Disk controller og DMA 180
15.10 ATA/IDE, SATA og SCSI 181
15.10.1 ATA/IDE 181
15.10.2 SATA 181
15.10.3 SCSI 182
15.11 KiB, MiB og GiB 182
15.12 Sammenligning av overføringshastigheter på minne-enheter 185
15.12.1 Sammenligning av disker 186
15.13 RAID 186
15.13.1 Ytelse 187
15.13.2 Paritet 187
15.13.3 Eksempel på paritetsberegning 187

16 Forelesning 14/5-24(2 timer). Prøveeksamen 2022 **189**

16.1 Forelesningsvideoer 189

17 Forelesning 14/5-24(2 timer). Prøveeksamen 2023 **189**

17.1 Forelesningsvideoer 189

18 Forelesning 21/5-24(2 timer). Prøveeksamen 2024 **189**

18.1 Forelesningsvideoer 190

19 Forelesning 7/5-24(2 timer). Prøveeksamen 2025 (og 20 og 21) **190**

19.1 Prøveeksamen 2025 190

19.1.1 Forelesningsvideoer 190

19.2 Prøveeksamen 2020 190

19.2.1 Forelesningsvideoer 190

19.3 Prøveeksamen 2021 191

19.3.1 Forelesningsvideoer 191

1 Forelesning 15/1-24(2 timer). Introduksjon til operativsystemer

`Avsnitt fra Tanenbaum: 1.1, 1.2, 1.5.1`

Slides brukt i forelesningen[1]

1.1 Forelesningsvideoer

Uredigert opptak av hele første time av forelesningen (00:35:31)[2]

Uredigert opptak av hele andre time av forelesningen (00:43:06)[3]

Opptak av forelesningen inndelt etter tema:

os1del1.mp4[4] (06:10) Innledning om kurssider, forelesninger, obliger
os1del2.mp4[5] (01:52) Pensum og kompendier
os1del3.mp4[6] (01:36) Emneevaluering
os1del4.mp4[7] (00:47) Øvinger, studentassistenter og lab
os1del5.mp4[8] (01:58) Eksamen
os1del6.mp4[9] (06:03) Slides: Operativsystemer - motivasjon
os1del7.mp4[10] (02:21) Slides: Mer om kurset
os1del8.mp4[11] (01:43) Slides: Mål for kurset, pensumlitteratur

[1] https://os.cs.oslomet.no/os/osintro.pdf
[2] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1.mp4
[3] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1b.mp4
[4] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del1.mp4
[5] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del2.mp4
[6] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del3.mp4
[7] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del4.mp4
[8] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del5.mp4
[9] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del6.mp4
[10] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del7.mp4
[11] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del8.mp4

os1del9.mp4[12] (02:06) Slides: Eksamen og obliger
os1del10.mp4[13] (01:05) Slides: Nyttige personer
os1del11.mp4[14] (04:03) Slides: Hva er et operativsystem og hvor stort er det?
os1del12.mp4[15] (03:15) Slides: User mode og Kernel mode, OS forenkler
os1del13.mp4[16] (09:21) Slides: OS-definisjon, Prinsippskisse av Linux
os1del14.mp4[17] (02:08) Slides: Sentralt begrep: prosess
os1del15.mp4[18] (15:17) En filosofisk analogi: hvis koden er DNA, hva er prosessen?
os1del16.mp4[19] (03:10) Slides: Eksempler på prosesser, Linux og Windows, hierarkier
os1del17.mp4[20] (12:05) Slides: Datamaskinarkitektur og logiske kretser

1.2 Om kurset

- Kurset består av to relativt uavhengige deler
  1. Operativsystemer(OS)
  2. Praktisk bruk av operativsystemer (Linux/Windows/Docker)
- Foreleser: Hårek Haugerud, haugerud@oslomet.no, rom SG214, SG29
- Hver onsdag fra 8:30: Flipped classroom-forelesninger
- Videoer hvor alt fagstoff foreleses legges ut uken før
- All kursinfo: `https://www.cs.oslomet.no/~haugerud/os` og Canvas
- Viktig: Jobb med **oppgaver**!!
- grunnlag for valgfaget Nettverks- og systemadministrasjon
- grunnlag for OsloMet-mastergraden Cloud-based services and operations (tidligere Network and System administration)

1.3 Vesentlige mål for kurset

1. Lære å bruke kommandolinje, inkludert script (Mest Linux og Docker, noe Windows)
2. Lære hvordan en datamaskin virker på alle nivåer, fra transistorer og opp til operativsystemet

1.4 Pensumlitteratur

- To kompendier dekker pensum
  - os.pdf
  - linux.pdf
  - Versjonene fra 2024 ligger under filer på Canvas og under Forelesninger på kurs-siden

[12] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del9.mp4
[13] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del10.mp4
[14] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del11.mp4
[15] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del12.mp4
[16] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del13.mp4
[17] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del14.mp4
[18] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del15.mp4
[19] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del16.mp4
[20] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os1del17.mp4

    - Kompendiene blir fortløpende oppdatert i løpet av semesteret med årets endringer

- Anbefalt støtteliteratur: Tanenbaum, Andrew S.: Modern operating systems : Global edition, 4th edition, 2014
- Omfattende og dyr, men en meget god bok

1.5 Eksamen

- 3 timers skriftlig Inspera eksamen (teller 100%)
- Ingen hjelpemidler tillatt
- Linux kommandolinje tilgjengelig under eksamen i Silurveien

1.6 Obligatoriske gruppe-innleveringer

- Uke-oppgavene som er markert som obligatoriske for hver uke samles opp og leveres ved hver innlevering.
- Alle obliger MÅ være godkjent for å kunne melde seg opp til eksamen

1.7 Obligatoriske individuelle innleveringer

Individuelle Multiple Choice tester med tidsbegrensning

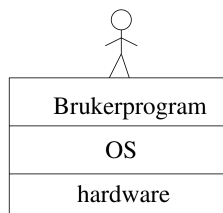
- Utgår trolig i 2025
- 3 korte Multiple Choice tester (7-10 minutter)
- Trekkes tilfeldig fra en database av spørsmål
- Må svare riktig på minst 7 av 10 for å få godkjenning
- Hvis ikke MÅ studentassistent kontaktes. Hen går igjennom svarene og anbefaler hva som bør jobbes med og oppdaterer databasen slik at du får en ny sjanse

1.8 Nyttige personer

- Foreleser
- Studentassistenter, i øvingstimene
- Hedda Marie Westlin, heddamar@oslomet.no, Linux drift (data2500-server, etc)

1.9 Hva er et operativsystem (OS)?

Et OS er et software-grensesnitt mellom brukeren og en datamaskins hardware.



1.10 Hvor stort er et operativsystem?

Kildekoden til et moderne operativsystem som Linux eller Windows er på omtrent fem millioner linjer kode. Det tilsvarer omtrent 100 Tanenbaum-bøker som hver er på 1000 sider med 50 linjer pr side.

5m

Så stor er alene kildekoden for selve operativsystemkjernen. Hvis man tar med GUI, biblioteker og annen nødvendig system software (som Windows explorer), blir størrelsen ti til tyve ganger større.

1.11 OS-definisjon

Forsøk på definisjon: OS er programvare hvis hensikt er:

A Gi applikasjonsprogrammer og brukere enhetlig, enklere og mer abstrakt adgang til maskinens ressurser

B Administrere ressursene slik at prosesser og brukere ikke ødelegger for hverandre når de skal aksessere samme ressurser.

Eksempler:

A filsystemet som gir brukerne adgang til logiske filer slik at brukerne slipper å spesifisere disk, sektor, sylinder, lesehode osv.

B Et system som sørger for at brukerne ikke skriver over hverandres filer; fordeling av CPU-tid.

1.12 Prinsippskisse av Linux

GNU er en rekursiv forkortelse (høy nerdefaktor) og står for 'GNU's not Unix'. Bakgrunnen for begrepet GNU/Linux er at GNU-prosjektet som ble startet av Richard Stallmann i 1983. GNU sto bak mange av de veldig viktige delene som ligger utenfor kjernen, som shellet bash og kompilatoren gcc. Operativsystemkjernen kan for eksempel ikke gjøre så mye fornuftig hvis man ikke kan kompilere programmer slik at de kan kjøres. Det er fortsatt noen som bruker begrepet GNU/Linux når de omtaler operativsystemet Linux, men det vanligste er å bare si Linux. Og bruken av GNU/Linux kan sammenlignes med distribusjoner av Linux som Ubuntu og Red Hat. Disse distribusjonene gjør også en rekke tilpasninger og lager systemprogrammer som skal gjøre det enklere, bedre og sikrere å bruke Linux-kjernen.

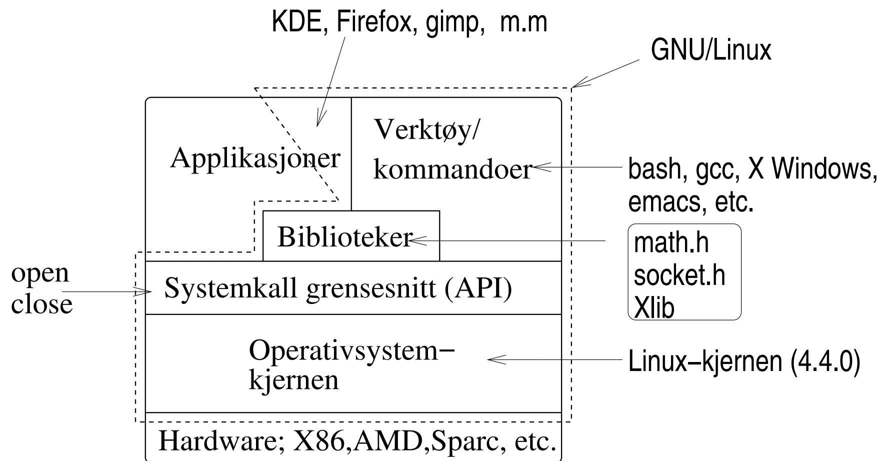


Figure 1: Prinsippskisse for et IT-system. GNU/Linux distribusjonen er markert med stiplede linjer. API = Application Programming Interface.

1.13 Sentralt begrep: prosess

Prosess er et svært viktig Operativsystem-begrep.

1.13.1 Linux prosesser

```
 21:49:07 up 7 days,  7:05,  2 users,  load average: 0.01, 0.02, 0.00
66 processes: 64 sleeping, 2 running, 0 zombie, 0 stopped
CPU states:   3.8% user,   2.4% system,   0.0% nice,  93.8% idle
Mem:    901440K total,   875496K used,    25944K free,    18884K buffers
Swap:   128516K total,     2252K used,   126264K free,   681000K cached

  PID USER      PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
17938 root       11 -10 93532  10M  2000 S <    2.3  1.2   0:31 XFree86
18958 haugerud  17   0  8800 8800  7488 R      2.3  0.9   0:01 kdeinit
18788 haugerud  11   0  3548 3548  2572 S      0.7  0.3   0:03 artsd
19272 haugerud  12   0   956  956   748 R      0.3  0.1   0:00 top
    1 root        8   0   484  456   424 S      0.0  0.0   0:00 init
    2 root        9   0     0    0     0 SW     0.0  0.0   0:00 keventd
    3 root       19  19     0    0     0 SWN    0.0  0.0   0:00 ksoftirqd_CPU0
    4 root        9   0     0    0     0 SW     0.0  0.0   0:29 kswapd
    5 root        9   0     0    0     0 SW     0.0  0.0   0:00 bdflush
    6 root        9   0     0    0     0 SW     0.0  0.0   0:19 kupdated
  123 daemon      9   0   432  428   356 S      0.0  0.0   0:00 portmap
  130 root        9   0     0    0     0 SW     0.0  0.0   0:01 rpciod
  131 root        9   0     0    0     0 SW     0.0  0.0   0:00 lockd
  196 root        9   0   872  868   724 S      0.0  0.0   0:02 syslogd
  199 root        9   0  1092 1088   420 S      0.0  0.1   0:00 klogd
  204 root        9   0   700  700   604 S      0.0  0.0   0:00 rpc.statd
  209 root        9   0   944  940   628 S      0.0  0.1   0:06 inetd
  293 root        9   0  2076 1860  1608 S      0.0  0.2   0:02 sendmail
  314 root        8   0  1280 1224  1068 S      0.0  0.1   0:00 sshd
  319 root        9   0  3028 2208   596 S      0.0  0.2   0:00 xfs
  321 root        9   0  1968 1968  1748 S      0.0  0.2   0:00 ntpd
```

1.13.2 Windows XP prosesser

```
PS C:\Documents and Settings\mroot> ps

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    105       5     1176       3616    32     0,07   1212 alg
    342       5     1512       3180    22    20,56    688 csrss
    118       4     1056       2808    21     0,90    972 csrss
    144       3     1812       3548    21     0,44   1132 csrss
     76       4     1004       3684    30     0,05    376 ctfmon
     72       4      964       3452    30     0,02    460 ctfmon
     86       2     1420       2200   413     0,02   2032 cygrunsrv
    157       4     1952       6180    44     0,07   1776 DW20
    352      10     8772      14460    85     0,66    520 explorer
    362      10     8036      14856    84     0,75   1864 explorer
      0       0        0         28     0               0 Idle
    164       6     3168       4724    38     2,65   1040 logonui
    389       9     3908       2284    41     0,38    768 lsass
    276       9    27568      25488   140     1,68   2132 powershell
     79       3     1196       3576    34     0,02    232 rdpclip
    106       4     1392       4384    35     0,03   1800 rdpclip
    154       5     4348       5884    56     0,08   2080 rundll32
    356       8     3328       5116    35     1,24    756 services
     40       2      400       1504    11     0,01    248 shutdownmon
     31       1      152        412     3     0,04    616 smss
    120       5     3148       4780    41     1,23   1468 spoolsv
     86      23     2092       3428   413     0,05    488 sshd
    263       6     2908       5452    61     0,11    924 svchost
    239      13     1724       4248    34     0,31   1080 svchost
   1561      62    15192      25432   140     5,15   1168 svchost
     76       3     1308       3584    29     1,11   1264 svchost
    161       5     1492       3912    34     1,23   1372 svchost
```

1.13.3 Definisjon av prosess

Alternative definisjoner:

1. Et program som kjører
2. Arbeidsoppgavene en prosessor gjør på et program
3. (a) Et kjørbart program
   (b) Programmets data (variabler, filer, etc.)
   (c) OS-kontekst (tilstand, prioritet, prosessor-registre, etc.)
4. Et programs ånd/sjel

   I en analogi hvor programmet som kjører er et menneskes DNA, vil prosessen være hele livet et menneske lever:

   Program = DNA

   Prosess = livet

   Hardware = Organer/Universet/hus/mat/bygninger

   OS = staten/lovverket

kill, Ctrl-C = drap
root/Administrator = Gud
CPU =Hjerne
kriminalitet = Black hat hacking

1.14 Abstraksjon og hierarkier

Disse begrepene som er illustrert i Figur 2 er generelt sett blant de viktigste innen all databehandling, og spesielt innen OS.

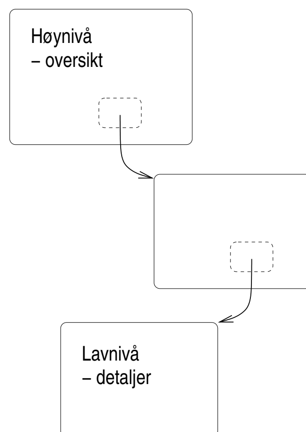


Figure 2: Abstraksjoner i et hierarki

1.14.1 Linux-eksempel på hierarki

Med verktøyet Bourne Again-shell (bash):

```
$ cat /etc/motd
```

Hjelpeprogrammet cat bruker flere systemkall for å skrive /etc/motd til skjermen. *Et C-program kan gjøre direkte kall til Linux-kjernen. F. eks. klargjør systemkallet open for å lese fra en fil:*

```
open("/etc/motd", O_RDONLY|O_LARGEFILE) = 3
```

Kommandoen cat består av en serie systemkall:

- open
- read
- close
- etc.

Hvilke systemkall et Linux-program gjør, kan man se med kommandoen `strace`:

```
$ strace cat /etc/motd
execve("/bin/cat", ["cat", "/etc/motd"], [/* 36 vars */]) = 0
uname({sys="Linux", node="rex", ...})   = 0
brk(0)                                  = 0x804d000
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40017000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=67455, ...}) = 0
old_mmap(NULL, 67455, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40018000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\360^\1"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1244688, ...}) = 0
old_mmap(NULL, 1254852, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40029000
old_mmap(0x40151000, 32768, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x127000) = 0x40151000
old_mmap(0x40159000, 9668, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40159000
close(3)                                = 0
munmap(0x40018000, 67455)               = 0
brk(0)                                  = 0x804d000
brk(0x806e000)                          = 0x806e000
brk(0)                                  = 0x806e000
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 25), ...}) = 0
open("/etc/motd", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=712, ...}) = 0
read(3, "Linux rex 2.6.1skas #3 SMP Mon A"..., 4096) = 712
write(1, "Linux rex 2.6.1skas #3 SMP Mon A"..., 712) = 712
read(3, "", 4096)                       = 0
close(3)                                = 0
close(1)                                = 0
exit_group(0)                           = ?
```

1.15 Datamaskinarkitektur

Operativsystemkjernen styrer maskinens hardware slik at programmene som skal kjøres kan få utført det de ønsker uten at de ødelegger for hverandre. For å forstå hvordan et operativsystem virker, må man derfor også ha noe forståelse for de grunnleggende delene av en datamaskin og ikke minst forstå datamaskinens hjerne, CPU-en.

1.16 Digitalteknikk og konstruksjon av kretser

Alle tall i en datamaskin er representert i det binære tallsystem med nuller og enere og fysisk representeres det med ingen eller positiv elektrisk spenning i forhold til jord. Ved å sette slike bit med verdi null eller en ved siden av hverandre, kan man velge å la dem representere binære tall. For eksempel vil 32 bit ved siden av hverandre kunne representere alle heltall fra 0 til $2^{32} - 1 = 4\ 294\ 967\ 295$. Når man har en slik definisjon av tall, kan man definere alt man trenger i en datamaskin, desimaltall, bokstaver (for eksempel ASCIII, 80 = P), eller pixler og farger i grafikk. Alt representeres med tall som igjen representeres binært med bit som er av eller på, representert ved 0 Volts spenning elle 5 Volts spenning. Et tall kan representeres med 4 bits som vist i figur 1.16.

For å kunne lage en datamaskin må man kunne utføre logiske og matematiske operasjoner på slike samlinger av bit. For eksempel må man kunne sammenligne, addere, subtrahere, multiplisere, dividere og gjøre shift-operasjoner. For å få til dette må man lage elektriske kretser som utfører denne type

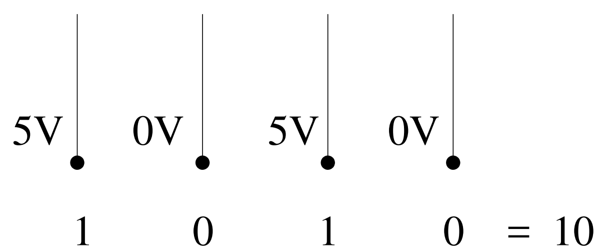


Figure 3: Et fire bits tall representer ved ulike spenningsforskjeller

operasjoner. Det må for eksempel være mulig å få en CPU til å legge sammen to tall og gi et resultat. Dette kan man få til ved digitale kretser også kalt logiske kretser. La oss først se på hva man trenger for å kunne addere sammen to tall.

1.17 Logiske porter og binær logikk

Alt CPU-en eller prosessoren i en datamaskin gjør er å manipulere på nuller og enere og ved hjelp av dette kan all verdens tenkelige beregninger utføres. Det teoretiske grunnlaget for manipulasjon av nuller og enere, binær logikk, ble utviklet på 1800-tallet. Ved hjelp av tre logiske operatorer, AND, OR og NOT, kan alle logiske beregninger utføres. Så ved å bygge disse tre logiske operatorene i hardware og i tillegg ha en enhet som kan lagre verdien 1 eller 0, har man alt som skal til for å bygge en datamaskin. En 0 blir representert ved ingen spenning og 1 ved for eksempel 5 Volts spenning. Så kan man ved hjelp av matematisk logikk konstruere en datamaskin som kan legge sammen, trekke fra og sammenligne binære tall og det er stort sett alt man trenger.

Den fysiske implementasjonen av AND, OR og NOT operatorene kalles porter, det kommer en eller to ledninger inn i den ene enden og går en ledning ut av den andre. Teoretisk fysikks store oppdagelse i det tyvende århundre, kvantemekanikken, la grunnen for halvlederteknologien og transistoren. I 1956 fikk Shockley, Bardeen og Brattain Nobelprisen i Fysikk for å konstruere transistoren som har gjort det mulig å lage slike porter ekstremt små. På en CPU-chip med noen få kvadratcentimeters overflate kan det være plass til 100 millioner slike porter og med ledninger mellom dem som er så tynne som 5 nanometer. Et hårstrå er til sammenligning 100 000 nm. De fysiske grenesene for hvor lite det er mulig å lage de integrert kretsene begynner å nærme seg og derfor klarer man ikke å øke klokkefrekvensen så hurtig som tidligere.

I Fig. 4 ser vi de tre grunnleggende portene og hvordan man kan sette dem sammen til mer kompliserte kretser.

Logikken til disse operatorene og tilsvarende porter kan beskrives ved såkalte sannhetstabeller, ved binær logikk og ved å tegne portene som vist i følgende avsnitt.

1.17.1 AND

Sannhetstabell:

A	B	Ut = $A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

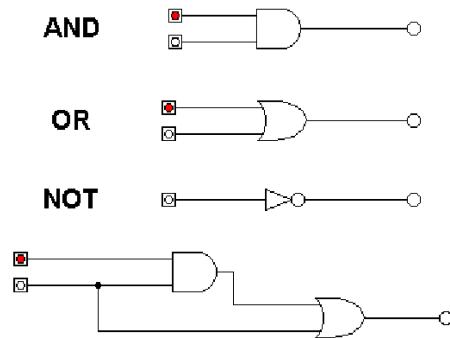


Figure 4: De tre grunnleggende logiske byggestenene i en prosessor, AND, OR og NOT-porter.

Logisk AND-operator:

$$
\begin{aligned}
0 \cdot 0 &= 0 \\
0 \cdot 1 &= 0 \\
1 \cdot 0 &= 0 \\
1 \cdot 1 &= 1
\end{aligned}
$$

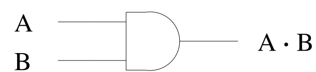


Figure 5: Tegning av AND-port

1.17.2 OR

Sannhetstabell:

A	B	Ut $= A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Logisk OR-operator:

$$
\begin{aligned}
0 + 0 &= 0 \\
0 + 1 &= 1 \\
1 + 0 &= 1
\end{aligned}
$$

$$1 + 1 = 1$$

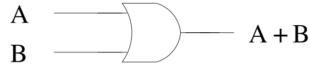


Figure 6: Tegning av OR-port

1.17.3 NOT

Sannhetstabell:

A	Ut $= \overline{A}$
0	1
1	0

Logisk NOT-operator:

$$\overline{0} = 1$$
$$\overline{1} = 0$$

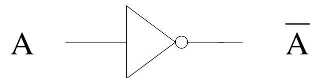


Figure 7: Tegning av NOT port

I tegningen av OR-porten, betyr det for eksempel at hvis det kommer en spenning 0 inn på inngang A og en spenning 1 inn på inngang B, vil det gå en spenning 1 ut. Slike porter kan man så sette sammen til kompliserte kretser. Det er for eksempel ikke veldig mange porter som skal til for å bygge en krets som kan legge sammen to firebits tall, slik CPU-en omtalt nedenfor kan. Alle tall i en datamaskin er representert i det binære tallsystem med nuller og enere, eller som vi har sett, med ingen eller positiv elektrisk spenning.

I figurene ser man hvordan det er vanlig å tegne AND, OR og NOT-porter i krets-diagram.

2 Forelesning 22/1-24(2 timer). Transistorer, porter og krets som adderer tall

`Avsnitt fra Tanenbaum: 1.3.1`

2.1 Sist

- Operativsystemkjernen
- Prosesser
- Logiske porter

Slides brukt i forelesningen[21]

2.2 Forelesningsvideoer

Uredigert opptak av hele forelesningen (01:34:15)[22]

Opptak av forelesningen inndelt etter tema:

os2del1.mp4[23] (04:12) Innledning om kurssider, digitale forelesninger, oppgaver
os2del2.mp4[24] (03:55) Mål for forelesningen: adderings-krets
os2del3.mp4[25] (08:12) Slides: Transistoren, Moores lov og Integrerte kretser
os2del4.mp4[26] (03:37) Slides: CMOS, NMOS og PMOS
os2del5.mp4[27] (05:06) Slides: Hvordan bygge NOT-port or OR-port av CMOS-transistorer
os2del6.mp4[28] (01:13) Slides: Binær funksjon
os2del7.mp4[29] (02:15) Slides: Kort repetisjon av logikken for AND, OR og NOT
os2del8.mp4[30] (06:47) Slides: Sannhetstabell, logisk uttrykk og logisk sum
os2del9.mp4[31] (02:17) Slides: Tegne krets av logisk uttrykk
os2del10.mp4[32] (02:21) Slides: Tegne og forenkle kretser
os2del11.mp4[33] (02:56) Spørsmål fra pausen: Hvorfor blir Y = 1 når X = 0 i NOT-porten?
os2del12.mp4[34] (04:59) Gjennomgang av 2 poll-spørsmål (litt lite bilde i opptaket) (Eksamensoppgave 2 og 3 fra Eksamen våren 2020)
os2del13.mp4[35] (00:50) Spørsmål: Er de to kretsene like? Begge = B
os2del14.mp4[36] (02:01) Slides: Forenkle kretser med Boolsk algebra
os2del15.mp4[37] (02:39) Binær addisjon

[21] https://www.cs.oslomet.no/ haugerud/os/2porter.pdf
[22] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2.mp4
[23] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del1.mp4
[24] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del2.mp4
[25] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del3.mp4
[26] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del4.mp4
[27] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del5.mp4
[28] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del6.mp4
[29] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del7.mp4
[30] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del8.mp4
[31] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del9.mp4
[32] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del10.mp4
[33] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del11.mp4
[34] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del12.mp4
[35] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del13.mp4
[36] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del14.mp4
[37] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del15.mp4

os2del16.mp4[38] (01:36) Addisjonskrets
os2del17.mp4[39] (02:40) Generell metode for å lage krets: Sannhetstabell-boolsk uttrykk-forenkling-tegne krets
os2del18.mp4[40] (03:28) Bruk av Full Adder for å legge sammen tall
os2del19.mp4[41] (02:46) Adderings-krets og andre nødvendige kretser
os2del20.mp4[42] (08:06) Demo: Hvordan lage og bruke kretser i Digital Works

2.3 Porter og transistorer

Ved hjelp av AND, OR og NOT-porter kan man helt generelt uttrykke alle mulige logiske sammenhenger. Fysisk sett er disse logiske portene bygd fra enda mindre byggestener, transistorer. Dette er selve grunn-byggestenen i en datamaskin på lignende måte som atomer er grunn-byggestenen i alle materialer. En transistor er egentlig bare en av/på bryter hvor en innkommende ledning er en bryter som avgjør om det ledes strøm eller ikke. Dette er omtrent som en vanlig elektrisk bryter på veggen, men i transistorens tilfelle styres bryteren av om det kommer strøm inn eller ikke. Slike transistorer kan man sette sammen i kretser og bygge logiske porter som igjen kan brukes til all den logikk en CPU trenger.

I de aller første datamaskiner ble radiorør brukt som slike brytere. I 1948 ble halvleder-transistoren oppfunnet av Bardeen, Brattain og Shockley, noe de fikk nobelprisen i fysikk for i 1956. Man kan argumentere for at dette er den viktigste enkeltstående oppfinnelsen noen sinne. Den største forskjellen fra radiorør er at transistorene kan lages ekstremt små; i 2017 klarte Intel og pakke mer enn 100 millioner transistorer på en kvadratmillimeter. Dette var med såkalt 10 nanometer teknologi hvor størrelsen på komponenter er helt nede i 10 nanometer (en nanometer er 10^{-9} meter). I 2020 begynte TSMC (Taiwan Semiconductor Manufacturing Company) produksjon av 5 nanometer silisium-brikker (chips). TSMC er verdens ledende halvleder-produsent og produserer for kunder som AMD og Apple og produserer også noe for Intel og andre firma som i hovedsak har egen produksjon.

I CPU-en Intel 4004 fra 1971 var det 2.300 transistorer og en CPU klokkefrekvens på 500 KHz, mens det i en Intel Xeon fra 2016 var 7.2 milliarder transistorer og en klokkefrekvens på 3 GHz. Mindre avstander gjør det mulig å øke klokkefrekvensen, men etter 2005 har den ikke økt fordi det genreres for mye varme om man gjør det. Frem til 2005 kunne man løse dette problemet med å redusere størrelsen, men på denne tiden begynte man å nå de fysiske grensene for hvor lite noe kunne lages, siden bredden på ledningene bare utgjorde noen titalls atomers bredde. I følge Moores lov så dobler antall transistorer i integrerte kretser seg hvert andre år og den loven har blitt fulgt ganske nøyaktig siden 70-tallet. Riktignok har det meste av ekstra transistorer i moderne CPUer blitt brukt til cache, hurtigminne i CPUene. Fig. 8 [43] viser eksempler på mikroprosessorer som følger Moores lov.

Og snart når man de fysiske yttergrensene. Bohr-radius, avstand fra kjerne til elektron i hydrogen, er 0.05 nanometer. Radius til et Silisium-atom er 0.11 nanometer. Komponentene har nå blitt så små at man må ta hensyn til fenomener som kvante-tunnelering og man kan ikke gå særlig mye lenger ned i størrelse på transistor-teknologien.

[38] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del16.mp4
[39] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del17.mp4
[40] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del18.mp4
[41] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del19.mp4
[42] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os2del20.mp4
[43] https://en.wikipedia.org/wiki/Moore%27s_law

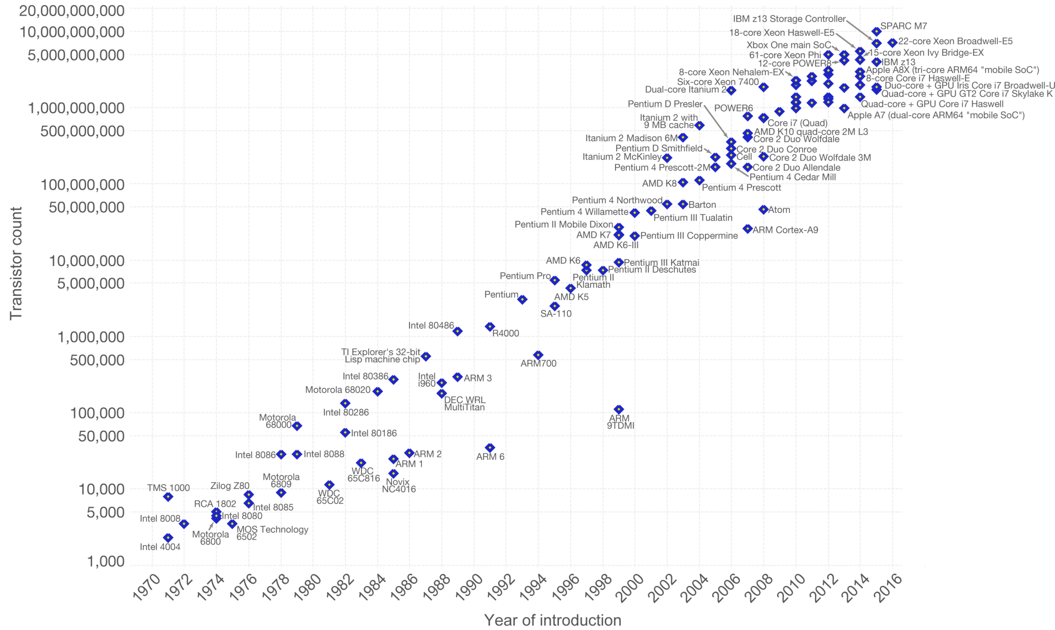


Figure 8: Moores lov: antall transistorer i integrerte kretser dobler seg hvert andre år. T

2.4 CMOS

CMOS er en teknologi for å lage integrerte kretser som brukes i alle mikroprosessorer. At den er komplementær betyr at den setter sammen to motsatte typer transistorer, NMOS og PMOS. En NMOS-transistor er egentlig en n-type metal oxide semiconductor field effect transistor (MOSFET). Dette er egentlig en ekstremt liten bryter, noen tiltalls nanometer stor. Hvis det er null spenning inn på transistoren, er de to andre utgangene isolert fra hverandre, bryteren er av. Hvis man sender positiv spenning inn skrus bryteren på og de to andre utgangene vil lede strøm, akkurat som når man trykker på en lysbryter. En n-type transistor er vist i Fig. 9.

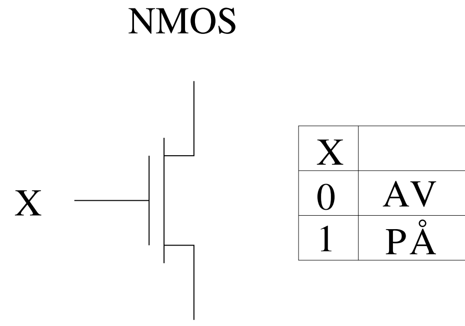


Figure 9: NMOS transistor. Når det ikke er spenning inn, er øvre og nedre del isolert, bryteren er av. Når X kobles til spenning går bryteren på og strøm ledes mellom nedre og øvre del.

Det at man kan lage så ekstremt små transistorer har gjort det mulig å legge milliarder av dem på samme

chip og lage enormt kraftige mikroprosessorer. En p-type transistor er veldig lik, men den virker helt motsatt, den er komplementær. En slik transistor er vist i Fig. 10.

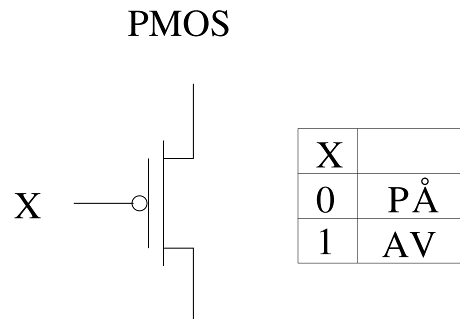


Figure 10: PMOS transistor. Virker motsatt av NMOS, når det er spenning inn, er øvre og nedre del isolert, bryteren er av.

Ved å sette sammen NMOS og PMOS sammen viste det seg at man unngår unødvendig strømføring i kretsene og at man dermed reduserer varmeproduksjonen som er et vesentlig problem for integrerte kretser. Man kan lage den enkleste logiske porten, NOT, ved å sette sammen en p-type og en n-type transistor som vist i Fig. 11.

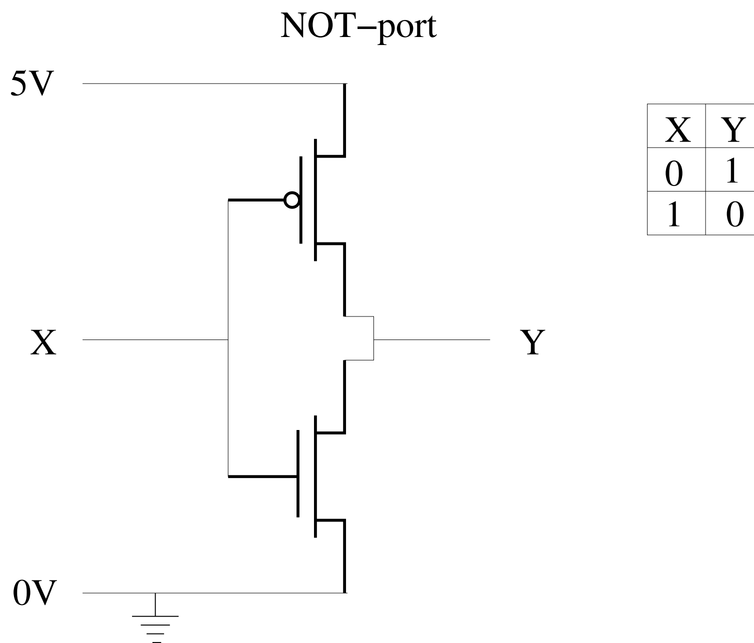


Figure 11: NOT-port. Når X kobles til positiv spenning, vil den øverste PMOS-transistoren isolere, mens den nederste NMOS-transistoren gir 0 spenning ved Y. Og det motsatte skjer når X kobles til jord.

Dermed har man en viktig byggestein. I figuren ser man også hvordan de forskjellig delene kobles sammen i en krets og kobles til spenning eller jord, null spenning. Hvis man kobler X-inngangen til NOT-porten til jord, null volt, vil man måle 5 volt ved utgangen Y. Så kan man bygge videre, ved for eksempel å sette en NOT-port til etter den første. Men hvis man ønsker å kunne lage alle mulige logiske binære funksjoner, trenger man også AND- og OR-porter. Det viser seg at man kan lage AND og OR-porter med 3 CMOS-par av transistorer og hvordan man kan lage OR er vist i Fig. 12. Omtrent tilsvarende kan man lage en AND-port.

Dermed kan man lage alle mulige logiske operasjoner bygget på transistorer ved å sette sammen systemer

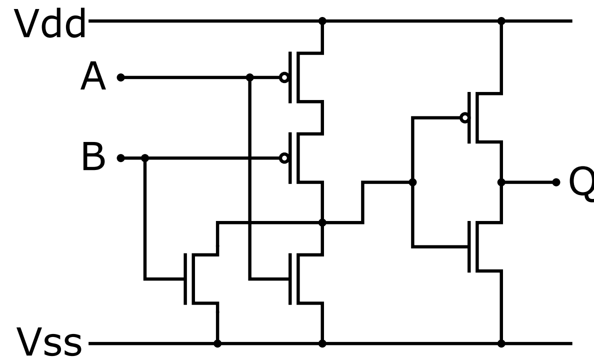


Figure 12: En OR-port kan lages av 6 transistorer. Vss er jord, null spenning, og Vdd er positiv spenning.

av NOT, AND og OR-porter. For å bygge en CPU som kan gjøre operasjoner som å legge sammen tall, trenger man generelt å lage generelle binære funksjoner som vist i Fig. 14.

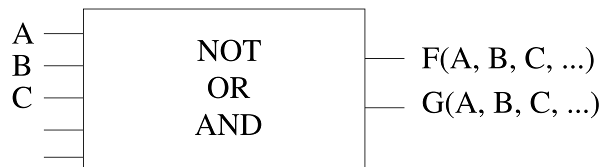


Figure 13: En generell binær funksjon.

Generelt kan en boolsk funksjon defineres ved en sannhetstabell som definerer hva output av funksjonen skal være for alle mulige kombinasjoner av binær input.

2.5 Boolsk algebra

I boolsk algebra uttrykkes AND, OR og NOT på følgende måte

- AND: $A \cdot B$ ($A \wedge B$)
- OR: $A + B$ ($A \vee B$)
- NOT: $\overline{A}$ ($\neg A$)

De boolske operatorene som står i parentes er slik de blir brukt i kurset Diskret matematikk. I digitalteknikk er det vanligst å bruke konvensjonen for boolske operatorer som brukes i dette kurset.

2.6 Fra sannhetstabell til logisk krets.

Når man skal lage en komponent av en CPU, må man spesifisere nøyaktig hvordan denne kretsen skal virke. Og generelt vet man nøyaktig hva man ønsker. Hvis en krets for eksempel skal sammenligne to bit for å se om de er like, vil disse to bit være input til kretsen. Output vil være 1 (sann) hvis de to innkommende bit er like og 0 (usann) ellers. Tilsvarende kan man for mer komplekse kretser alltid kunne

skrive ned hva som skal være output for hver eneste mulige input. På samme måte som for AND, OR og NOT-porter, kan ønsket om hvordan kretsen skal fungere formuleres som en såkalt sannhetstabell. Funksjonen F(A,B) til en krets som har som input A og B og som skal være 1 (sann) når A og B er like og 0 (usann) når A og B er forskjellige kan skrives slik:

A	B	F(A,B)
0	0	1
0	1	0
1	0	0
1	1	1

Ut ifra en slik sannhetstabell kan man alltid skrive ned et logisk uttrykk for funksjonen F(A,B). For å få til det, benytter man seg av egenskapene til AND og OR operatorene. Et produkt $A \cdot B$ vil kun være lik 1 hvis både A og B er lik 1, i alle andre tilfeller vil produket være lik 0. Hvis en linje i sannhetstabellen viser at F skal være lik 1, kan man derfor skrive ned et produkt som gir 1 når man ganger sammen (AND'er) nøyaktig de to verdiene i denne linjen av sannhetstabellen. For første linje i sannhetstabellen over, må man derfor skrive ned uttrykket

$$\overline{A} \cdot \overline{B} \tag{1}$$

som første ledd av uttrykket F(A,B), siden dette gir 1 for verdiene A = 0 og B = 0 i første linje og 0 i alle andre tilfeller ($\overline{A}$ betyr ikke A eller NOT A og er lik 1 når A = 0). Hvis man skriver ned tilsvarende uttrykk for alle linjer som gir 1 i sannhetstabellen, kan man til slutt legge sammen alle leddene med OR-operatoren. Dette blir da tilsammen et korrekt uttrykk for funksjonen F, fordi om minst ett av leddene i et OR-uttrykk er 1 vil det totale uttrykket også bli 1. Dermed vil en slik sum av produkter alltid gi den riktige verdien for funksjonen F. I sannhetstabellen for sammenligningskretsen, gir også den siste linjen 1. Denne linjen gir derfor uttrykket $A \cdot B$ som er 1 når A og B er 1 og ellers 0. Dermed kan det logiske uttrykket for funksjonen skrives ned som

$$F(A,B) = \overline{A} \cdot \overline{B} + A \cdot B \tag{2}$$

For å overbevise seg om at dette er riktig, kan man ganske enkelt teste at hvert ledd i sannhetstabellen er oppfylt. En stor fordel med logiske operatorer er at antall mulig input-verdier er begrenset, i motsetning til for kontinuerlige funksjoner.

Når man har klart å skrive ned et logisk uttrykk for sannhetstabellen, kan man ganske enkelt tegne et diagram for kretsen ved å erstatte operatorene AND, OR og NOT med de tilsvarende logiske portene. Dermed kan man tegne følgende krets for ligning Eq. 2:

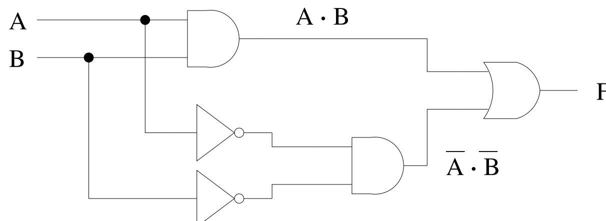


Figure 14: Tegning av den logiske kretsen $F(A,B) = A \cdot B + \overline{A} \cdot \overline{B}$.

Slike kretsdiagram kan brukes til å spesifisere enkeltdelene i en større krets som en CPU og til slutt kan tilsvarende deler lages i hardware. Først legges de logiske delene som skal med og så kobles de sammen;

'place and route' kalles generelt denne teknikken. Det finnes også programmeringsspråk som brukes til å definere kretser og verktøy som automatiserer deler av prosessen fra en logisk krets til en fysisk implementasjon som en integrert krets.

2.7 Forenkling av logiske uttrykk.

Man kan generelt skrive ned logiske uttrykk som beskrevet over fra en sannhetstabell, også om den har 3, 4 eller enda flere input. Men da blir de resulterende logiske kretsene generelt svært omfattende. Men ved hjelp av Boolsk algebra og andre metoder er det mulig å redusere størrelsen før kretsene lages. Konstruksjonen av følgende krets er et eksempel på dette. Anta at man ønsker å lage en krets som tilfredsstiller følgende sannhetstabell:

A	B	F(A,B)
0	0	0
0	1	1
1	0	0
1	1	1

Ved å følge metoden forklart i forrige avsnitt, kan man da skrive ned den logiske funksjonen for kretsen som en sum av de to leddene som gir verdien 1 for andre og fjerde linje, slik at sannhetstabellen oppfylles:

$$F(A,B) = \overline{A} \cdot B + A \cdot B \tag{3}$$

Dermed kan man tegne følgende krets for ligning Eq. 3:

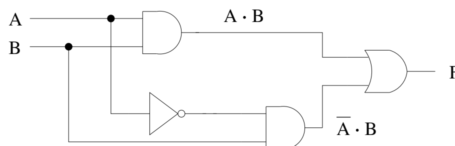


Figure 15: Tegning av den logiske kretsen $F(A,B) = A \cdot B + \overline{A} \cdot B$.

I motsetning til uttrykket i forrige avsnitt, kan dette uttrykket forenkles, slik at hele kretsen kan forenkles. Som i vanlig algebra, finnes det metoder for å faktorisere og forenkle uttrykk. Ved å bruke slike metoder kan dette uttrykket forenkles som følger:

$$F(A,B) = \overline{A} \cdot B + A \cdot B = (\overline{A} + A) \cdot B = 1 \cdot B = B \tag{4}$$

Her har vi brukt at at $\overline{A} + A = 1$ som gjelder kun for Boolsk algebre. Som i vanlig algebra kan man faktorisere og som i vanlig algebra gjelder $1 \cdot X = X$. I dette tilfellet er altså funksjonen F = B og kretsen blir ekstremt forenklet som vist i Fig. 16.

Man kan på samme måte lage boolske funksjoner og tegne kretser med utgangspunkt i en sannhetstabell når man har 3, 4 og også flere variabler. Med 3 og 4 variabler kan man bruke såkalte Karnaugh-diagram for systematisk forenkling, noe som er raskere og enklere enn å bruke Boolsk algebra direkte på utrykkene. Ofte er 4 variabler nok til å lage den enheten man ønsker og siden kan man bruke 'place and route' for å koble sammen alle enhetene til en fullstendig krets, slik som en CPU.

Figure 16: Ekstrem forenkling av kretsen til $F(A, B) = B$.

2.8 Hvordan kan man få en logisk krets til å addere?

I digital logikk er tall representert binært, som spenninger av og på. En måte å få en krets til å legge sammen binære tall på, er å la den gjennomføre operasjonene som man bruker når man legger sammen binære tall med penn og papir. Fig. 17 viser hvordan man legger sammen $1 + 3 = 4$ binært. Det rød ett-tallet over nullen er mente fra første operasjon der man gjør $1 + 1 = 10$ binært og får en i mente. Tilsvarende er det andre røde tallet mente fra neste operasjon.

$$\begin{array}{r} {}^{1}\ \ 0^{1}1 \\ +\ 1\ 1 \\ \hline 1\ 0\ 0 \end{array}$$

Figure 17: Binær addisjon som gir $1 + 3 = 4$

Nå vil det være mulig å lage en digital krets som utfører en enkelt av disse repeterende operasjonene som man gjør for å legge sammen to tall. Man tar med mente fra høyre, legger sammen med de to tallen som står under hverandre og det resulterer i ett binært siffer som settes under brøkstreken og eventuelt mente til neste operasjon. Det betyr at input for kretsen er mente fra forrige operasjon (som vi kaller Z) og de to tallene som skal adderes, dem kaller vi X og Y. Output fra kretsen er ett tall S som skal under brøkstreken og resulterende mente som vi kaller C (engelsk: carry). Operasjonen som skal utføres er vist i Fig 18.

$$\begin{array}{r} C\ \ X\ \ Z \\ +\ Y\ \ \ \\ \hline S\ \ \ \end{array}$$

Figure 18: En enkelt operasjon i binær addisjon som må repeteres for hvert siffer i de binære tallene.

Når man forstår algoritmen for å legge sammen to tall, er det rett frem å skrive ned sannhetstabellen for en krets som skal gjøre akkurat denne operasjonen, med input X, Y og Z og output S og C. Den blir som følger:

X	Y	Z	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

For eksempel i den siste linjen, når alle tre input er 1, forplanter mente seg videre til C = 1 og også S blir 1. Gitt en slik sannhetstabell kan man med metodene vist i de forrige avsnittene konstruere en logisk krets ved å

1. Skrive ned boolske uttrykk for funksjonene S(X,Y,Z) og C(X,Y,Z).
2. Forenkle uttrykkene med Boolsk algebra (eller Karnaugh-diagram).
3. Tegne en krets basert på det Boolske uttrykket som utfører nøyaktig denne operasjonen.

Punkt 1 vil være å ta utgangspunkt i de fire linjene i sannhetstabellen som gir 1 for funksjonene S og C. Dermed kan man skrive ned uttrykkene:

$$\begin{aligned} S(X,Y,Z) &= \overline{X}\cdot\overline{Y}\cdot Z + \overline{X}\cdot Y\cdot\overline{Z} + X\cdot\overline{Y}\cdot\overline{Z} + X\cdot Y\cdot Z \\ C(X,Y,Z) &= \overline{X}\cdot Y\cdot Z + X\cdot\overline{Y}\cdot Z + X\cdot Y\cdot\overline{Z} + X\cdot Y\cdot Z \end{aligned}$$

deretter kan uttrykkene forenkles. Det er ved hjelp av boolsk algebra (eller Karnaugh-diagram) mulig å forenkle C til

$$C(X,Y,Z) = X\cdot Y + Z\cdot Y + X\cdot Z$$

og dermed tegne kretsen som gir funksjonen C, som vist i Fig. 19. Det er uten for dette kursets pensum å utføre denne type litt mer kompliserte forenklinger, men det er viktig å vite at man systematisk kan utføre denne type forenklinger ved hjelp av boolsk algebra eller enda mer effektivt ved hjelp av Karnaugh-diagram.

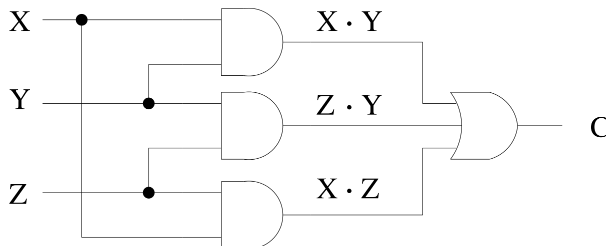


Figure 19: Krets som lager funksjonen $C(X,Y,Z) = X\cdot Y + Z\cdot Y + X\cdot Z$.

Uttrykket for S kan også forenkles noe. Deretter kan man tegne kretsene for disse uttrykkene sammen i en boks. I dette tilfellet kalles kretsen en Full Adder (FA) og en boks om gjør denne operasjonen (og hvor innholdet i kretsen med alle AND, OR og NOT-porter er skjult), kan se ut som i Fig. 20.

Ved å sette sammen to slike Full Adder kretser ved å koble carry fra den høyre boksen til mente i den venstre boksen, som vist i Fig. 21, vil vi få en krets som legger sammen to binære tall X_1X_0 og Y_1Y_0 og som gir som resultat det binære tallet $S_2S_1S_0$.

Kretsen med de to FA'ene utfører regnestykket som er vist i Fig. 22

For å addere tall med flere bit, kan man bare legge til flere slike bokser, en for hvert tall. Om man skjøter sammen 64 slike Full Adder bokser, får man en krets som legger sammen 64-bits tall. I Fig. 23 legger kretsen sammen to 3-bits tall.

Her har vi sett hvordan man kan lage en krets som legger sammen to binære tall. Tilsvarende kan man lage kretser som gjør alle andre operasjoner man trenger i en CPU, som å sammenligne, subtrahere, multiplisere, dividere og så videre. For x86 arkitekturen finnes det hundrevis av instruksjoner, noen

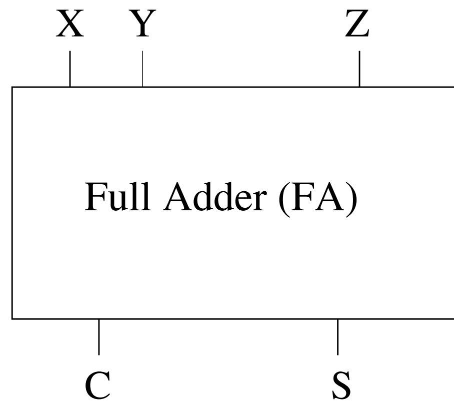


Figure 20: En Full Adder krets som oppfyller sannhetstabellen vist over.

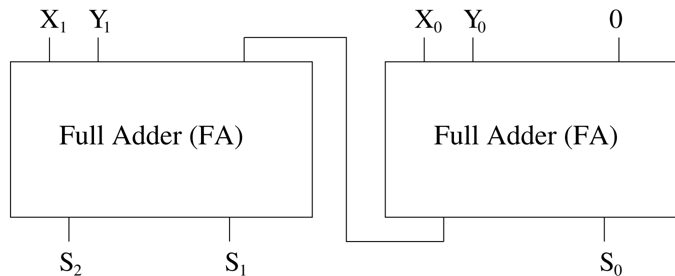


Figure 21: To Full Adder kretser skjøtet sammen slik at de regner ut det binære regnestykket $X_1X_0 + Y_1Y_0 = S_2S_1S_0$ korrekt for alle mulige verdier av X og Y.

krever kompliserte kretser andre er enklere. Ikke alle instruksjoner opererer på tall og data, noen gjør ikke annet enn å endre på verdien av ett enkelt bit. Og kretsene kan utnytte hverandre. Som et eksempel kan man klare å lage en instruksjon som subtraherer ved å kode negative tall med en metode som kalles toerkomplement, kan man bruke en addisjons-krets til å trekke fra hverandre tall også. Kretser for alle de logiske og matematiske instruksjonene legges inn i det som kalles ALU (Arithmetic Logic Unit) og ved å sende riktig styringsbit til ALU, får man utført den instruksjonen man ønsker.

$$\begin{array}{rrr} & X_1 & X_0 \\ + & Y_1 & Y_0 \\ \hline S_2 & S_1 & S_0 \end{array}$$

Figure 22: Det binære regnestykket $X_1X_0 + Y_1Y_0 = S_2S_1S_0$. Kretsen i Fig. 21 utfører dette.

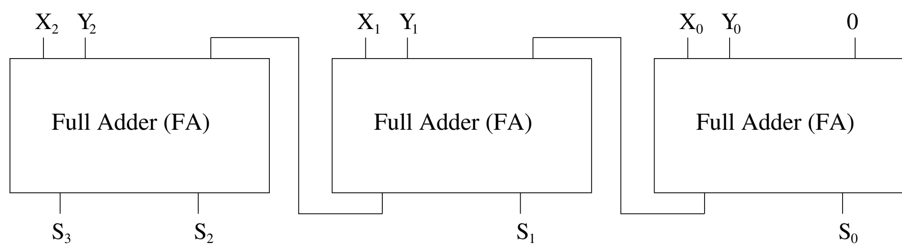


Figure 23: Tre Full Adder kretser skjøtet sammen slik at de regner ut det binære regnestykket $X_2X_1X_0 + Y_2Y_1Y_0 = S_3S_2S_1S_0$ korrekt for alle mulige verdier av X og Y.

3 Forelesning 29/1-24(2 timer). Vipper og registre, CPU-arkitektur

Avsnitt fra Tanenbaum: 1.3.1

Slides brukt i forelesningen[44]

3.1 Forelesningsvideoer

Uredigert opptak av første del av forelesningen (00:45:07)[45]
Uredigert opptak av andre del av forelesningen (00:59:02)[46]

Opptak av forelesningen inndelt etter temaer:

os3del1.mp4[47] (06:26) Innledning om forelesninger, oblig1 og os-grupper i Canvas
os3del2.mp4[48] (02:28) Om dagens forelesning
os3del3.mp4[49] (05:49) Slides: Sist, Adderings-krets, numerisk overflow
os3del4.mp4[50] (05:45) Slides: Arithmetic Logic Unit (ALU), ALU-operasjoner og tidlig ALU-design
os3del5.mp4[51] (03:28) Slides: Vipper og registere
os3del6.mp4[52] (01:19) Slides: Første forsøk på å lagre en bit
os3del7.mp4[53] (03:17) Slides: Andre forsøk på å lagre en bit
os3del8.mp4[54] (05:26) Slides: Tredje forsøk, D-latch
os3del9.mp4[55] (01:21) Slides: Spørsmål: hva var problemet med de to første løsningene?
os3del10.mp4[56] (07:50) Slides: Shift-register av D-latch'er. Menneskelige-D-latcher. Løsning: D-vippe med klokke
os3del11.mp4[57] (01:15) Gjennomgang av svaret for første poll om port-diagram.
os3del12.mp4[58] (01:32) Gjennomgang av svaret for andre poll om port-diagram.
os3del13.mp4[59] (06:11) Slides: Sammenslåing av to D-latcher til en D-vippe
os3del14.mp4[60] (01:53) Slides: Tellere
os3del15.mp4[61] (04:34) Slides: Von Neumann arkitektur
os3del16.mp4[62] (04:05) Slides: Beregningsenheter
os3del17.mp4[63] (05:07) Slides: Simulering av CPU, Datapath
os3del18.mp4[64] (03:28) Slides: Høynivåkode og maskininstruksjoner
os3del19.mp4[65] (06:24) Slides: Assemblykode og maskinkode for simulerings-CPU'en

[44] https://www.cs.oslomet.no/ haugerud/os/Forelesning/os3.pdf
[45] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3.mp4
[46] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3b.mp4
[47] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del1.mp4
[48] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del2.mp4
[49] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del3.mp4
[50] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del4.mp4
[51] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del5.mp4
[52] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del6.mp4
[53] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del7.mp4
[54] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del8.mp4
[55] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del9.mp4
[56] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3del10.mp4
[57] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel1.mp4
[58] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel2.mp4
[59] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel3.mp4
[60] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel4.mp4
[61] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel5.mp4
[62] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel6.mp4
[63] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel7.mp4
[64] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel8.mp4
[65] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel9.mp4

os3del20.mp4[66] (01:49) Slides: Branch control
os3del21.mp4[67] (05:09) Demo: Hvordan kjøre og programmere simulerings-CPU'en med maskinkode

3.2 Sist

- Transistorer
- Porter
- Sannhetstabell
- Skrive ned krets med AND, OR og NOT-porter
- Forenkle kretsen med Boolsk algebra
- Tegne logisk krets
- Logisk krets som adderer tall

I forrige forelesning så vi på hvordan man med datamaskinarkitekturens atomer, transistorer, kan lage logiske porter. Videre så vi på hvordan enhver logisk enhet kan defineres utifra en sannhetstabell som gitt hvilke logiske signaler som kommer inn i enhet sier hva som skal komme ut. Utifra en slik sannhetstabell kan man skrive ned et uttrykk ved hjelp av binær logikk og forenkle dette utrykket med boolsk algebra og andre metoder. Man ender da opp med et uttrykk beskrevet med de logiske operatorene AND, OR og NOT. Til slutt kan man konstruere en logisk krets utifra det logiske uttrykket og utifra dette kan man lage en fysisk enhet som nøyaktig oppfyller den sannhetstabellen man satt opp. Dette kan gjøres på en kretskortfabrikk hvor hver eneste transistor legges inn på kretskortet på samme måte som i den logiske kretsen. Dermed kan man lage hvilke som helst type logiske enheter. Ved å fysisk koble sammen slike enheter med ledninger, kan man lage komplekse kretser som for eksempel legger sammen to 64-bits tall. Men etter å ha lagt sammen to tall trenger vi å kunne lagre dette resultatet og umiddelbart etterpå for å kunne bruke resultatet i nye beregninger. For å kunne lage en datamaskin trenger vi derfor lagerenheter.

3.3 ALU

Vi har vist at ved å sette sammen flere bokser som adderer enkelt-siffer med mente, kan man lage en adderer som ser omtrent ut som i Fig. 24.

En slik løsning kan systematisk utvikles til å legge sammen 32 og 64 bits tall. Når det gjelder addisjon, gjøres det noen forbedringer på metoden vi brukte for at det ikke skal ta for lang tid for mente å bevege seg fra siste til første siffer. Men det vil bruke mye plass om man må ha en slik enhet for enhver matematisk eller logisk operasjon som man ønsker å utføre. Det viser seg at man kan konstruere en krets som ved hjelp av små endringer både kan addere, subtrahere, øke med en, sammenligne, gange med 2, og så videre. Man styrer hvilken av disse operasjonene enheten skal utføre ved hjelp av kontroll-bits. I Fig. 25 ser vi en tilsvarende krets med to ekstra kontroll-bit.

Dette gjør det mulig å kode inn flere lignende operasjoner inn i kretsen, og den blir til en liten ALU (Arithmetic Logic Unit). Eksakt hvilken funksjon som utføres bestemmes av kontroll-bitene. En ALU er selve hjernen i en CPU hvor alle beregninger blir gjort. I tillegg inneholder CPU registre (4-bits registre i figuren) for å lagre data og en kontrollenhet som oversetter maskininstruksjoner til styringsbit for registre og ALU slik at riktig instruksjon blir utført. Konseptet ALU ble innført av matematikeren

[66] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel10.mp4
[67] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os3bdel11.mp4

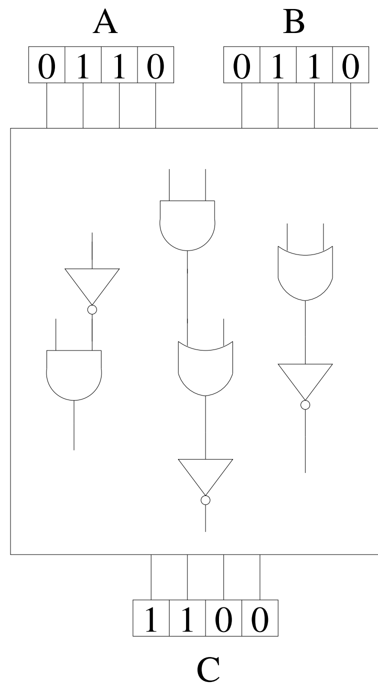


Figure 24: C = A + B (12 = 6 +6). Logiske kretser med AND, OR og NOT er konstruert inne i boksen på en slik måte at de alltid regner ut riktig resultat. Kretsen leser fra og skriver til 4-bits registre.

John von Neumann i forbindelse med konstruksjonen av en av verdens første datamaskiner, EDVAC i 1945. Det følgende er operasjoner som alle ALU-er kan utføre:

- Adder
- Subtraher
- Inkrement (++)
- Dekrement (−−)
- Multipliser
- Divider
- Shift (flytt alle bit i en retning)
- Sammenligne
- AND, OR, NOT, XOR

Neste steg er å forstå hvordan man kan bruke AND, OR og NOT-porter til å lage registre som kan lagre data inne i CPUen.

3.4 Lagring av data: vipper og registre

Man kan lagre og representere nuller og enere ved å bruke små kondensatorer som lagrer elektrisk ladning. Men dette er ikke like raskt som om man bruker logiske porter lagd av transistorer. I tillegg må

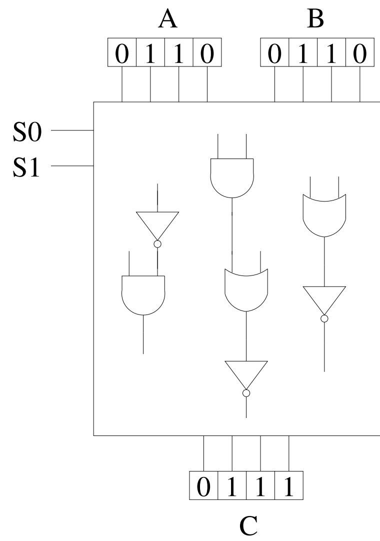


Figure 25: En ALU med fire funksjoner. C = A + 1 (7 = 6 + 1). Her styrer kontroll-bitene S0 og S1 kretsen til å øke A med en og legge resultatet i C. Dette kan for eksempel skje ved S0 = 0 og S1 = 0, mens ved S0 = 0 og S1 = 1 ville ALU-en legge sammen tallene.

kondensator-lagerenheten jevnlig oppfriskes, typisk 10 ganger i sekundet. Men det er denne teknologien som brukes i RAM og dette skal vi se på senere. Hovedproblemet er at denne teknologien gjøer RAM tregere. For å lage en lagerplass som hurtig kan leses å endres må vi bruke logiske porter. Av slike porter kan man lage en lagerenhet som kan lagre nuller og enere og disse kalles vipper(engelsk: flip-flops). Dette er den siste byggestenen vi trenger for å kunne lagre dataene i beregningene som kretsene gjør. Det er mulig å lage en slik lagerenhet ved hjelp av porter. Denne vil da bli ekstremt hurtig, like hurtig som resten av CPU-en og langt hurtigere en lagringsenhetene i RAM. En vippe er den grunnleggende lagringsenheten i CPU-en og brukes til all lagring av data internt, inkludert cache (mellomlagring) i CPU-en og cache mellom CPU og RAM. I de neste avsnittene skal vi se på hvordan en slik lagringsenhet kan konstrueres. Først lager vi en enhet som kan lagre en bit, slike kan settes sammen til store registre med 32 og 64 bit.

3.4.1 Lagringsenhet for en bit, D-lås(D-latch)

For å lagre noe med porter, må vi lage en lukket krets slik at bit-verdiene bevares. Et første forsøk er vist i Fig. 27.

Men et opplagt problem med denne lagringen er at verdien ikke kan endres. Ved å legge inn en OR-port foran NOT-porten [68] får man en mulighet til å legge inn en verdi D, som vist i Fig. 28.

For denne lagringsenheten kan man endre verdien. Hvis man sender inn D=0 som på figuren vil den etterfølgende NOT-porten sende en ener inn i den øverste OR-porten. Hvis det kommer en ener inn i en av inngangene til en OR-port vil det alltid gå en ener ut, og dermed vil kretsen lagre 0 etter at eneren har gått igjennom den siste øverste NOT-porten. Hvis vi prøver å lagre D=1, kan man se at den nederste OR-porten alltid vil gi en ener ut. Dermed sendes en null opp til den øverste OR-porten som sammen med en null fra NOT-porten rett etter D gir en null og dermed en ener ved Q[69].

Dermed kan vi legge inn verdire i lagringsenheten, men problemet nå er at vi alltid vil legge inn det som kommer inn i D. Noen ganger ønsker vi å bevare det vi har lagret til senere og for å få til den muligheten legger vi inn et kontroll signal C som er slik at

[68] En OR-port etterfulgt av en NOT-port kan forenkles til en NOR-port
[69] man kan også se at det må bli slik ved at kretsen er symmetrisk

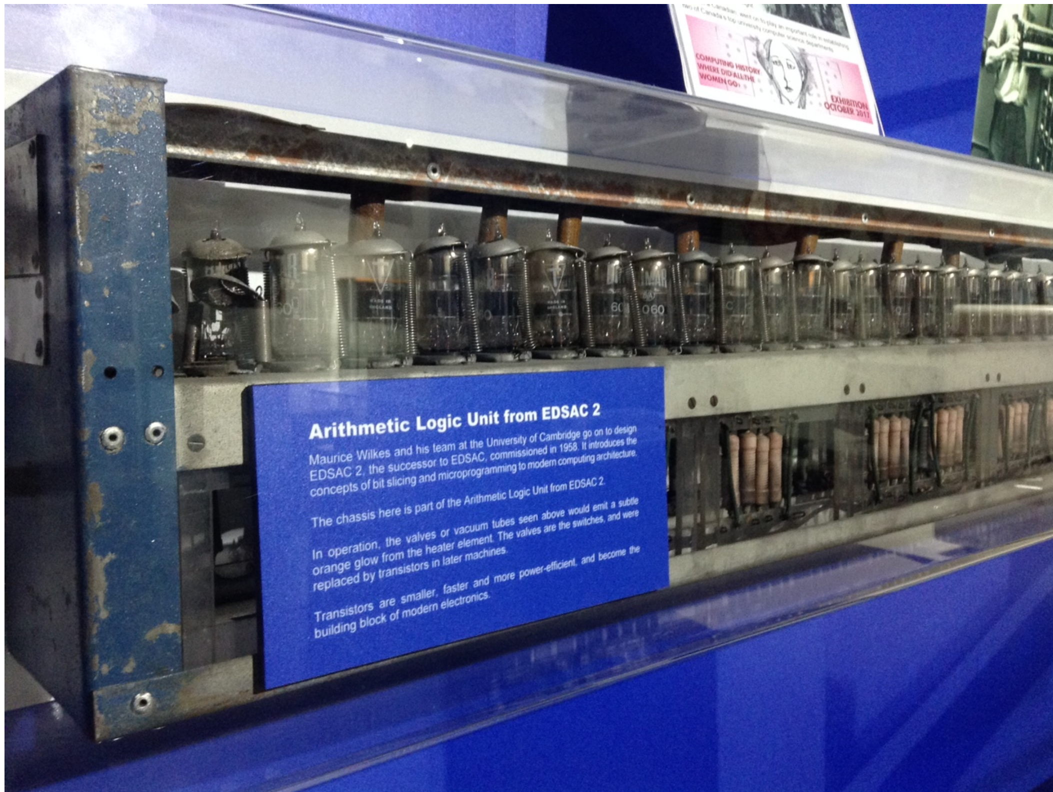


Figure 26: ALU-en til EDSAC 2 fra 1958 hadde radiorør, som senere ble erstattet av transistorer.

- C = 1 ⟶ verdien inn fra D lagres
- C = 0 ⟶ den lagrede verdien beholdes, uavhengig av verdien inn fra D

Det kan gjøres som vist i Fig. 29.

Hvis C = 1 så vil kretsen fungere nøyaktig som kretsen over i Fig. 28, fordi en ener inn i en AND-port alltid vil gi verdien til den andre inngangen ut og dermed er alt nøyaktig som i kretsen over. Hvis C = 0 vil kretsen fungere nøyaktig som den første kretsen i Fig. 27. Dette er fordi en null inn i en AND-port alltid gir 0 ut, og dermed kommer det en null inn i begge OR-portene. Men en null inn i en OR-port gir alltid verdien til den andre inngangen og dermed fungerer kretsen som den aller første og enkleste, kretsen beholder bare den verdien den har lagret.

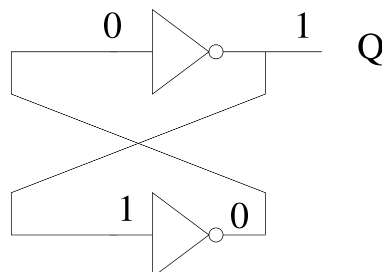


Figure 27: Lagring av verdien Q = 1

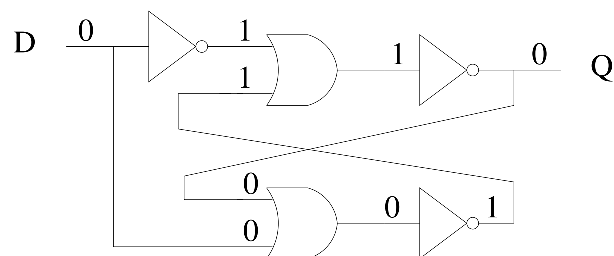


Figure 28: Endring av verdien Q til 0

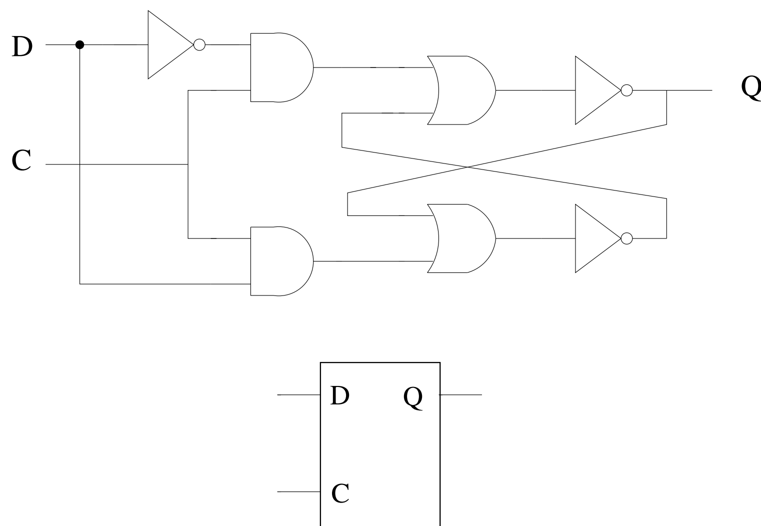


Figure 29: D-lås(latch). D står for data. Dette er en lagringskrets som kan skru lagring av og på. Den nederste boksen symboliserer hele kretsen.

3.4.2 Simulering av shift-register med studenter

Et problem med å bruke en D-lås til å lagre data i en CPU er at man trenger kontroll på når data lagres av en D-lås og når data leses av neste D-lås. Om data i tillegg går igjennom kretser som endrer data blir problemet større, for da vil det være ulik tid som går med til å fullføre beregninger av ulik type. For eksempel tar en multiplikasjon av to tall lenger tid enn en addisjon. Under forelesningen skulle hver av de åtte deltagerne fungere som en D-lås. Det vil si, de skulle kun se på D-låsen foran seg og endre sin verdi til dens verdi når input fra C er 1. Når foreleser løftet hånden, betydde det C = 1 og at alle D-låser skulle virke ved å endre sin verdi til den samme verdien som nærmeste D-lås hadde. På samme måte som i Fig. 32 der hver av D-låsene hele tiden leser av verdien Q til den D-låsen som står til venstre for seg og dette blir til input D for den selv. Når C = 1 vil den derfor endre sin Q-verdi til samme Q-verdi som D-låsen til venstre for seg. På samme måte stilte åtte studenter, dobbelt så mange som i figuren, opp etterhverandre som vist i Fig. 31.

Når foreleser hever armen og alle begynner å virke, er problemet at det kan være litt ulikheter mellom hvor fort de menneskelige D-låsene reagerer på endringer til D-låsen foran dem. Dermed kan for eksempel en endring fra figuren over bli til resultatet i Fig. **??**, at en ekstra verdi 1 dukker opp. Det kan skje hvis den kvinnelige vippen nummer 4 fra venstre leser av sin verdi raskere enn kvinnelig vippe nr 6. Og mannlig vippe nr 5 i mellom dem er raskere enn mannlig vippe nr 7. En slik måte å propagere data på vil være ustabil i forhold til selv svært små forskjeller i tidsbruk og vil også være helt ubrukelig som metode i en CPU.

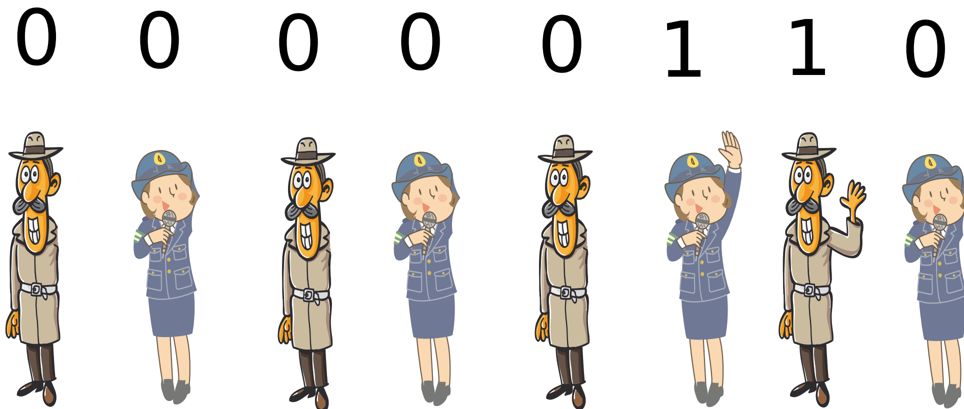


Figure 30: Åtte menneskelige D-låser på rad. Armen opp betyr 1, ned betyr 0. Hver D-lås leser verdien til D-låsen til høyre for seg (motsatt retning som for D-låsene i i Fig. 32).

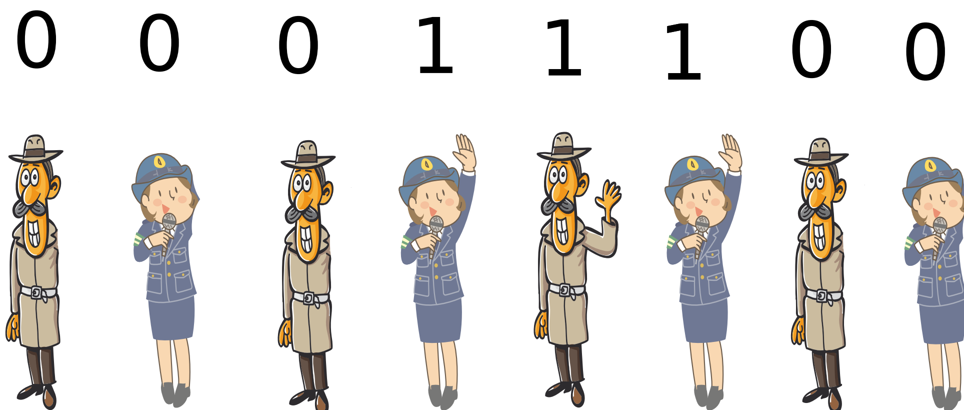


Figure 31: De menneskelige D-låsene reagerer ikke nøyaktig like fort og dermed kan dette bli neste tilstand for arrayet av bits, der en ener for mye har dukket opp.

For å kunne kontrollere nøyaktig når dataene blir overført og gjøre overføringen uavhengig av små svingninger i tempo på de involverte kretsene, innfører man en klokke som med jevne av og på singnaler styrer dette. I forelesningen ble dette gjort ved at foreleser svinger armen opp og ned i et jevnt tempo med en frekvens på omtrent en halv Hertz, altså hvert andre sekund. Når dette gjøres og alle studentene virker som de skal, kan man se at de to bit'ene beveger seg bortover uten at noe informasjon blir borte. Svingningene opp og ned med armen tilsvarer den berømte CPU-klokken, som typisk svinger med en frekvens på 2-4 GHz, noe som betyr flere milliarder svingninger av og på i løpet av et sekund.

3.4.3 Vipper(flip flops)

Nå er vi nesten fremme; vi har sett hvordan en krets kan gjøre operasjoner som å legge sammen to tall og vi har nå en lagringsenhet som input kan tas fra og resultatet lagres i. Men det er et problem som gjenstår. Hvis vi bare kobler sammen disse enhetene, har man ikke kontroll på den logiske flyten av data. Selvom signalene går med nesten lysets hastighet, tar det litt tid fra signalene kommer inn på den ene siden av en regnekrets til det rette svaret kommer ut i den andre enden. Et eksempel på hvordan flyten

blir ukontrollert får man om man kobler sammen fire D-låser i hensikt å lage et shift-register som vist i Fig. 32. Vi ønsker å kunne utføre en operasjon som 1010 -> 0101 som med binære tall er det samme

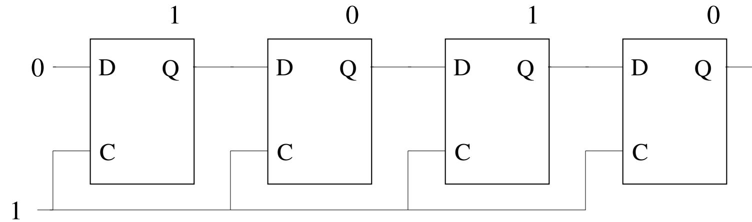


Figure 32: D-latch(lås). Fire D-låser settes sammen til et shift-register.

som å dele på 2. Men om vi sender en 1'er inn i C (som betyr at D-låsen skal lese inn en ny verdi) som i figuren, har vi ikke kontroll på hvor lang tid det tar for den nye verdien å bli lest, og med en gang den blir lest, vil neste D-lås lese den nye verdien og så videre[70]. Dermed har man ikke kontroll på hvor langt informasjonen går før man eventuelt skal sende inn en 0'er inn i C for å stoppe innlesningen og lagre resultatet. I mer kompliserte kretser vil det også være et problem at det kan variere mye hvor lang tid det tar før beregningen er ferdig og neste bit klar for lesing. Dette problemet løser man i en CPU ved å sette sammen D-låser to og to og å ha en klokke som sender et av og på signal med en viss frekvens, for eksempel 1 GHz, eller en milliard endringer mellom null og en i sekundet. Hvordan dette kan gjøres er vist i Fig. 33. Når klokken sender en 1'er, vil en slave lese inn og lagre det som master lagret når

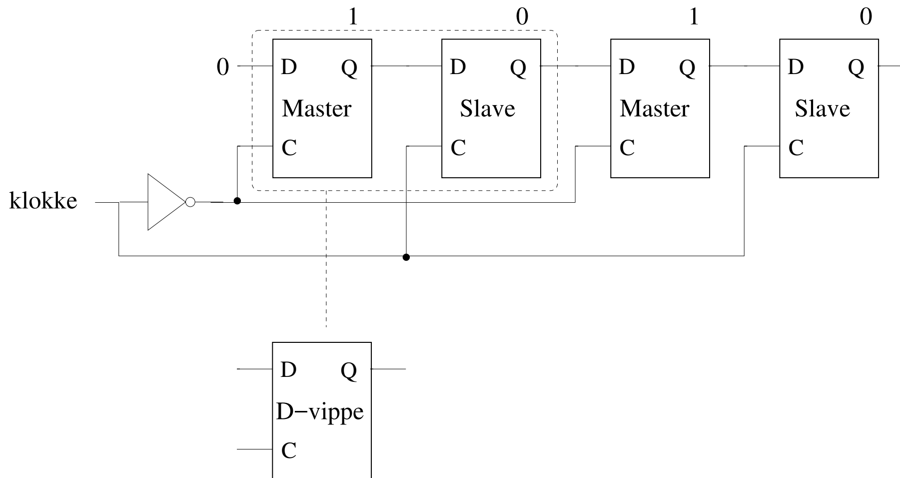


Figure 33: To master og slave D-låser blir til en D-vippe. Klokken går av og på og styrer når de endelige dataene lagres av slaven. Dette er selve CPU-klokken, som typisk har en frekvens på 1-3 GHz.

klokken rett før sendte en 0. Dermed vil data kunne bevege seg fra D-vippe til D-vippe på en kontrollert måte. Det er slaven som holder den gjeldene verdien for vippen som den leser fra master i starten av en klokkesyklus. Dette er verdien til bit'en i denne klokkesyklusen. Et shift til høyre vil forflytte alle slave-verdiene synkronisert, slik at de flytter seg hver gang klokken går over fra å sende 0 til å sende 1. Til en viss grad kunne man se at det fungerte under student-simuleringen.

Oppsummert vil tiden deles inn i små klokke-tikk og innenfor et slikt tikk må

- Når klokken sender en 0: alle beregninger ferdigstilles ved å strømme igjennom kretsene som adderer

[70] I eksempelet på forelesning så man hvor vanskelig det var når personene som simulerte D-låser skulle sende informasjon nedover rekken

eller gjør annen logikk, sluttresultatet lagres hos master. Det må være ferdig før klokken switcher til 1.

- Når klokken sender en 1: slaven leser verdien fra master og lagrer den. Den begynner straks å sende dette resultatet, som er det gjeldende resultatet, ut i kretsene som er koblet til utgangen for nye beregninger.

CPU-klokken er helt essensiell for å synkronisere dataene, for hvert tikk av klokken kan et nytt sett av beregninger gjøres, for eksempel å utføre en maskin-instruksjon.

Data lagres i CPU-en med slike vipper og slike samlinger av vipper som vi lar representere tall kalles registre. En 32bits CPU har registre som består av 32 vipper og dermed kan lagre 32bits tall. En 64bits CPU har 64bits registre. Vi skal nå se på en simulering av en virkelig CPU laget med verktøyet Digital Works. Denne CPU-en er svært liten, det er en 4bits CPU og den har altså 4bits registre. Men bortsett fra størrelsen, kan den i prinsippet gjøre alle beregninger som en moderne CPU kan gjøre og den virker på helt samme måte. Grunnen til at en moderne CPU er mye mer komplisert er i tillegg til at de er større, er at arkitekturen er endret for at beregningene skal gå rasker. Registrene sitter i begge ender av beregningene. Hvis to tall skal legges sammen, kobles output for to registre til ALU-en som inneholder addisjons-logikken. Utgangen av ALU kobles så til registeret som skal lagre resultatet. I denne simulerte maskinen gjøres en ny operasjon eller instruksjon hvert klokke-tikk.

3.5 Tellere

For å kunne løpe gjennom instruksjoner i et program, trenger man en teller som kan telle oppover for hvert klokke-tikk. Et program som kjøres av en CPU består av en rekke instruksjoner som skal gjøres etterhverandre. Den starter med instruksjon nummer en som ligger i RAM og så teller den seg oppover til instruksjon nummer 2 og så videre ved hjelp av telleren. Noen ganger er det behov for å hoppe i koden, som ved en If-test, og da settes telleren til det instruksjonsnummeret det skal hoppes til og teller videre derfra. I Fig. 34 ser vi en 2-bits teller som kan telle fra 0 til 3. Ved å la verdien disse vippene har, D-vipper i dette tilfellet, representere hvert sitt siffer i et binært tall, kan de tilsammen representere tallene 0-3. For å lage større tall, trenger vi bare flere vipper.

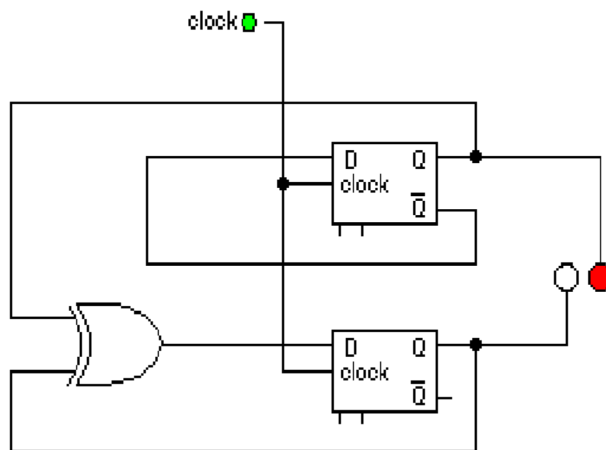


Figure 34: En 2-bits teller

3.6 CPU-arkitektur

Den vanligste datamaskinarkitekturen som brukes, i hvertfall i hovedtrekk, av de fleste av dagens datamaskiner, er von Neumann-arkitekturen. Den ble definert i rapporten "First Draft of a Report on the EDVAC" av matematikk-professoren John von Neumann i 1945. Fig. 35 viser en skisse av denne arkitekturen. De viktigste delene av von Neumann arkitekturen er

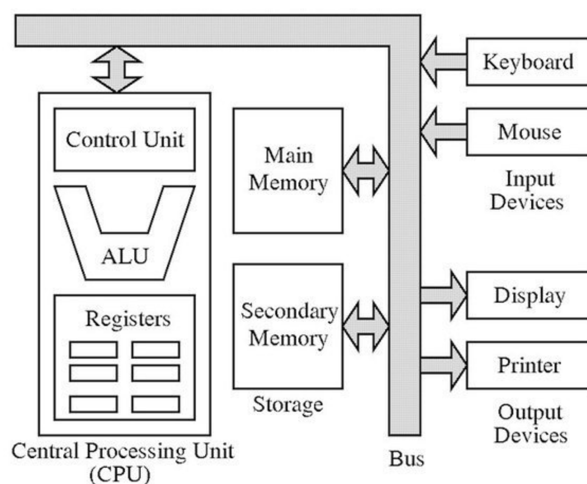


Figure 35: Von Neumann arkitektur

- Et arbeidsminne (internminnet/RAM) som inneholder både instruksjoner og kode.
- En aritmetisk/logisk enhet (ALU - Arithmetic Logic Unit) som kan utføre matematiske og logiske operasjoner.
- En kontrollenhet som henter inn instruksjoner fra RAM, dekoder dem og sender signaler som gjør at instruksjonen blir utført.
- Registre, internlager for både instruksjoner og data inne i CPU-en.
- Enheter for input og output som gjør at CPU kan kommunisere med harddisk, tastatur, nettverk, etc.

CPU(Central Processing Unit) inneholder kontrollenheten, ALU og registre (de to siste utgjør tilsammen datapath). Et problem med denne arkitekturen blir omtalt som 'the Von Neumann bottlenck'. Det kommer av at instruksjoner og data deler samme data-buss. I Hardvard arkitekturen er strømmen av instruksjoner og data inn til CPU fysisk adskilt. I de fleste moderne løsninger er en Modified Harvard architecture tatt i bruk som løser flaskehalsen ved å ha forskjellige cache-kanaler for instruksjoner og data.

3.7 Beregningsenheter

Følgende er noen typer beregningsenheter med økende grad av kapasitet. En CPU er meget anvendelig ved at den er så generell at den kan programmeres til å gjøre alle mulige beregninger. De etterfølgende enhetene er i økende grad spesialiserte og dermed raskere til å utføre de spesielle beregningene de er lagd for å gjøre.

- ALU (Arithmetic Logic Unit) CPU-ens hjerne
- CPU (Central Processing Unit)
- FPU (Floating-Point Unit) vanligvis integrert i CPU
- GPU (Graphics Processing Unit) tusenvis av cores
- FPGA (Field-programmable Gate Array) programmerbar logikk
- ASIC (Application-Specific Integrated Circuit)

3.8 En simulering av en datamaskin

Fig. 36 viser arkitekturen til en komplett CPU som kan utføre programmer skrevet i såkalt maskinkode. Det er gjort visse endringer i forhold til von Neumann arkitekturen, den vesentligste er at maskininstruksjonene er lagret i en ROM (Read Only Memory) inne i CPUen. Vanligvis hentes de fra RAM fortløpende og lagres i instruksjonsregisteret i CPU-en før de kjøres.

I en reell CPU består portene av fysiske halvledere og ledningene av fysiske elektriske ledninger brent inn i kretskort i ekstremt liten skala. Alt denne maskinen forstår er nuller og enere og for å gi den beskjed om hva den skal gjøre, må disse beskjedene gis i form av en binær kode. I denne maskinen består instruksjonene av 8 bit og for eksempel betyr maskininstruksjonen 01001011 at den skal legge sammen tallet som er lagret i register R2 og tallet som er lagret i register R3 og lagre resultatet i R2. Instruksjonen er delt opp slik at de 4 første bit'ene 0100 betyr ADD og at operasjonen 'legg sammen' skal utføres. De to neste bitene 10 betyr R2 og de to siste 11 betyr R3. Det eneste CPU-en gjør er å utføre denne type instruksjoner om og om igjen.

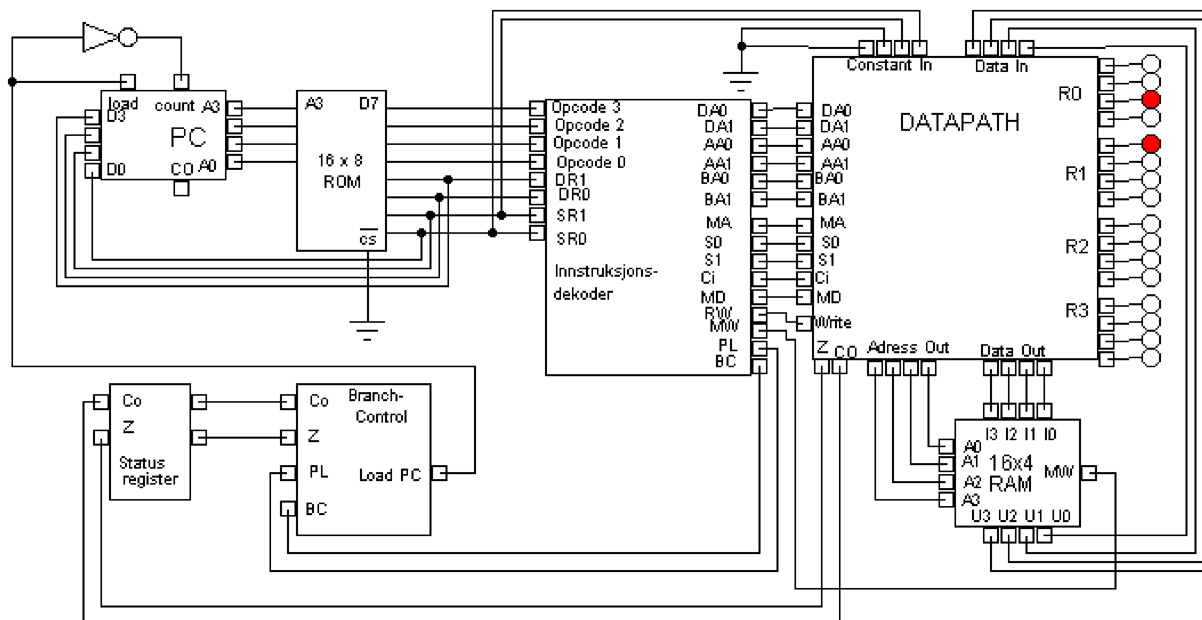


Figure 36: En komplett CPU tilkoblet RAM

En samling slike instruksjoner utgjør et program og i denne maskinen er de lagret i boksen merket ROM. Vanligvis ligger maskinkoden som skal utføres sammen med programmets data i RAM, boksen nede i høyre hjørnet. I denne datamaskinarkitekturen ligger bare programmets data i RAM og det finnes instruksjoner som flytter data mellom RAM og registrene, men vi skal i første omgang ikke bruke dem

her. Kanalene som går mellom datapath og RAM er data-bussen. I en von Neuman-arkitektur sendes både data og instruksjoner over denne bussen. Men i vårt tilfelle ligger ikke instruksjonene i RAM men i en spesiallaget ROM, dette er ikke så vanlig i CPU-arkitekturer. Men totalt sett minner denne arkitekturen derfor mer om Harvard-arkitekturen hvor instruksjoner og data sendes inn til CPU-en på to forskjellige busser. Det finnes typisk noen hundre registre i en standard CPU, men disse blir bare brukt til mellomlager, mer permanente verdier som variabler i et program, lagres i RAM som har mye større kapasitet. Enda større datamengder som man ønsker å lagre når maskinen skrus av, lagres på disk.

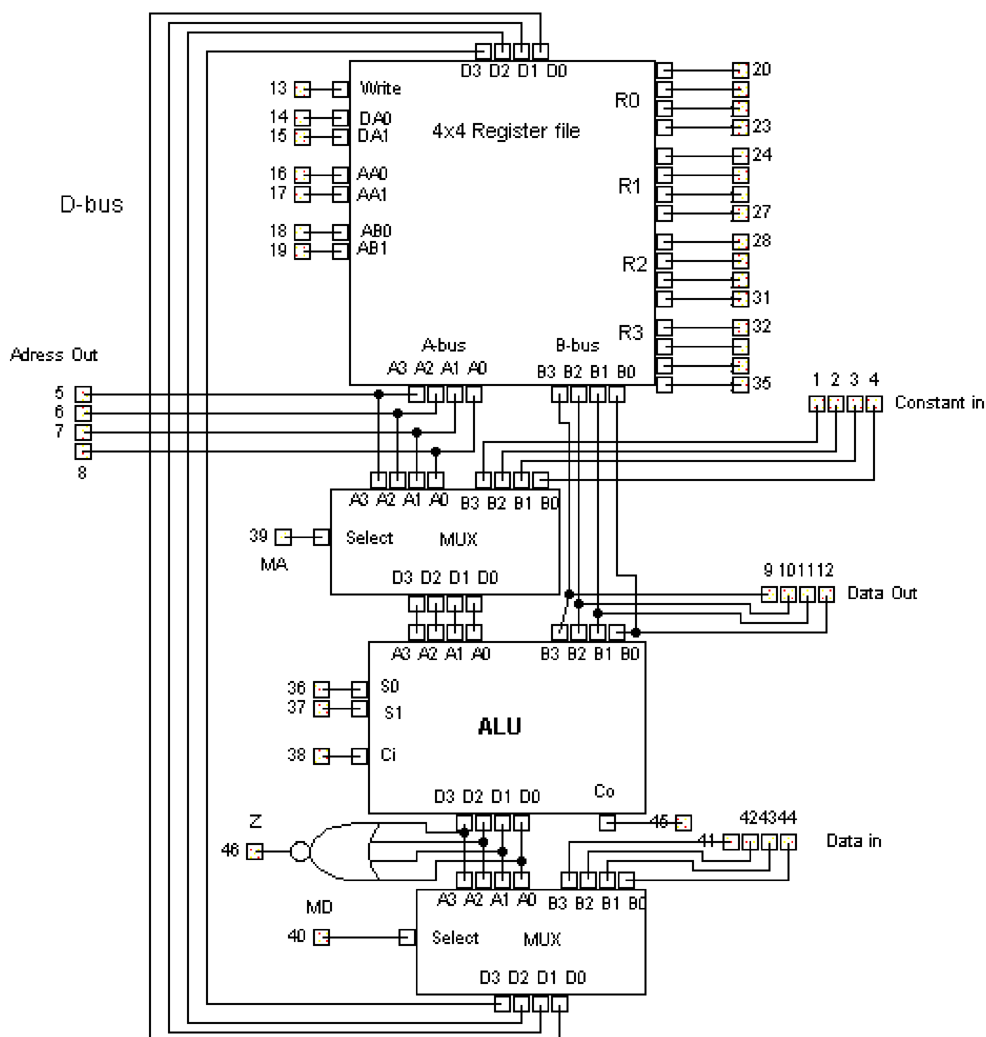


Figure 37: Datapath med fire 4-bits registre og ALU. En ofte brukt operasjon består i at data føres inn i ALU fra registrene via A og B og at resultatet etter beregningen lagres i et register.

Men hvor er selve hjernen til CPU-en som kan legge sammen tall, trekke dem fra hverandre, sammeligne dem og så videre? Den sitter inne i Datapath der inngangen til to valgte registre kan kobles til ALU-en (Arithmetic Logic Unit). Og inne i denne skjer selve operasjonen. Verdien på input-ledningene S0 og S1 avgjør hvilke operasjoner som utføres, ADD, SUB, SHIFT, etc. Andre input-ledninger avgjør hvilke registre som ledes inn i ALU og i hvilket register resultatet lagres. Disse input'ene kommer fra instruksjonsdekoderen som for hver instruksjon trykker på de rette knappene for å få den utført. For eksempel sørger instruksjonsdekoderen for at maskininstruksjonen 01001011 nevnt ovenfor virkelig fører til at ALU legger sammen tallet som er lagret i register R2 med tallet som er lagret i register R3 og lagrer resultatet i R2.

3.9 Fra høynivåkode til CPU-beregning

Så godt som all programvare blir skrevet i et høynivåspråk som Java eller C. Dette er språk som vanlige CPU-er ikke forstår og høynivåkode må derfor oversettes til maskinkode for at den skal kunne utføres. Oppgaven med å oversette fra høynivåkode til maskinkode utføres vanligvis av et program, en kompilator. Vi skal se på et enkelt eksempel og vise hvordan en liten bit høynivåkode kan oversettes til et kjørbart program i vår virtuelle Digital Works prosessor. Vi starter med følgende kodesnutt:

```
S = 0;
for(i=1;i < 4;i++)
{
   S = S + i;
}
```

Det en programmerer ønsker CPU-en skal utføre er følgende:

```
i = 1: S = 0 + 1 = 1
i = 2: S = 1 + 2 = 3
i = 3: S = 3 + 3 = 6
```

og så avslutte. Om dette var et C-program, ville variablene S og i lagres i RAM og lastes inn i registrene under beregningene. Nå skal vi skrive en optimalisert maskinkode for vår maskin, hvor S og i lagres i registrene Dette gjør at beregningen går hurtigere, fordi det tar relativt lang tid å lese og skrive til RAM. Følgende er et såkalt Assembly-program som utfører denne beregningen. Assembly er et språk som ligger svært nær maskinspråk, det bruker symboler for maskininstruksjoner slik at det er lettere å lese for et menneske. Det har i motsetning til høynivåspråk en enkel en-til-en oversettelse til maskinspråk.

```
0 MOVI R0 <- 3          (MOV Integer. maksverdien i for-løkken legges i R0)
1 MOVI R1 <- 1          (tallet som i økes med for hver runde i løkken)
2 MOVI R2 <- 0          (variabelen i lagres i R2)
3 MOVI R3 <- 0          (S = 0)
4 ADD R2 <- R2 + R1     (i++)
5 ADD R3 <- R3 + R2     (S = S + i)
6 CMP R2 R0             (COMPARE, er i = 3 ? )
7 JNE 4                 (Jump Not Equal 4, hopp til linje 4 hvis i != 3)
```

Dette er slik programmet kan gjennomføres med maskin-instruksjoner og for å kunne programmer vår maskin, må nå dette programmet skrives binært med enere og nuller. Da trenger vi å vite litt av instruksjonssettet for maskinen. Vår CPU er konstruert slik at instruksjonene består av 8 bit. De fire første bit'ene definerer hvilket nummer instruksjonen er i rekken av instruksjoner. Bit nummer 5 og 6 er første operand og nr 7 og 8 er andre operand. Vanligvis bestemmer det to-bits tallet i operanden hvilket register som er involvert. Instruksjonene vi trenger til vårt program er:

binært Nr	operand1	operand2	Nr	Navn
0010	DR	tall	2	MOVI
0100	DR	SR	4	ADD
1100	DR	SR	12	CMP
1111	nr	nr	15	JNE

Den første linjen betyr at MOVI er instruksjon nummer 2 og at den gjør at tallet gitt ved de to siste bit'ene i instruksjonen legges i register nr DR (Destination Register). Instruksjonen ADD er nummer 4 og den legger sammen DR og SR(Source Register) og legger svaret i DR. Instruksjon nr 12 er CMP og den sammenligner DR og SR. Til slutt er instruksjon nummer 15 JNE(Jump Not Equal) som hopper til linjenummeret definert av de fire siste bit'ene i instruksjonen, hvis registrene sammenlignet i forrige

instruksjon var ulike. Når vi vet dette kan vi oversette Assembly-programmet vi skrev til maskinkode som kan lastes inn i maskinens ROM:

Linje Nr	I Nr	DR	SR
0	0010	00	11
1	0010	01	01
2	0010	10	00
3	0010	11	00
4	0100	10	01
5	0100	11	10
6	1100	10	00
7	1111	01	00

Første instruksjon er MOVI (0010) og den legger tallet 3 (11) i register nummer 0 (00, det vil si R0 Og tilsvarende for de andre instruksjonene. En kompilator oversetter direkte fra høynivåkode til tilsvarende maskinkode som kan kjøres direkte av datamaskinene. Ved å laste maskinkoden inn i Digital Works-simuleringen, kan man se hvordan dette programmet kjøres instruksjon for instruksjon og til slutt produserer resultatet S = 6.

I prinsippet fungerer moderne CPUer på samme måte som i denne simuleringen. En vesentlig forskjell er at også maskininstruksjonene hentes fra RAM. Da dette tar litt tid og da noen instruksjoner er litt mer omfattende en andre, kan det i noen tilfeller ta mer enn en klokkesyklus å få utført en instruksjon.

3.10 Løkker og branch control

Legg merke til at siste instruksjon hopper til linje 4, basert på sammenligningen av to registre i forrige instruksjon. Denne muligheten er svært viktig, for den gjør det mulig å utføre alle slags løkker, som for og while-løkker og i tilleg if-tester. I alle disse tilfellen må utførelsen kunne velge å hoppe til et annet sted i programmet basert på et resultat. JNE-instruksjonen (Jump Not Equal) gjør at man hopper til en adresse hvis sammenligningen av to registre i forrige instruksjon viste at de er forskjellige.

Hvis man ikke kunne hoppe i koden, ville man bare kunne utføre instruksjoner etterhverandre fra første til siste instruksjon og programmene måtte være enormt lange. Et program med en milliard linjer ville kunne utføres på omtrent ett sekund. Fig. 36 ser vi at det er to bit i DATAPATH, Z og co, som leder til en liten del av CPU-en som kalles branch controll. Disse styrer sammen med to bit fra instruksjonsdekoderen om programkontrollen bare skal oppe til neste instruksjon som den vanligvis gjør når program counter teller oppover, eller om PC istedet skal hoppe til det 4-bits tallet som kommer inn fra de 4 siste bit i ROM der instruksjonen inneholder adressen det eventuelt skal hoppes til.

Branch control gjør at denne enkle CPU-en kan utføre if, for og while og med det i prinsippet kan utføre alle mulige dataprogrammer. Begrensningen en firebits registerstørrelse utgjør er det lett å fjerne ved å bare øke registerstørrelsen til for eksempel 64 bit som de fleste vanlige Intel og AMD CPUer bruker. Logikken er den samme, det blir bare veldig mange flere ledinger å holde styr på.

Totalt kan denne maskinen forøvrig gjøre åtte instruksjoner(men store og load som laster inn og ut fra RAM har bugs):

binært Nr	operand1	operand2	Nr	Navn	Funksjon
0000	DR	SR	0	MOV	R[DR] $\longleftarrow$ R[SR]
0010	DR	tall	2	MOVI	R[DR] $\longleftarrow$ tall
0100	DR	SR	4	ADD	R[DR] $\longleftarrow$ R[DR] + R[SR]
0110	DR	SR	6	SUB	R[DR] $\longleftarrow$ R[DR] - R[SR]
1000	DR	SR	8	LOAD	R[DR] $\longleftarrow$ M[R[SR]]
1010	DR	SR	10	STORE	M[R[DR]] $\longleftarrow$ R[SR]
1100	DR	SR	12	CMP	R[DR] - R[SR] = 0?
1111	nr	nr	15	JNE	PC = nr nr hvis like

Hvilke instruksjoner som kan utføres, hvordan de er nummerert, hva operandene betyr og hvordan de skal tolkes, utgjør tilsammen maskinarkitekturen. Den vanligste arkitekturen i våre dager i PCer og servere er X86-arkitekturen som ble innført av Intel i 1978 og som også brukes av AMD. Men den aller mest produserte prosessoren er basert på ARM (Advanced RISC Machine) som brukes i mobiler. Det var i 2017 produsert mer enn 100 milliarder ARM-prosessorer og 200 milliarder ble passert i 2021.

4 Forelesning 5/2-24(2 timer). C, maskinkode og assembly

`Avsnitt fra Tanenbaum: 1.3.1`

4.1 Forelesningsvideoer

Opptak av forelesningen:

os4del1.mp4[71] (35:42) Uredigert opptak av første time av forelesningen.
os4del1.mp4[72] (52:37) Uredigert opptak av andre time av forelesningen.

Opptak av forelesningen inndelt etter temaer:

os4del1.mp4[73] (1:27) Intro om oppgaver, oblig-innlevering, os-grupper
os4del2.mp4[74] (2:10) Dagens tema; om maskinarkikturer, x86 og ARM
os4del3.mp4[75] (03:47) Demo: Mer om CPU-simuleringen, instruksjonsdekoderen (Night King som gjesteforeleser; litt feil på chroma key instillingene :)
os4del4.mp4[76] (02:45) Spørsmål: Kan man legge inn tallet 4 i et register?
os4del5.mp4[77] (00:58) Spørsmål: Er ledningene som går mellom boksene data-bussen?
os4del6.mp4[78] (01:42) Spørsmål: Er dette Harvard CPU-arkitektur?
os4del7.mp4[79] (02:40) Demo: CPU-simuleringen: visualisering av branch-control
os4del8.mp4[80] (13:09) Demo: Innlegging av en Load-innstruksjon som lagrer et resultat fra et register i RAM

[71] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4time1.mp4
[72] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4time2.mp4
[73] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del1.mp4
[74] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del2.mp4
[75] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del3.mp4
[76] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del4.mp4
[77] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del5.mp4
[78] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del6.mp4
[79] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del7.mp4
[80] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del8.mp4

os4del9.mp4[81] (02:49) Demo: Kompilering av C-versjonen av "Hello World!" med gcc
os4del10.mp4[82] (01:40) Spørsmål i pausen: Hvorfor ligger RAM inne i CPU?
os4del11.mp4[83] (01:10) Spørsmål i pausen: Hva er forskjellen på Adress Out og Data Out som går inn til RAM?
os4del12.mp4[84] (02:03) Spørsmål i pausen: Tvilsvarer branch-control i simuleringen Control Unit i Figuren i avsnitt 3.6?
os4del13.mp4[85] (07:05) Demo: C-programmering og hex-dump av maskinkoden a.out
os4del14.mp4[86] (05:10) Demo: C-program som summerer opp til S = 6 i en for-løkke. Variabler og kompilering med gcc
os4del15.mp4[87] (05:16) Demo: Oppslitting av sum.c i to deler, sumMain.c og sumFunksjon.c. Kompilering av hver del og linking
os4del16.mp4[88] (10:15) Demo: Hvordan be gcc-kompilatoren om å lage Assembly-kode? Innledning om assembly.
os4del17.mp4[89] (05:49) Demo: Kompilering og kjøring av Assembly-programmet as.s
os4del18.mp4[90] (04:57) Demo: Forklaring av Assembly-programmet as.s
os4del19.mp4[91] (02:31) Spørsmål: Hvordan vet vi at registere og data i RAM ikke blir overskrevet av andre prosesser?
os4del20.mp4[92] (01:41) Spørsmål: Hvordan vet man hva som returneres når man avslutter et Assembly-program med ret?
os4del21.mp4[93] (01:18) Forskjellen på kompilator og assembler

4.2 Simulerings-CPU og RAM

I maskinkoden som summerer opp summen S = 1 + 2 + 3 ved hjelp av en løkke som er mulig å få til på grunn av branch-kontrollen, foregår alle beregninger inne i CPU-en. Det vil si at alle tallene først legges inn i registerne og at alle resultatene fra mellomregninger før man kommer frem til den endelige summen ligger i registerne lokalt i CPU. Slik er det for virkelige CPU-er laget av Intel og AMD også. Registerne er meget hurtige, men det er et begreneset antall man kan ha inne i en CPU. En alternativ lagringsplass for beregningsdata er internminne eller RAM. Her er det plass til Milliarder av bytes (8 bit) med data, men det tar omtrent ti ganger så lang tid å lagre noe i RAM. Det optimale er derfor å alltid bruke registre for å lagre midlertidige data, slik som den stadig økende summen i eksempelet vi ser på. Men hvordan høynivåkoden oversettes til maskinkode avgjøres av kompilatoren. Dette er et program som systematisk kan oversette alle mulige varianter av høynivåkode til maskinkode som utfører det som høynivåkoden ber om når det kompilerte programmet kjøres i en datamaskin. Detaljer som om beregningsdata skal lagres i registere eller i RAM, avgjøres av kompilatoren. Men som vi skal se senere er det mulig å be kompilatoren om å lage maskinkode som skal gå hurtigst mulig. Og hvis man gjør det når man kompilerer vil kompilatoren lage kode som lagrer alle mellomregninger i registerne og først skriver resultatene til variabler i RAM når beregningene er fullført.

Når man deklarerer variabler som for eksempel i et C-program på følgende måte

[81] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del9.mp4
[82] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del10.mp4
[83] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del11.mp4
[84] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del12.mp4
[85] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del13.mp4
[86] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del14.mp4
[87] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del15.mp4
[88] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del16.mp4
[89] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del17.mp4
[90] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del18.mp4
[91] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del19.mp4
[92] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del20.mp4
[93] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os4del21.mp4

```
int sum=0;
int i;
```

vil det settes av 4 byte i RAM til denne variabelen og der initialiseres den til å ha verdien 0. RAM er ganske enkelt et enormt array av bytes som ligger etterhverandre. Den minste lagerenheten er en byte som består av 8 bit. At en integer skal være 32 bit er en konvensjon for programmeringsspråket C, men disse konvensjonene kan variere mellom forskjellige språk og også mellom forskjellige implementasjoner av C. Andre konvensjoner er at en long long int bruker 8 byte og at flytt-tall lagringsenhetene float og double er henholdsvis 32 og 64 bit lange.

4.3 C-programmering

Siden Dennis Ritchie på starten av 70-tallet laget programmeringsspråket C, har det vært tett knyttet til Unix-operativsystemer. De fleste Unix-programmer er skrevet i C og de fleste systemkall har korresponderende C-funksjoner med samme navn. Vi skal her bruke C-program som eksempler på høynivåkode og se hvordan de må kompileres til maskinkode for å kunne kjøres av en datamaskin.

4.3.1 hello.c

Et `Hello World` C-program ser slik ut:

```
/* filnavn: hello.c */

#include <stdio.h>

int main()
{
   printf("Hello world!\n");
}
```

Den første linjen inkluderer standard-biblioteket `stdio.h` som blant annet inneholder funksjoner for å kunne skrive til et terminal-vindu. Alle C-program har en main-funksjon. Den kan inneholde all koden eller inneholde kall til andre funksjoner. For å kunne kjøre et C-program, må det først kompileres til maskinkode og det kan man i et Linux-shell gjøre slik:

```
$ gcc hello.c
```

Det lages da maskinkode som lagres i en fil ved navn `a.out`. Den kan kjøres med

```
$ ./a.out
Hello world!
```

Filen `a.out` inneholder maskinkode i form av maskin-instruksjoner for en prosessor med såkalt x86-arkitektur som ble introdusert av Intel i 1978. Det finnes mange forskjellige CPU-arkitekturer, som ARM, SPARC og PowerPC, men x86 er den som nå brukes i nesten alle PCer og servere. Andre arkitekturer har andre maskin-instruksjoner og de kan derfor ikke kjøre maskinkode for x86, slik som innholdet i `a.out`. Maskinkode for Hello World er på mange tusen byte og den inneholder blant annet kode for å kommunisere med operativsystemet. Dette er nødvendig for eksempel for å kunne skrive ut noe. Man kan se på direkte på koden og følgende er deler av innholdet:

```
$ xxd a.out
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 3e00 0100 0000 3004 4000 0000 0000  ..>.....0.@.....
00000020: 4000 0000 0000 0000 d819 0000 0000 0000  @...............
00000030: 0000 0000 4000 3800 0900 4000 1f00 1c00  ....@.8...@.....

00000230: 0100 0000 0000 0000 2f6c 6962 3634 2f6c  ......../lib64/l
00000240: 642d 6c69 6e75 782d 7838 362d 3634 2e73  d-linux-x86-64.s
00000250: 6f2e 3200 0400 0000 1000 0000 0100 0000  o.2.............
00000260: 474e 5500 0000 0000 0200 0000 0600 0000  GNU.............

000005b0: f3c3 0000 4883 ec08 4883 c408 c300 0000  ....H...H.......
000005c0: 0100 0200 4865 6c6c 6f20 776f 726c 6421  ....Hello world!
000005d0: 0000 0000 011b 033b 3000 0000 0500 0000  .......;0.......
```

Deler av programmet inneholder data, som strengen `Hello world!` og andre deler inneholder maskin-instruksjoner. Disse tilsvarer på alle måter maskin-instruksjonene i den simulerte maskinen vi har sett på. Den hadde kun 8-bits instruksjoner, x86-instruksjoner er av variabel lengde mellom 8 og 48 bit. Etterhvert skal vi se på Assembly-kode og der korresponderer hver x86 assembly-instruksjon som ADD, MOV, CMP, JNE osv. til en bestem maskin-instruksjon. Dette er også helt tilsvarende som i CPU-simuleringen.

4.3.2 Et C-program som summerer

Tidligere oversatte vi høynivåkode, en for-løkke med summering, til maskinkode for den simulerte CPUen. Den prosessen vi da gjennomførte, er det samme som gcc-kompilatoren gjorde for C-programmet over. Følgende er et C-program vi kaller `sum.c` som utfører den samme beregningen. Vi kunne skrevet all koden i main-funksjonen, men lager en egen funksjon som vi kaller `sum()` for enklere å kunne analysere hva som skjer i denne spesielle kode-biten:

```
/* filnavn: sum.c */

#include <stdio.h>

int sum()
{
   int S=0,i;
   for(i=0;i<4;i++)
   {
      S = S + i;
   }
   return(S);
}

int main()
{
   int Sum;
   Sum = sum();
   printf("Sum = %d \n",Sum);
}
```

Variabler må deklareres i C. Hvis man ikke definerer funksjonen før main(), kan man få en warning fra gcc. Man kan kompilere og kjøre programmet med

```
$ gcc sum.c -o sum
```

```
$ ./sum
Sum = 6
```

Opsjonen `-o` brukes til å gi det kjørbare programmet et annet navn enn default verdi `a.out`.

4.3.3 Kompilering av C-funksjoner

Når programmet over kompileres, lages det først maskinkode av C-koden i `sum.c` og så linkes denne koden sammen med kode fra standard-biblioteket `stdio.h` til ferdig maskinkode som er klar til å lastes inn i RAM og kjøres. Det er også mulig å legge en C-funksjon i en egen fil og så kompilere den til en egen maskinkode-fil. Hvis vi kaller følgende fil `sumFunksjon.c`

```
/* filnavn: sumFunksjon.c */

int sum()
{
   int S=0,i;
   for(i=0;i<4;i++)
   {
      S = S + i;
   }
   return(S);
}
```

kan vi kompilere den med

```
$ gcc -c sumFunksjon.c -o funksjon
```

Opsjonen `-c` gir kompilatoren `gcc` beskjed om å ikke linke programmet, men bare kompilere det og legge maskinkoden i filen `funksjon`. Deretter kan vi lage en fil til som vi kan kalle `sumMain.c`

```
/* filnavn: sumMain.c */

#include <stdio.h>

extern int sum();

int main(void)
{
   int summ;
   summ = sum();
   printf("Sum = %d \n",summ);

}
```

så kan vi kompilere den med

```
$ gcc -c sumMain.c -o main
```

og lage en maskinkode-fil med navn `main`. Til slutt kan vi skjøte sammen og be kompilatoren om å linke disse to filene sammen til et kjørbart sum-program og kjøre det:

```
$ gcc funksjon main -o sum
$ ./sum
Sum = 6
```

Vi kunne gjort disse tre operasjonene, kompilering av de to programmen og linking, i en operasjon med

```
$ gcc sumFunksjon.c sumMain.c -o sum
```

men vi velger å gjøre det slik for å kunne erstatte beregningene i `funksjon` med Assembly-kode. Maskinkoden i `funksjon` tilsvarer den maskinkoden vi la inn i CPU-simuleringen, dermed kan vi i detalj sammenligne x86-Assembly med vårt eget assembly-språk for den simulerte CPUen.

4.4 Assembly

Kompendiet i INF2270 datamaskinarkitektur på UiO[94] inneholder nyttig informasjon, blan annet alle X86 instruksjonene. Forelesningsnotatene til Erik Hjelmås, OS-kompendium2018.pdf, som ligger under filer i Canvas, inneholder noen avsnitt om Assembly.

Det finnes mange andre gode kilder på nettet, blant annet denne introduksjonen til Assembly.[95]

Vi skal ikke gå veldig dypt inn i x86-assembly, men ved hjelp av noen få av de tilgjengelige Assembly-instruksjonene skrive kode som tilsvarer noen enkel eksempler på høynivåkode.

Idag trenger vi bare å kjenne noen få assembly-instruksjoner som ligner på dem vi lagde for simulerings-CPU-en:

Instruksjon	source	destination	resultat
mov	s	d	verdien av s legges i d
add	s	d	d = d + s
cmp	s	d	sammenlign (compare) s og d
jne	label		Jump Not Equal, hvis s ulik d i forrige linje, hopp til label

Her kan s være en konstant (et tall skrevet som $34 for tallet 34), et register (`%rax, %rbx, %rcx, %rdx`) eller en referanse til et sted i RAM. Det siste kan være definert som et variabelnavn eller på formen `-4(%rbp)`, som betyr fire byte fra starten av stack for programmet. Stack er et område i RAM der variabler for metoder lagres og rbp står for Register Base Pointer og peker på starten av stacken.

4.4.1 Summerings-funksjonen skrevet i Assembly

Følgende x86-Assembly kode utfører nøyaktig det samme som maskinkoden i filen `funksjon` i avsnittet over. Assemblerkode ligger svært tett opp til den maskinkoden som kjører i CPU-en man programmerer for og koden kan kun kjøre på CPUer som har nøyaktig den arkitekturen og dermed de maskininstruksjonene som koden inneholder. Nesten alle data som instruksjonene i maskinkode virker på er lagret i selve CPU-en og lagringsenhetene for disse dataene er registre. I vår simulerte CPU kalte vi registrene R0, R1, R2 og R3. I x86-arkitekturen finnes det fire generelle registre som er svært mye brukt i all Assembly-programmering og de kalles ax, bx, cx og dx. Opprinnelig ble disse betegnelsene brukt om 16-bits registre på den tiden dette var den vanlige størrelsen for en x86-CPU. Ganske snart økte størrelsen til

[94] https://www.uio.no/studier/emner/matnat/ifi/INF2270/v16/pensumliste/kompendium-inf2270.pdf
[95] https://www.cs.oberlin.edu/ bob/cs331/Notes%20on%20x86-64%20Assembly%20Language.pdf

32-bit og disse registrene ble da betegnet eax, ebx, etc. En moderne 64-bits prosessor har 64-bits registre og de kalles rax, rbx, rcx og rdx og det er disse vi bruker i koden nedenfor. Når denne koden assembles til maskinkode, vil maskinkoden utføre den samme beregningen som maskinkoden i filen `funksjon` i forrige avsnitt som regner ut summen S.

Følgende kode utgjør Assembly-programmet `as.s`:

```
# filnavn: as.s

.globl sum
# C-signatur:int sum ()

# 64 bit assembly

# b = byte (8 bit)
# w = word (16 bit, 2 bytes)
# l = long (32 bit, 4 bytes)
# q = quad (64 bit, 8 bytes)


# Opprinnelige 16bits registre: ax, bx, cx, dx
# ah, al 8 bit
# ax 16 bit
# eax 32 bit
# rax 64 bit

sum:                # Standard

mov   $3, %rcx      # 3 -> rcx, maks i løkke
mov   $1, %rdx      # 1 -> rdx, tallet i økes med for hver runde
mov   $0, %rbx      # 0 -> rbx, variabelen i lagres i rbx
mov   $0, %rax      # 0 -> rax, summen = S

# løkke
start: # label
add  %rdx, %rbx # rbx = rbx + rdx (i++)
add  %rbx, %rax # rax = rax + rbx (S = S + i)
cmp  %rcx, %rbx # compare, er i = 3?
jne  start      # Jump Not Equal til start:

ret  # Verdien i rax returneres
```

Assembly-programmet `as.s` utfører nøyaktig det samme som C-programmet `sumFunksjon.c` listet øverst i avsnitt 4.3.3.

Om vi sammenligner med summerings-koden for den simulerte CPU-en i avsnitt 3.7, vil man se at de åtte Assembly-linjene etter `sum:` tilsvarer linje for linje koden der (om man ser bort ifra linjen som inneholder `start:`. Registeret `%rcx` tilsvarer R0, `%rdx` tilsvarer R1, `%rbx` tilsvarer R2 og `%rax` tilsvarer R3. Vi har skrevet programmet slik at summen S lagres i nettopp registeret `%rax` fordi verdien som ligger i nettopp `%rax` er den verdien som returneres til main-funksjonen som utfører kallet på funksjonen `sum()`.

For å kunne kjøre funksjonen vi har skrevet i Assembly-programmet `as.s` må man be gcc-kompilatoren om å assemble den. Det kan man gjøre slik:

```
$ gcc -c as.s -o as
```

Dette gjør at gcc oversetter Assembly-koden til maskinkode og lagrer denne maskinkoden i filen `as`.

Prosessen med å assemble Assembly-kode til maskinkode er mye enklere enn kompilering fordi det er en ganske enkel oversettelse som stort sett skjer linje for linje. For eksempel vil en linje som inneholder instruksjonen ADD ganske enkelt oversettes til oppcode som inneholder hvilket nummer instruksjonen ADD har i x86-arkitekturen etterfulgt av rett rekkefølge på registrene som er involvert. Helt på samme måte som vi gjorde med koden for den simulerte CPUen.

Til slutt kan man så linke maskinkoden i filen `as` sammen med main-maskinkoden for å få et kjørbart program:

```
$ gcc main as -o sum
$ ./sum
Sum = 6
```

på samme måte som med C-programmene, kunne man også gjort disse tre operasjonene, kompilering av main, assembly av `as.s` og linking av de to, i en operasjon:

```
$ gcc sumMain.c as.s -o sum
$ ./sum
Sum = 6
```

Man kunne også skrive hele hovedprogrammet i Assembly, men for å forenkle kodingen, har vi konsentrert oss om kun den koden som utføres av sum-funksjonen.

Et viktig poeng er at maskinkoden `as` laget fra Assembly funksjonelt sett utfører den samme beregningen som maskinkoden `funksjon` som kompilatoren lagde. Men det finnes mange mulige varianter av både Assembly-kode og maskinkode som utfører nøyaktig det som høynivåkoden sier skal gjøres. Men hva som er den optimale maskinkoden som både er raskest og tar minst plass, er langt fra opplagt. Veldig mye forsking og utvikling er blitt brukt på å lage kompilatorer som genererer best mulig maskinkode. Likevel kan gode Assembly-programmerer i noen tilfeller lage enda bedre kode enn en kompilator, spesielt om de har innsikt i nøyaktig hva som er hensikten med programmet.

4.5 Assembly-kode generert av en kompilator

Man kan også be en kompilator om å stoppe kompileringen før den assembler koden til maskinkode. Det kan man med gcc få til med opsjonen -S og den lager da en fil med filendelse s som innholder Assembly-kode som tilsvarer den maskinkoden den ville laget om man bare kompilerte med opsjonen -c.

```
$ gcc -S sumFunksjon.c
$ cat sumFunksjon.s
.file "sumFunksjon.c"
.text
.globl sum
.type sum, @function
sum:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $0, -8(%rbp)
```

```
movl $0, -4(%rbp)
jmp .L2
.L3:
movl -4(%rbp), %eax
addl %eax, -8(%rbp)
addl $1, -4(%rbp)
.L2:
cmpl $3, -4(%rbp)
jle .L3
movl -8(%rbp), %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size sum, .-sum
.ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0 20160609"
.section .note.GNU-stack,"",@progbits
$
```

5 Forelesning 12/2-24(2 timer). C, maskinkode og pipelining

`Avsnitt fra Tanenbaum: 1.3.2 - 1.3.7`

Slides brukt i forelesningen[96]

Opptak av forelesningen:

os5time1.mp4[97] (48:26) Uredigert opptak av første time av forelesningen.
os5time2.mp4[98] (48:09) Uredigert opptak av andre time av forelesningen.

Opptak av forelesningen inndelt etter temaer:

os5del1.mp4[99] (03:33) Intro om oblig-innlevering, godkjentkrav for grupper med kun en student, VMer
os5del2.mp4[100] (01:25) dagens tema, innledning
os5del3.mp4[101] (14:46) Demo: Optimalisering av maskinkode ved kompilering med gcc -O, fibo()
os5del4.mp4[102] (03:30) Demo: Optimalisering av SumFunksjon.c med gcc -O
os5del5.mp4[103] (09:35) Demo: En linje høynivåkode kan bli til flere linjer maskinkode
os5del6.mp4[104] (05:00) Demo: Assemblykode for enlinje()-funksjonen (NB! Må nå kompileres med `gcc -no-pie`)
os5del7.mp4[105] (02:09) Demo: Hva skjer om man prøver å addere to RAM-variabler med en instruksjon?
os5del8.mp4[106] (02:32) Beskjed fra Ine: vi gir kommentarer til obligene, les dem!
os5del9.mp4[107] (01:08) Spørsmål: Hva betyr .quad?
os5del10.mp4[108] (09:01) Tegning og forklaring om CPU, registere og RAM
os5del11.mp4[109] (18:17) Demo: En if-test i Assembly skrevet fra scratch (NB! Må nå kompileres med `gcc -no-pie`)
os5del12.mp4[110] (01:07) Demo: Hvorfor er main() av typen int? Eksempel med returverdi fra C-program.
os5del13.mp4[111] (01:22) Slides: Forenklinger ved CPU-simuleringen
os5del14.mp4[112] (01:41) Slides: CPU-løkke (hardware-nivå)
os5del15.mp4[113] (05:15) Slides: Pipelining
os5del16.mp4[114] (06:53) Slides: Superscalar arkitektur

5.1 Maskinkode optimalisert for å kjøre hurtigst mulig

For gcc-kompilatoren er default virkemåte at den skal kompilere hurtigst mulig, det vil si at selve kompileringen skal gå så fort som mulig. Det er vanlgivis ønskelig når man utvikler et program, slik at man

[96] https://www.cs.oslomet.no/ haugerud/os/Forelesning/os5.pdf
[97] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5time1.mp4
[98] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5time2.mp4
[99] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del1.mp4
[100] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del2.mp4
[101] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del3.mp4
[102] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del4.mp4
[103] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del5.mp4
[104] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del6.mp4
[105] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del7.mp4
[106] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del8.mp4
[107] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del9.mp4
[108] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del10.mp4
[109] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del11.mp4
[110] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del12.mp4
[111] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del13.mp4
[112] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del14.mp4
[113] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del15.mp4
[114] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os5del16.mp4

minimaliserer ventetiden før man kan prøvekjøre siste versjon. Når man derimot har et helt ferdig versjon, er det mest naturlig å be kompilatoren å lage kode som kjører raskest mulig og er mest mulig effektiv. Generelt vil man med opsjonen -O be gcc om å lage så hurtig kode som mulig; altså lage et program som utfører beregningen den skal gjøre som raskt som mulig.

Følgende er en C-funksjon som regner ut tall nummer 'last' i Fibinacci-rekken:

```
int fibo(int last)
{
   int i;
   int a=1,b=1,c;
   /* Har allerede de første to */
   for(i=3;i <= last;i++)
       {
  c = a;      /* b skal etterpå få denne */
  a = a + b;  /* Neste tall */
  b = c;      /* b fortsatt nest siste tall */
       }
   return(a);
}
```

Hvis man kompilerer denne koden med `gcc -S fibo.c` for å se hva slags maskinkode kompilatoren vil lage, får man følgende:

```
movl %edi, -20(%rbp)
movl $1, -12(%rbp)
movl $1, -8(%rbp)
movl $3, -16(%rbp)
jmp .L2
.L3:
movl -12(%rbp), %eax
movl %eax, -4(%rbp)
movl -8(%rbp), %eax
addl %eax, -12(%rbp)
movl -4(%rbp), %eax
movl %eax, -8(%rbp)
addl $1, -16(%rbp)
.L2:
movl -16(%rbp), %eax
cmpl -20(%rbp), %eax
jle .L3
movl -12(%rbp), %eax
popq %rbp
ret
```

Her kan man se at adderingsoperasjonene utføres direkte på variablene som ligger i RAM, de flyttes ikke først inn i registerne for så å utføre regneoperasjonene internt inne i CPU-en. Det finnes forøvrig ingen X86-instruksjon som direkte legger sammen to tall som ligger i RAM og så lagrer resultatet i RAM etterpå, derfor må minst ett av tallene i en addisjon ligge i RAM. Men hvis man istedet kompilerer denne koden med opsjonen -O: `gcc -O -S fibo.c` får man følgende resultat:

```
movl $1, %esi
movl $1, %ecx
movl $3, %edx
jmp .L3
```

```
.L5:
movl %eax, %ecx
.L3:
leal (%rcx,%rsi), %eax
addl $1, %edx
movl %ecx, %esi
cmpl %edx, %edi
jge .L5
rep ret
```

Her ser vi at alle beregningene skjer i registerne og dette gir kode som raskere leverer sluttresultatet.

Hvis man gjør dette med vår funksjon, sumFunksjon.c, får man en meget kort Assembly-kode som resultat:

```
$ gcc -O -S sumFunksjon.c
$ cat sumFunksjon.s
.file "sumFunksjon.c"
.text
.globl sum
.type sum, @function
sum:
.LFB0:
.cfi_startproc
movl $6, %eax
ret
.cfi_endproc
.LFE0:
.size sum, .-sum
.ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0 20160609"
.section .note.GNU-stack,"",@progbits
$
```

Hvorfor er koden så kort og hva er den meget effektive beregningen kompilatoren har funnet frem til?

5.2 En linje høynivåkode kan gi flere linjer maskininstruksjoner

Vi tar utgangspunkt i følgende C-program `main.c` som kaller en ekstern funksjon `enlinje()`:

```
#include <stdio.h>

extern int enlinje();

int main (void) {

int svar;
printf("Kaller enlinje()...\n");
svar = enlinje();
printf("Svar = %d\n", svar);
}
```

Denne funksjonen, her lagret i filen `enlinje.c`, legger sammen to variabler som er lagret i RAM og som heter svar og memvar og returnerer svaret:

```
int enlinje()
{
   int svar = 32;
   int memvar = 10;

   svar = svar + memvar;

   return(svar);
}
```

Vi skal nå se at en enkelt C-instruksjon som linjen

```
svar = svar + memvar;
```

ikke nødvendigvis fører til en enkelt linje med maskinkode. I dette tilfellet er det faktisk ikke mulig å gjøre denne operasjonen med en linje maskinkode, fordi det ikke finnes noen x86-instruksjon som kan utføre denne operasjonen.

Om man prøver å legge sammen to variabler som ligger i internminnet(RAM), slik som dette

```
add memvar, svar     # svar = svar + memvar
```

får man følgende feilmelding

```
Error: too many memory references for ‘add’
```

fordi det x86-instruksjonen add ikke kan operere på to referanser i minnet samtidig. Det ville tatt for lang tid og en slik instruksjon finnes derfor ikke. Man må først hente inn en av variablene fra minnet og det må også koden en kompilator lager gjøre.

Følgende assembly-fil, en.s, inneholder assemblykode som gjøre det samme som enlinje.c:

```
.globl enlinje
# C-signatur:int enlinje ()

enlinje:        # Standard start av funksjon

mov memvar, %rbx # Man trenger to linjer kode for å
add %rbx, svar   # gjøre en høynivålinje svar = svar + memvar
mov svar, %rax   # Returnerer svar

ret  # Verdien i rax returneres

# Følgende avsnitt av koden viser hvordan man definerer
# variabler som lagres i minnet.
# Andre linje tilsvarer linjen
# int svar=32;
# i et C-program
# Dette avsnittet kunne også stått øverst i filen

.data
svar:   .quad 32   # deklarerer variabelen svar i RAM
memvar: .quad 10   # 8 byte = 64 bit variable
```

C-koden main.c i starten av avsnittet kan brukes for å kalle assembly-funksjonen over med

```
gcc -no-pie main.c en.s
```

Hvis man kompilerer enlinje med

```
gcc -S enlinje.c
```

Får man følgende kode:

```
.file "enlinje.c"
.text
.globl enlinje
.type enlinje, @function
enlinje:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $32, -8(%rbp)
movl $10, -4(%rbp)
movl -4(%rbp), %eax
addl %eax, -8(%rbp)
movl -8(%rbp), %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size enlinje, .-enlinje
.ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609"
.section .note.GNU-stack,"",@progbits
```

Hvordan utføres addisjonen her?

NB! Fra og med 2022 har default konfigurasjon av kompilatoren gcc endret seg, slik at man nå må legge på opsjonen -no-pie for å kompilerere assembly-kode som deklarere variabler i et data-segment. Derfor må man kompilere med

```
gcc -no-pie main.c en.s
```

og også bruke `gcc -no-pie` når man loader sammen en slik kompilert assembly-fil med en kompilert C-fil. Hvis ikke får du en feilmelding om relocation R_X86_64_32S against '.data' can not be used.

5.3 If-test

Følgende kode viser hvordan en if-test kan lages i assembly. Tilsvarende som for while- og for-løkker, må man ha en test som hopper til et annet sted i koden avhengig av resultatet av testen.

```
.globl iftest
# C-signatur:int iftest ()

iftest:        # Standard start av funskjon

# Returnerer 1 hvis svar > 42, ellers 0
# if(svar > 42){
#    return(1):
# }
# else{
#    return(0);
#}

mov $42, %rbx

cmp  %rbx, svar # compare
jg  greater     # Jump Greater, hvis svar > 42

mov $0, %rax    # 42 eller mindre hvis her
jmp return

greater:
mov $1, %rax

return:
ret  # Verdien i rax returneres

.data
svar:  .quad 40   # deklarerer variabelen svar i RAM
```

5.4 Forenklinger ved CPU Simuleringen

Iforhold til vår forenklede simulering er de fleste CPUer som Intel og AMD mer komplekse:

- Instruksjoner bruker mer tid enn en CPU-sykel på å utføres
- Hver instruksjon hentes inn fra RAM før den utføres (ikke ROM)
- En x86-instruksjon deles inn i flere små deler (mikro-operasjoner), uops

5.4.1 CPU-løkke (hardware-nivå)

En datamaskin med en CPU gjennomfører en evigvarende løkke som utfører en maskininstruksjonen av gangen helt til maskinen skrus av. En demo av en CPU-løkke (uten interrupts) kan sees her, men krever flash.[115].

Pseudo-kode for den evige hardware-løkken som CPU-en kjører:

```
while(not HALT)
{
   IR = mem[PC];   # IR = Instruction Register
   PC++;           # PC = Program counter
   execute(IR);
```

[115]https://nexus.cs.oslomet.no/ haugerud/os/demoer/iecycle.swf

```
    if(InterruptRequest)
    {
       savePC();
       loadPC(IRQ); # IRQ = Interrupt Request
                    # Hopper til Interrupt-rutine
    }
}
```

5.5 Pipelining

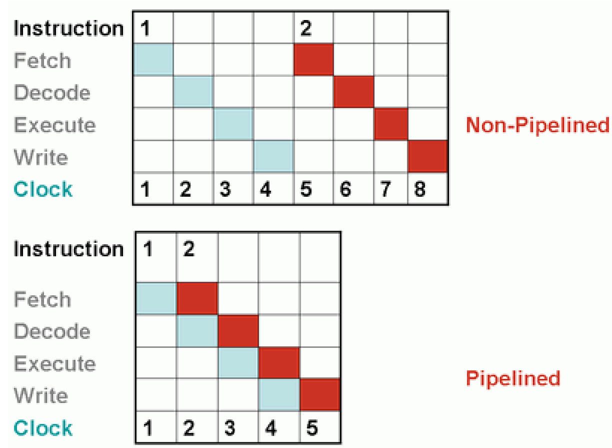
En instruksjon kan deles inn i flere deler, stages, 14 er vanlig i Intel-CPUer.

Eksempel med 4 stages:

- Fetch (hent instruksjonen fra RAM)
- Decode (hvilke knapper skal trykkes på i ALU og Datapath)
- Execute (utfør instruksjonen)
- Write (skriv resultater til RAM)

Tid spares ved at neste instruksjon starter før den første er ferdig.

5.6 Pipelining

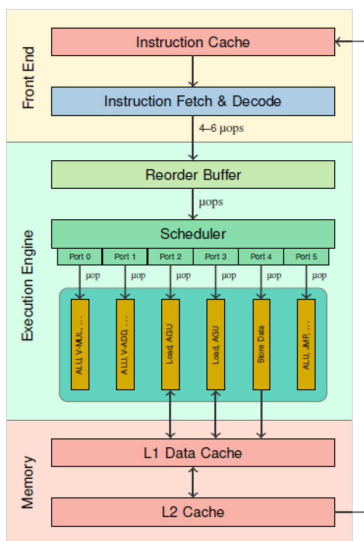


5.7 Intel mikroarkitekturer

En mikroarkitektur er hvordan et instruksjonsett er implementert i en CPU.

År	arkitektur (CPU)	pipeline stages	Max MHz	nm
1978	8086	2	5	3000
1985	486	5	33	1000
1995	P6 (Pentium Pro)	14	450	250
2000	NetBurst (Pentium 4)	20	2000	180
2004	NetBurst (Pentium 4)	31	3800	90
2011	Sandy Bridge (core i7)	14	4000	32
2015	Skylake (core i7)	14	4200	14
2019	Cascade Lake (core i9)	14	4400	14

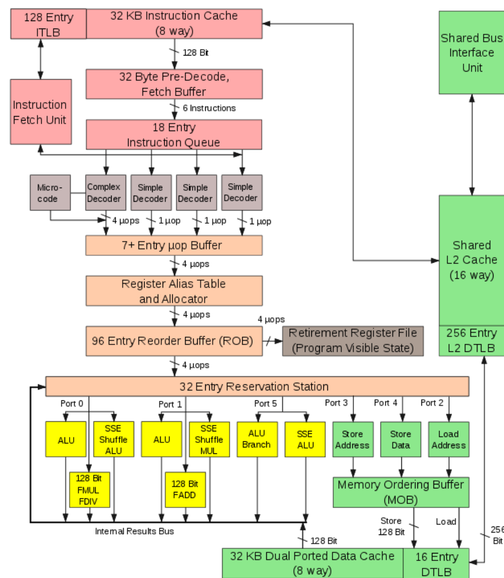
5.8 Superscalar arkitektur



5.9 Superscalar arkitektur

- En skalær prosessor utfører instruksjoner en for en (som simuleringen)
- En superskalær prosessor har flere parallelle enheter som utfører mikro-operasjoner
- For eksempel 2 ALU-er, 1 FPU, load, store
- Utfører operasjoner samtidig
- Operasjoner kan utføres out-of-order (i en annen rekkefølge enn det sekvensielle programmet tilsier)

5.10 Intel Core 2



Intel Core 2 Architecture

6 Forelesning 26/2-24(2 timer). Branch prediction, Multitasking

`Avsnitt fra Tanenbaum: 1.3-1.6`

Slides brukt i forelesningen[116]

6.1 Forelesningsvideoer

Opptak av forelesningen:

os6time1.mp4[117] (43:50) Uredigert opptak av første time av forelesningen.
os6time2.mp4[118] (54:03) Uredigert opptak av andre time av forelesningen.

Opptak av forelesningen inndelt etter temaer:

os6del1.mp4[119] (05:05) Praktisk info, MC1, oblig2
os6del2.mp4[120] (03:31) Intro om Linux-VMer (Docker containere)
os6del3.mp4[121] (07:10) Demo: Innlogging på Linux-VMer (Docker containere)
os6del4.mp4[122] (02:04) Intro, hva vi gjorde sist
os6del5.mp4[123] (08:08) Slides: Branch prediction og Meltdown

[116] https://www.cs.oslomet.no/ haugerud/os/Forelesning/os6.pdf
[117] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6time1.mp4
[118] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6time2.mp4
[119] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del1.mp4
[120] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del2.mp4
[121] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del3.mp4
[122] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del4.mp4
[123] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del5.mp4

os6del6.mp4[124] (13:19) Demo: branch prediction i praksis
os6del7.mp4[125] (04:21) Spørsmål, hvordan få os-passord og logge seg inn på VM (repetisjon)
os6del8.mp4[126] (03:06) Slides: Oppsummering av datamaskinarkitektur
os6del9.mp4[127] (09:29) Slides: OS-historie, Microsoft OS
os6del10.mp4[128] (02:45) Slides: OS-historie, Unix OS
os6del11.mp4[129] (09:19) Slides: Intro til multitasking
os6del12.mp4[130] (06:18) Slides: Context Switch og PCB (Prosess Control Block)
os6del13.mp4[131] (03:30) Slides: Multitasking i praksis
os6del14.mp4[132] (08:42) Demo: Multitasking i praksis
os6del15.mp4[133] (01:09) Spørsmål: Hvilke prosesser er ikke CPU-avhengige?

6.2 Sist

- En C-linje kan gi mange linjer assembly/maskinkode
- Kompilering av høynivåkode til maskinkode er komplisert og ikke entydig
- Kompilering av assembly til maskinkode er direkte og entydig
- En enkelt assembly instruksjon er delt inn i mikro-operasjoner og utføres i parallelle pipelines

6.3 branch prediction

- Ved en branch i programmet (if-test), vet man ikke hva neste instruksjon er
- Stort problem for pipelining, må vente på resultatet fra forrige instruksjon
- Gjetter, basert på erfaring, hvilken branch (gren) som følges i programmet og utfører den
- Speculative execution, må gjøres om hvis feil
- I et superskalær arkitektur kan begge grener delvis utføres på forhånd

Følgende `C++` program inneholder kode som viser hva konsekvensene av branch prediction kan være. I starten av programmet lages et data-array med tilfeldig trukkede heltall mellom 0 og 255. De 10 første tallene blir skrevet ut og så gjentas en ytre løkke 100.000 ganger for at man skal få mer nøyaktige målinger av hvor lang tid den indre løkken tar. Den indre løkken består av at man går igjennom hvert element i det store arrayet og legger til verdien `data[c]` til en variable sum hvis verdien er større enn 127. I praksis vil dette skje omtrent halvparten av gangene.

```
#include <algorithm>
#include <iostream>

using namespace std;
```

[124] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del6.mp4
[125] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del7.mp4
[126] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del8.mp4
[127] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del9.mp4
[128] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del10.mp4
[129] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del11.mp4
[130] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del12.mp4
[131] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del13.mp4
[132] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del14.mp4
[133] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os6del15.mp4

```
int main()
{
   // Lager et data-array
   int i,c;
   int arraySize = 32768;
   int data[arraySize];

   for (c = 0; c < arraySize; ++c)
     {
       data[c] = rand() % 256;
     }

   // Gir tilfeldig tall mellom 0 og 255
   // Gir samme array med tall for hver kjøring

   // sort(data, data + arraySize);
   // sorterter data-arrayet

   // Skriver ut de 10 første verdiene
   for (c = 0; c < 10; c++)
     cout << data[c] << "\n";

   // Legger sammen alle tall større enn 127
   long sum = 0;

   // Ytre løkke for at det skal ta litt tid...
   for (i = 0; i < 100000; ++i)
     {
        // Indre løkke
        for (c = 0; c < arraySize; ++c)
           {
             if (data[c] > 127)
             sum += data[c];
           }
     }

   cout << "sum = " << sum << "\n";
}
```

Deretter kompileres C++ programmet og kjøres på en Linux-maskin:

```
$ g++ b.cpp
$ time ./a.out
103
198
105
115
81
255
74
236
41
205
sum = 314931600000
Real:17.095 User:17.096 System:0.000 100.00%
```

Kommandoen `time` tar tiden på programmet som kjøres og gir som resultat at programmet har brukt

17.096 sekunder CPU-tid og at det har brukt CPU-en hele tiden (100%). Utskriften av de 10 første tallene viser at verdiene kommer i en tilfeldig rekkefølge og at de er over og under 127, slik at if-testen vil slå til en gang i blant og i gjennomsnitt ca halvparten av gangene.

Deretter gjøres en enkelt endring på koden ved at kommentartegnet foran linjen

```
sort(data, data + arraySize);
```

fjernes, slik at data-arrayet blir sortert før programmet kjører de to løkkene. Dette vil endre rekkefølgen for når if-testen slår til og dataene adderes til sum-variabelen, men det vil skje nøyaktig like mange ganger og som vi ser blir også summen nøyaktig den samme:

```
0
0
0
0
0
0
0
0
0
sum = 314931600000
Real:6.285 User:6.280 System:0.000 99.91%
```

Utskriften av de første 10 elementene viser at data-arrayet er sortert og starter med alle 0-verdiene. Men det overraskende er at denne kjøringen går mer en dobbelt så fort og nesten bare tar en tredjedel av tiden til den første kjøringen. Og dette til tross for at nøyaktig de samme instruksjonene blir utført i begge tilfeller og det samme regnestykket gir samme resultat. Hva kan dette skyldes?

6.4 Meltdown

- Et hardware-sikkerhetshull funnet i 2018
- Rammet Intel, ARM og IBM-prosessorer
- Meltdown utnytter at både koden som sjekker om prosessen kan lese fra RAM og lesingen fra RAM delvis utføres
- Meltdown kan dermed lese data fra andre prosesser som er cache't men ennå ikke fjernet pga feil branch
- Spectre brukte lignende metoder til å lese passord og sensitive data
- Betegnet som sikkerhets-katastrofe
- Både CPU design og operativsystemer ble endret for å hindre Meltdown og Spectre i å virke

6.5 Viktig å huske fra datamaskinarkitektur

På veien videre er det viktigste å huske fra datamaskinarkitektur at alt CPU-en gjør er å slavisk utføre maskininstruksjoner en for en i en evigvarende løkke; ihvertfall til maskinen skrus av. Legg også merke til at det ikke er noen en til en forbindelse mellom instruksjoner i høynivåkode og maskinkode. En linje kode i høynivåspråk fører ofte til mange maskininstruksjoner i det kompilerte programmet.

6.6 OS historie

Det er nyttig å vite litt om historien til noen av de mest brukte operativsystemene.

6.6.1 Microsoft Desktop-OS

MS-DOS 1981, 16-bit

Windows 1.0 i 1985, 3.0 i 1990, GUI på toppen av DOS

Windows 95 Noe 32-bit kode, mye 16-bit Intel assembler,DOS-filsystem, bruker DOS til å boote

Windows 98 essensielt som 95, desktop/Internett integrert

Windows Me essensielt som 98, mer multimedia og nettverk support

Windows 2000 Første (ikke så vellykkede) forsøk med Desktop OS basert på NT (5.0).

Windows XP Oktober 2001, Desktop-OS som kombinerer NT 5.1 kode med Win 9x. Home edition: 1 CPU, XP Professional: 2CPU-er, logge inn utenfra, 32 og 64 bit

Windows Vista januar 2007. Kernel NT 6.0, Booter raskere, bedre filsøk, Ingen suksess.

Windows 7 oktober 2009, kjernen er Windows NT 6.1, PowerShell 2.0 default, 7 editions, Service Pack 1

Windows 8 oktober 2012, NT 6.2, Start Screen, touch screen, USB 3.0, Windows Store, Windows RT for ARM

Windows 8.1 oktober 2013, NT 6.3

Windows 10 July 2015, NT 10.0(!), Microsoft Edge, virtual desktops, native Ubuntu bash shell(samarbeid med Canonical) gjennom Windows Subsystem for Linux

Windows 11 October 2021, NT 10.0(!), ikke lenger støtte for 32-bit x86 CPUs, Internet Explorer ikke inkludert

Intels første 32-bit maskin var 386 fra 1985. Generelt problem før XP: Windows er bakover-kompatibelt til DOS, alle Win-prosesser kan ødelegge for kjernen og ta ned OS.

6.6.2 Microsoft Server-OS

NT 3.1 1993 32-bit, skrevet fra scratch i C (lite assembler), David Cutler (VAX-VMS designer), mye bedre sikkerhet og stabilitet enn Windows. 3.1 millioner linjer kode.

NT 4.0 1996, Samme GUI som Win-95, 16 millioner linjer, portabelt `->` alpha, PowerPC

Windows 2000 NT 5.0, opp til 32 CPU'er, features fra Win-98, Plug and Play, `\winnt\system32\ntoskrnl.exe`. 29 millioner linjer. *MS-DOS borte, men 32-bits kommando-interface med samme funksjonalitet.*

Windows Server 2003 bygger på 2000 server, NT 5.2, design med tanke på .NET: web, XML, C#, 32 og 64 bit, SP1, SP2, R2 i 2006

Windows Server 2008 februar 2008, felles basis med Vista, første OS med PowerShell, Kan installeres som **Server core** og styres fra CLI (Command Line Interface), Hyper-V virtualisering

Windows Server 2008 R2 oktober 2009, NT 6.1 (som Win 7), PowerShell 2.0 default, kun 64 bit

Windows Server 2012 september 2012, NT 6.2 (som Win 8), cloud computing, oppdatert Hyper-V, nytt filsystem: ReFS

Windows Server 2012 R2 oktober 2013, NT 6.3 (som Win 8.1)

Windows Server 2016 september 2016, NT 10.0, Windows Defender, nano server: uten gui, fjernstyres med PowerShell

Windows Server 2019 oktober 2018, NT 10.0, Windows Admin Center

Windows Server 2022 August 2021, NT 10.0.2

6.6.3 Unix operativsystemer

Dagens Unix-versjoner har utviklet seg fra to som dominerte rundt 1980:

- system V (AT&T Bell Labs)
- BSD (University of California at Berkely)

De fleste av dagens varianter er bygd på SVR4 som er en blanding. Følgende er kommersielle 64 bit Unix-OS for RISC-prosessorer som var dominerende i server-markedet et stykke inn på 2000-tallet:

OS	Eier	hardware
AIX	IBM	RS6000, Power
Solaris	Sun	Sparc, intel-x86
HP-UX	Hewlett-Packard	PA-RISC, Itanium(IA-64)
Tru64 UNIX(Digital Unix)	HP(Compaq(DEC))	Alpha
IRIX	Silicon Graphics	SGI

Oracle stanset vidreutviklingen av Solaris, det største av Unix-operativsystemene, i 2017. Idag er det x86 servere som dominerer og de fleste med Linux. Frie Unix-kloner for mange plattformer:

OS	hardware
FreeBSD	x86, Alpha, Sparc
OpenBSD	(sikkerhet) x86, Alpha, Sparc, HP, PowerPC, mm
NetBSD	x86, Alpha, Sparc, HP, PowerPC (Mac), PlayStation, mm
Darwin	(basis for Mac OS X og iOS, kjernen, XNU, bygger på FreeBSD og Mach 3 microkernel) , intel x86, ARM, Powe
Linux	x86, Alpha, Sparc, HP, PowerPC, PlayStation 3, Xbox, stormaskin, mm

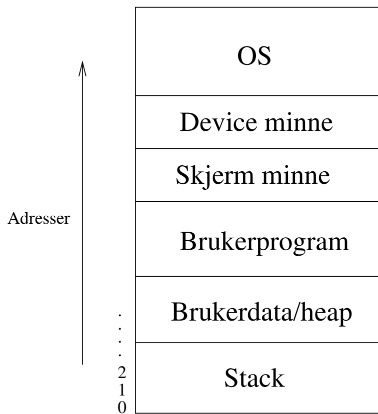
6.6.4 Interrupts (avbrytelser)

- Signal fra hardware
- CPU-en avbrytes for å håndtere signalet
- Lagrer adressen til neste instruksjon på stack og hopper til interrupt-rutinen
- Hvert interrupt-nr (IRQ) har sin rutine

6.7 Singletasking OS

Basis for flerprosess-systemer.

6.7.1 Internminne-kart



Stack: brukes bl. a. til å lagre adressen som skal returneres til ved subrutinekall.

6.8 Multitasking-OS

For å lage et system som kan kjøre n programmer samtidig, må vi få en enprosess maskin til å se ut som n maskiner.

Bruker software til å fordele tid mellom n programmer og å dele ressurser; minne, disk, skjerm etc. OS-kjernen utfører denne oppgaven.

Samtidige prosesser må tildeles hver sin del av minne:

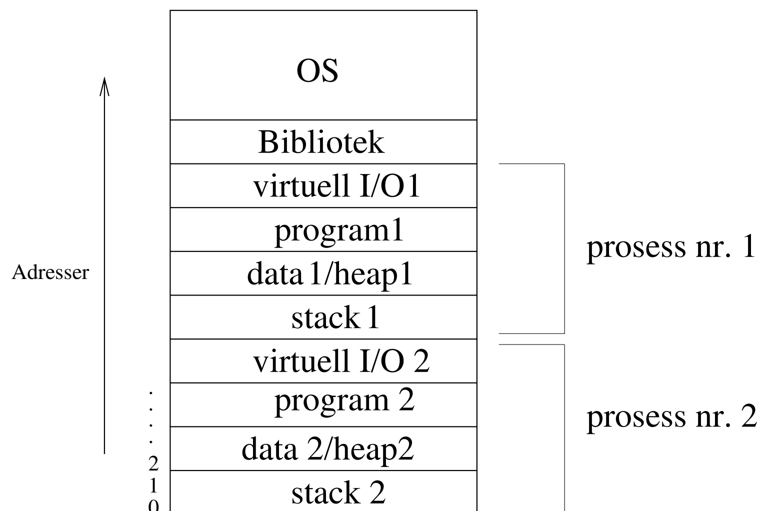


Figure 38: Minnekart for et multitasking system

6.9 Multitasking

Multitasking gjør at man kan kjøre flere programmer samtidig selvom man bare har en CPU. I prinsippet er CPU-en meget enkel på den måten at den gjør en og en maskinistruksjon av gangen. Som for eksempel

å legge sammen to tall, å sammenligne to bit-strenger (1010001101101110 = 1010001100101110?) eller å lagre en streng med binære tall i internminnet(RAM). Et multitasking operativsystem får det til å se ut som om mange programmer kan kjøre samtidig ved å dele opp tiden i små biter (timeslices) og la hver prosess som kjører få en bit CPU-tid (typisk et hundredels sekund) av gangen i et køsystem (såkalt **Round Robin** kø). Metoden som alle moderne OS bruker er Preemptive multitasking. Metoden består i at en hardware timer (klokke) jevnlig sender et interrupt-signal som gjør at første OS-instruksjon legges inn i CPU-en. Dermed unngås det at vanlige brukerprosesse tar over kontrollen. OS lar hver prosess etter tur bruke CPU-en i et kort tidsintervall. Alle prosesser ser da ut til å kjøre samtidig. Når OS switcher fra prosess P1 til prosess P2 utføres en såkalt Contex Switch (kontekst svitsj).
Typisk tid for context-switch: 0.001 ms (ms = millisekunder = tusendels sekund). Timeslice = 10 ms

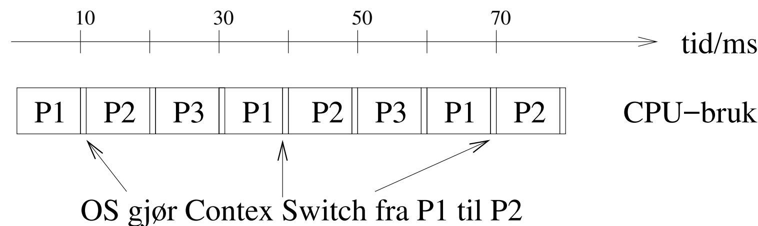


Figure 39: Prosessene P1, P2 og P3 kjører samtidig under et multitasking OS. En Context Switch utføres hver gang en prosess gis CPU-tid.

for Linux på Intel.

6.10 PCB -Process Control Block

Process Control Block (PCB) er Prosessens tilstandsbeskrivelse: prioritet, prosessormodus, minne, stack, åpne filer, I/O, etc. PCB inneholder bl. a. følgende:

- CPU registre
- pekere til stack
- prosesstilstand (sleep, run, ready, wait, new, stopped)
- navn (PID)
- eier (bruker)
- prioritet (styrer hvor mye CPU-tid den får)
- parent prosess
- ressurser (åpne filer, etc.)

6.11 Timesharing og Context Switch

CPU-scheduling = å fordele CPU-tid mellom prosessene = Time Sharing
Metoden som alle moderne OS bruker er Preemptive multitasking med en **Round Robin** kø. OS lar hver prosess etter tur bruke CPU-en i et kort tidsintervall (timeslice). Alle prosesser ser da ut til å kjøre samtidig. Når OS switcher fra prosess P1 til prosess P2 utføres en Contex Switch. Typisk tid for context-switch: 0.001 ms. Timeslice = 10 ms for Linux på Intel.

All PCB-info må lagres i en Context Switch $\longrightarrow$ *tar tid* $\longrightarrow$ *systemoverhead*

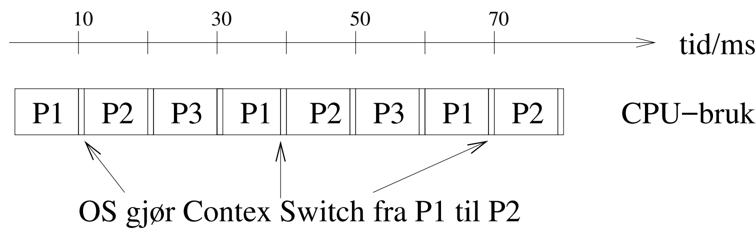


Figure 40: Prosessene P1, P2 og P3 kjører samtidig under et multitasking OS. En Context Switch utføres hver gang en prosess gis CPU-tid.

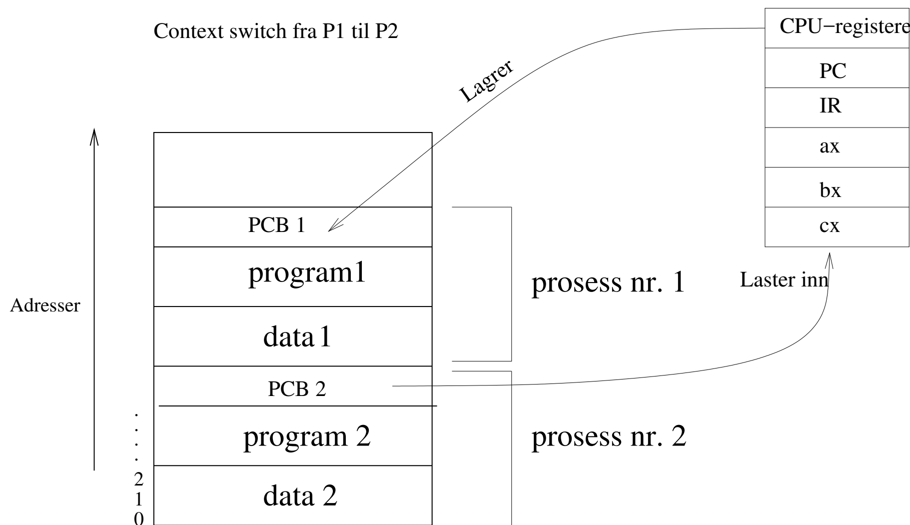


Figure 41: CPU info lagres i PCB ved en Context Switch

6.12 Multitasking i praksis, CPU-intensive programmer

Et program som oversetter kildekode til maskinkode (kompilator) eller et program som hele tiden regner med tall, vil bruk så mye CPU-tid som det klarer å få tak i. Prosesser som kjører slike programmer kalles CPU-intensive. De fleste vanlige programmer som browsere, tekstbehandlingsprogrammer, tengeprogram etc. bruker lite CPU og det er dette som gjør at multitasking av hundretalls samtidige prosesser går helt greit uten at brukeren oppfatter datamaskinen som treg. Vi skal nå se hva som skjer når vi kjører flere instanser av et mulititasking program på systmer med en eller flere CPU'er.

Programmet vi bruker er et lite shell-script som står i en løkke og regner og regner. Da vil det hele tiden ha behov for CPU-en. Siden prosessen aldri har behov for å vente på data fra disk, tastatur eller andre prosesser, kan den regne uten stans. Programmet heter `regn` og ser slik ut:

```
#! /bin/bash

# regn (bruker CPU hele tiden)

(( max = 100000 ))
(( i = 0  ))
(( sum = 0  ))

echo $0 : regner....
while (($i < $max))
```

```
do
        (( i += 1 ))
        (( sum += i  ))
done
echo $0, resultat: $sum
```

6.13 Multitasking eksempel

Bare rene regneprosesser bruker CPU hele tiden. Vanlige prosesser venter mye på I/O (Input/Output fra disk, nettverk etc.) og multitasking gir da mer effektiv utnyttelse av CPU.

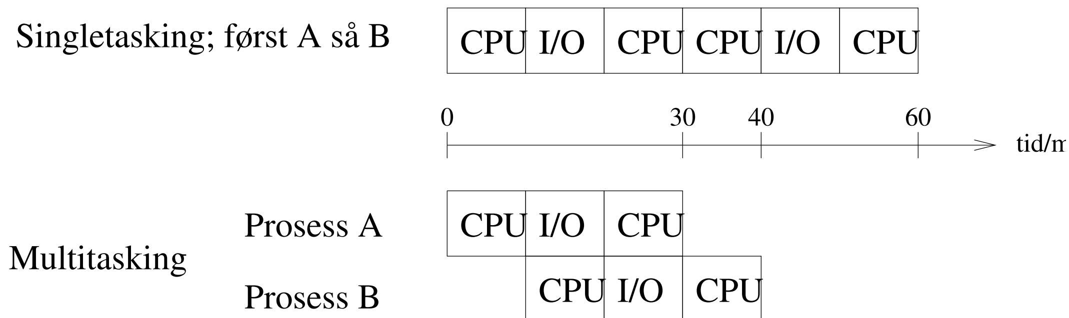


Figure 42: Prosessene A og B kjørt med single og multitasking

6.14 CPU-intensiv prosess på system med èn CPU

Med kommandoen `lscpu` kan man hente ut mye nyttig informasjon om cpu og cache:

```
user@chokeG7:~$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
Address sizes:       40 bits physical, 48 bits virtual
CPU(s):              1
On-line CPU(s) list: 0
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):           1
NUMA node(s):        1
Vendor ID:           AuthenticAMD
CPU family:          15
Model:               65
Model name:          Dual-Core AMD Opteron(tm) Processor 2216
Stepping:            3
CPU MHz:             2400.114
BogoMIPS:            4800.22
Hypervisor vendor:   Xen
Virtualization type: full
L1d cache:           64K
L1i cache:           64K
L2 cache:            1024K
```

```
NUMA node0 CPU(s):   0
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2
```

Dette viser at denne Linux-maskinen har èn enkelt 64-bits CPU med en klokkefrekvens på 2.4 GHz. Utskriften viser også at dette er en virtuell maskin, man kan se at den er virtualisert med Xen utifra de to linjene

```
Hypervisor vendor:   Xen
Virtualization type: full
```

Videre kan man se størrelsen på L1 og L2 cache. En rask og tydelig oversikt kan man få med kommandoen `lstopo`:

```
lstopo --no-io
```

som gir følgende figur

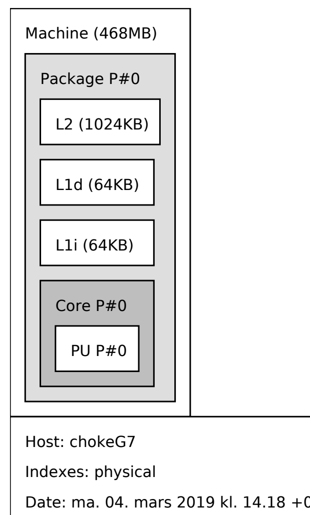


Figure 43: CPU-topologi generert av lstopo den virtuelle maksinen chokeG7.

Vi starter en instans av programmet `regn` som er CPU-intensivt og bruker så mye CPU det kan få:

```
mroot@chokeG7:~$ ./regn
./regn, resultat: 3125001250000
```

Samtidig startes `top` og man kan se at prosessen med PID (Process ID) 18908 forsyner seg grovt av CPU-en og klarer å karre til seg 99.3% av CPU-tiden. Verdien som vises er gjennomsnittsverdien for de siste 3 sekunder.

```
top - 14:32:39 up 10 days, 23:56,  2 users,  load average: 0,09, 0,12, 0,05
Tasks:  74 total,   1 running,  73 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,0 us,  0,7 sy,  0,0 ni, 99,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,3 st

PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
18908 mroot     20   0    9504   3244   3008 R  99,3   0,7   0:07.16 regn
18903 mroot     20   0   16668   4668   3536 S   0,3   1,0   0:00.04 sshd
```

Vi starter så en instans til av programmet som regner i vei, uavhengig av den første. Da ser vi at OS fordeler CPU-en likt mellom de to prosessene og de får i underkant av 50% hver av CPU-tiden.

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
18912 mroot     20   0    9504   3224   2988 R  49,8   0,7   0:13.14 regn
18913 mroot     20   0    9504   3312   3080 R  49,8   0,7   0:12.84 regn
```

Dette betyr at reelt sett er det til enhver tid bare en prosess som kjører, men det oppleves som om de kjører samtidig fordi OS deler opp tiden i små biter(1-10 hundredels sekunder) og lar dem bruke CPU-en annenhver gang. Husk at for en prosess er ett hundredels sekund lang tid, den kan rekke å utføre millioner av maskininstruksjoner på den tiden.

7 Forelesning 5/3-24(2 timer). Multitasking, cache, hyperthreading

Avsnitt fra Tanenbaum: 1.5 - 1.6

Slides brukt i forelesningen[134]

Opptak av forelesningen:

os7time1.mp4[135] (43:50) Uredigert opptak av første time av forelesningen.
os7time2.mp4[136] (54:03) Uredigert opptak av andre time av forelesningen.

Opptak av forelesningen inndelt etter temaer:

os7del1.mp4[137] (07:03) Velkommen og intro, oppsummering av forrige forelesning
os7del2.mp4[138] (11:11) Demo: CPU intensiv regnejobb på 1, 2 og 4 CPUer
os7del3.mp4[139] (02:11) Spørsmål: i top virker det ikke å taste f. Demo på Linux-VM, virker der
os7del4.mp4[140] (01:28) Spørsmål: Er en regne-enhet en ALU? Ja,...
os7del5.mp4[141] (02:12) Demo: CPU-fordeling på VM/containere
os7del6.mp4[142] (06:36) Slides: Internminne og Cache, Minnepyramiden
os7del7.mp4[143] (02:18) Slides: SRAM og DRAM
os7del8.mp4[144] (06:00) Slides: L1 og L2 Cache
os7del9.mp4[145] (03:20) Spørsmål: Hvorfor er ikke Real = User + System for time-kommandoen?
os7del10.mp4[146] (01:14) Poll: Hvis en CPU-avhengig prosess bruker 18 sekunder på en CPU, hvor lang tid bruker da 5 prosesser på 4 CPU-er?
os7del11.mp4[147] (04:57) Poll: Utregning av svar og kjøring av eksperiment
os7del12.mp4[148] (06:16) Slides: Multitasking og Multiprocessing, Multiprosessor og Multicore, Intel Core og AMD K10
os7del13.mp4[149] (04:08) Demo: amdock, serveren med 96 CPU-er som drifter VM-containerene
os7del14.mp4[150] (03:34) Slides: Hyperthreading
os7del15.mp4[151] (10:51) Demo: Hyperthreading på Linux-desktop rex med 8 CPU-er (eller 4?)
os7del16.mp4[152] (08:02) Demo: Hyperthreading og taskset

[134]https://www.cs.oslomet.no/ haugerud/os/Forelesning/os7.pdf
[135]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7time1.mp4
[136]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7time2.mp4
[137]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del1.mp4
[138]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del2.mp4
[139]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del3.mp4
[140]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del4.mp4
[141]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del5.mp4
[142]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del6.mp4
[143]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del7.mp4
[144]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del8.mp4
[145]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del9.mp4
[146]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del10.mp4
[147]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del11.mp4
[148]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del12.mp4
[149]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del13.mp4
[150]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del14.mp4
[151]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del15.mp4
[152]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os7del16.mp4

7.1 CPU-intensiv prosess på Mac med to CPU'er

Mac OS X sin operativsystemkjerne heter Darwin og bygger blant annent på FreeBSD-kjernen som er en Unix-kjerne bygd på den opprinnelig BSD Unix-versjonen. Dermed følger det som standard også med mye som er kjent fra Linux. For eksempel kan får man opp et bash-shell når man starter opp et Mac OS X-terminalvindu:

```
harek-haugeruds-macbook:~ hh$ uname -a
Darwin dhcp-202-136.wlan.hio.no 9.5.1 Darwin Kernel Version 9.5.1: Fri Sep 19 16:19:24 PDT 2008; root:xnu-1228.8.30~1/
```

Som vi ser er dette kjerneversjon 9.5.1 av Darwin og denne har en rettferdig måte å dele to CPU'er mellom tre prosesser på. Når man kjører det samme regn-scriptet, får man følgende resultat:

```
$ top -o cpu
Processes:  48 total, 5 running, 43 sleeping... 176 threads            21:43:01
Load Avg:  3.16,  1.85,  0.83    CPU usage: 89.27% user, 10.73% sys,  0.00% idle

  PID COMMAND      %CPU   TIME   #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
  170 bash        63.9%  2:33.80   1    13     19  192K   704K   692K    18M
  168 bash        63.8%  2:56.34   1    13     19  192K   704K   692K    18M
  169 bash        62.1%  2:35.43   1    13     19  192K   704K   692K    18M
```

Man ser at tiden deles praktisk talt likt mellom de tre prosessene. Dette gjøres ved at tre prosessene med jevne mellomrom bytter på hvilken CPU de kjører på. Til en hver tid vil det kjøre to prosesser på samme CPU, men OS-scheduler bytter likt mellom dem, slik at de skifter på hvilken av prosessene som kjører alene på den andre CPUen.

7.2 Fem CPU-intensive prosesser på host med 4 CPUer

Serveren studssh har fire CPUer som man kan se fra lscpu:

```
haugerud@studssh:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             2
NUMA node(s):          1
Vendor ID:             AuthenticAMD
CPU family:            15
Model:                 6
Model name:            Common KVM processor
Stepping:              1
CPU MHz:               2294.248
BogoMIPS:              4588.49
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             64K
L1i cache:             64K
L2 cache:              512K
```

```
L3 cache:              16384K
NUMA node0 CPU(s):     0-3
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse ss
```

Det vil si den har to sockets (to forskjellige brikker) med to Cores (CPUer) på hver. Men som man kan se av linjen hvor det står KVM, så er den egentlig en virtuell maskin (KVM står for Kernel-based Virtual Machine og er en type Linux virtualisering) og har derfor fått tildelt disse CPUene. Tidligere i semesteret var studssh konfigurert med bare to vCPUer (virtualCPU), dette kan konfigureres.

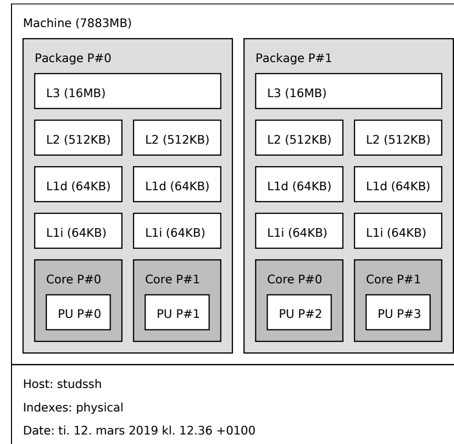


Figure 44: CPU-topologi generert av lstopo på den virtuelle maksinen studssh.

Ved å kjøre en CPU-intensiv prosess, får man følgende resultat:

```
haugerud@studssh:~$ time ./regn
Real:9.962 User:9.940 System:0.004 99.82%
```

Den bruker ca 10 sekunder. Hvis man kjører fem slike prosesser samtidig,

```
top - 13:04:52 up 27 days,  1:31, 19 users,  load average: 1,11, 0,86, 0,61
Tasks: 268 total,   6 running, 261 sleeping,   1 stopped,   0 zombie
%Cpu0  : 98,3 us,  0,3 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  1,3 st
%Cpu1  : 97,7 us,  0,3 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,3 si,  1,7 st
%Cpu2  : 95,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  5,0 st
%Cpu3  : 98,7 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,3 si,  1,0 st
KiB Mem :  8174752 total,  4926648 free,   363368 used,  2884736 buff/cache
KiB Swap:   950268 total,   846168 free,   104100 used.  7368460 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND                 P
 7495 haugerud  20   0   14648   3284   3092 R  85,1  0,0   0:04.47 regn                    1
 7493 haugerud  20   0   14648   3184   2980 R  81,8  0,0   0:04.38 regn                    0
 7497 haugerud  20   0   14648   1040    940 R  80,9  0,0   0:04.32 regn                    3
 7498 haugerud  20   0   14648   3228   3036 R  79,5  0,0   0:03.86 regn                    0
 7491 haugerud  20   0   14648   3280   3084 R  71,0  0,0   0:03.93 regn                    2
```

vil de fem prosessen kjøre samtidig på de fire CPUene, slik at det til en hver tid er to prosesser på en av CPUene. I top-utskriften over, kan man se av P-kolonnen helt til høyre at det er CPU nr 0 som har to prosesser. OS skifter fortløpende på hvilken CPU som har to prosesser, slik at alle de fem prosessene får omtrent like mye CPU-tid hver, ca 80%.

Når en prosess tar 10 sekunder, vil det kreves 5x10 = 50 CPU-sekunder for å fullføre alle de fem jobbene. Arbeidsmengden blir fordelt likt på fire CPUer og det vil da ta 50/4 = 12.5 sekunder å fullføre hver jobb. Og dette resultatet får man om man kjører fem jobber samtidig.

```
haugerud@studssh:~$ for i in {1..5}; do time ./regn& done

Real:12.208 User:10.060 System:0.016 82.54%
Real:12.241 User:10.060 System:0.020 82.34%
Real:12.366 User:9.964 System:0.024 80.76%
Real:12.869 User:10.200 System:0.008 79.32%
Real:13.143 User:10.076 System:0.032 76.90%
```

Som forventet bruker jobbene ca 12.5 sekunder på å bli ferdig. Man kan også beregne at om man bruker 80% CPU i 12.5 sekunder vil man tilsammen bruke 10 CPU-sekunder. Det kan man også se av output, User: viser hvor mange CPU-sekunder hver prosess har brukt.

7.3 Internminnet og Cache

Vi har sett at CPU-en kan lese og utføre maskininstruksjoner og disse blir hentet inn til registerne fra internminnet. Dette minnet blir også kalt RAM, en forkortelse for Random Access Memory.'Random' fordi hvilken som helst byte kan leses ut eller aksesseres like raskt som enhver annen byte. Maskinkoden for operativsystemet og andre programmer som skal kjøres må først lastes inn i internminnet fra disk. Andre deler av programmene kan hentes inn fra disk senere ved behov. Selvom det går omtrent hundre tusen ganger raskere å hente data fra internminnet enn å hente data fra harddisken, tar det alt for lang tid i forhold til hvor fort moderne CPU'er kan behandle data. De raskeste prosessorene yter mer enn 1000 MIPS (Million Instructions Per Second), det vil si mer enn en milliard (10^9) instruksjoner i sekundet. Hvis CPU-en måtte vente på data fra RAM for hver instruksjon den skulle gjøre, ville mange prosesser gått ti ganger så sakte som de gjør nå. For å kunne mate en hurtig prosessor med instruksjoner og data raskt nok, bruker man flere nivåer av mellomlagring av data, såkalt cache-minne. Ordet cache kommer fra fransk og betyr et hemmelig lager. Det går vesentlig raskere å hente minne fra cache-minnet enn fra internminnet. I tillegg har det vist seg at de fleste programmer i 90% av tiden utfører instruksjoner innenfor 10% av det totale minnet. Når CPU ber om en instruksjon som ligger et bestemt sted i minnet, hentes derfor ikke bare denne instruksjonen, men for eksempel 32 KByte av minnet. Alt dette lagres i cache og når CPU ber om neste instruksjon, ligger den ofte i cache, slik at den ikke må hentes fra internminnet. Fig. 45 viser noen typiske størrelser og aksesstider for de sentrale lagringsmedien som finnes i en datamaskin, fra registre til harddisk. Legg spesielt merke til den store forskjellen i aksesstid mellom internminnet og harddisk.

Både CPU-registre og cache er laget av SRAM (Static RAM). Aksess er meget hurtig og SRAM er statisk i den betydning at det ikke trenger å oppfriskes, slik DRAM (Dynamic RAM) må. Mer en 10 ganger i sekundet må DRAM opplades, ellers forsvinner informasjonen. SRAM består av 6 transistorer for hver bit som lagres, til sammenligning består en NOT-port av to transistorer og AND og OR-porter av 4. Men DRAM trenger bare en transistor og en kapasitator(lagrer elektrisk ladning) for å lagre en bit. Derfor er DRAM billigere, mindre og bruker mindre effekt og kan derfor lages i større enheter. Internminnet består derfor av DRAM eller forbedrede varianter av DRAM. DDR4 SDRAM (Double-Data Rate generation 4 Synchronus Dynamic RAM) ble lansert i 2014 og i 2020 kom DDR5 som er det foreløpig siste av leddene i kjedene av forbedrede utgaver av DRAM.

Cache inneholder både data og instruksjoner og deler av MMUs (Memory Management Unit) page-tables i TLB (Translation Lookaside Buffer). I L1 cache er ofte disse separert i egne enheter, mens L2 cache pleier å være en enhet. I de senere årene har man klart å få plass til L2 på selve prosessorchip'en (den lille brikken som utgjør mikroprosessoren, bare noen kvadratcentimeter stor).

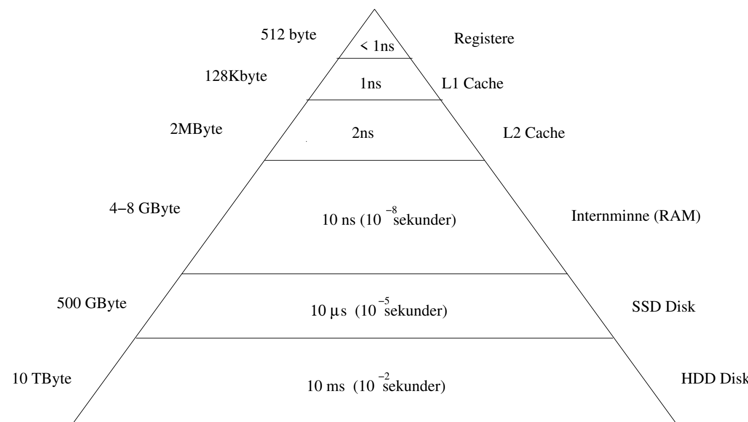


Figure 45: Minne-pyramiden. Størrelsen og tiden det tar å hente data øker nedover pyramiden.

Arkitekturen til en moderne prosessor kan da i grove trekk se ut som i Fig. 78.

Noen arkitekturer har i tillegg enda et lag i minnehierarkiet, en offchip L3 cache som sitter mellom mikroprosessoren og RAM.

7.4 Multitasking og Multiprocessing

- **Multitasking/Multiprogramming** Software(OS) brukes til å fordele tid fra samme CPU mellom flere prosesser
- **Multiprocessing** To eller flere CPU'er i samme computersystem kjører flere prosesser virkelig samtidig, på samme tidspunkt

- **Symmetric Multiprocessing** SMP, to eller flere prosessorer deler samme internminnet og kjører flere prosesser virkelig samtidig, på samme tidspunkt
- **Multi Core Multiprocessing** To eller flere prosessorer på samme brikke deler cache og databus og kjører flere prosesser samtidig. Regnes også som SMP.

7.5 Intel Core og AMD K10

Vi skal se litt på Intel Core i7 og AMD Opteron K10. De har mange likheter, begge er quad core, det vil si har 4 kjerner og har opptil 3.2 GHz klokkefrekvens. Cache-arkitekturen ligner også på hverandre, begge ser ut som i figuren:

En forskjell er at Intel Core i7 har hyperthreading i motsetning til de foregående Intel Core 2 prosessorene. Dermed kan den kjøre åtte prosesser samtidig. Men som vi skal se senere, for svært CPU-krevende prosesser har ikke dette så stor betydning, de må dele på beregnings-enheten, ALU.

Merk forøvrig: 30 MHz var maks klokkefrekvens for Intel i 1992 og den ble mer enn tredvedoblet på åtte år fram til 2000 hvor de første GHz prosessorene kom. Men på de neste åtte årene ble frekvensen bare tredoblet. Det at det er vanskelig å øke klokkefrekvensen har gjort at man istedet har økt kapasiteten med multi core og hyperthreading.

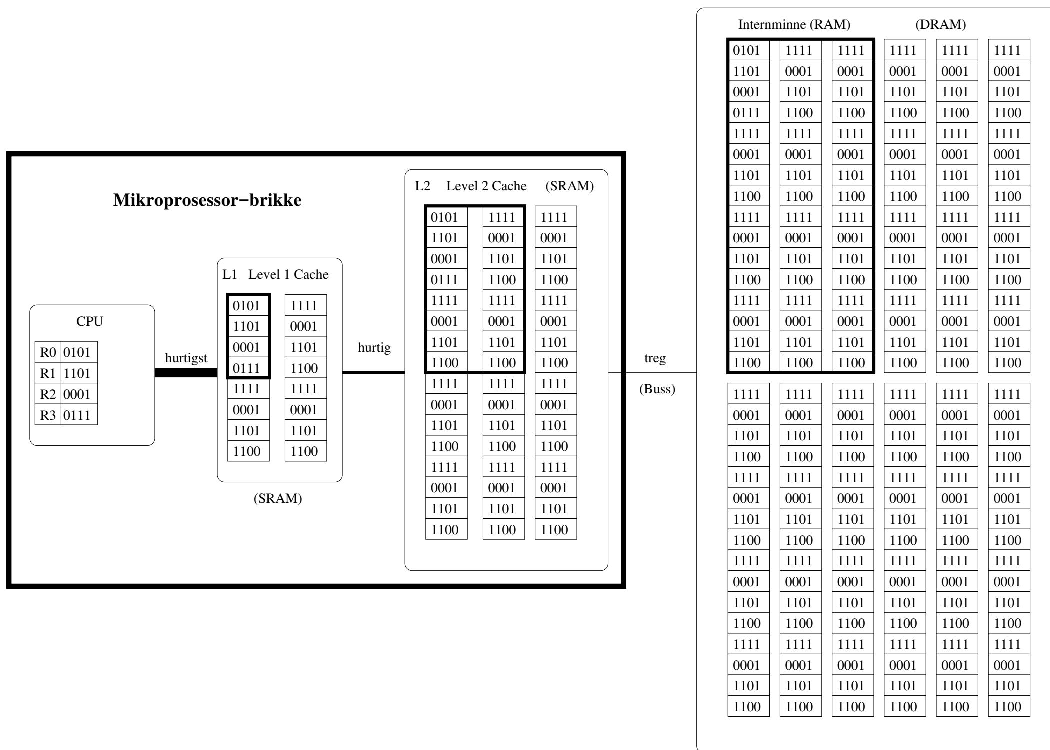


Figure 46: Level 1 cache (L1) ligger nærmest CPU. L2 er større, men har lengre aksesstid. Større deler av instruksjoner og data blir hentet av gangen fordi det ofte blir brukt for dette senere.

7.6 Mikroarkitektur

Vi har sett at hardware definerer et bestemt instruksjonssett og sett på noen av instruskjonene som er definert i X86-instruksjonssettet. Men i praksis finnes det mange måter å fysisk implementere et gitt instruksjonssett. Måten en produsent av CPU-er, som Intel eller AMD, implementerer et instruksjonssett kalles en mikroarkitektur. Forskjeller i for eksempel hvordan mikro-operasjoner utføres og hvordan pipelining eller chache-nivåer er lagt opp utgjør forskjeller i mikroarkitekturen. Man kan si at datamaskinarkitekturen utgjør kombinasjonen av instruksjonssettet og mikroarkitekturen.

Serveren amdock som er fysisk server for alle Linux-VMene (som egentlig er docker containere) har en AMD CPU modell som heter EPYC 7552 og det er en 64-bit 48-core x86 server-mikroprosesser designet av AMD i 2019. Hver core(kjerne) har SMT (Simultanious Multithreading) slik at OS ser 96 CPU-er. Den er basert på AMDs Zen 2 mikroarkitektur. Den har 3MB L1-cache, 24 MB L2 cache og 192 MB L3 cache. Videre har den 768 GB RAM.

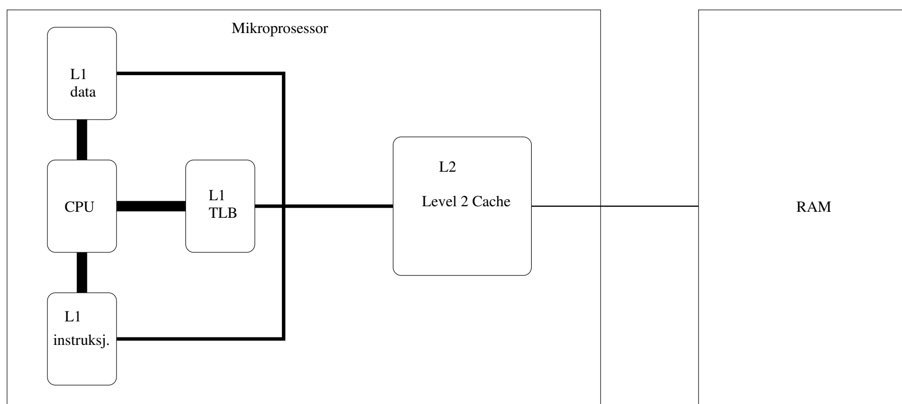


Figure 47: Level 1 cache (L1) bestående av tre deler. I AMD Athlon 64 er TLB i tillegg delt i to deler, en for adresser til instruksjoner og en for adresser til data.

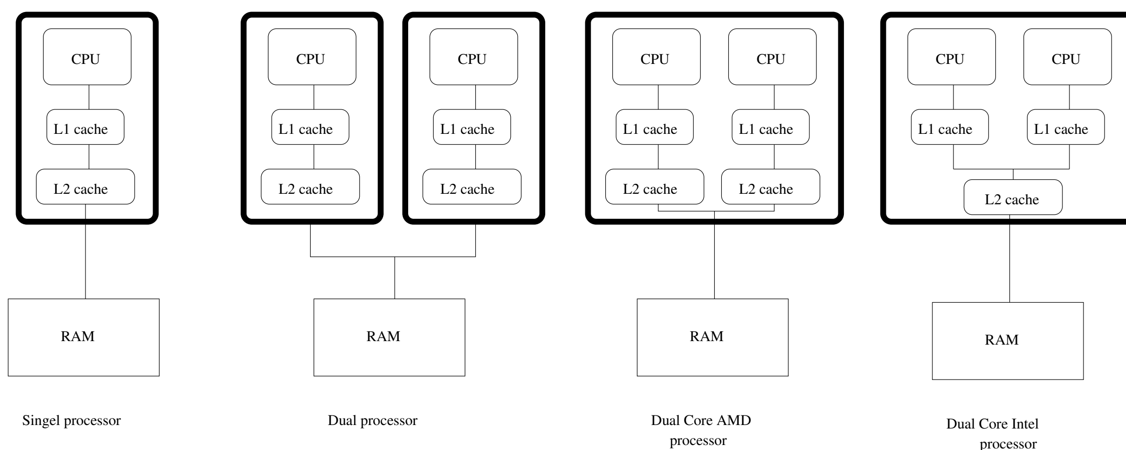


Figure 48: Single og dual prosessor og dual core prosessorer. Den tykke linjen markerer grensen for brikken/chip'en

CPU CPU CPU CPU

64KB L1 cache 64KB L1 cache 64KB L1 cache 64KB L1 cache

256 KB L2 cache 256 KB L2 cache 256 KB L2 cache 256 KB L2 cache

8MB L3 cache

Figure 49: Intel Core i7 og AMD Opteron K10. K10 kjernen har 512KB L2 cache og i7 kjernene har hyperthreading, ellers er de i store trekke relativt like.

7.7 Hyperthreading

Flere av Intels prosessorer som Pentium 4 og Xeon har såkalt hyperthreading teknologi (Hyper-Threading er slik Intel selv betegner teknologien). Det vil si at deler av CPU-en er duplisert, som alle registerne, men for eksempel ikke ALU-en. Det gjør at en CPU kan inneholde to prosesser samtidig, slik at hvis den ene prosessen for eksempel bruker tid på å hente noe fra minne, kan CPUen ekstremt raskt switche over og la den andre prosessen bruke ALU-en. En slik overgang styres av hardware og ikke av operativsystemeet og skjer i løpet av et nanosekund eller to. En normal context switch utført av OS tar flere tusen ganger så lang tid, flere mikrosekunder. Det å utføre instruksjoner for en prosess kalles en thread eller tråd. Det er basert på bildet av den linjen(tråden) som følges når et program utføres ved å hoppe i mellom instruksjonene som programmet består av. Mer om threads senere. I en CPU med hyperthreading er ikke fullstendig delt i to enheter, slik som kjernene i en dobbel core prosessorer hvor de er helt separate enheter. Et operativsystem oppfatter en hyperthreading CPU på samme måte som en duo core, som to CPU'er. Det kan derfor være vanskelig å se forskjellen, men det vil kunne sees av ytelsen.

Hyperthreading er Intels eget markedsføringsbegrep for denne teknologien. Den generelle betegnelse er SMT (Simultaneous multithreading) og AMD har implementert SMT i noen av sine mikroartkitekturer som i Zen.

Desktop'en rex har en Intel i7 prosessor som har 4 kjerner(cores) som er hyperthreading og lscpu gir følgende:

```
rex:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Stepping:              9
CPU MHz:               1711.156
CPU max MHz:           3900,0000
CPU min MHz:           1600,0000
BogoMIPS:              6785.02
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
```

Dette er ikke en VM men en fysisk node som har en socket med 4 cores og 2 threads per core. Det gir tilsammen 8 CPUer, output fra lscpu omtaler hver regneenhet som en CPU. Men 2 threads per core betyr i denne sammenhengen at hver core er hyperthreading som forklart over og egentlig er en regneenhet med en enkelt ALU men dobbelt sett av registre slik at en core kan kjøre to prosesser samtidig. Hvordan dette ser ut i praksis skal vi her teste.

Som før bruker vi følgende CPU-slukende program:

```
#! /bin/bash
```

```
(( max = 300000 ))
(( i = 0  ))
(( sum = 0  ))

while (($i < $max))
do
        (( i += 1 ))
        (( sum += i  ))
done
echo $0, resultat: $sum
```

Slik ser det ut på rex om man starter 10 CPU-intensive regnejobber:

```
rex:~/regn$ for i in {1..10}; do time ./regn & done
```

```
top - 21:19:30 up 26 days,  9:10,  2 users,  load average: 3,72, 2,59, 2,12
Tasks: 395 total,  11 running, 384 sleeping,   0 stopped,   0 zombie
%Cpu0  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu1  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu2  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu3  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu4  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu5  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu6  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu7  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem : 16385632 total,  5192648 free,  5303528 used,  5889456 buff/cache
KiB Swap: 16730108 total, 16698824 free,    31284 used. 10271192 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND                   P
32596 haugerud  20   0   14068    908    804 R  90,0  0,0   0:17.55 regn                      0
32599 haugerud  20   0   14068   3020   2808 R  83,7  0,0   0:16.54 regn                      4
32598 haugerud  20   0   14068    912    808 R  83,4  0,0   0:17.08 regn                      6
32584 haugerud  20   0   14068    940    836 R  82,7  0,0   0:16.71 regn                      1
32595 haugerud  20   0   14068   2872   2660 R  79,7  0,0   0:15.93 regn                      3
32587 haugerud  20   0   14068   2932   2720 R  79,4  0,0   0:17.39 regn                      1
32597 haugerud  20   0   14068   2944   2728 R  78,1  0,0   0:18.73 regn                      5
32600 haugerud  20   0   14068    900    800 R  78,1  0,0   0:15.61 regn                      3
32588 haugerud  20   0   14068   2956   2740 R  75,7  0,0   0:16.41 regn                      2
32592 haugerud  20   0   14068    904    800 R  68,8  0,0   0:15.54 regn                      7
```

Linux-kjernen betrakter dette som åtte uavhengige CPU'er og kjører prosesser på alle åtte. Av kolonnen P kan vi se at OS fordeler prosesser på alle CPUene og dermed må det være to prosesser på to av dem, i dette tilfellet på prosessor nummer 1 og 3. OS bytter med jevne mellomrom på hvilke CPUer som har to prosesser, slik at som man kan se av time-kolonnen, alle prosessene får omtrent like mye CPU-tid og avslutter samtidig. CPU-kolonnen viser andel CPU de siste 3 sekundene og her kan vi se at det er litt forskjell. Men i snitt får de omtrent 4/5 dels eller 80% CPU-tid. Det koster litt overhead å flytte en prosess fra en CPU til en annen, så det skjer ikke altfor ofte. (Når man kjører top må man taste 1 for å se de 8 øverste CPU-linjene og f fulgt av p og return for å se hvilke prosessorer som brukes).

7.7.1 Kjører en CPU med hyperthreading to prosesser reelt sett samtidig?

Men regner disse prosessene reelt sett samtidig? Hvis man setter igang åtte regnejobber viser `top` at de jobber på hver sin CPU og at de hver får 100% CPU-tid. Men hvordan kan man finne ut om de virkelig gjør det?

På samme måte som om man ønsker å finne ut om åtte arbeidere man har ansatt for å skrelle poteter virkelig jobber samtidig. Man tar tiden på dem. Åtte personer bør bruke like lang tid på å skrelle åtte sekker poteter som fire stykker bruker på fire sekker poteter. Ihvertfall hvis de har en potetskreller (ALU) hver. Men hvis to og to av arbeiderne må dele på samme potetskreller, tar det dobbelt så lang tid.

Så vi setter igang fire regnejobber som skreller ivei på hver sin av de fire CPU-ene:

```
rex:~/regn$ for i in {1..4}; do time ./regn & done

Real:18.152 User:18.144 System:0.000 99.96%
Real:18.401 User:18.392 System:0.004 99.97%
Real:18.417 User:18.412 System:0.000 99.97%
Real:18.516 User:18.508 System:0.000 99.96%
```

Jobben går unna på litt i overkant av 18 sekunder og det bør ikke ta lenger tid for åtte prosesser hvis de reelt sett jobber samtidig:

```
rex:~/regn$ for i in {1..8}; do time ./regn & done
Real:35.048 User:35.008 System:0.000 99.88%
Real:35.222 User:35.144 System:0.000 99.78%
Real:35.246 User:35.104 System:0.000 99.59%
Real:35.270 User:34.976 System:0.020 99.22%
Real:35.500 User:34.888 System:0.008 98.29%
Real:35.562 User:34.840 System:0.012 98.00%
Real:35.606 User:35.448 System:0.000 99.55%
Real:35.796 User:35.140 System:0.012 98.20%
```

Men det tar nesten dobbelt så lang tid. Som beskrevet tidligere, CPU-en har lastet inn to prosesser samtidig, men internt må de bytte på å bruke ALU-en og for slike prosesser som hele tiden bruker CPU har hyperthreading liten effekt. Det blir som om arbeiderne må bytte på å bruke samme potetskreller og da tar det dobbelt så lang tid å bli ferdig. Litt effekt har dog hyperthreading, det går litt mindre enn dobbelt så lang tid, som ville vært nærmere 37 sekunder.

7.8 Hyperthreading med prosess som bruker mye RAM

Hyperthreading har altså kun en liten positiv effekt for programmer som hele tiden bruker ALU siden de to prosessene da må bytte på å bruke ALUen og dermed bruker dobbelt så lang tid. Men for programmer som ofte venter noen klokkesykler på å få lest fra eller skrevet til RAM kan hyperthreading ha stor effekt. Det følgende programmet skriver om og om igjen tall til et array som ligger i RAM:

```
rex:~/regn$ cat ram.c
#include <stdio.h>

int array[102400];

void main(){
   int i,k;
   for(k=0;k<2000000;k++){
      for(i = 0;i < 1024;i++){
          array[i] = i;
     }
  }
}
```

Hvis man kompilerer og kjører det, bruker det ca 4 sekunder på å kjøre ferdig.

```
rex:~/regn$ gcc ram.c
rex:~/regn$ time ./a.out
Real:4.021 User:4.016 System:0.004 99.97%
```

Hvis man starter fire slike prosesser, sørger operativsystemet sin scheduler for at de kjører på hver sin ALU og alle bruker omtrent fire sekunder.

```
rex:~/regn$ for i in {1..4}; do time ./a.out& done
rex:~/regn$
Real:4.060 User:4.056 System:0.000 99.89%
Real:4.103 User:4.100 System:0.000 99.91%
Real:4.137 User:4.132 System:0.000 99.87%
Real:4.141 User:4.136 System:0.000 99.88%
```

Hvis man kjører åtte slike RAM-brukende prosesser samtidig, vil to og to av dem kjøre på samme core og dermed bytte på å bruke den samme ALU-en. Men i dette tilfellet vil det hele tiden være litt venting på RAM, slik at et lynhurtig bytte av hvem som bruker ALU kan gi en positiv effekt. Og det er hardware som utfører dette byttet, det blir i motsetning til for en vanlig prosess context switch ikke utført av operativsystemet.

```
rex:~/regn$ for i in {1..8}; do time ./a.out& done
rex:~/regn$
Real:4.345 User:4.336 System:0.000 99.80%
Real:4.370 User:4.332 System:0.000 99.12%
Real:4.371 User:4.364 System:0.000 99.85%
Real:4.371 User:4.368 System:0.000 99.94%
Real:4.425 User:4.420 System:0.000 99.89%
Real:4.437 User:4.424 System:0.000 99.70%
Real:4.479 User:4.328 System:0.000 96.62%
Real:4.496 User:4.328 System:0.000 96.27%
```

Og nå ser vi at hyperthreading har en meget stor effekt, til tross for at de må dele ALU bruker prosessen bare omtrent 10 % mer CPU-tid på å fullføre sammenlignet med når en prosess kjører helt alene på en core.

7.9 Deaktivering av hyperthreading

Følgende informasjon er hentet fra samme Intel desktop med navn rex:

```
rex:~$ lscpu | grep name
Model name:            Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz

rex:~$ lscpu | grep Thread
Thread(s) per core:    2

rex:~$ grep "" /sys/devices/system/cpu/cpu*/topology/thread_siblings_list
/sys/devices/system/cpu/cpu0/topology/thread_siblings_list:0,4
/sys/devices/system/cpu/cpu1/topology/thread_siblings_list:1,5
/sys/devices/system/cpu/cpu2/topology/thread_siblings_list:2,6
```

```
/sys/devices/system/cpu/cpu3/topology/thread_siblings_list:3,7
/sys/devices/system/cpu/cpu4/topology/thread_siblings_list:0,4
/sys/devices/system/cpu/cpu5/topology/thread_siblings_list:1,5
/sys/devices/system/cpu/cpu6/topology/thread_siblings_list:2,6
/sys/devices/system/cpu/cpu7/topology/thread_siblings_list:3,7
```

Dette viser at CPUen har hyperthreading, siden den angir "2 threads per core". Når to prosess-enheter deler samme ALU, kalles de "thread siblings" og listen over viser hvilke som hører sammen og deler ALU. Den samme informasjonen kan man få grafisk med `lstopo`:

```
lstopo --no-io --no-caches
```

gir følgende figur

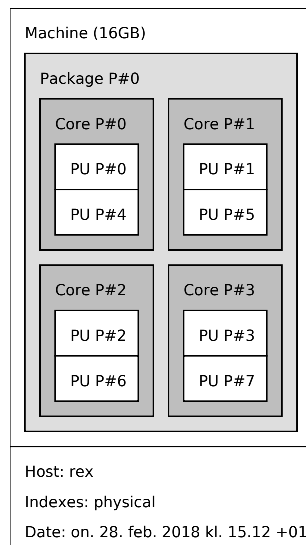


Figure 50: CPU-topologi generert av lstopo for rex med hyperthreading aktivert

For å skru av hyperthreading, kan man fjerne en PU (Processing Unit) fra hver av de fire siblings-parene i listen. Alternativt en av PU'ene fra hver core i lstopo-figuren. Dette kan gjøres ved som root å overskrive en setting i `/sys/devices/system/cpu`:

```
# for i in 4 5 6 7; do echo 0 > /sys/devices/system/cpu/cpu$i/online ; done
```

Etter man har gjort dette, vil rex kun ha fire cores uten hyperthreadin og OS vil kun se disse fire og schedulere prosesser på disse fire.

Om man nå starter 8 regn-jobber på rex, ser det slik ut:

```
top - 12:10:49 up 27 days, 2 min,  1 user,  load average: 4,66, 1,71, 0,95
Tasks: 345 total,   9 running, 336 sleeping,   0 stopped,   0 zombie
%Cpu0  : 99,7 us,  0,3 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu1  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu2  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu3  : 99,3 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,7 si,  0,0 st
```

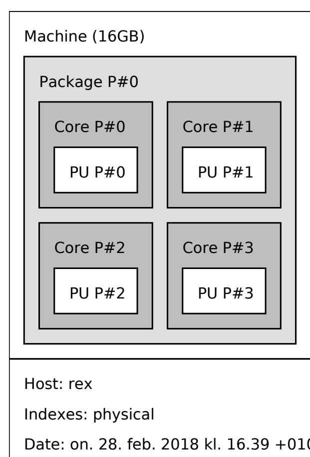


Figure 51: CPU-topologi generert av lstop for rex med hyperthreading deaktivert

```
KiB Mem : 16385632 total,  4838284 free,  5646572 used,  5900776 buff/cache
KiB Swap: 16730108 total, 16725972 free,     4136 used.  9934028 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
29234 haugerud  20   0   14068    884    780 R  50,2  0,0   0:07.19 regn
29232 haugerud  20   0   14068    940    836 R  49,8  0,0   0:07.10 regn
29237 haugerud  20   0   14068   2956   2740 R  49,8  0,0   0:07.16 regn
29239 haugerud  20   0   14068    940    836 R  49,8  0,0   0:07.33 regn
29241 haugerud  20   0   14068   1056    956 R  49,8  0,0   0:07.10 regn
29243 haugerud  20   0   14068    884    780 R  49,8  0,0   0:07.10 regn
29242 haugerud  20   0   14068    888    784 R  49,5  0,0   0:07.26 regn
29244 haugerud  20   0   14068    912    808 R  49,2  0,0   0:07.07 regn
```

De åtte jobbene deler på fire CPUer, to kjører på hver og får 50% CPU-tid. Totaltiden blir naturlig nok ganske nøyaktig det dobbelte av når bare fire prosesser kjører:

```
Real:36.553 User:18.260 System:0.004 49.96%
Real:36.679 User:18.340 System:0.012 50.03%
Real:36.839 User:18.356 System:0.000 49.82%
Real:36.842 User:18.232 System:0.000 49.48%
Real:36.854 User:18.216 System:0.008 49.45%
Real:36.915 User:18.232 System:0.000 49.39%
Real:37.098 User:18.368 System:0.000 49.51%
Real:37.112 User:18.272 System:0.000 49.23%
```

Litt overhead blir det av context-switching når to prosesser deler samme CPU, men ikke mye. Og vi ser at totaltiden blir litt over ett sekund lenger enn når hyperthreading var på, så en liten effekt har hyperthreading, selvom de to prosessene må dele ALU. For andre jobber kan hyperhtreading ha større effekt, spesielt hvis programmene ofte bruker RAM.

7.10 RAM-prosess uten hyperthreading

Hvis vi nå kjører prosessen som hele tiden skriver til et array, så ser det helt likt ut når 4 prosesser kjører. De vil scheduleres av OS på hver sin core og bruker som før ca 4 sekunder:

```
rex:~/regn$ for i in {1..4}; do time ./a.out& done
rex:~/regn$
Real:4.063 User:4.036 System:0.000 99.34%
Real:4.080 User:4.056 System:0.004 99.50%
Real:4.171 User:4.036 System:0.000 96.76%
Real:4.177 User:4.088 System:0.000 97.86%
```

Men når vi nå kjører 8 samtidige prosesser må operativsysteme schedulere to prosesser på hver CPU (core/kjerne/regnenhet/ALU) og la de to prosessene bytte på å bruke den ved konvensjonell multitasking styrt av OS. Da vil det totalt sett naturlig nok ta dobbelt så lang tid å fullføre alle prosessene:

```
rex:~/regn$ for i in {1..8}; do time ./a.out& done
rex:~/regn$
Real:8.034 User:3.988 System:0.000 49.64%
Real:8.077 User:3.980 System:0.000 49.27%
Real:8.083 User:4.040 System:0.000 49.98%
Real:8.139 User:4.036 System:0.008 49.68%
Real:8.137 User:3.988 System:0.000 49.00%
Real:8.139 User:3.988 System:0.000 48.99%
Real:8.163 User:3.996 System:0.004 49.00%
Real:8.176 User:4.124 System:0.000 50.43%
```

Dette er resultatet av OS-styrt multitasking og er helt forskjellig fra hva vi fikk når hyperthreading var skrudd på, da fikk vi følgende når nøyaktig samme program kjøres på samme maskin:

```
rex:~/regn$ for i in {1..8}; do time ./a.out& done
rex:~/regn$
Real:4.345 User:4.336 System:0.000 99.80%
Real:4.370 User:4.332 System:0.000 99.12%
Real:4.371 User:4.364 System:0.000 99.85%
Real:4.371 User:4.368 System:0.000 99.94%
Real:4.425 User:4.420 System:0.000 99.89%
Real:4.437 User:4.424 System:0.000 99.70%
Real:4.479 User:4.328 System:0.000 96.62%
Real:4.496 User:4.328 System:0.000 96.27%
```

Det går nå nesten dobbelt så fort og viser at hardware hyperthreading er mye mer effektivt en OS multithreading med context switch. En hyperthreading switch bruker bare noen nanosekunder og kan utnytte at en thread venter på RAM. En OS-context switch tar minst tusen ganger så lang tid og kan derfor ikke brukes til å effektivisere bort svært korte pauser i prosesseringen på grunn av venting på RAM.

7.11 Taskset

Med kommandoen `taskset` kan man eksplisitt forlange at en prosess kjører på en bestemt CPU. Det vil da medføre at man overstyrer OS sin scheduling av prosesser; OS vil alltid prøve å fordele arbeidsmengden så jevnt som mulig. Taskset kan være nyttig i mange situasjoner og kan også brukes til å belyse forskjellen på multithreading og hyperthreading. OS nummererer CPU-ene fra 0 til 7 om man har åtte av dem. Man kan låse en regnejobb til CPU nummer 0 og kjøre og ta tiden på den med

```
rex:~/regn$ time taskset -c 0  ./regn
Real:18.042 User:18.036 System:0.000 99.96%
```

(på forelesning ble rekkefølgen på time og taskset byttet og da blir tidsanvisningen anderledes) Hvis man tvinger to slike regnejobber til å kjøre på samme CPU vil de bruke dobbelt så lang tid:

```
rex:~/regn$ for i in 1 2; do time taskset -c 0  ./regn& done
rex:~/regn$
Real:36.088 User:18.036 System:0.000 49.97%
Real:36.206 User:18.164 System:0.000 50.16%
```

Av prosenttallet ser vi at de får 50% CPU hver, likt fordelt fra OS som schedulerer under restriksjonen att begge alltid må kjøre på OS nr 0. Hvis man overlot scheduling til OS, ville de blitt plassert på hver sin core og fullført dobbelt så fort:

```
rex:~/regn$ for i in 1 2; do time  ./regn& done
rex:~/regn$
Real:18.431 User:18.428 System:0.000 99.98%
Real:18.453 User:18.448 System:0.000 99.97%
```

Fra før vet vi at CPU 0 og CPU 4 er siblings, det vil si sitter på samme core. Hvis vi med hyperthreading aktivert bruker taskset til å låse prosessene til disse to, vil det gå nesten like sakte som om de ble satt på samme CPU:

```
rex:~/regn$ for cpu in 0 4; do time taskset -c $cpu  ./regn& done
rex:~/regn$
Real:35.075 User:35.072 System:0.000 99.99%
Real:35.080 User:35.072 System:0.004 99.99%
```

Det kommandoen over tekninsk sett gjør er å gi variabelen cpu verdien 0 og 4 og løkken setter igang to prosesser, en på CPU 0 og en på CPU 4, som er siblings og sitter på samme core med en felles ALU.

Men hvis vi låser prosessene på nesten samme måte, men til CPU 1 og 4, vil det igjen gå nesten dobbelt så fort fordi vi nå eksplisitt har plassert dem på hver sin core, slik også OS gjør når to prosesser scheduleres:

```
rex:~/regn$ for cpu in 1 4; do time taskset -c $cpu  ./regn& done
rex:~/regn$
Real:18.470 User:18.468 System:0.000 99.98%
Real:18.762 User:18.756 System:0.000 99.97%
```

8 Forelesning 12/3-24(2 timer). Systemkall, Scheduling og vaffelrøre

I år (2025) vil denne simuleringen gjennomføres i levende live i PH170 onsdag 12. mars og ikke bare som videoopptak slik som under pandemien.

`Avsnitt fra Tanenbaum: 1.4, 1.6`

8.1 Slides og opptak

8.1.1 Slides brukt i forelesningen

Slides brukt i forelesningen[153]

Opptak av forelesningen:

os8time1.mp4[154] (38:03) Uredigert opptak av første time av forelesningen.
os8time2.mp4[155] (48:56) Uredigert opptak av andre time av forelesningen.

Opptak av forelesningen inndelt etter temaer:

os8del1.mp4[156] (07:27) Intro, om kurset fremover, om dagens temaer og vaffelrørelaging
os8del2.mp4[157] (02:11) Oppsummering av forrige forelesning
os8del3.mp4[158] (09:53) Hvorfor kan ikke en prosess unytte to CPU-er
os8del4.mp4[159] (05:54) Samtidige prosesser og prosessormodus
os8del5.mp4[160] (02:01) Viktig spørsmål: Hvis du kjører et script som root, kjører du da i kernel mode? Nei!
os8del6.mp4[161] (08:41) Hvordan kan OS effektivt kontrollere brukerprosesser?
os8del7.mp4[162] (01:27) Spørsmål: Eksempler på hyperthreading; to ytterpunkter: CPU og RAM-avhengig

os8del9.mp4[163] (01:12) Spørsmål: Man kan switche fra kernel mode til user mode. Men ikke tilbake?
os8del10.mp4[164] (10:38) Systemkall
os8del11.mp4[165] (10:44) Linux-scheduling
os8del12.mp4[166] (03:27) Prioritet
os8del13.mp4[167] (02:21) Need resched
os8del14.mp4[168] (07:55) Introduksjon til vaffelrøre-simuleringen av et operativsystem

[153] https://www.cs.oslomet.no/ haugerud/os/Forelesning/os8.pdf
[154] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8time1.mp4
[155] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8time2.mp4
[156] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del1.mp4
[157] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del2.mp4
[158] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del3.mp4
[159] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del4.mp4
[160] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del5.mp4
[161] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del6.mp4
[162] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del7.mp4
[163] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del9.mp4
[164] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del10.mp4
[165] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del11.mp4
[166] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del12.mp4
[167] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del13.mp4
[168] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os8del14.mp4

8.1.2 Vaffel-video

Video som demonstrerer multitasking av forelesning og vaffelrørelaging[169]

8.2 Sist

- Multitasking
- CPU og Cores
- Hyperthreading

8.3 Hvorfor kan ikke en prosess bruke to CPU-er?

Det ville vært ønskelig om et vilkårlig program kjøres fire ganger så raskt på en maskin med en quadcore prosessor(4 CPU'er på en brikke) som på en maskin med en enkelt CPU av samme type. Men det gjør generelt et program ikke, det bruker like lang tid om det har fire prosessorer, for det klarer bare å utnytte en prosessor av gangen. Slik at man bare får en gevinst av de fire prosessorene om man har flere programmer som kjører samtidig.

For å undersøke hvorfor det er slik, ser vi på assemblerkoden for et program som regner ut Fibonacci-rekken, 1 1 2 3 5 8 13 21 34 55 ..... I denne rekken er neste tall summen av de to foregående. Assemblerkode ligger tett opp til maskinkode, det er en litt mer lesbar utgave av maskininstruksjoner og kan sees på som den koden som CPU-en utfører en for en:

```
1. mov 1, %ax     # %ax = 1
2. mov 1, %bx     # %bx = 1
3. add %ax, %bx   # %bx = %bx + %ax
4. add %bx, %ax   # %ax = %ax + %bx
5. jmp 3
```

Etter hver runde i denne evige løkken, vil ax og bx være siste og nest siste ledd i Fibonacci-rekken som er regnet ut. I instruksjon 3 settes bx lik summen av de to og i nummer 4 settes ax lik summen av de to og dermed har vi kommet to stepp videre i beregningen. Vi ser at det ikke er mulig for et operativsystem å fordele bergningene i en slik algoritme på flere CPU'er. Neste ledd i beregningen avhenger av det forrige og det tar uforholdsmessig lang tid å flytte verdier av registre fra en CPU til en annen. I dette tilfellet lar ikke algoritmen seg naturlig dele opp i separate bergningsdeler og da vil det også være vanskelig for en programmerer å dele opp beregningen i flere prosesser for å utnytte flere prosessorer. Følgende eksempel som regner ut summen

$$S = 1 + 2 + 3 + 4 + \cdots + 2000 = \sum_{i=1}^{2000} i$$

er det i prinsippet letter å dele opp eller parallellisere som det kalles:

```
1. mov 2001, %ax
2. mov 1, %bx
3. mov 0, %cx
3. add %bx, %cx    # %cx += %bx
4. inc %bx         # %bx++
5. cmp %bx %ax
6. jne 3           # Hopp til linje 3 hvis %bx ikke er lik 2001
```

[169] https://os.cs.oslomet.no/os/vaffel.mp4

Etter dette programmet er avsluttet vil registreret cx være lik $S = \sum_{i=1}^{2000} i$. Det er lett å se at denne algoritmen i prinsippet kan deles i to. En CPU kan regne ut $\sum_{i=1}^{1000} i$ og en annen CPU kan regne ut $\sum_{i=1001}^{2000} i$ og så legger man sammen svarene til slutt. Men poenget er at operativsystemet ikke har noen anelse om hva som foregår i et vilkårlig program. Det bare sørger for at prosessene får utført sine instruksjoner. Derfor er det programmereren som eksplisitt må skrive programmet slik at det kjøres som to uavhengige prosesser for at det skal kunne utnytte flere CPU'er. En annen løsning er at programmet inneholder flere tråder(threads) som kan kjøres på hver sin CPU, dette kommer vi tilbake til senere. Threads i denne sammenhengen har ikke noe med hyperthreads å gjøre, disse threads styres av OS og ikke av prosessoren. Det har også blitt utviklet kompilatorer som til en viss grad klarer å parallelisere kode. Men operativsystemet kan ikke gjette seg til hva programmet gjør og kan derfor ikke på egenhånd få en enkelt prosess til å utnytte flere prosessorer.

Moderne spillkonsoller inneholder ofte mange CPU'er, XBOX 360 har tre og Playstation 3 har åtte CPU'er. For å kunne utnytte disse må spill-programmen som kjøres på dem skrives slik at de kan utnytte alle prosessorene. Programmererne deler da opp oppgavene i uavhengige deler slik at de kan beregnes hver for seg. Dette kalles å parallellisere koden. Tidligere var dette bare viktig i såkalte clustere satt sammen av mange datamaskiner, men med dagens utvikling hvor etterhvert alle datamakiner har flere CPU'er er dette viktig for alle programmer.

8.4 Samtidige prosesser

To prosesser (tasks) må ikke ødelegge for hverandre:

- skrive til samme minne
- kapre for mye CPU-tid
- få systemet til å henge

Beste løsning:

All makt til OS = **Preemptive multitasking**

Linux, Windows, Mac Preemptive multitasking, *"Preemptive"= rettighetsfordelende. Opprinnelig betydning: Preemption = Myndighetene fordeler landområde.*

Windows 3.1 Cooperative multitasking, prosessene samarbeider om å dele på CPU-en. *En prosess kan få hele systemet til å henge.*

8.5 Prosessor modus

Alle moderne prosessorer har et modusbit[170] som kan begrense hva som er lov å gjøre. Modusbit switcher mellom bruker og priviligert modus. Dette kalles også protection hardware og er nødvendig for å kunne kjøre multitasking.

- Bruker modus: User mode. begrenset aksess av minne og instruksjoner; må be OS om tjenester.
- Priviligert modus: Kernel mode. Alle instruksjoner kan utføres. Alt minne og alle registre kan aksesseres.

[170] 8086 og 8088 hadde ikke modusbit, på Intel-prossorer kom det først med 286

Se avsnittet Protection Hardware i 1.5.7 i Tanenbaum. Se også bruken av hjelm som hardware protection senere i forelesning.

8.6 Hvordan kan OS effektivt kontrollere brukerprosesser?

Problem: OS kan ikke tillate en prosess/bruker å ta kontroll over maskinen. Men hvis OS skal kontrollere *hver* instruksjon en bruker-prosess utfører (emulering) gir det meget stor systemoverhead.

Effektiv løsning: OS bruker en hardware timer til å gi et begrenset tidsintervall til en brukerprosess og switcher til brukermodus og laster inn 1. brukerinstruksjon. Når tiden er ute, hopper CPU til OS-kode og OS overtar. Hver eneste CPU får sine individuelle timer-interrupt, slik at OS tar over kontrollen med jevne intervaller; typisk hvert hundredels skund.

8.7 Bruker/Priviligert minne-kart

Det er helt vesentlig at operativsystemet har støtte fra hardware for å kunne kontrollere at vanlige programmer ikke skriver til vilkårlige deler av internminnet. Det varierer hvor stor del av de forskjellige operativsystemene som finnes som ligger i kjernen, det vil si, ligger i den priviligerte delen av minnet som kun kan nås i priviligert prosessormodus.

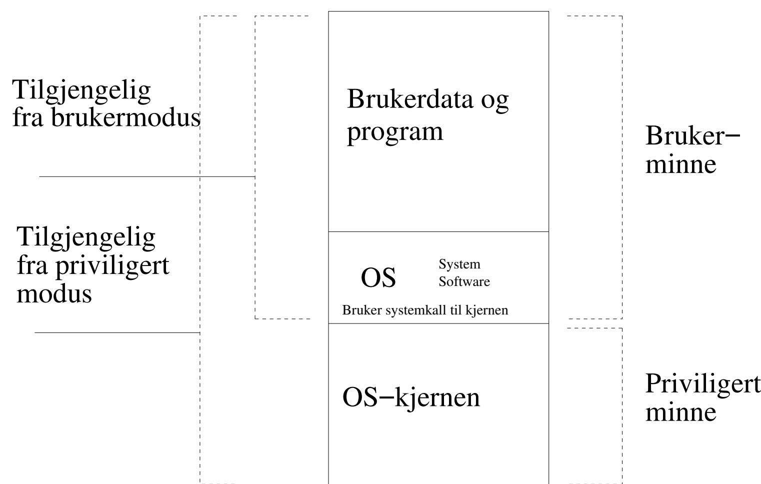


Figure 52: Modusbit må switches til priviligert modus for å kunne kjøre kode fra priviligert del av minne (OS-kjernen). Deler av OS ligger utenfor kjernen og kan kjøres fra brukermodus.

8.8 Systemkall

Anta at en vanlig bruker utfører Linux-kommandoen `ls`. En slik kommando ber om å lese noe som ligger på harddisken og er da nødt til å få hjelp av OS-kjernen for brukerprogrammer har aldri direkte aksess til eksterne enheter. Programmet `ls` inneholder derfor bl. a. systemkallet `readdir()` som ber kjernen om å lese hva en katalog inneholder. Før kjernekoden kjøres, må modus-bit settes til kernel. Men hvis det fantes en instruksjon SWITCH_TO_KERNELMODE, ville et program i usermodus kunne gjøre denne instruksjonen og deretter få fri tilgang til systemet. Hvordan løses dette dilemmaet: sette modusbit til kernel og deretter være sikker på at det kun er kjernekode som kjøres?

OS må igjen ha hjelp fra hardware i form av en spesiell instruksjon, trap, som i samme operasjon switcher modussbit til kernel og hopper til ett av flere predefinert steder i minnet hvor det ligger kode for systemkall,

som vist i figur 56. Det er da ikke mulig å switche til kernelmodus og kjøre vilkårlig kode etterpå for vanlige brukerprogrammer. Etter systemkallet er utført, switcher OS modusbit til brukermodus og returnerer til der i koden systemkallet ble utført.

På Linuxinstallasjoner der kildekoden er med, inneholder filen `syscall_table_32.S` (med path /usr/src/linux-source-3.2/arch/x86/kernel e.l.) som er en del av kildekoden for Linux som Linus Torvalds har skrevet, som definerer hver av de 348 Linux-systemkallene. Den tilsvarende filen for 4.4 kjernen heter

```
linux-source-4.4.0/arch/x86/entry/syscalls/syscall_32.tbl
```

og har 376 systemkall. I tillegg finnes det noen spesifikke systemkall for 64 bits prosessorer.

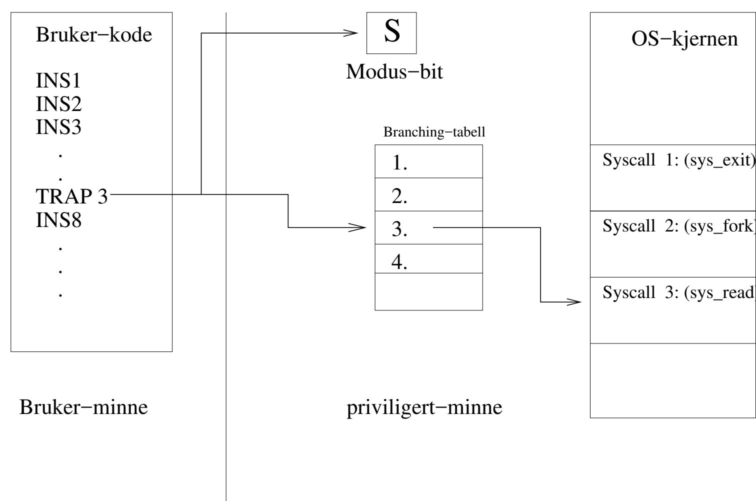


Figure 53: Trap-instruksjonen sørger for at en brukerprosess ikke kan oppnå full kontroll over en maskin ved å switche til kernelmodus.

8.9 Prioritet i Linux-scheduling

Alle mulititasking operativsystem bruker en hardware timer til å dele opp tiden i små enheter. Med faste intervall vil denne timeren sende et interrupt til CPU-en som gjør at den hopper til operativsystemet-kode for dette intertuptet. Linux-kjernen deler tiden opp i ticks eller jiffies som vanligvis varer i 10 millisekunder eller ett hundredels sekund. Dette er tiden som går mellom hver gang hardwaretimeren sender et interrupt. Denne hardwaretimeren er faktisk programmerbar, slik at OS-kjernen kan sette verdien for denne når maskinen booter. Det har variert litt hva som har vært standard lengde på en Linux-jiffie, men i det siste har det vært 10 ms. I noen tidligere versjoner var det 2.5 ms. Det er mulig å konfigurere Linux kjerneversjon 2.6 til å bruke jiffie-lengde 10, 2.5 og 1 ms.

- Tiden deles i epoker
- Hver prosess tildeles et time-quantum målt i et helt antall jiffies som legges i variabelen counter. F. eks. 20 i enheter av ticks = 10 ms = timer-intervall
- OS kjører Round Robin-scheduling. Prosessen som kjører mister ett tick (counter reduseres med en) for hvert timer-tick.
- For hvert timer-tick sjekkes det om kjørende prosess har flere ticks, counter $>$ 0

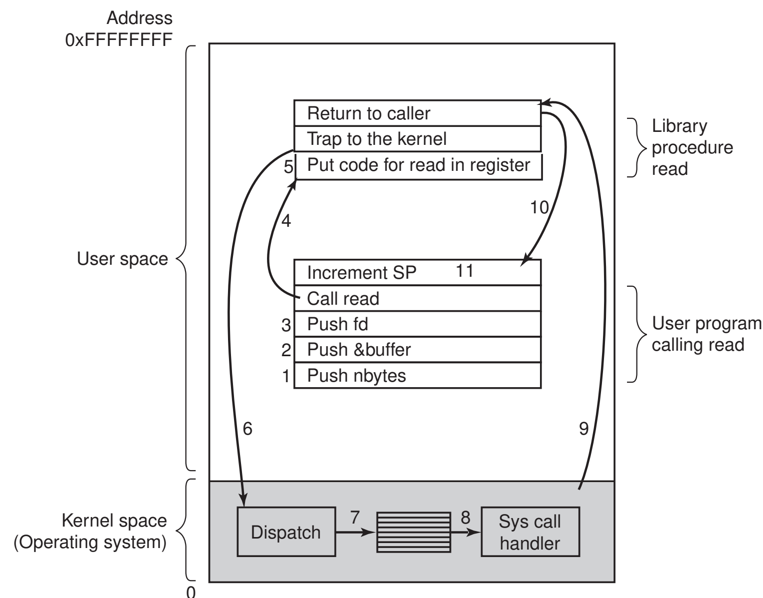


Figure 54: Fig 1-17 fra Tanenbaum

- Hvis counter > 0 fortsetter prosessen, hvis ikke kalles schedule() som velger en ny
- Epoken er over når alle prosesser har brukt opp sin tid (counter = 0)
- Antall ticks som deles ut før hver epoke bestemmes av prioriteten og lagres i en variabel med navn 'priority'
- En vanlig brukerprosess kan senke sin egen prioritet
- Prioritet kan dermed endres dynamisk (har en prosess brukt mye CPU, kan den f. eks. få nedsatt prioritet)
- Gjennomsnittlig time-quantum for 2.4 kjernen var ca 210 ms
- Gjennomsnittlig time-quantum for 2.6 kjernen var ca 100 ms

8.10 need resched

Linux 2.6 kjernen deler dynamisk prosessene inn i 140 forkjellige prioritetsklasser. Hver prioritetsklasse tilsvarer et antall tildelte ticks i starten av en epoke. Hver gang scheduleren kalles, velges prosessen som har høyest prioritet og den kjører til den har brukt opp alle sine tildelte ticks. Deretter kalles scheduleren på nytt. Men hvis det i løpet av epoken kommer et interrupt, for eksempel fra tastaturet, vil et flagg `need_resched` bli satt og scheduler på grunn av dette kjøres etter neste timer-tick. En interaktiv prosess vil få mange ticks hver epoke og komme i en høy prioritetsklasse. Dermed vil den alltid velges før CPU-intensive prosesser etter en context switch. Men om ikke `need_resched` ble satt ved et interrupt fra for eksempel tastaturet, ville den interaktive prosessen som skulle hatt og prosessert dette tegnet måtte vente helt til en kjørende CPU-intensiv prosess var ferdig med alle sine tick i en epoke, før den slapp til etter en contex switch. At `need_resched` settes gjør at det skjer en context switch med en gang ved neste timer tick og interaktive prosesser får dermed en mye bedre responstid.

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure 55: Fig 1-18 fra Tanenbaum

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Figure 56: Fig 1-23 fra Tanenbaum

8.11 Simulering av hvordan man ved hjelp av et operativsystem kan holde forelesning og lage vaffelrøre samtidig

I år (2025) vil denne simuleringen gjennomføres i levende live i PH170 onsdag 12. mars og ikke bare som videopptak slik som under pandemien.

Video som demonstrerer multitasking av forelesning og vaffelrørelaging[171]

Når en datamaskin skal kjøre to prosesser samtidig, må den strengt skille mellom oppgavene de to gjør og fordele CPU-ressurser mellom de to. Et menneske er relativt dyktig til å gjøre to ting samtidig, men om man f. eks. skal forelese og lage vaffelrøre samtidig, kan det være nyttig (vel.....) å ha et program (operativsystem) som systematisk fordeler hjernebruken. Nedenfor følger instruksjoner for vaffelrøre, forelesning samt et styringsprogram som illustrerer hvordan et OS gjør scheduling. Sidene nedenfor kan sammenlignes med internminnet på en maskin og hardware utfører instruksjoner herfra i en evig løkke.

```
Operativsystem, data
mem[1] = OST1  # timer
mem[2] = OSM1  # I/O melk

Operativsystem, kode
OSI1 stack = PC                      # I = Interrupt. Hjelm!
OSI2 disableInterrupts
OSI3 JMP mem[IRQ]

OSM1 legg melk i buffer              # M = melk
OSM2 set needResched
OSM3 enableInterrupts
OSM4 switch to user modus            # Ta av hjelm
OSM5 PC = stack

OST1 updateCounterCurrentProcess        # T = timer kall
OST2 If(countersum == 0)
        counter = priority;  JMP OSS1   # ny epoke
OST3 If(counter == 0)
        removeFromReadyList; JMP OSS1   # Call scheduler
OST4 if(! needResched)
        switch to user modus            # Ta av hjelm
        PC = stack

OSS1 Les Ready List                     # S = scheduler
OSS2 Velg prosess
OSS3 if(ny prosess)
            Context Switch
            OSS3a  stack -> OldPCB.PC   # Lagre gammel PCB
            OSS3b  NyPCB.PC -> Stack    # Laste ny PCB
OSS4 Sett på timer                      # ringer om ett minutt = tick/jiffie
OSS5 enableInterrupts
OSS6 switch to user modus               # Ta av hjelm
OSS7 PC = stack

OSmelk1 stack = PC
OSmelk2 disableInterrupts
```

[171] https://os.cs.oslomet.no/os/vaffel.mp4

```
OSmelk3 hent melk                     # Be I/O melkemannen om melk
OSmelk4 block prosess                 # Fjern fra Readylist
OSmelk5 JMP OSS1                      # Call scheduler

OSegg1 stack = PC
OSegg2 disableInterrupts
OSegg3 knus egg
OSegg4 enableInterrupts
OSegg5 switch to user modus           # Ta av hjelm
OSegg6 PC = stack
```

Vaffelprosess (V) med PCB (Process Control Block, process descriptor)

```
PCB
counter  0
priority 3
PC V1

V1  100g sukker
V2  100g mel
V3  visp
V4  blandet?
V5  JMPNB V3  (Jump Not Blandet)
V6  syscall trap OSmelk1
V7  1 dl melk
V8  Visp
V9  blandet
V10 JMPNB V8
V11 syscall trap OSegg1
V12 Visp
V13 blandet?
V14 JMPNB V12
V15 JMPNOK V6
```

Forelesningsprosess. Siden den har prioritet 2, får den i snitt kjøre litt mindre enn vaffelprosessen.

```
PCB
counter  0
priority 2
PC F1

F1 Utviklingen av Linux
F2 Linux ble utviklet av datastudenten
F3 Linus Torvalds på en hybel i Helsinki
F4  år   Versjon brukere kodelinjer
F5  1991  0.01   1       10K
F6  1992  0.96   1000    40K
F7  1994  1.0    100.000 170K
F8  1996  2.0    1.5M    400K
F9  1999  2.2    10M     1M
F10 2001  2.4    20M     3.3M
F11 2003  2.6    25M     5.9M
```

```
F12 2009  2.6.32 ?       9.7M
F13 2010  2.6.36 ?      10.9M
F14 2011  3.0    ?      11.9M
F15 2012  3.3    59M    12.3M
F16 2015  4.0    ?      15.5M
F17 2017  4.10   89M    17.0M
F18 2019  4.19   ?      17.3M
```

```
Operativsystem, data}
ready list: F, V
needResched
stack:                         # Prosessen det ble hoppet fra

CPU-registre}
PC Program Counter
(pekes på av kulepenn)

IRQ:
```

Operativsystemet opererer i priviligertmodus og bruker derfor sykkelhjelm. Når vaffelrøre prosessen stoppes må hvilken instruksjon den har kommet til lagres i PCB, slik at prosessen fortsetter fra der den slapp neste gang. Deretter leses ready-list. Hvis forelesningen ikke er ferdig, står den der og adressen til neste instruksjon er lagret i denne prosessens PCB. OS oppdaterer Ready list, setter på timer, switcher til bruker modus og legger neste F-instruksjon i PC. Forelesningen fortsetter til timeren ringer og scheduler kalles på nytt. Ved ønske om å knuse egg (som er for risikofylt for at en vanlig bruker kan gjøre det) må vaffelprosessen gjøre et systemkall (OSegg). Da switches modus, systemkallet utføres, modus switches tilbake og vaffelprosessen kan fortsette. counter minskes med en for hver gang timeren ringer for den prosessen som kjøres. Når alle countere er null er en epoke over og en ny innledes ved at alle countere blir satt lik prosessenes prioritet.

Video som demonstrerer multitasking av forelesning og vaffelrørelaging[172]

8.12 Dagens faktum: Linux

Linux er nå snart 32 år gammelt:

```
> From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
> Newsgroups: comp.os.minix
> Summary: small poll for my new operating system
> Date: 25 Aug 91 20:57:08 GMT
>
> Hello everybody out there using minix -
>
> I'm doing a (free) operating system (just a hobby, won't be big and
> professional like gnu) for 386(486) AT clones.  This has been brewing
> since april, and is starting to get ready.  I'd like any feedback on
> things people like/dislike in minix, as my OS resembles it somewhat
> (same physical layout of the file-system (due to practical reasons)
> among other things).
>
> I've currently ported bash(1.08) and gcc(1.40), and things seem to work.
```

[172] https://os.cs.oslomet.no/os/vaffel.mp4

```
> This implies that I'll get something practical within a few months, and
> I'd like to know what features most people would want.  Any suggestions
> are welcome, but I won't promise I'll implement them :-)
>
>               Linus (torvalds@kruuna.helsinki.fi)
```

Linux ble utviklet av Linus Torvalds (f. 1969) på en hybel i Helsinki.

år	Versjon	brukere	kodelinjer
1991	0.01	1	10K
1992	0.96	1000	40K
1994	1.0	100.000	176K
1995	1.2	500.000	311K
1996	2.0	1.5M	400K
1999	2.2	10M	1M
2001	2.4	20M	3.3M
2003	2.6	25M	5.9M
2009	2.6.32	?	9.7M
2010	2.6.36	?	10.9M
2011	3.0	?	11.9M
2012	3.3	59M	12.3M
2015	4.0	?	15.5M
2017	4.10	89M	17.0M
2019	4.19	?	17.3M
2022	5.16	?	22.6M
2024	6.7.8	?	?

Torvalds leder fortsatt utviklingen av kjernen. Linux kildekode er fri, GNUs public licence, GPL. Tusenvis av programmerere fra hele verden har bidratt til prosjektet. En Eu-studie fra 2006 anslår at det ville koste 882M euro å utvikle 2.6.8-kjernen på nytt fra bunnen av.

9 Forelesning 19/3-24(2 timer). Prosesser, OS-arkitektur

`Avsnitt fra Tanenbaum: 2.1 - 2.2`

9.1 Video av forelesningen

Opptak av forelesningen:

os9time1.mp4[173] (41:31) Uredigert opptak av første time av forelesningen.
os9time2.mp4[174] (53:05) Uredigert opptak av andre time av forelesningen.

Opptak av forelesningen inndelt etter temaer: (lydkvaliteten blir desverre litt dårligere, men høyere, mitt i avsnittet om nice)

os9del1.mp4[175] (03:10) Intro, om kurset fremover, om dagens temaer, eksamen
os9del2.mp4[176] (07:09) Om ukens oppgaver, docker og ryddescript, dockerfiles, lite disk på serveren
os9del3.mp4[177] (18:15) Demo av systemkall med getppid.c
os9del4.mp4[178] (04:15) Slides: Prioritet og scheduling-algoritmer
os9del5.mp4[179] (07:19) Slides og demo: nice; lydkvaliteten blir desverre dårligere mitt i opptaket
os9del6.mp4[180] (10:01) Demo: hvorfor ingen effekt av nice? nice med taskset
os9del7.mp4[181] (01:00) Slides: prioritet i Windows
os9del8.mp4[182] (04:02) Slides og demo: Prosessforløp
os9del9.mp4[183] (00:58) Slide: Schedulingbegreper
os9del10.mp4[184] (05:16) Slides: Lage nye prosesser, parent og child, Linux fork
os9del11.mp4[185] (01:22) Slide: Windows CreateProcess
os9del12.mp4[186] (04:41) Slides: avslutte prosesser og signaler
Reprise fra femte linux-forelesning: linux5del15.mp4[187] (04:51) Demo: Signaler og trap
os9del13.mp4[188] (03:46) Slide: Linux arkitektur
os9del14.mp4[189] (03:55) Slide: Windows arkitektur
os9del15.mp4[190] (13:59) Demo: C-program med fork

9.1.1 Slides brukt i forelesningen

Slides brukt i forelesningen[191]

[173] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9time1.mp4
[174] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9time2.mp4
[175] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del1.mp4
[176] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del2.mp4
[177] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del3.mp4
[178] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del4.mp4
[179] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del5.mp4
[180] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del6.mp4
[181] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del7.mp4
[182] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del8.mp4
[183] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del9.mp4
[184] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del10.mp4
[185] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del11.mp4
[186] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del12.mp4
[187] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/linux5del15.mp4
[188] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del13.mp4
[189] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del14.mp4
[190] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os9del15.mp4
[191] https://www.cs.oslomet.no/ haugerud/os/Forelesning/os9.pdf

9.2 Sist

- Systemkall
- Scheduling
- OS-vaffelsimulering

9.3 Systemkall og timer ticks

Et eksempel på et systemkall er getppid som returnerer PID for prosessens parent, foreldre-prosessen som startet prosessen.

```
rex:~$ grep getppid /usr/src/linux-source-4.4.0/arch/x86/entry/syscalls/syscall_64.tbl
110 common getppid sys_getppid
```

Som vi ser er getppid systemkall nummer 110 for Linux sin 4.4 kjerne. Programmeringsspråket C ble laget samtidig som operativsystemet Unix og er dermed også tett beslektet med Linux. Mange av systemkallene i Linux er egne funksjoner i C. Det er tilfellet for getppid og følgende er et C-program som gjør dette systemkallet 10 millioner ganger.

```
#include<unistd.h>

int main(void) {
   int i;
   for (i=0; i<10000000; i++) {
      getppid();
   }
   return(0);
}
```

Hvis vi ser på timer-ticks før og etter dette programmet, vil vi se at de fleste av tickene finner sted i kernel eller system mode. Vi tester dette med følgende script `sys.sh`:

```
#! /bin/bash

grep cpu3 /proc/stat
taskset -c 3 ./getppid
grep cpu3 /proc/stat
```

Dette scriptet starter getppid-programmet på CPU 3 og leser ticks fra den CPUen før og etter. Ett tick (også kalt jiffie) varer i 10 millisekunder (ett hundredels sekund) og om for eksempel en CPU bruker all tid i user mode når en prosess kjøres på den, vil denne kolonnen øke med 100 ticks i løpet av ett sekund. De fire første kolonnene i `/proc/stat` er som følger:

- user (1) Time spent in user mode.
- nice (2) Time spent in user mode with low priority (nice).
- system (3) Time spent in system mode.
- idle (4) Time spent in the idle task.

og resultatet av kjøringen er:

```
rex:~$ ./sys.sh
cpu3 6354414 5212787 2789939 218067966 804787 0 198336 0 0 0
cpu3 6354490 5212787 2790067 218067966 804787 0 198336 0 0 0
```

Vi ser at det kun er kolonnene user (1) og system (3) som har endret seg. Av differansen mellom de to første kolonnene ser vi at ved 76 av tickene som ble foretatt mens programmet kjørte, ble programmet kjørt i user-mode. Av differansen i tredje kolonne, ser vi at 128 av tickene skjedde mens programmet var i system elle kernel-mode. Det siste betyr at operativsystemet utførte et systemkall, getppid, på vegne av programmet som ble startet opp i user-mode. Totalt antall ticks adderer opp til 204 og det betyr at programmet totalt brukte 2.04 sekunder CPU-tid på CPU nr 3.

Omtrent det samme kan vi konkludere om vi bruker time-kommandoen til å ta tiden på prosessen:

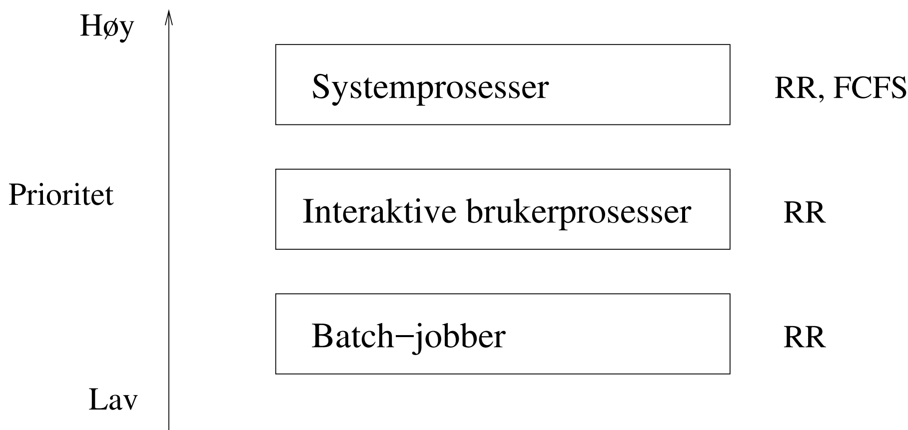
```
rex:~/undervisning/OSogUNIX/unixnotater/virt$ time ./getppid
Real:2.068 User:0.772 System:1.292 99.83%
```

som viser at programmet bruker 0.77 sekunder i user-mode og 1.29 sekunder i kernel-mode. Hvis man tar tiden på sys.sh scriptet ser man at det stemmer meget godt overens.

```
rex:~$ time ./sys.sh
cpu3 6355584 5212992 2790765 218120204 804824 0 198434 0 0 0
cpu3 6355662 5212992 2790894 218120205 804824 0 198434 0 0 0
Real:2.072 User:0.780 System:1.284 99.59%
```

9.4 Prioritet

Det er vanlig å kategorisere prosesser og gi dem prioritet etter hvilken kategori de tilhører.



Hvis en prosess i en høyere prioritetsklasse ønsker å kjøre, for eksempel en kjerne-prosess, overtar den med en gang for prosesser fra en lavere prioritetsklasse og kjører til den er ferdig. Innen samme prioritetsklasse tildeles flere timeslices til prosesser med høyere prioritet, slik at de innen samme Round Robin-kø får mer CPU-tid men kjøres samtidig. FCFS (First Come First Served) er en annen scheduler algoritme som brukes for kjerne-tråder (kthreads) som må bli helt ferdige med det de skal gjøre før de avsluttes.

9.4.1 Scheduling-algoritmer

Scheduling (skedulering) betegner organiseringen av hvordan man tildeler ressurser til en arbeidsoppgave som skal gjennomføres. I mange sammenhenger trenger man algoritmer som sørger for at en arbeidsoppgave blir effektivt fullført og som fordeler tid eller andre ressurser, organiserer jobbflyt og hvordan prosesser utføres og det trenger ikke nødvendigvis å være prosesser i en datamaskin. Noen vanlige algoritmer er:

- RR (Round Robin) Prosesser kjører på omgang, litt tid hver runde
- FCFS (First Come First Served) Den første prosessen blir først prosessert
- FIFO (First In Firs Out) Samme som FCFS
- SJF (Shortest Job First) Den prosessen som tar kortest tid er den neste som kjøres

9.4.2 Linux-eksempel: nice

```
$ nice -n 9 regn     # Setter nice-verdi til 9 for prosess regn
$ renice +19 25567   # Endrer nice-verdi til 19
```

- nice $\longrightarrow$ vær snill med andre prosesser
- Høyere niceverdi gir mindre CPU-tid til prosessen
- default niceverdi er 0
- top viser niceverdier

9.5 Prosess-prioritet i Windows

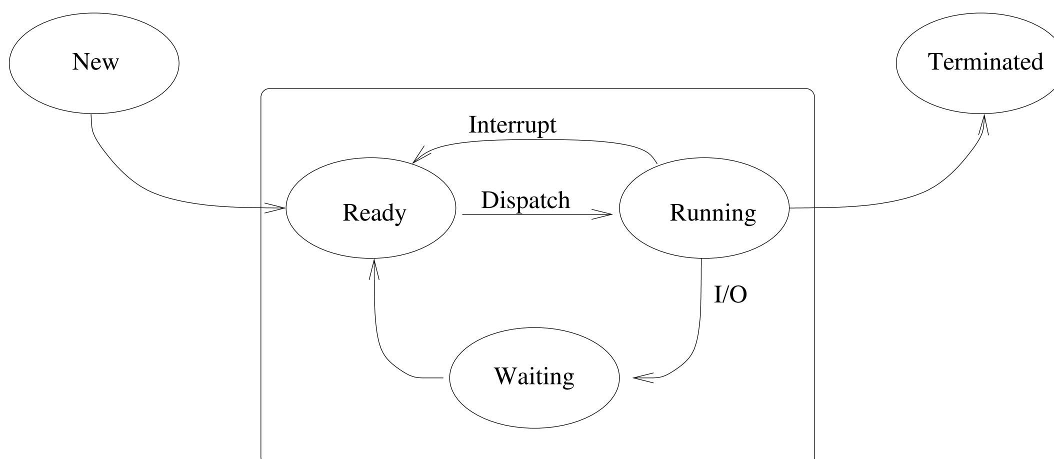
Windows scheduling har store likeheter med Linux. Prosesser får i utgangspunktet en prioritet som kan endres dynamisk. CPU-prosesser gis gradvis mindre, I/O prosesser gradvis mer. Det finnes 4 forhåndsdefinerte prioritetsklasser:

- IDLE PRIORITY CLASS (prioritet 4)
- NORMAL PRIORITY CLASS (8)
- HIGH PRIORITY CLASS (13)
- REALTIME PRIORITY CLASS (24)

Prioritet for en prosess settes til verdier mellom 1 og 31. Prioritet endres dynamisk og prosessen som kjører et aktivt vindu, gis økt prioritet. Dette kan endres fra task-manager hvis man har admin-rettigheter og først velger 'go to details'. Men det er svært stor forskjell på prioritestnivåene som blir satt i task-manager.

9.6 Prosessforløp

Denne figuren viser de viktigste tilstandene i et prosessforløp. Prosesser som ligger i ready-list ønsker så snart som mulig å bli tildelt tid i prosessoren og dermed komme over i tilstand running. Prosesser som venter av fri vilje eller som for eksempel må vente på Input/Output (I/O) settes i waiting-tilstand.



9.6.1 Sentrale schedulingbegreper

Enqueuer
- Legger i kø
- Beregner prioritet

Dispatcher
- Velger prosess fra READY LIST; liste med prosesser som er klare til å kjøre

9.6.2 Prosessforløp-demo

En mp4-demo av dette prosessforløpet kan sees her[192]

9.7 Lage en ny prosess

Det mest sentrale konseptet for et operativsystem er prosessen. De følgende avsnitt viser hvordan prosesser fødes, lever og dør på Linux og Windows. Alle moderne OS har en mekanisme for å lage nye prosesser. Prosesser lages ved

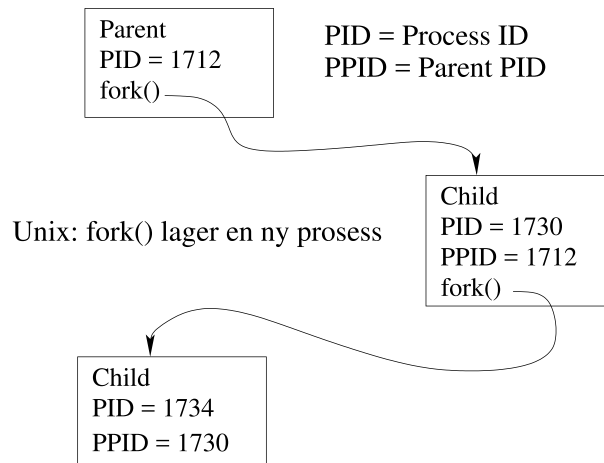
- System oppstart (Linux: init-prosessen)
- En kjørende prosess utfører et systemkall som staret en ny prosess
- En bruker ber om at en prosess startes

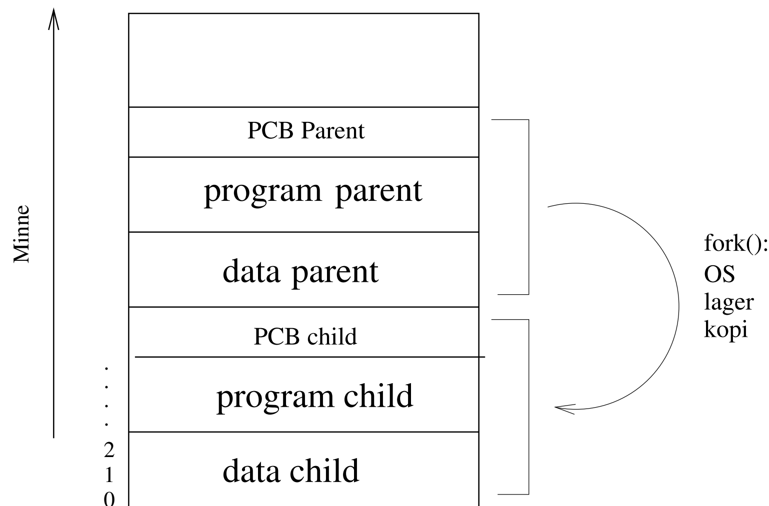
Bortsett fra ved systemoppstart er det alltid en prosess som lager en annen og den prosessen kalles en forelder-prosess. I prosessverdenen er det bare en forelder til ett eller flere barn. En parent-prosess lager en child-prosess.

9.7.1 Linux: fork()

fork() er et Linux-systemkall for å lage en child-prosess. fork() lager en kloning, identisk prosess med kopi av program, data og PCB (bortsett fra PID, PPID og noen andre).

[192]https://os.cs.oslomet.no/os/demoer/prosess.mp4





Vanligvis starter child med å laste inn koden for det programmet det skal kjøre, slik at child-prosessen likevel blir forskjellig fra parent-prosessen.

Linux-prosesser lager på denne måten et hierarki av prosesser med barn og barnebarn. Det kan da være mulig å sende signaler til alle prosessene i et hierarki.

9.7.2 Windows: CreateProcess

Win 32 API'et har også støtte for fork(), men standardmetoden for å lage en ny prosess er å gjøre et kall til `CreateProcess` med 10 parametre. Da lages et nytt prosess-objekt; hvilket program som skal kjøres, vinduer som skal åpnes, prioritet mm overføres med parameterene til kallet. Bindingen mellom parent og child er ikke like sterk som under Linux. Windowsprosesser kan gjøre sine barn arveløse.

9.8 Avslutte prosesser

Vanligvis avsluttes prosesser når jobben er ferdig. Noen prosesser, såkalte daemons, kjører hele tiden mens systemet er oppe. Prosesser avsluttes ved:

- Normal avslutning. Frivillig. Linux: exit, Windows: ExitProcess
- Avslutning ved feil. Frivillig. (f. eks. 'file not found')
- Fatal feil. Ufrivillig. (division by zero, Segmentation fault, core dumped)
- Drept av annen prosess. Ufrivillig. Linux: kill, Windows: TerminateProcess

9.8.1 Signaler

Prosesser kan kommunisere med hverandre ved hjelp av signaler. En bruker kan også sende et signal til en prosess, CTRL-C er et eksempel på et slikt signal som prøver å avslutte prosessen. Under Linux kan en prosess selv velge hva den skal gjøre når den mottar et signal. For eksempel er standard oppførsel for en prosess å avslutte når den mottar et CTRL-C signal, men den kan velge å ignorere det. Med kommandoen kill kan man sende mange forskjellige signaler og `kill -9` er et såkalt ustoppelig signal som dreper prosessen enten den vil eller ikke. For kill-signaler er det en forutsetning at den som sender signalet har riktig rettigheter, ellers vil signalet ikke ha noen effekt.

9.8.2 Signaler og trap i bash-script

En prosess kan stoppes av andre prosesser og av kjernen. Det gjøres ved å sende et signal. Alle signaler bortsett fra SIGKILL (`kill -9`) kan stoppes og behandles av bash-script med kommandoen `trap`. Følgende script kan bare stoppes ved å sende (`kill -9`) fra et annent shell.

```
#! /bin/bash

# definisjoner fra fra /usr/src/linux-2.2.18/include/asm-i386/signal.h:
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGTERM         15      /* Termination (ANSI).  */
#define SIGTSTP         20      /* Keyboard stop (POSIX).  */

trap 'echo -e "\rSorry; ignores kill -1 (HUP)\r"' 1
trap 'echo -e "\rSorry; ignores kill -15 (TERM)\r"' 15
trap 'echo -e "\rSorry; ignores CTRL-C\r"' 2
trap 'echo -e "\rSorry; ignores CTRL-Z\r"' 20
trap 'echo -e "\rSorry; ignores kill - 3 4 5\r"' 3 4 5
trap 'echo -e "\rCannot stop kill -9\r"' 9

while [ true ]
do
   echo -en "\a quit? Answer y or n: "
   read answer
   if [ "$answer" = "y" ]
        then break
   fi
done
```

9.9 OS arkitektur

Den vanligste om ikke den beste måten å designe et OS på er en såkalt monolittisk arkitektur. Linux er et typisk eksempel på dette. Alle metoder i koden har tilgang til alle datastrukturer i kjernen. Dette gjør at kjernen blir hurtig og effektiv, men åpner for flere bugs. Windows NT kjernen er objektorientert og skrevet i C++ og har gjennom dette en ryddigere struktur. En ulempe med Windows er at GUI delvis inngår i kjernen slik at den blir mye mer omfattende. En bedre arkitektur er en såkalt microkernel. Bare det helt essensielle utføres av kjernen, som multitasking og behandling av interrupts. Resten utføres utenfor kjernen av prosesser i usermode som kommuniserer med hverandre og mikrokjernen.

9.9.1 Linux arkitektur

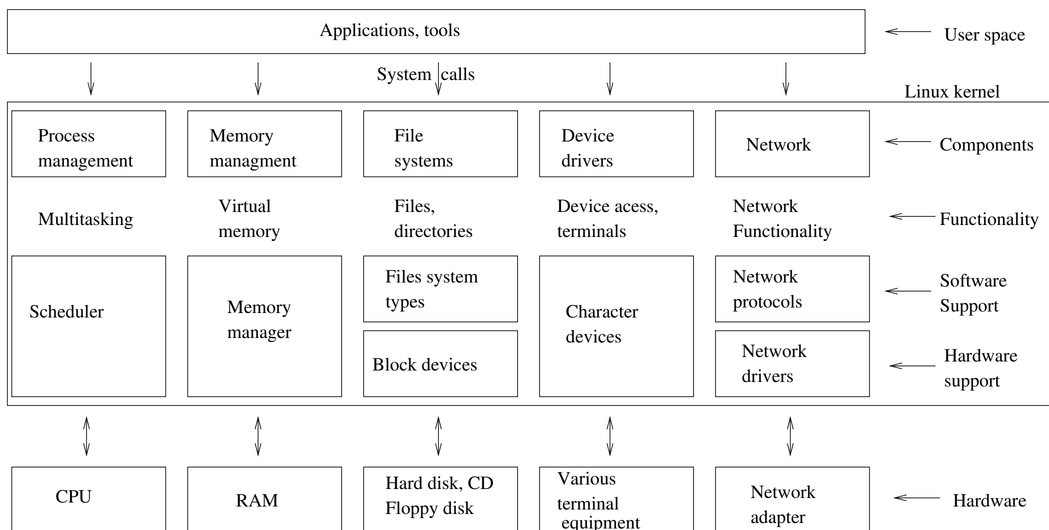


Figure 57: Linux arkitektur

9.9.2 Windows arkitektur

9.10 Design av systemkommandoer

Analogt med forholdet mellom privilegert modus og usermodus er forholdet mellom superuser og en vanlig bruker. Men et program som kjører som root har bare utvidede rettigheter i forhold til filsystemet og devicer, det kjøres ikke som en del av OS-kjernen. To muligheter (eksempler fra Linux) ved design av systemkommandoer.

1. Root-rettigheter (noen få: ping, mount, su, ...)
   - tilgang på alle filer
   - mye sikkerhets-overhead (alt må sjekkes)
   - stor sikkerhetsrisiko (kan være sikkerhetshull)
   - betrodd software
2. Vanlig bruker-rettigheter (de fleste: ls, ps, mv,...)

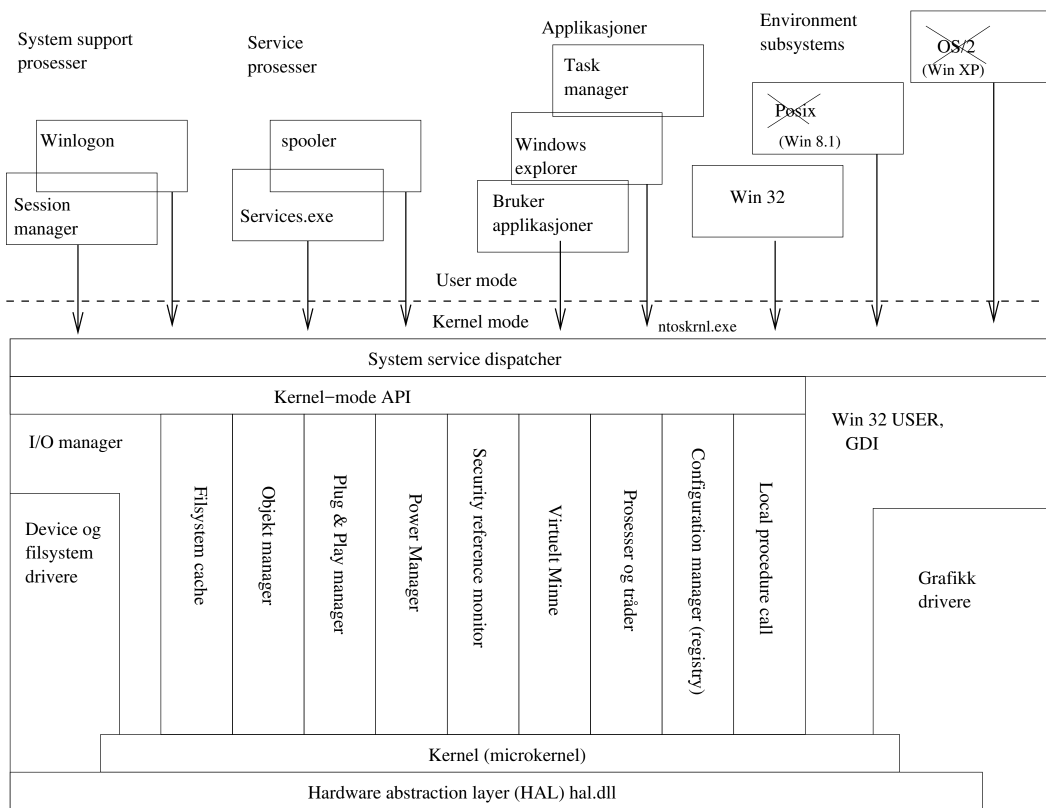


Figure 58: Windows arkitektur

- Sikrere (*begrensede rettigheter*)
- Lite feilsjekking $\longrightarrow$ lite overhead

9.10.1 Linux-eksempel: setuid-bit

```
group1@osG1:~$ ls -l /bin/ls
-rwxr-xr-x 1 root root 114032 jan.  26  2013 /bin/ls
group1@osG1:~$ ls -l /bin/ping
-rwsr-xr-x 1 root root 36136 april 12  2011 /bin/ping
```

Kommandoen /bin/ls kjører med vanlige brukerrettigheter, men for /bin/ping betyr s'en på eierrettighetene at setuid-bit er satt. Dette betyr at en vanlig bruker kjører ping med root-rettigheter. Setuid-programmer er en stor sikkerhetsrisiko. Setuid-bit settes med

```
$ chmod 4755 program  # Setter SETUID-bit
$ chmod 0755 program  # Skrur av SETUID-bit
```

9.11 Utbredelse av Operativsystemer

Det finnes mange undersøkelser om utbredelsen av operativsystemer og det er vanskelig å få en eksakt oversikt fra åpne kilder.

9.11.1 Desktop OS

Følgende er statistikk for desktop-OS basert på hva slags OS de som er inne på web-sider bruker. Slik så tallene ut i 2007:

```
Windows XP      85.30%
Windows 2000    5.00%
Mac OS          4.15%
Windows 98      1.77%
MacIntel        1.52%
Windows ME      0.89%
Windows NT      0.68%
Linux          0.37%
Windows Vista   0.16%
```

Tallene er hentet fra marketshare.hitslink.com. [193]. De blir laget månedlig fra statistikk over hva slags OS som blir brukt av ca 160 millioner besøkende pr måned som er innom mer enn 40.000 forskjellige webservere verden over. En annen kilde med lignende statistikk er https://gs.statcounter.com. [194].

I 2009 så det slik ut:

```
Windows XP      63.76%
Windows Vista   22.48%
Mac OS X 10.5   5.28%
Mac OS X 10.4   2.74%
Windows 2000    1.37%
Mac OS X        1.00%
Linux           0.83%
```

2012:

```
Desktop Operating System Market Share February, 2012
Windows XP     45.39%
Windows 7      38.12%
Windows Vista  8.10%
Mac OS X 10.6  3.00%
Mac OS X 10.7  2.69%
Linux          1.16%
Mac OS X 10.5  0.95%
Mac OS X 10.4  0.23%
Windows 2000   0.15%
Windows NT     0.06%
Windows 98     0.05%
```

2017

```
TOTAL MARKET SHARE
Windows 7      48.41%
Windows 10     25.19%
Windows XP     8.45%
```

[193]https://marketshare.hitslink.com
[194]https://gs.statcounter.com

```
Windows 8.1 6.87%
Mac OS X 10.12  2.91%
Linux           2.05%
Windows 8       1.65%
Mac OS X 10.11  1.55%
Mac OS X 10.10  1.00%
Windows Vista   0.78%
Windows NT      0.39%
Mac OS X 10.9   0.35%
```

2019

```
Windows 7 40.17%
Windows 10 37.35%
Windows 8.1 4.87%
Mac OS X 10.13 4.36%
Windows XP 3.91%
Mac OS X 10.14 1.88%
Mac OS X 10.12 1.51%
Linux         1.49%
Windows 8 0.99%
Mac OS X 10.11 0.98%
```

2021

```
Windows 10 55.17%
Windows 7 27.09%
Windows 8.1 3.41%
Mac OS X 10.14 3.34%
Mac OS X 10.15 2.96%
Mac OS X 10.13 1.52%
Linux Linux 1.37%
Windows XP 1.35%
Linux Ubuntu 0.81%
Mac OS X 10.12 0.65%
```

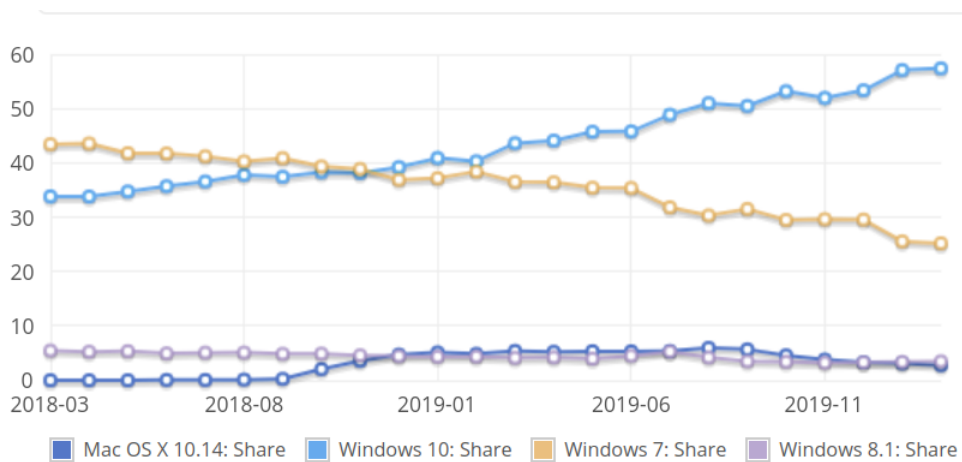


Figure 59: Slik så trenden ut mellom 2018 og 2019, ikke så store endringer frem til 2021.

Tilsvarende tall for webbrowsere er som følger;

```
2007
Internet Explorer  79.64%
Firefox            14.00%
Safari             4.24%
Opera             0.87%
Netscape           0.85%
Mozilla            0.22%


2009
Internet Explorer  67.55%
Firefox            21.53%
Safari             8.29%
Chrome             1.12%
Opera              0.70%


2012
Internet Explorer  52.84%
Firefox            20.92%
Chrome             18.90%
Safari             5.24%
Opera              1.71%


2017
Chrome             58.22%
Internet Explorer  19.45%
Firefox            11.73%
Microsoft Edge     5.51%
Safari             3.46%
Opera              1.26%


2019
Chrome           65.00%
Internet Explorer 10.23%
Firefox           9.72%
Edge           4.33%
Safari           3.74%
Opera           1.57%


2021
Chrome                 68.75
Firefox                 7.91
Edge                 7.36
Internet Explorer 5.86
Safari                 3.64
QQ                 1.76
Sogou Explorer         1.67
Opera                 1.22
```

9.11.2 Server OS

Servertall er vanskligere å finne fra åpne kilder. Netcrafts rapport fra Mars 2008 visete at av de 162 millioner webserverene de hadde mottatt respons fra i sin undersøkelse, kjørte ca 60% apache(typisk

med Linux som OS), ca 30% Microsoft webserver og 10% på andre plattformer. Fire år etter så det litt anderledes ut, apache hadde økt andelen noe. (Se detaljer her.[195]) I 2019 hadde bildet endret seg med flere Microsoft webservere og (detaljene kan sees her.[196])

I 2020 har bildet igjen endret seg og Nginx er den mest brukte webserveren. (detaljene kan sees her.[197])

Det mest korrekte bildet får man trolig av å se på 'active sites' som har reelle webservere som leverer innhold og ikke kun er reklame for f.eks. salg av domenenavn.

En wikipedia artikkel[198] prøver å gi en oversikt over utbredelse av server-OS.

[195] http://news.netcraft.com/archives/2012/03/05/march-2012-web-server-survey.html
[196] https://news.netcraft.com/archives/2019/02/28/february-2019-web-server-survey.html
[197] https://news.netcraft.com/archives/2020/
[198] https://en.wikipedia.org/wiki/Usage_share_of_operating_systems

10 Forelesning 26/3-24(2 timer). Plattformavhengighet og Threads

Avsnitt fra Tanenbaum: 2.2

Slides brukt i forelesningen[199]

10.1 Video av forelesningen

Uredigert opptak av hele første time av forelesningen (00:35:31)[200]

Uredigert opptak av hele andre time av forelesningen (00:43:06)[201]

Opptak av forelesningen inndelt etter temaer:

os10del1.mp4[202] (04:18) Intro om dagens temaer, plattformavhengighet og threads, oppsummering av det vi gjorde sist
os10del2.mp4[203] (13:52) Web-notater: Å kjøre Java, C og bash-programmer under forskjellige OS
os10del3.mp4[204] (23:12) Demo: Test av C, Java, Python og bash på 5 plattformer (desverre blir lydkvaliteten plutselig dårlig 14 minutter ut i dette opptaket)
os10del4.mp4[205] (02:57) Gjennomgang av spørsmålene til poll om plattformavhengighet (se figur under)
os10del5.mp4[206] (03:37) Spørsmål om python2 vs python3 og print med og uten parenteser.
os10del6.mp4[207] (04:03) Gjennomgang av svarene til poll om plattformavhengighet (se figur under)
os10del7.mp4[208] (02:00) Demo: ARM assembly instruksjoner for sum.c
os10del8.mp4[209] (15:43) Slides: Tråder (threads) vs prosesser, definisjoner, single og multithreading, fordeler med threads
os10del9.mp4[210] (05:09) Slides: Java-threads og scheduling, variabler og tråder
os10del10.mp4[211] (02:16) Notater: Om hvordan Calc.java starter og kjører to tråder

10.2 Sist

- Systemkall
- Prioritet
- Nice
- Prosessforløp
- OS-arkitektur

[199] https://www.cs.oslomet.no/ haugerud/os/Forelesning/10.pdf
[200] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10time1.mp4
[201] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10time2.mp4
[202] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del1.mp4
[203] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del2.mp4
[204] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del3.mp4
[205] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del4.mp4
[206] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del5.mp4
[207] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del6.mp4
[208] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del7.mp4
[209] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del8.mp4
[210] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del9.mp4
[211] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os10del10.mp4

10.3 Å kjøre Java, C og bash-programmer under forskjellige OS

Utgangspunktet er det samme "Hello world!" programmet skrevet i C, Java og Bash. Vi skal prøve å kjøre disse på tre systemer:

- Operativsystemet Linux kjørt på en vanlig PC med Intel Pentium prosessor som kun forstår X86 maskin-instruksjoner
- Operativsystemet Windows kjørt på den samme Intel-PC-en
- Operativsystemet Solaris kjørt på en Sun Sparcstation 20 med en sparc-prosessor som kun forstår Sparc maskin-instruksjoner

Idag er Sun og sparc-prosessor en utdøende rase og det finnes ikke så mange av dem lenger. Men det er et eksempel på en datamaskinarkitektur som er helt forskjellig fra x86 og Sparc CPUer var tidligere veldig mye i bruk i Unix-servere. Nå har dette markedet i stor grad blitt tatt over av Linux- og Windows-servere som stort sett kjører på Intel og AMD CPUer med x86-arkitektur.

10.3.1 Hello.java

Dette Java-programmet ser slik ut

```
$ cat Hello.java
class Hello
{
    public static void main(String args[])
    {
        System.out.println("Java: Hello world!");
    }
}
```

og som de to andre, skriver det bare ut en linje til skjermen. For å kjøre dette programmet må det kompileres og vi gjør det på Linux-maskinen:

```
$ javac Hello.java
```

da lages det en binær fil med navn `Hello.class` som inneholder såkalt bytekode. Denne koden er instruksjoner til en såkalt Java Virtual Machine (JVM) som er et program som kjører bytekoden og får det underliggende systemet til å utføre det denne koden sier skal gjøres. For hvert operativsystem er det en egen JVM som sørger for dette slik som vi ser i figuren:

Dermed kan `Hello.class` filen som ble kompilert på en Linux-maskinen faktisk kjøre på alle de tre plattformene. Enheten OS + Hardware blir ofte omtalt som en plattform. På grunn av dette sier vi at Java er plattformuavhengig. Det samme er også Perl og C#, deres kompilerte kode kjøres av virtuelle maskiner.

10.3.2 hello.c

Dette programmet ser slik ut

```
cube$ cat hello.c
#include <stdio.h>
main()
{
    printf("c: Hello world!\n");
}
```

For å kjøre dette programmet må det kompileres og vi gjør det på Linux-maskinen:

```
$ gcc hello.c
```

da lages det en binær fil med navn `a.out` som inneholder maskin-instruksjoner og deler av disse kan lastes inn og kjøres direkte på CPU-en til plattformen den er kompilert på. Nå har vi kompilert programmet på en X86-prosessor og da vil `a.out` inneholde X86-instruksjoner som på assembler-form kan se ut som f. eks. `mov %eax,%ebx` og henviser til registrene denne CPU-en har. Kompilatoren oversetter direkte fra kildekode til maskinkode. Assembly er et språk i mellom disse, men som ligger svært nær maskinkoden, og som gjør at vi enkelt kan se nøyaktig hva maskinkoden gjør. For å få kjørbar kode som er veldig kompakt og eller gjør nøyaktig det vi ønsker, er det mulig å skrive programmer direkte i assembly og så få en assembler til å oversette dette til maskinkode og dermed et kjørbart program.

Ved hjelp av reverse engineering kan man prøve å se hva binærkoden til et kompilert program (ofte kalt objekt-kode) inneholder. Programmet `objdump` er en disassembler som oversetter maskinkoden til en for mennesker mer leselig assembly-kode. Dette er det motsatte av hva en assembler gjør.

```
Linux$ objdump -d a.out

8048278:       83 ec 04                    sub    $0x4,%esp
 804827b:       e8 00 00 00 00          call   8048280 <_init+0xc>
 8048280:       5b                      pop    %ebx
 8048281:       81 c3 d8 12 00 00       add    $0x12d8,%ebx
 8048287:       8b 93 fc ff ff ff       mov    -0x4(%ebx),%edx
```

Dette er bare et lite utsnitt av koden og det sentrale i vår sammenheng er at dette er instruksjoner fra det såkalte X86-instruksjonssettet som alle vanlige Intel og AMD-prosessorer bruker.

Dermed kan det umulig gå bra å kjøre `a.out` på en Sparc-prosessor, for den forstår overhode ikke maskin-instruksjonene som blir gitt. Det er rett og slett helt gresk for Sparc-prosessoren. Den forstår bare Sparc-instruksjoner som er et helt annent instruksjonessett som vi kan se hvis vi kompilerer programmet og dumper objekt koden under Solaris:

```
Solaris$  gcc a.out
Solaris$  objdump -d a.out
   10440:       e0 03 a0 40     ld  [ %sp + 0x40 ], %l0
   10444:       a2 03 a0 44     add  %sp, 0x44, %l1
   10448:       9c 23 a0 20     sub  %sp, 0x20, %sp
   1044c:       80 90 00 01     tst  %g1
```

Dette ligner, men instruksjonene er delvis forskjellige og definsjonene av hvilke bit-strenger som betyr en bestemt instruksjon eller et bestemt register er helt forskjellige.

I tillegg inneholder `a.out` kode som ber Linux-operativsystemet om å skrive ut til skjermen og dette er spesielle systemkall som er helt spesifikke for Linux og som andre OS ikke forstår. Dermed går det heller ikke å kjøre dette programmet på Windows-maskinen, selvom maskin-instruksjonene er de samme, alle er hentet fra X86-instruksjonssettet. Som vi ser i figuren er det kun på plattformen programmet er kompilert

at det kan kjøres og vi sier at C er plattformavhengig. Om vi skal kjøre dette programmet på de to andre plattformen, må det kompileres på nytt med en C-kompilator på både Windowsmaskinen og på Solaris-maskinen. På Windows kan det gjøres for eksempel med tinyCC (tcc), eller med en komersiell kompilator som Visual C++. På Solaris finnes det som regel en C-kompilator, det er vesentlig for Unix-maskiner. Da lages det instruksjoner som snakker med det rette OS'et og som inneholder de rette maskininstruksjonene, slik som i figuren.

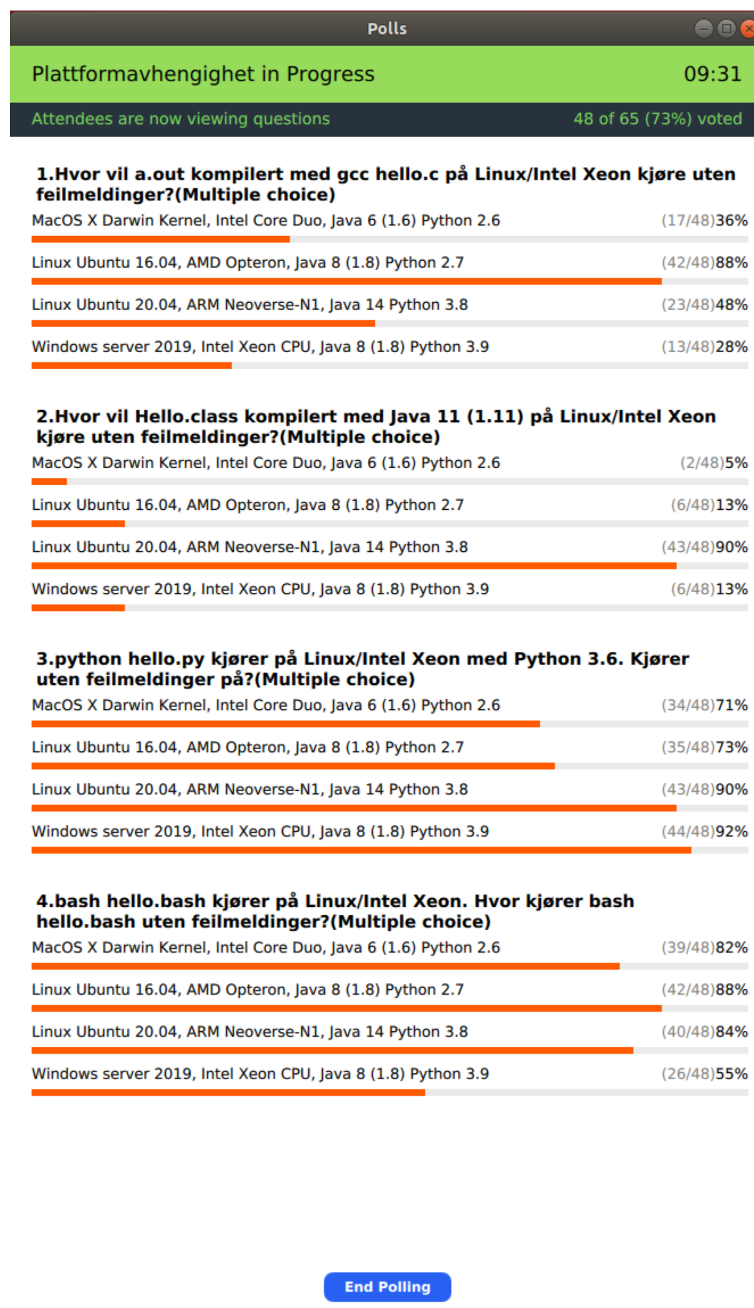


Figure 60: Spørsmål og resultater fra poll om plattformuavhengighet.

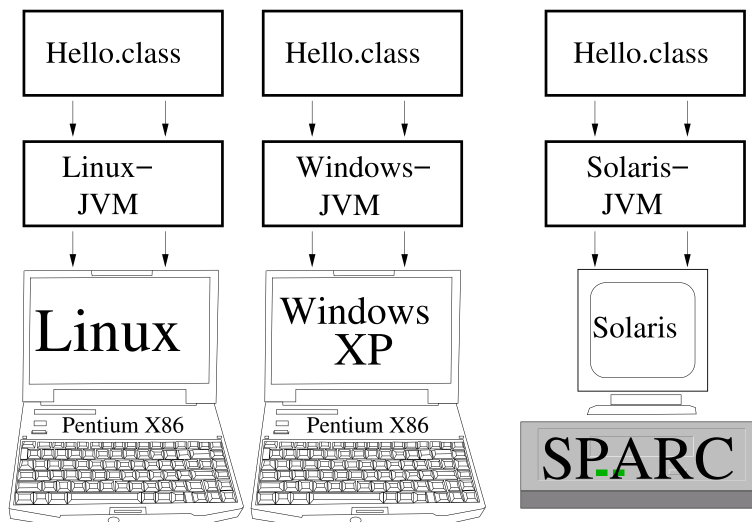


Figure 61: Java er plattformuavhengig og samme Hello.class fil kan kjøres på alle de tre plattformene.

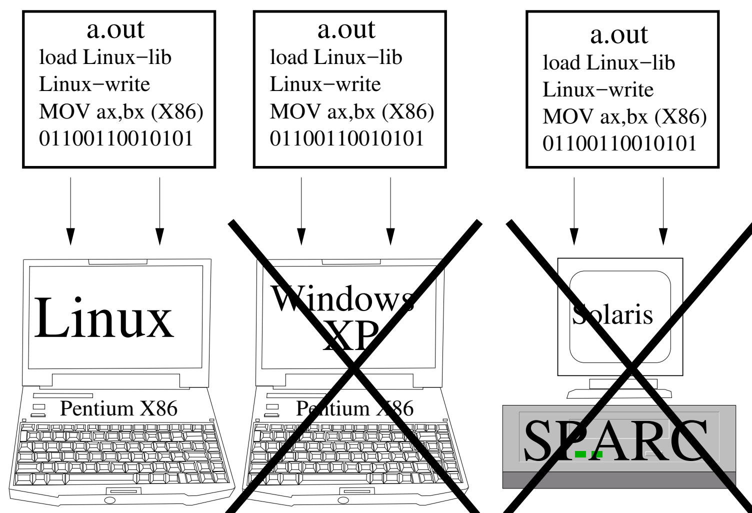


Figure 62: C er ikke plattformuavhengig og samme a.out fil kan ikke kjøres på alle de tre plattformene.

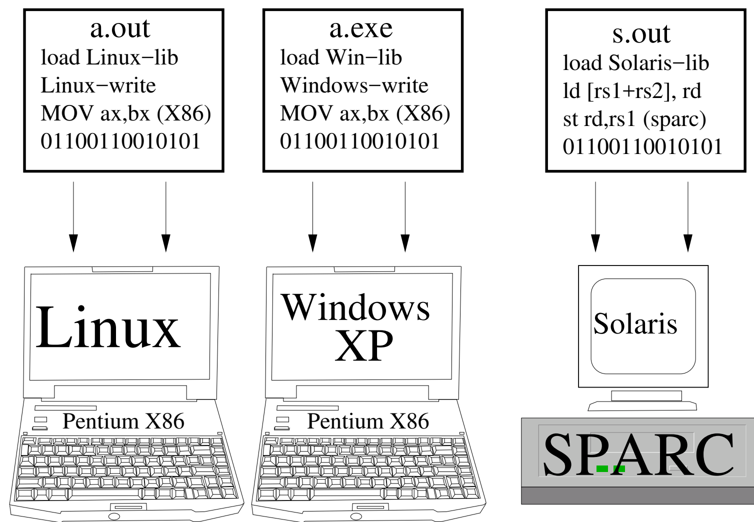


Figure 63: C-programmet må kompileres på hver av de tre plattformene, først da kan de kjøres.

10.3.3 hello.bash

Dette programmet ser slik ut

```
cube$ cat hello.bash
#! /bin/bash
echo "bash: Hello world!"
```

og er avhengig av at shellet bash er installert på plattformen det skal kjøres. Dette programmet tolker da linje for linje og utfører den. Dette er delvis analogt til Java, bortsett fra at mellomleddet med å kompilere og lage bytekode er fjernet. Programmet bash erstatter JVM og kjører koden. Dermed er også bash-script plattformuavhengig. Tidligere var det ikke mulig å kjøre bash på Windows, men i de siste årene har Microsoft samarbeidet mye med Ubuntu og det er nå mulig å aktivere et fullverdig bash-shell i Windows. Men det er ikke aktivert som default.

10.4 Test av C, Java, Python og bash på 5 plattformer

I forelesningen ble et Python 'Hello world' program testet i tillegg til de tre språkene beskrevet over. De fem forskjellige plattformene som var involvert i testen var de følgende.

- Linux Ubuntu 18.04, Intel Xeon, Java 11 Python 3.6 (HP laptop)
- MacOS X Darwin Kernel, Intel Core Duo, Java 6 (1.6) Python 2.6 (Gammel MacBook Pro)
- Linux Ubuntu 16.04, AMD Opteron, Java 8 (1.8) Python 2.7 (Dell server med 48 CPUer)
- Linux Ubuntu 20.04, ARM Neoverse-N1, Java 14 Python 3.8 (Amazon EC2, London)
- Windows server 2019, Intel Xeon CPU, Java 8 (1.8) Python 3.9 (Amazon EC2, London)

Utganspunktet varr at alle programmene ble kompilert og kjørt på den førstnevnte HP-laptopen som kjører Ubuntu 18.04. Kompileringen av programmene ble gjort med

```
gcc hello.c
javac Hello.java
```

og kjøringen med

```
./a.out
java Hello
python hello.py
bash hello.bash
```

10.5 Threads (tråder)

- prosess = At en kokk lager en porsjon middag i et kjøkken
- CPU = kokk
- ressurser = kjøkken, matvarer, oppskrift

- thread/tråd = den sammenhengende serien av hendelser som skjer når kokken lager en porsjon

Når porsjonen er ferdig er porsessen avsluttet. Alternativer:

1. To uavhengige prosesser = to kjøkken, kokken løper frem og tilbake og lager en porsjon i hvert kjøkken. Følger en oppskrift i hvert kjøkken(men oppskriften er den samme).
2. En prosess med to threads = Ett kjøkken, kokken bytter på å jobbe med de to porsjonene og lager to porsjoner fra samme oppskrift med felles ressurser for de to porsjonene.

Med en tradisjonell prosess kan kun en kokk jobbe i et kjøkken og der lage kun en porsjon. Ønsker man å lage flere porsjoner, så må man lage flere kjøkken. Innfører man tråder kan flere kokker jobbe i samme kjøkken med flere porsjoner på en gang.

10.6 Definisjoner av threads

- den sammenhengende rekken av hendelser/instruksjoner som utføres når et program kjøres
- "tråden" som følges når et program utføres
- Lettvekts prosess

Programmereren (og ikke OS) vet hva som skal gjøres. Han kan detaljstyre threads til å samarbeide om oppgaver som skal utføres.

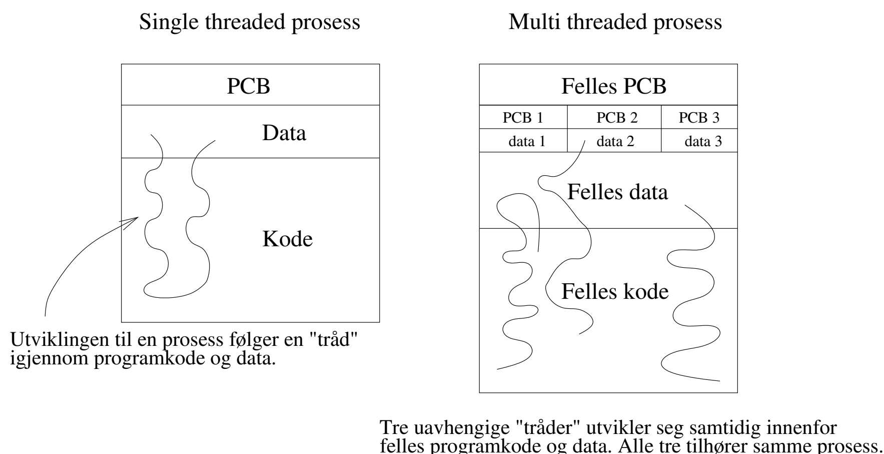


Figure 64: Single og multithreading

Figur 64 viser for en single threaded prosess hvordan programmet gjennom å utføre instruksjoner beveger seg igjennom koden og også frem og tilbake og til RAM. Om man tegner opp denne bevegelsen får man en enkelt "tråd" som beveger seg rundt og illustrerer hvordan programmet kjøres. Om man kjører et program flere ganger, kan det følge forskjellige tråder hvis for eksempel input er forskjellig fra gang til gang. Hvis man kjører en prosess som kan ha flere tråder kan man istedet for å kjøre et program tre ganger, kjøre tre tråder samtidig inne i den samme koden. Den høyre delen av figuren viser en multi threaded prosess hvor tre instanser av samme program kjører samtidig og følger hver sin tråd under utførelsen. Det meste av koden kan deles med de andre trådene, men alle data som er spesielle for den enkelte kjøringen må lagres hver for seg, slik som PCB for tråden. For eksempel vil disse tre trådene hele

tiden ha forskjellige verdier i registerene og de vil kunne gjøre kall til forskjellige metoder. Dermed må de ha hver sin stack definert i RAM (hvor metode-kallene og metode-variabler lagres) og de må ha sine helt egne verdier i registerene som lagres i PCB når de ikke kjører. Den vanligste måten for operativsystemet å schedulere tråder er å betrakte dem som uavhengig enheter slik at tre tråder innen samme prosess kan kjøre på tre forskjellige CPUer.

10.7 Fordeler med threads

Ressursdeling Flere tråder eksisterer innenfor samme prosess. Deler på kode, data og delvis PCB.

Respons Interaktive applikasjoner kan ha en tråd med høy prioritet som kommuniserer med brukere og lavprioritettråder som gjør grovarbeid.

Effiktivitet Tar mindre tid å lage nye threads og mindre tid å context-switche mellom threads. Kan typisk ta 30x så lang tid å lage en ny prosess som å lage en ny thread. Context switch kan ta 5x så lang tid.

Multiprosessor Hver tråd kan tildeles en egen CPU.

Felles variabler Ofte nyttig med felles minne for prosesser, men det er tungvint å sette opp. Dette er trivielt for threads.

10.8 Java-threads

For å lage Java-threads må man arve klassen Thread. Viktige Thread-metoder:

start() Allokerer minne, stack etc. og kaller run().

run() Her uføres jobben tråden skal gjøre.

yield() Tråden gir fra seg CPU-en.

setPriority() Setter thread-prioritet. Min = 1, Max = 10, default = 5.

sleep(ms) Tråden sover i ms millisekunder

10.8.1 Prioritet

Vanligvis scheduleres to Java-tråder av OS, etter en-til-en modellen slik at de to trådene kjører uavhengig av hverandre og samtidig. Dette kalles native threads. Det finnes implementasjoner hvor Java kjører en prosess og schedulerer trådene selv, såkalte green-threads. jdk1.1 var implementert slik på Linux.

Det går ikke klart frem av spesifikasjonene for JVM (Java Virtual Machine) hvordan prioritet skal implementeres og her kan det være forskjeller.

10.8.2 Java på Linux

```
$ emacs Calc.java&
$ javac Calc.java   # Calc.class lages; bytecode
$ java Calc         # Starter JVM (Java Virtual Machine) som kjører byte-koden
```

10.8.3 Variabler

Variabler som blir definert som static vil være felles for alle trådene. Andre vil kun kunne brukes av den enkelte tråd.

```
   static int count;
   int id;
```

I eksempelet oppdateres `count` av begge trådene, mens det eksisterer en `id` for hver tråd.

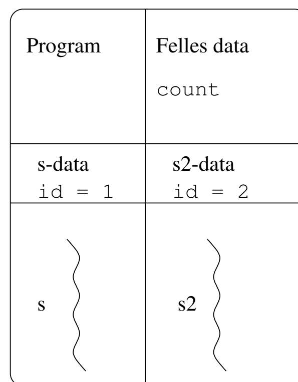


Figure 65: Deklareres en variabel som static blir den felles for alle tråder.

10.9 Java thread eksempel: Calc.java

```
import java.lang.Thread;

class CalcThread extends Thread
{
   static int count = 0;
   int id;

   CalcThread()
        {
         count++;
         id = count;
        }

   public void run()
        {
         System.out.println("Thread nr." + id + " is starting");
         System.out.println("Thread nr." + id + " calculated " + work());
        }

   private float work()
        {
         int i,j;
           float res = 0;
            System.out.println("Thread nr." + id + " calculating");
            for(j = 1;j < 5;j++)
                {
```

```
                    for(i = 1;i < 30000000;i++)
                        {
                            res += 1.0/(1.0*i*i);
                        }
                     System.out.println("Thread nr." + id + " calculating" + j);
                 }
          return(res);
         }
}

class Calc
{
   public static void main(String args[])
   {
    System.out.println("Starts two threads !\n");
    CalcThread s = new CalcThread();
    System.out.println("Thread s has id " + s.id + "\n");
    s.start(); // Allokerer minne og kaller s.run()

    CalcThread s2 = new CalcThread();
    System.out.println("Thread s2 has id " + s2.id + "\n");
    s2.start();
    System.out.println("s2 started !\n");

    }
}
```

Kjører man dette programmet på en maskin med to CPU'er, vil de to trådene kunne kjøre på hver sin CPU og dermed utnytte ressursene optimalt. Et java-program med en tråd vil kun kunne utnytte en av CPU-ene.

11 Forelesning 2/4-24(2 timer). Java threads og synkronisering

`Avsnitt fra Tanenbaum: 2.2`

Slides brukt i forelesningen[212]

11.1 Forelesningsvideoer

Uredigert opptak av hele første time av forelesningen (00:38:56)[213]

Uredigert opptak av hele andre time av forelesningen (00:45:01)[214]

Opptak av forelesningen inndelt etter temaer: os11del2.mp4[215] (01:37) Sist, plattformuavhengighet og tråder
os11del3.mp4[216] (07:45) Demo: CalcMany.java, et java-program som starter 20 tråder
os11del4.mp4[217] (09:58) Demo: Prioritet av Java-tråder på Linux
os11del5.mp4[218] (05:28) Slides: Blokkerende systemkall og tråd-modeller
os11del6.mp4[219] (04:57) Slides: Synkronisering og serialisering
os11del7.mp4[220] (01:30) Spørsmål: Hva er meningen med tråder når prioriteten ikke tas hensyn til?
os11del8.mp4[221] (02:11) Spørsmål: Hvordan kopiere filer fra egen Windows til studssh? Demo av WinSCP og scp i PowerShell
os11del9.mp4[222] (03:35) Demo: Prioritet av Java-tråder på Windows
os11del10.mp4[223] (04:28) Slides: Problem med to web-prosesser som skriver ut billetter. Må serialiseres.
os11del11.mp4[224] (07:05) Slides: Race condition med en kodelinje, to prosesser oppdaterer en felles variabel saldo; en million blir borte!
os11del12.mp4[225] (03:29) Demo: Saldo.java, to tråder oppdaterer felles variabel saldo og resultatet blir forskjellig hver gang det kjøres.
os11del13.mp4[226] (04:52) Demo: Hvordan ser Java-byte koden som kjøres ut? Kan sees med javap -private. Programmet er ikke trådsikkert
os11del14.mp4[227] (17:27) Demo: Race condition med C, to pthreads og én instruksjon

11.2 Sist

- Plattformavhengighet
- Tråder
- Java-threads

[212]https://www.cs.oslomet.no/ haugerud/os/Forelesning/12.pdf
[213]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11time1.mp4
[214]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11time2.mp4
[215]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del2.mp4
[216]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del3.mp4
[217]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del4.mp4
[218]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del5.mp4
[219]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del6.mp4
[220]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del7.mp4
[221]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del8.mp4
[222]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del9.mp4
[223]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del10.mp4
[224]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del11.mp4
[225]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del12.mp4
[226]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del13.mp4
[227]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os11del14.mp4

11.3 Mange samtidige Java-tråder

Man kan starte et gitt antall tråder med

```
.
.
.
    static double saldo[] = new double[5000000]; // Felles array. 5M*8 byte = 40MByte
.
.
.
        int threads = 20;
        CalcThread tr[] = new CalcThread[threads];
        System.out.println("Starts " +threads + " threads !\n");

        for(k = 0;k < threads;k++)
            {
                tr[k] = new CalcThread();
                System.out.println("Thread has id " + tr[k].id + "\n");
                tr[k].start();
            }
```

Dette vil lage en prosess med mange tråder. Med top på Linux så dette tidligere ut som 20 uavhengige prosesser, selvom de egentlig var tåder. På en Linux host med to CPU'er ser det idag slik ut:

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4448 haugerud  20   0  246m  59m  15m S  199  2.9   0:16.19 java
```

Legg merke til at prosessen bruker 199% CPU, det skyldes at de 20 trådene tilsammen bruker så mye CPU ved at de av OS-kjernen scheduleres på begge CPU-ene. Om man taster H får man se alle trådene:

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4458 haugerud  20   0  246m  59m  15m R 10.8  2.9   0:59.51 java
 4466 haugerud  20   0  246m  59m  15m R 10.8  2.9   0:59.38 java
 4450 haugerud  20   0  246m  59m  15m R 10.5  2.9   0:59.60 java
 4454 haugerud  20   0  246m  59m  15m R 10.5  2.9   0:59.54 java
 4465 haugerud  20   0  246m  59m  15m R 10.5  2.9   0:59.34 java
 4455 haugerud  20   0  246m  59m  15m R 10.2  2.9   0:59.53 java
 4459 haugerud  20   0  246m  59m  15m R 10.2  2.9   0:59.49 java
 4460 haugerud  20   0  246m  59m  15m R 10.2  2.9   0:59.49 java
 4451 haugerud  20   0  246m  59m  15m R  9.9  2.9   0:59.58 java
 4461 haugerud  20   0  246m  59m  15m R  9.6  2.9   0:52.16 java
 4467 haugerud  20   0  246m  59m  15m R  9.3  2.9   0:52.01 java
 4452 haugerud  20   0  246m  59m  15m R  9.0  2.9   0:52.18 java
 4456 haugerud  20   0  246m  59m  15m R  9.0  2.9   0:52.18 java
 4462 haugerud  20   0  246m  59m  15m R  9.0  2.9   0:52.14 java
 4463 haugerud  20   0  246m  59m  15m R  9.0  2.9   0:52.13 java
 4464 haugerud  20   0  246m  59m  15m R  9.0  2.9   0:52.13 java
 4468 haugerud  20   0  246m  59m  15m R  9.0  2.9   0:52.08 java
 4469 haugerud  20   0  246m  59m  15m R  9.0  2.9   0:52.01 java
 4453 haugerud  20   0  246m  59m  15m R  8.7  2.9   0:52.22 java
 4457 haugerud  20   0  246m  59m  15m R  8.7  2.9   0:52.15 java
```

Det ser ut som det er 20 prosesser som bruker 59MByte hver, men de deler i virkeligheten på et stort felles array, `saldo[]`. Legg merke til at hver tråd har sin egen PID. Det er slik OS-kjernen ser trådene, de scheduleres som selvstendige enheter og får like mye CPU hver. Hvis man i top taster f og velger

```
TPGID   = Tty Process Grp Id
nTH     = Number of Threads
```

vil man se at dette er PID for den opprinnelige prosessen som startet alle de andre trådene, tråd nummer 21 i listingen nedenfor:

```
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND          TPGID nTH
 7617 haugerud  20   0 8241296  60408  16644 R 43,9  0,4   1:30.37 java              7586  40
 7623 haugerud  20   0 8241296  60408  16644 R 41,6  0,4   1:29.58 java              7586  40
 7624 haugerud  20   0 8241296  60408  16644 R 41,6  0,4   1:31.18 java              7586  40
 7607 haugerud  20   0 8241296  60408  16644 R 41,3  0,4   1:31.09 java              7586  40
 7622 haugerud  20   0 8241296  60408  16644 R 41,3  0,4   1:30.59 java              7586  40
 7619 haugerud  20   0 8241296  60408  16644 R 40,3  0,4   1:32.93 java              7586  40
 7625 haugerud  20   0 8241296  60408  16644 R 40,3  0,4   1:32.86 java              7586  40
 7606 haugerud  20   0 8241296  60408  16644 R 39,9  0,4   1:31.49 java              7586  40
 7614 haugerud  20   0 8241296  60408  16644 R 39,9  0,4   1:30.02 java              7586  40
 7615 haugerud  20   0 8241296  60408  16644 R 39,9  0,4   1:30.50 java              7586  40
 7620 haugerud  20   0 8241296  60408  16644 R 39,9  0,4   1:30.47 java              7586  40
 7609 haugerud  20   0 8241296  60408  16644 R 38,9  0,4   1:31.44 java              7586  40
 7610 haugerud  20   0 8241296  60408  16644 R 38,9  0,4   1:30.85 java              7586  40
 7611 haugerud  20   0 8241296  60408  16644 R 38,3  0,4   1:30.66 java              7586  40
 7613 haugerud  20   0 8241296  60408  16644 R 38,0  0,4   1:30.76 java              7586  40
 7616 haugerud  20   0 8241296  60408  16644 R 38,0  0,4   1:30.37 java              7586  40
 7608 haugerud  20   0 8241296  60408  16644 R 37,3  0,4   1:30.91 java              7586  40
 7618 haugerud  20   0 8241296  60408  16644 R 37,3  0,4   1:29.09 java              7586  40
 7621 haugerud  20   0 8241296  60408  16644 R 37,3  0,4   1:29.92 java              7586  40
 7612 haugerud  20   0 8241296  60408  16644 R 37,0  0,4   1:30.44 java              7586  40
 7586 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7587 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.06 java              7586  40
 7588 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7589 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7590 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7591 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7592 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7593 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7594 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7595 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7596 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:16.10 java              7586  40
 7597 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7598 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7599 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7600 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.02 java              7586  40
 7601 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.02 java              7586  40
 7602 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7603 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.02 java              7586  40
 7604 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.00 java              7586  40
 7605 haugerud  20   0 8241296  60408  16644 S  0,0  0,4   0:00.05 java              7586  40
```

Og hvis man taster H igjen, er det bare denne prosessen man ser som bruker nesten 800% CPU, siden serveren har 8 CPU'er i dette tilfellet.

```
 PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND      TPGID nTH
 7586 haugerud  20   0 8241296  60548  16644 S 790,4  0,4  86:37.05 java         7586  40
```

11.4 Java threads-eksempel: Prioritet

En Java-tråd kan tilordnes prioritet med `setPriority()`. Eksempelet under viser også at en tråd selv kan endre sin prioritet. Når en tråd med høyere prioritet starter er meningen at den skal ta over CPU fra tråder med lavere prioritet, eventuelt gis mer CPU-tid. Men dette er ikke et absolutt krav i spesifikasjonen og praksis viser at dette ikke er tilfelle for alle JVM'er.

Om man hadde kjørt programmet i eksempelet med tråder implementert i user-space, ville OS ikke vært involvert i scheduleringen. Med Java green-threads (jdk1.1) startes først tråd 1 med prioritet 5 og tråd 2 med prioritet 10. Tråd 2 "sover" i 3 sekunder, så i starten kjører tråd 1 alene. Når tråd 2 våkner, tar den fullstendig over CPU-en og utfører `work()` helt til den setter sin egen prioritet til 1. Da tar tråd 1 over igjen til den er helt ferdig og til slutt fullføres tråd 2.

Kjøres det samme med native-threads, som er implementert i alle nyere JVM'er, vil trådene scheduleres av OS-kjernen og dermed timeslices og kjøre samtidig. På Linux vil prioriteten i praksis ikke ha noen innflytelse, trådene kjøres samtidig og deler likt på CPU, uansett hva prioriteten settes til. Dette skjer også om trådene tvinges til å kjøre på samme CPU med taskset. Det ville vært mulig å bruke `nice` til å implementere prioritet i en Linux JVM, men man har valgt å ikke gjøre det som default siden dette ikker er et absolutt krav i spesifikasjonen for JVM-implementasjonen. Se ukeoppgavene om hvordan man kan få prioritet til å virke under Linux.

Under Windows vil trådene både timeslice og respektere prioriteten, men i så stor grad at threads med lavere prioritet nesten ikke slipper til. Og denne prioriteringen skjer kun hvis trådene deler samme CPU. Om det er flere CPUer tilgjengelig, kjører trådene på hver sin CPU og får like mye CPU-tid hver. Moralen er: Java Threads er ikke plattformuavhengig og avhengig av hvordan JVM i samarbeid med det underliggende OS schedulerer trådene.

```
import java.lang.Thread;

class PriorThread extends Thread
{
   static int count = 0;
   int id,mil;
   int max = 400000000;

   PriorThread(int millisek)
{
 count++;
 id = count;
 mil = millisek;
}

   public void run()
{
 try {sleep (mil);} catch (Exception e) {}
 System.out.println("Thread nr." + id + " med prioritet " + getPriority() + " starter");
 System.out.println("Thread nr." + id + " regnet ut " + work()+ "\n");
 if(id == 2)
    {
    setPriority(1);
    System.out.println("\nEndrer prioritet for Thread nr." + id + ". Prioritet er nå "+getPriority()+"\n");
    }
 System.out.println("Thread nr." + id + " regnet ut " + work()+ "\n");
}

   private float work()
{
```

```
 int i,j;
 float res = 0;
 for(j=1;j<=8;j++)
     {
 for(i = 1;i < max;i++)
     {
 res += 1.0/(1.0*i*i);
     }
 System.out.println("Thread nr." + id + " avsluttet work(" + j + ")");
     }
 return(res);
}
}

class Prior
{
   public static void main(String args[])
   {
    System.out.println("\nStarter to threads!\n");
    PriorThread s1 = new PriorThread(1);
    s1.start();
    s1.setPriority(5);
    System.out.println("Default prioritet er " + s1.NORM_PRIORITY + " for en thread");
    System.out.println("Max er " + s1.MAX_PRIORITY + " og min er " + s1.MIN_PRIORITY + "\n");

    PriorThread s2 = new PriorThread(0);
    s2.setPriority(10);
    s2.start();
   }
}
```

11.5 Prior.java kjørt på Linux

Man ser av kjøringen under at selvom man tvinger trådene til å dele samme CPU, har ikke prioriteten noen effekt:

```
rex:~/threads$ taskset -c 0 java Prior

Starter to threads!

Default prioritet er 5 for en thread
Max er 10 og min er 1

Thread nr.2 med prioritet 10 starter
Thread nr.1 med prioritet 5 starter
Thread nr.1 avsluttet work(1)
Thread nr.2 avsluttet work(1)
Thread nr.1 avsluttet work(2)
Thread nr.2 avsluttet work(2)
Thread nr.1 avsluttet work(3)
Thread nr.2 avsluttet work(3)
Thread nr.1 avsluttet work(4)
Thread nr.2 avsluttet work(4)
Thread nr.1 avsluttet work(5)
Thread nr.2 avsluttet work(5)
Thread nr.1 avsluttet work(6)
Thread nr.2 avsluttet work(6)
```

```
Thread nr.1 avsluttet work(7)
Thread nr.2 avsluttet work(7)
Thread nr.2 avsluttet work(8)
Thread nr.1 avsluttet work(8)
Thread nr.1 regnet ut 13.15576

Thread nr.2 regnet ut 13.15576

Endrer prioritet for Thread nr.2. Prioritet er nå 1

Thread nr.1 avsluttet work(1)
Thread nr.2 avsluttet work(1)
Thread nr.1 avsluttet work(2)
Thread nr.2 avsluttet work(2)
Thread nr.2 avsluttet work(3)
Thread nr.1 avsluttet work(3)
Thread nr.2 avsluttet work(4)
Thread nr.1 avsluttet work(4)
Thread nr.2 avsluttet work(5)
Thread nr.1 avsluttet work(5)
Thread nr.2 avsluttet work(6)
Thread nr.1 avsluttet work(6)
Thread nr.2 avsluttet work(7)
Thread nr.1 avsluttet work(7)
Thread nr.2 avsluttet work(8)
Thread nr.2 regnet ut 13.15576

Thread nr.1 avsluttet work(8)
Thread nr.1 regnet ut 13.15576
```

11.6 Prior.java kjørt på Windows 10

Kjører man PriorThread-eksempelet under Windows, på en maskin med kun en CPU, vil prioriteten tas hensyn til, men tråd med høyere prioritet tar da over nesten all CPU-tiden og slipper ikke den andre tråden til:

```
PS C:\Users\os\threads> java Prior

Starter to threads!

Default prioritet er 5 for en thread
Max er 10 og min er 1

Thread nr.2 med prioritet 10 starter
Thread nr.2 avsluttet work(1)
Thread nr.2 avsluttet work(2)
Thread nr.1 med prioritet 5 starter
Thread nr.2 avsluttet work(3)
Thread nr.2 avsluttet work(4)
Thread nr.2 avsluttet work(5)
Thread nr.2 avsluttet work(6)
Thread nr.2 avsluttet work(7)
Thread nr.2 avsluttet work(8)
Thread nr.2 regnet ut 13.15576
```

```
Thread nr.1 avsluttet work(1)
Thread nr.1 avsluttet work(2)

Endrer prioritet for Thread nr.2. Prioritet er nå 1

Thread nr.1 avsluttet work(3)
Thread nr.1 avsluttet work(4)
Thread nr.1 avsluttet work(5)
Thread nr.1 avsluttet work(6)
Thread nr.1 avsluttet work(7)
Thread nr.1 avsluttet work(8)
Thread nr.1 regnet ut 13.15576

Thread nr.1 avsluttet work(1)
Thread nr.1 avsluttet work(2)
Thread nr.1 avsluttet work(3)
Thread nr.1 avsluttet work(4)
Thread nr.1 avsluttet work(5)
Thread nr.1 avsluttet work(6)
Thread nr.1 avsluttet work(7)
Thread nr.1 avsluttet work(8)
Thread nr.1 regnet ut 13.15576

Thread nr.2 avsluttet work(1)
Thread nr.2 avsluttet work(2)
Thread nr.2 avsluttet work(3)
Thread nr.2 avsluttet work(4)
Thread nr.2 avsluttet work(5)
Thread nr.2 avsluttet work(6)
Thread nr.2 avsluttet work(7)
Thread nr.2 avsluttet work(8)
Thread nr.2 regnet ut 13.15576
```

Man ser at når thread nr. 2 endrer prioritet, kjører thread nr. 1 to runder før thread nr 2 får tid til å skrive ut sin kommentar om at prioriteten er senket.

Kjører man eksemplet med mange tråder, kan man på Windows taskmanager se hvor mange tråder JVM bruker. For å få til det må man høyreklikke på en av kolonne-navnene og be om at thread-kolonnen vises. I utganspunktet bruker Windows JVM 10 tråder. Hvis man starter 20 egne tråder viser dermed Taskmanager 30 tråder.

11.7 Blokkerende systemkall

Den viktigste årsaken til at man i det hele tatt begynte med threads var såkalte blokkerende I/O requests. Blokkerende betyr at applikasjonen som ber om I/O blir satt på vent av operativsystemet til resultatet fra I/O operasjonen returnerer. Blokkerende I/O forespørsler kan for eksempel være å lese fra en fil eller fra tastaturet. Programmet kan da ikke kjøre videre før det får resultatet. Generelt leder forespørsler om I/O til systemkall og disse er da blokkerende eller ikke-blokkerende systemkall. Eksempler på blokkerende systemkall:

- read/write
- wait
- sleep

De to første er alltid blokkerende, mens read og write kan bli gjort til nonblocking ved å bruke buffere og ordninger som sender signaler når lesingen/skrivingen er ferdig. Ikke blokkerende systemkall:

- getpid
- gettimeofday
- setuid

Dette er systemkall som ikke trenger å vente på I/O, andre prosesser eller noe annent for å fullføre. Uansett vil et systemkall føre til en trap til OS-kjernen.

11.8 Thread-modeller

Det finnes tre hovedmetoder for hvordan et operativsystem skedulerer threads. De tre metodene er forskjellige med hensyn på hvor uavhengig av hverandre trådene skeduleres.

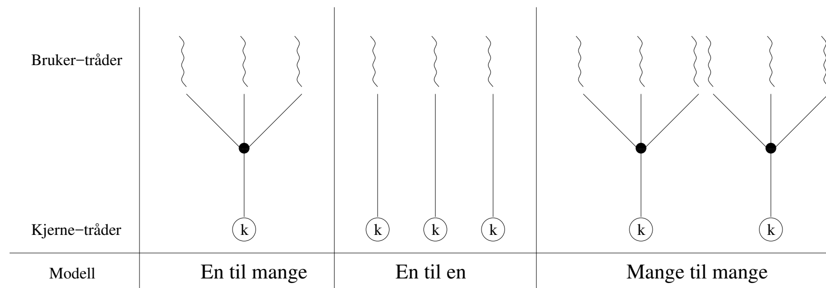


Figure 66: Thread-modeller i OS-kjernen

en til mange Alle trådene skeduleres som en prosess, en enhet. Java: green-threads, JVM sørger selv for skedulering; ingen multitasking. Default på gamle versjoner av Linux(Debian) og Solaris.

en til en Den mest brukte modellen. Hver tråd skeduleres uavhengig av de andre. Windows Java-threads, Linux native Java-threads, Linux Posix-threads (pthreads)

mange til mange Tråder skeduleres uavhengig om de ikke er for mange. Kjernen kan begrense antall tråder i RR-køen. Solaris, Digital Unix, IRIX pthreads

Et operativsystem som bruker en-til-mange metoden vil behandle en prosess som inneholder mange tråder akkurat som en prosess med bare en tråd. Den gir prosessen biter av CPU-tid som alle andre prosesser og bryr seg ikke om at prosessen egentlig består av flere tråder. Det gjør at tråd-programmet selv må sørge for skedulering. En måte å gjøre dette på er at trådene kaller på `yield()` for å signalisere til de andre trådene at den er ferdig med sin del av jobben. Alternativt kan tråd-programmet selv lag en round robin schedulering som fordeler tid mellom trådene. For Java green-threads, som var default metode i de første Java-versjonene, var det JVM som selv sørget for skedulering. Hvis man i en slik versjon av Java setter igang to tråder, vil de ikke jobbe annen hver gang som man ville forvente, men første tråd kjøre til den er ferdig og så vil trå nummer to ta over.

De aller fleste av dagens operativsystemer bruker en-til-en metoden hvor det er en kjerne-tråd for hver bruker-tråd. Det betyr ikke at det for hver bruker-prosess er en egen prosess eller tråd i kjernen, men at kjernen for hver tråd har lagret data som registerverdier, program counter, tilstand, tråd-ID, prioritet

og så videre og skedulerer hver tråd uavhengig av de andre trådene. For prosesser med bare en tråd, har kjernen lagret disse dataene i en tabell som inneholder informasjon om alle prosessene som kjører. For prosesser som har flere tråder, har kjernen for hver av disse en tabell som inneholder data for de individuelle trådene. Når operativsystemets scheduler skal velge hvilken tråd den skal kjøre på en CPU, velger den mellom alle tilgjengelige tråder, uavhengig av hvilken prosess de tilhører og uavhengig av hvor mange tråder hver prosess har. OS-kjernen fordeler CPU-tid jevnt mellom alle trådene og ikke mellom prosessene. En prosess som har veldig mange tråder vil derfor få mer CPU-tid enn en prosess med få tråder.

11.9 Synkronisering

Samtidige prosesser som deler felles ressurser/data må **synkroniseres**.

- prosesser må ikke endre felles data samtidig
- en prosess bør ikke lese felles data mens en annen endrer dem
- en prosess må kunne vente på (f. eks. resultater fra) en annen prosess

Distribuerte systemer mer og mer vanlig. Synkronisering er da essensielt.

11.10 Serialisering

Prosesser/tråder som aksesserer felles data må serialiseres; jobbe en av gangen på felles data. Problemstillingen kalles **Race Condition** (konkurranse om felles ressurser). *Brukeren må selv serialisere sine prosesser. OS legger mulighetene til rette.*

11.10.1 Eksempel: To web-prosesser som skriver ut billetter

Anta at man kan kjøpe billetter på en web-side. På web-serveren starter det da opp en prosess for hver bruker som bestiller en billett. Disse prosessene er helt uavhengige, bortsett fra at de har en felles variabel `LedigeBilletter` som er antall ledige billetter. Koden prosessene kjører kan se omtrent slik ut:

```
if(LedigeBilletter > 0){
   LedigeBilletter--;
   SkrivUtBillett();
}
```

Dette fungerer greit når bare en slik prosess kjøres av gangen, men det kan oppstå problemer hvis de kjører samtidig(husk at `LedigeBilletter` er en felles variabel):

P1-kode	P2-kode	LedigeBilletter
if(LedigeBilletter > 0){		1
—Context Switch⟹	if(LedigeBilletter > 0){	1
	LedigeBilletter—;	0
	SkrivUtBillett();	0
	}	0
LedigeBilletter—;	⟸Context Switch—	-1
SkrivUtBillett();		-1
}		-1

En Context Switch kan forekomme når som helst og hvis den skjer rett etter at P1 har sjekket at `(LedigeBilletter > 0)`, men før den har senket verdien med en, vil P1 og P2 skrive ut den samme (og siste) billetten til to forskjellige kunder. Dette er opplagt galt. Prosessene må serialiseres, slik at denne kodebiten (som kalles et kritisk avsnitt) gjøres av en prosess av gangen.

11.10.2 Eksempel: to prosesser som oppdaterer en felles variabel

Det er viktig å huske at en linje høynivåkode ofte oversettes til mange linjer makinkode. Dermed kan en Race Condition oppstå selv inni en kodelinje, fordi en Context Switch kan oppstå mellom to hvilke som helst maskininstruksjoner. CPU-en ser kun maskininstruksjoner og aner ikke noe om høynivåkoden som ligger bak. Anta at to prosesser P1 og P2 kjører følgende høynivåkode som oppdaterer en konto:

P1-kode	P2-kode
static int saldo;	static int saldo;
.	.
.	.
.	.
saldo = saldo - mill;	saldo = saldo + mill;

Variabelen `saldo` *er da en felles variabel begge kan endre.*

Problem Hva skjer om OS switcher fra P1 til P2 mens P1 ufører `saldo = saldo - mill` ?

Hvorfor? Prosessen utfører maskinkode, linje for linje, og kan bli avbrutt etter en instruksjon.

P1	P2
saldo = saldo - mill;	saldo = saldo + mill;
mov saldo,%ax	mov saldo,%ax
mov mill,%bx	mov mill,%bx
sub %bx,%ax	add %bx,%ax
mov %ax,saldo	mov %ax,saldo

Anta at P1 blir Context switchet etter å ha utført `mov mill,\%bx` og P2 overtar. Og videre at saldo er 5 til å begynne med og mill er 1. Følgende skjer, sett fra prosessoren:

Prosess som kjører	Instruksjon (IR)	%ax	%bx	saldo
P1	mov saldo,%ax	5	0	5
P1	mov mill,%bx	5	1	5
OS	Context switch	0	0	5
P2	mov saldo, %ax	5	0	5
P2	mov mill,%bx	5	1	5
P2	add %bx,%ax	6	1	5
P2	mov %ax,saldo	6	1	6
OS	Context switch	5	1	6
P1	sub %bx,%ax	4	1	6
P1	mov %ax,saldo	4	1	4

Når P2 er ferdig vil P1 bruke den gamle saldoverdien 5 og sluttresultatet blir saldo = 4. Det burde ha blitt saldo = 5 og en mill er borte!! Konklusjon: må serialisere aksess til felles data!

11.11 Kritisk avsnitt

To prosesser P1 og P2 kjører:

P1-kode	P2-kode
static int saldo;	static int saldo;
.	.
.	.
.	.
saldo = saldo - mill;	saldo = saldo + mill;

Utregningen av saldo er et **kritisk avsnitt** i koden til P1 og P2. Et kritisk avsnitt **må fullføres** av prosessen som utfører det uten at andre prosesser slipper til; prosessene må serialiseres.

11.12 Kritisk avsnitt: Java-eksempel

Anta at to tråder samarbeider om et felles int-variabel saldo. Den ene tråden øker en million ganger saldo med 1, mens tråd nummer to minker verdien av saldo med en million ganger. Koden kan se slik ut:

```
// Kompileres med  javac NosynchThread.java
// Run: java NosynchThread

import java.lang.Thread;

class SaldoThread extends Thread
{
    static int MAX = 1000000; // En million
    static int count = 0;
    public static int saldo; // Felles variable, gir race condition
    int id;

    SaldoThread()
    {
        count++;
        id = count;
    }

    public void run()
    {
System.out.println("Tråd nr. "+ id +", med prioritet " + getPriority() + " starter");
updateSaldo();
    }

    private void updateSaldo()
    {
int i;
if(id == 1)
{
    for(i = 1;i < MAX;i++)
    {
saldo++;
    }
}
else
```

```
{
    for(i = 1;i < MAX;i++)
    {
        saldo--;
    }
}
System.out.println("Tråd nr. " + id + " ferdig. Saldo: " + saldo);
    }
}

class NosynchThread extends Thread
{
   public static void main (String args[])
   {
       int i;
       System.out.println("Starter to tråder!");

       SaldoThread s1 = new SaldoThread();
       SaldoThread s2 = new SaldoThread();
       s1.start();
       s2.start();

       try{s1.join();} catch (InterruptedException e){}
       try{s2.join();} catch (InterruptedException e){}

       System.out.println("Endelig total saldo: " +SaldoThread.saldo);
   }
}
```

Her stod det inntil 14 april 2024 i koden `public synchronized void updateSaldo()` *men det var trolig fra en test; det hjelper ikke å bruke synchronized slik, se mer om dette i neste forelesning. I videoen er koden korrekt presentert, uten synchronized.*

Man skulle tro at saldo dermed ender opp som 0, men en kjøring kan gi noe slikt som:

```
rex:~/threads/nosync$ java NosynchThread
Starter to tråder!
Tråd nr. 2, med prioritet 5 starter
Tråd nr. 1, med prioritet 5 starter
Tråd nr. 2 ferdig. Saldo: -7831
Tråd nr. 1 ferdig. Saldo: -4892
Endelig total saldo: -4892
```

eller

```
rex:~/threads/nosync$ java NosynchThread
Starter to tråder!
Tråd nr. 1, med prioritet 5 starter
Tråd nr. 2, med prioritet 5 starter
Tråd nr. 1 ferdig. Saldo: 8055
Tråd nr. 2 ferdig. Saldo: 3727
Endelig total saldo: 3727
```

Altså flere tusen feil og varierende resultat fra gang til gang. Dette skyldes at trådenes lesning og lagring av den felles variabelen ikke er synkronisert.

11.12.1 Årsaken: race conditions

Ser man på bytekoden som kjøres vil den delen av updateSaldo() som legger til 1 se slik ut

```
\normalsize
$ javap -private -c SaldoThread
 -private for å vise alle metoder, ellers vises ikke updateSaldo()
.
.
      17: getstatic     #17                 // Field saldo:I
      20: iconst_1
      21: iadd
      22: putstatic     #17                 // Field saldo:I
.
.
```

og tilsvarende del av koden for subtraksjon ser slik ut

```
      40: getstatic     #17                 // Field saldo:I
      43: iconst_1
      44: isub
      45: putstatic     #17                 // Field saldo:I
```

JVM er en stack-maskin og getstatic laster verdien fra saldo på stacken før iadd øker verdien med en. Til slutt lagres verdien på stacken med putstatic. Årsaken til regnefeilene er at trådene når som helst kan context switches og om det skjer mellom getstatic og putstatic, vil regneoperasjonen bli usynkroniserte og trådene overser deler av hverandres regneoperasjoner.

11.13 Race condition med C, to pthreads og én instruksjon

Vi så i eksempelet med Java-tråder at instruksjonen

```
saldo++;
```

faktisk ikke utføres av en enkelt Java bytecode-instruksjon, men av flere. Dermed vil det selvom trådene kjører på samme CPU kunne skje en context switch rett etter at verdien på saldo er lastet inn og før verdien er lagret igjen. Når den andre tråen så går inn og leser verdien på saldo vil den bruke den ikke oppdaterte verdien og en race condition oppstår. Sluttresultatet vil avhenge av hvilken tråd som leser verdien først og vil dermed være forskjellig for hver gang programmet kjøres. Men hva om det faktisk bare var en enkelt instruksjon som blir utført i det kritiske avsnittet? Kan det også da oppstå en race condition?

For å undersøke det lager vi et ligenende program hvor vi bruker en enkelt assembly-instruksjon for å forsikre oss om at det kun er en enkelt instruksjon som utføres og at kompilatoren ikke lager maskinkode som involverer flere instruksjoner. Tråder er ikke inkludert som default i C, men man kan introdusere tråder ved hjelp av pthreads-biblioteket. Koden nedenfor viser et C-program som lager to tråder som begge oppdaterer en felles variabel med navn `svar`. I dette tilfellet øker begge tråder verdien til variabelen like mange ganger og dermed vet vi at verdien må bli det dobbelte av det hver tråd øker med hvis det ikke inntreffer en race condition.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int svar = 0;

extern void enlinje();

void *inc()
{
   printf("Starter; svar verdi: %d\n", svar);

   for (int i = 0; i < 10000000; i++)
     {
enlinje();
     }

   printf("Avslutter; svar verdi: %d\n", svar);
}


int main()
{
           pthread_t thread1, thread2;

          /* Lager uavhengige threads som utfører inc-funksjonen */

           pthread_create( &thread1, NULL, inc,NULL);
           pthread_create( &thread2, NULL, inc, NULL);

           /* Venter med join til begge tråder er ferdige */

           pthread_join( thread1, NULL);
           pthread_join( thread2, NULL);

           printf("Main avslutter; svar verdi: %d\n", svar);
           exit(0);
}
```

I første omgang skriver vi enlinje-funksjonen med vanlig C-kode i en fil med navn `en.c`, alt den gjør er å øke verdien av den felles variabelen `svar` med en:

```
void enlinje()
{
   extern int svar;
   svar++;
}
```

Når vi så kompilerer og kjører programmet på følgende måte:

```
rex:~/threads/lock$ gcc -pthread thread.c en.c
rex:~/threads/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 629979
Avslutter; svar verdi: 10026464
Avslutter; svar verdi: 12261597
```

```
Main avslutter; svar verdi: 12261597

rex:~/threads/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 204229
Avslutter; svar verdi: 7132793
Avslutter; svar verdi: 10668956
Main avslutter; svar verdi: 10668956

rex:~/threads/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 114562
Avslutter; svar verdi: 9936660
Avslutter; svar verdi: 10127784
Main avslutter; svar verdi: 10127784
```

ser vi at en race condition oppstår fordi svaret blir forskjellig for hver gang og avhenger av rekkefølgen de to trådene oppdaterer variabelen. At svaret er litt over 10 millioner er forenlig med det som skjer hvis begge henter ut verdien 1 omtrent samtidig og øker den til 2 og så skriver tilbake omtrent samtidig, vil den totale økningen være 1, mens den burde vært 2. Når trådene fortsetter slik uten å samarbeide, vil ca halvparten av økningene med 1 forsvinne. Men kan dette skyldes at kompilatoren lager maskinkode som involverer flere instruksjoner? Ja, det kan være årsaken (å undersøke om det virkelig er tilfelle at kompilatoren lager flere linjer, overlates til en av ukens oppgaver). For å være helt sikker på at enlinje-funksjonen kun utfører en enkelt insturksjon, erstatter vi `en.c` med assembly-filen `minimal.s`:

```
.globl enlinje
enlinje:
incl svar(%rip)
ret
```

Når vi nå kompilerer og kjører er vi sikker på at det kun er en enkelt instruksjon som oppdaterer variabelen `svar`, den blir ikke lagret i et register først. Likevel, når vi kompilerer og kjører oppstår det fortsatt en race condition:

```
rex:~/threads/lock$ gcc -pthread thread.c minimal.s
rex:~/threads/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 748235
Avslutter; svar verdi: 7065807
Avslutter; svar verdi: 10768013
Main avslutter; svar verdi: 10768013
```

Dette skyldes at de to trådene kjører på hver sin CPU og at det ikke er noen koordinering CPUene imellom om å vente på hverandre når en variabel skal hentes fra RAM. Men hva om man tvinger begge trådene til å kjøre på samme kjerne eller CPU? Da bør vel en race condition avverges? Det kan testes ved å starte prosessen med taskset, den vil tvinge prosessen og alle dens tråder til å kjøre på samme CPU. Og ganske riktig, da løses problemet:

```
rex:~/threads/lock$ taskset -c 0 ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 3258051
Avslutter; svar verdi: 17614192
Avslutter; svar verdi: 20000000
```

```
Main avslutter; svar verdi: 20000000

rex:~/threads/lock$ taskset -c 0 ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 3348312
Avslutter; svar verdi: 17502515
Avslutter; svar verdi: 20000000
Main avslutter; svar verdi: 20000000
```

Uansett hvor mange ganger man kjører dette, vil det gi riktig resultat. Dette er fordi en enkelt maskininstruksjon fullføres før en context switch skjer, dermed vil et kritisk avsnitt alltid fullføres før neste tråd tar over.

En mer genrell løsning består i å legge til maskininstruksjonen `lock` rett før oppdateringen av `svar`:

```
.globl enlinje
enlinje:
        lock
incl svar(%rip)
ret
```

Denne instruksjonen låser minnebussen slik at ingen andre prosesser får bruke den før den selv har utført neste instruksjon. Dermed løses problemet:

```
rex:~/threads/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 116546
Avslutter; svar verdi: 19886202
Avslutter; svar verdi: 20000000
Main avslutter; svar verdi: 20000000
```

Dette er en metode å løse race condition problemer på, i neste forelesning ser vi på denne og andre metoder. Forøvrig bruker programmet tre ganger så lang tid på å kjøre når man bruker `lock`. Dette er fordi det tar mye ekstra tid å synkronisere prosessene, de må vente på hverandre når minnebussen låses.

12 Forelesning 9/4-24(2 timer). Mutex, Semaforer, Deadlock

`Avsnitt fra Tanenbaum: 2.2 og 2.3`

Slides brukt i forelesningen[228]

12.1 Forelesningsvideoer

Uredigert opptak av hele første time av forelesningen (00:46:03)[229]

Uredigert opptak av hele andre time av forelesningen (00:46:37)[230]

[228] https://www.cs.oslomet.no/ haugerud/os/Forelesning/13.pdf
[229] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12time1.mp4
[230] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12time2.mp4

Opptak av forelesningen inndelt etter temaer:

os12del1.mp4[231] (02:08) Intro om planene fremover og om PowerShell
os12del2.mp4[232] (03:20) Om ukens oppgaver og dagens temaer
os12del3.mp4[233] (01:59) Spørsmål: Er uke 14 obligatorisk? Nei, oppgver merket (oblig) fra nå er de viktigste å få med seg.
os12del4.mp4[234] (01:20) Innledning om dagens temaer
os12del5.mp4[235] (08:07) Slides: Kritisk avsnitt, disableInterupts, Mutex
os12del6.mp4[236] (03:22) Slides: Linux og Windows eksempel
os12del7.mp4[237] (10:32) Slides: Softwareløsning av MUTEX, GetMutex(lock), forsøk 1
os12del8.mp4[238] (03:24) Slides: Hardware-støttet mutex og X86-instruksjonen lock
os12del9.mp4[239] (00:23) Spørsmål: Virker testAndSet også for flere CPU-er? -Ja
os12del10.mp4[240] (06:45) Demo: lock og pthreads, oppsummering av demo fra forrige uke, Men hva med flere linjer og samme CPU?
os12del11.mp4[241] (09:36) Demo: lock og pthreads (hvorfor lengre tid på en CPU?) Samme CPU og tre linjer for å øke svar
os12del12.mp4[242] (06:37) Tegning og forklaring av 2 CPU-er og låsing av minnebus med lock
os12del13.mp4[243] (05:50) Tegning og forklaring av en CPU og taskset slik at begge tråder kjører på samme CPU
os12del14.mp4[244] (01:14) Demo: Oppsummering av lock og pthreads med en og tre instruksjoner
os12del15.mp4[245] (05:39) Slides: semaforer og mutex, implementasjon av semafor i OS
NB! Slides: Moitorer og Java synkronisering er ikke klippet ut og er vist fra 38:00 i andre time:

Uredigert opptak av hele andre time av forelesningen (00:46:37)[246]

os13del3.mp4[247] (07:16) Slides: Deadlock
os13del4.mp4[248] (08:34) Slides: Dining Philosophers Problem og mulige løsninger på deadlock

12.2 Sist

- Synkronisering
- Serialisering
- Race condition Java
- Race condition C og pthreads
- Kritisk avsnitt

[231] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del1.mp4
[232] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del2.mp4
[233] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del3.mp4
[234] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del4.mp4
[235] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del5.mp4
[236] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del6.mp4
[237] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del7.mp4
[238] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del8.mp4
[239] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del9.mp4
[240] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del10.mp4
[241] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del11.mp4
[242] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del12.mp4
[243] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del13.mp4
[244] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del14.mp4
[245] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12del15.mp4
[246] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os12time2.mp4
[247] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del3.mp4
[248] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del4.mp4

12.3 Mulige måter å takle kritiske avsnitt

A Skru av scheduler før kritisk avsnitt. P1 kode:

```
disableInterupts();
saldo = saldo - mill;
enableInterupts();
```

OK for en OS-kjerne, men for farlig for brukerprosesser; de kan ta over styringen.

B Bruke en form for lås som gjør at bare en prosess av gangen har tilgang til felles data.

- MUTual EXclusion = MUTEX = gjensidig utelukkelse
- mest brukt
- mange implementasjoner

12.3.1 Linux-eksempel

File-lock for Linux-mail: hvis filen

```
/var/mail/haugerud.lock
```

eksisterer, kan inbox ikke leses/skrives til. *Sendmail og andre mailprogram lager denne filen før de skriver/leser mail og fjerner den når de er ferdige.*

12.3.2 Windows-eksempel

Win 32 API'et har to funksjonskall

- EnterCriticalSection
- LeaveCriticalSection

som applikasjoner kan kalle før og etter et kritisk avsnitt.

12.4 Softwareløsning for P1/P2 med MUTEX

Trenger to funksjoner `GetMutex(lock)` og `ReleaseMutex(lock)` som gjør at en prosess av gangen kan sette en lock. Da kan problemet løses med:

```
GetMutex(lock);      // henter nøkkel
KritiskAvsnitt();    // saldo -= mill;
ReleaseMutex(lock);  // gir fra seg nøkkel
```

12.4.1 Software-mutex, forsøk 1

```
static boolean lock = false; // felles variabel

GetMutex(lock)
   {
   while(lock){} // venter til lock blir false
   lock = true;
   }
ReleaseMutex(lock)
   {
   lock = false;
   }
```

Dette burde sikre at to prosesser ikke er i kritisk avsnitt samtidig? Men hva om det skjer en Context Switch rett etter `while(lock){}` når P1 kjører?

Da rekker ikke P1 å sette lock til true og P2 kunne gå inn i kritisk avsnitt, switches ut og P1 kan gå inn i kritisk avsnitt samtidig! Altså er ikke denne metoden korrekt. Dette er vanskelig å løse med en algoritme, som forsøkene i ukens oppgaver viser. Men Peterson-algoritmen gir en elegant løsning.

12.5 Hardware-støttet mutex

I praksis brukes som oftest hardwarestøttede løsninger, for alle softwareløsninger innebærer mange instruksjoner i tillegg til busy-waiting, som koster CPU-tid. Et unntak er synkronisering av fler-CPU maskiner, SMP, symmetric multiprocessing og flerkjerneprosessorer. . Kan lages med en egen instruksjon **testAndSet** også kalt TSL(Test and Set Lock). Tester og setter en verdi i samme maskininstruksjon. Låser minne-bussen slik at ikke andre CPUer kan endre eller lese verdien. GetMutex() kan da implementeres med:

```
GetMutex(lock)
   {
    while(testAndSet(lock)) {}
   }
```

og en context switch kan ikke ødelegge siden testen og endringen av lock skjer i samme instruksjon.

12.6 X86-instruksjonen lock

En annen X86 maskininstruksjon er `lock` som vi så på i forrige uke og som hindret race condition når kritisk avsnitt kun består av en enkelt instruksjon . Den vil for neste instruksjon som utføres låse minnebussen slik at instruksjoner på andre CPUer ikke samtidig kan hente eller lagre noe i RAM. Dette sikrer at instruksjonen etter lock som utføres på en variabel i minne får avsluttet hele sin operasjon uten at RAM endres. Dermed avverges problemet ved at det kritiske avsnittet fullføres før noen andre tråder slipper til.

12.7 Semaforer

En semafor er en integer S som signaliserer om en ressurs er tilgjengelig. To operasjoner kan gjøres på en semafor:

```
Signal(S): S = S + 1;                # Kalles ofte Up(), V()
Wait(S): while(S <= 0) {}; S = S - 1; # Kalles ofte Down(), P()
```

Signal og wait må være uninterruptible og implementeres med hardwarestøtte eller i kjernen for å være atomiske(umulige å avbryte).

Binær semafor S = 0 eller 1 (som lock) (initialiseres til 1)

Teller semafor S vilkårlig heltall (initialiseres til antall ressurser)

En semafor kan brukes slik til å takle et kritisk avsnitt(må da initialiseres til 1):

```
Wait(S);
KritiskAvsnitt();
Signal(S);
```

En semafor som er initialisert til S=1 og som ikke kan bli større enn 1, omtales ofte som en mutex.

12.7.1 Implementasjon av semafor i OS

Hvis en semafor implementeres i OS kan prosesser som venter legges i en egen kø, slik at de ikke bruker CPU mens de venter. Skjematisk sett kan implementasjonen gjøres slik:

```
Signal(S){
   S = S + 1;
   if(S <= 0){
      wakeup(prosess);
      # Sett igang neste prosess fra venteliste
   }
}


Wait(S){
   S = S - 1;
   if(S < 0){
      block(prosess);
      # Legg prosess i venteliste
   }
}
```

En mp4 demo kan sees på denne web-siden[249] som viser hvordan en implementasjon semaforer kan brukes av operativsystemet.

En flash-demo det samme kan sees på denne web-siden[250].

12.8 Bruk av semafor i kritisk avsnitt

Anta at semaforen S brukes til å beskytte en felles ressurs (variabel eller lignende) i et kritisk avsnitt. Prosess A og B må da kall Wait() før og Signal() etter kritisk avsnitt.

[249] https://www.cs.oslomet.no/ haugerud/os/demoer/mutex.mp4
[250] https://www.cs.oslomet.no/ haugerud/os/demoer/swf/sema.swf

PA	PB-kode
A1	B1
A2	Wait(S)
A3	K1
Wait(S)	K2
K1	K3
K2	Signal(S)
K3	B2
Signal(S)	B3
A4	B4

Slik beskyttes da det kritiske avsnittet (pilene angir Context Switch)

A	B	S
	B1	1
	Wait(S)	0
	K1	0
A1	$\Longleftarrow$	0
A2		0
A3		0
Wait(S)	OS legger A i kø	-1
$\Longrightarrow$	K2	-1
	K3	-1
	Signal(S)	0 (A ut av kø)
	B2	0
	B3	0
	B4	0
K1	$\Longleftarrow$	0
K2		0
K3		0
Signal(S)		1
A4		1

12.9 Bruk av semafor til å synkronisere to prosesser

En semafor kan brukes til å synkronisere to prosesser. Anta prosess B (PB) må vente til prosess A (PA) er ferdig med noe i sin kode (kodelinje A3 i eksempelet), før den kan gå videre (med kodelinje B2 i eksempelet). Med en semafor S initialisert til 0, kan de da synkroniseres som følger:

PA	PB-kode
A1	B1
A2	Wait(S)
A3	B2
Signal(S)	B3
A4	B4

PB kan ikke gjøre B2 før PA har satt S til 1. Pilene angir Context Switch. To mulige forløp. B når først fram:

A	B	S
A1		0
A2		0
$\Longrightarrow$	B1	0
	Wait(S)	-1
A3	$\Longleftarrow$	-1
Signal(S)		0
A4		0
$\Longrightarrow$	B2	0
	B3	0

A når først fram:

A	B	S
A1		0
A2		0
A3		0
Signal(S)		1
A4		1
$\Longrightarrow$	B1	1
	Wait(S)	0
	B2	0
	B3	0

12.10 Tanenbaums bruk av semaforer

Semaforene som er omtalt i læreboka er de samme som omtalt her men Tanenbaum bruker betegnelsene up og down istedet for signal og wait. I tillegg blir semaforen ikke mindre enn null, den beholder verdien null om en prosess kaller wait, men prosessen blir satt i kø. Men når en annen prosess gjør signal vil prosessen som lå i kø vekkes og de to kallene nulle ut verdien på semaforen. I praksis blir effekten den samme. Men vår bruk av semaforen viser tydligere hvor mange som ligger i kø.

12.11 Låse-mekanismer brukt i Linux-kjernen

Linux-kjernen kan selv skru av og på interrupts for å sikre at korte kode-biter ikke blir avbrutt. Flere CPUer kan samtidig kjøre kjerne-kode, derfor er låser mye i bruk for å unngå at datastrukturer aksesseres samtidig.

Atomiske operasjoner Operasjonen kan ikke interruptes, eksempel: `atomic_inc_and_test()`

Spinlocks Mest brukt. For korte avsnitt. Bruker busy waiting.

Semaforer Kjernen sover til semaforen blir ledig igjen om den er opptatt.

Reader/Writer locks Samtidige prosesser kan lese, men bare en CPU av gangen kan skrive

12.12 Monitorer og Java synkronisering

Det viser seg at i praksis er det vanskelig å skrive korrekte programmer med semaforer. Programmeren er helt overlatt til seg selv og ett signal for mye vil ødelegge hele systemet. Derfor ble konseptet monitor

laget. Dette er en del av et programmeringsspråk og kan sørge for at hele metoder eller deler av kode synkroniseres. Kun en monitor-metode kan kjøre av gangen og dermed sikres synkronisering på et høyere nivå.

Dette er implementert i Java som har et eget statement `synchronized` for å synkronisere bruken av felles variabler. Alle java-objekter har en egen monitor-lås, tilsvarende variabelen lock vi har brukt i tidligere eksempler.

Når man bruker statementet `synchronized()` må man derfor knytte det opp mot ett objekt og dermed bruke dette objektets lås. En integer verdi som variabelen `saldo` er ikke et objekt, i motsetning til for eksempel et array, og derfor lager vi et objekt som vi kaller `lock` for så å knytte `synchronized()` opp mot dette objektet:

```
public static int saldo;                  // Felles variable, gir race condition
public static Object lock = new Object(); // Argumentet til synchronized må være et objekt
```

Dermed kan man gjøre `synchronized(lock)` rundt en kodeblokk

```
synchronized(lock)
   {
      saldo++;
   }
```

hvor man synkronisere mot lock-objektets lås.

I praksis betyr det at om en tråd kjører instruksjoner inne i denne kodeblokken, vil andre tråder settes på vent om de prøver å kjøre det samme kodeavsnittet. Dette avsnittet er da et kritisk avsnitt. Det tilsvarer helt å kalle wait før og signal etter et kritisk avsnitt. Gjør man det med eksempelet fra forrige forelesning, tar beregningene lenger tid, men saldo blir 0 til slutt.

Ser man på java bytekoden, kan man se at koden som oppdaterer saldo da blir beskyttet i en monitor:

```
$ javap -private -c SaldoThread

      17: getstatic     #17                 // Field lock:Ljava/lang/Object;
      20: dup
      21: astore_2
      22: monitorenter
      23: getstatic     #18                 // Field saldo:I
      26: iconst_1
      27: iadd
      28: putstatic     #18                 // Field saldo:I
```

Man kan også definere hele metoder som synchronized. I Figure 2-35 i læreboka defineres i et eksempel metoden

```
public synchronized void insert(int val)
```

Dette forsikrer programmereren om at kun en tråd av gangen kan kjøre denne metoden.

En løsning på vårt problem kunne være å gjøre følgende:

```
  private static synchronized void upSaldo()
     {
saldo++;
     }

   private static synchronized void downSaldo()
     {
saldo--;
     }
```

og bruke disse metodene istedet for å endre variabelen direkte. Da ville også koden bli threadsafe og alltid gi saldo lik null til slutt.

12.13 Message passing

En annen metode for å sikre serialisering som også virker i distribuerte systemer er message passing. Konkurrerende tråder eller prosesser synkroniseres da ved å sende signaler til hverandre. En ulempe ved dette er at det generelt ikke er like effektivt som semaforer og monitorer.

12.14 Dining Philosophers Problem

Dette er et klassisk synkroniseringsproblem og som ofte har blitt brukt for å demonstrere synkroniseringsteknikker som semaforer og monitorer. Fem filosofer sitter rundt et bord og har fem gafler på deling. Når filosofene ikke tenker spiser de, men de trenger to gafler for å kunne spise. Hvordan kan man skrive fem filosof-prosesser som kjører samtidig og samtidig unngå en situasjon hvor alle griper en gaffel og blir sittende og vente? Dette er en såkalt deadlock-tilstand, vranglås, og det må unngås når man har samtidige tråder.

Filosoftilstander:

1. tenker (uten gafler)
2. Spiser spaghetti med 2 gafler

Tar opp en gaffel av gangen.

Problem Programmer en filosofprosess slik at 5 prosesser kan spise og tenke i tidsrom av varierende lengde og dele ressursene (gaflene) **uten** at deadlock (vranglås) kan oppstå.

12.15 Deadlock

To eller fler prosesser venter på hverandre, ingen kommer videre. Eks. 1: P1 venter på P2, P2 venter på P3 og P3 venter på P1 (sirkulær venting)

Eks. 2: Deadlock med to semaforer S1 og S2, initialisert til 1:

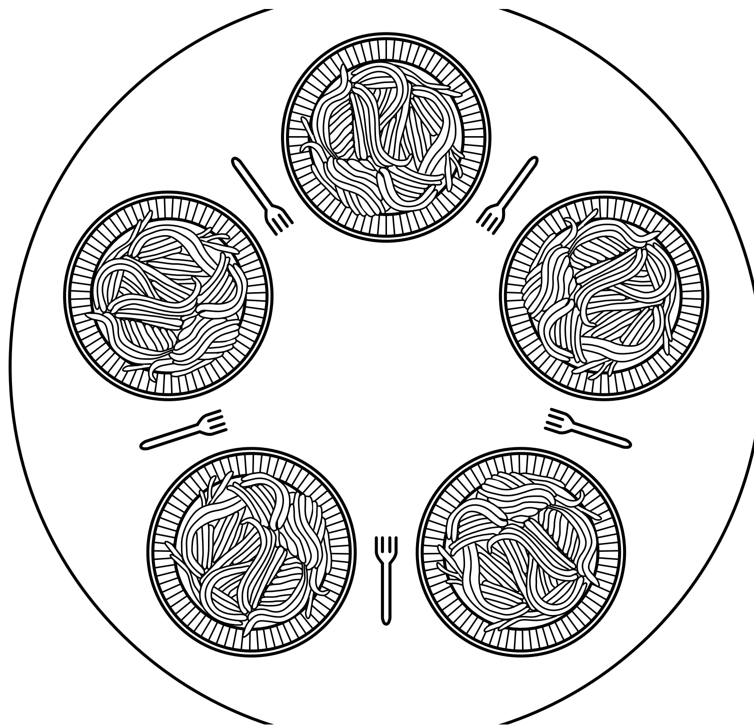


Figure 67: Fra Tanenbaum: Lunch time in the Philosophy Department.

PA	PB-kode
Wait(S1)	
	Wait(S2)
Wait(S2)	
	Wait(S1)
.	.
.	.
.	.
Signal(S1)	Signal(S2)
Signal(S2)	Signal(S1)

12.16 Kriterier for at deadlock kan oppstå

1. Mutex: ressurser som ikke kan deles
2. En prosess kan beholde sine ressurser mens den venter på andre.
3. En prosess kan ikke tvinges til å gi opp sin ressurs (felles minne, disk, etc.)

Med 1, 2 og 3 oppfylt, kan deadlock oppstå ved sirkulær venting! Mulige løsninger:

1. Forhindre. Internt i OS-kjernen må deadlock forhindres. Umulig å forhindre bruker-deadlock.
2. Løse opp deadlock. Generelt vanskelig.
3. Ignorere problemet (mest vanlig metode for et OS).

12.17 Tråder i Python

Global Interpreter Lock (GIL) er en mekanisme som brukes av CPython, den mest populære implementeringen av Python, for å sikre at kun én tråd utfører bytekodeinstruksjoner om gangen. Dette låser hovedinterpreterløkken, slik at tråder ikke kan utføre Python-bytecode parallelt, selv på en flertrådet prosessor.

import multiprocessing, gir en løsning som lar deg opprette prosesser, som hver har sin egen Python-interpreter og minneplass. Dette omgår GIL og lar deg utnytte flere CPU-kjerner.

Noen biblioteker som NumPy og Pandas er implementert i C og frigjør GIL under tunge beregninger. Dette kan tillate parallell utførelse av beregninger uten å være begrenset av GIL.

Hvis man kjører følgende kode på en server med flere CPUer, vil kun en av trådene kjøre av gangen:

```
import threading

class SaldoThread(threading.Thread):
    MAX = 200000000
    count = 0
    saldo = 0  # Felles variabel, gir race condition?

    def __init__(self):
        super().__init__()
        SaldoThread.count += 1
        self.id = SaldoThread.count

    def run(self):
        print(f"Tråd nr. {self.id} starter")
        self.update_saldo()

    def update_saldo(self):
        if self.id == 1:
            for _ in range(SaldoThread.MAX):
                SaldoThread.saldo += 1
        else:
            for _ in range(SaldoThread.MAX):
                SaldoThread.saldo -= 1
        print(f"Tråd nr. {self.id} ferdig. Saldo: {SaldoThread.saldo}")


class NosynchThread:
    @staticmethod
    def main():
        print("Starter to tråder!")

        s1 = SaldoThread()
        s2 = SaldoThread()
        s1.start()
        s2.start()

        s1.join()
        s2.join()

        print(f"Endelig total saldo: {SaldoThread.saldo}")


if __name__ == "__main__":
```

```
NosynchThread.main()
```

13 Forelesning 9/4-24. Internminne

Slides brukt i forelesningen[251]

13.1 Forelesningsvideoer

Uredigert opptak av hele første time av forelesningen (00:44:25)[252]

Uredigert opptak av hele andre time av forelesningen (00:55:46)[253]

os13del5.mp4[254] (07:59) Slide: Internminne og Cache
os13del6.mp4[255] (03:21) Slides: Minnepyramiden, Internminnet
os13del7.mp4[256] (00:07) Slide: Adresserommet
os13del8.mp4[257] (07:21) Slides: Virtuelt adresserom
os13del9.mp4[258] (01:26) Spørsmål: Hva er GDDR? Graphics DDR som brukes sammen med GPU-er
os13del10.mp4[259] (03:33) Spørsmål: Hva slags registere lagrer adresser? Vanlige CPU-registere. Demo av hvordan det ser ut i simulerings-CPUen
os13del11.mp4[260] (09:52) Slides: Virtuelle (logiske) adresser, Internminnet og kompilering, linking og loading
os13del12.mp4[261] (08:10) Demo: Generering og bruk av statisk og dynamisk bibliotek i C++
os13del13.mp4[262] (01:30) Slide: Linux-prosess segmentation
os13del14.mp4[263] (05:56) Slides: Minneadressering og MMU med lite eksempel
os13del15.mp4[264] (05:41) Slides: Paging, pages og page table entry
os13del16.mp4[265] (04:43) Slides: TLB - Translation Lookaside Buffer, ytelse og TLB-cache
os13del17.mp4[266] (05:54) Slides: Konkret eksempel med MMU-oversettelse med 64K virtuelt og 32K fysisk internminne
os13del18.mp4[267] (02:25) Slides: Paging og swapping, paging-algoritmer
os13del19.mp4[268] (01:04) Spørsmål: Er load 32 den designerte start-adressen? Ja, dette er en virtuell adresse som må oversettes
os13del20.mp4[269] (01:03) Spørsmål: Hva om en prosess bruker adresse som ikke er egne? Segmentation fault
os13del21.mp4[270] (03:20) Spørsmål: Hvorfor lager man ikke større L1, for eksempel 1 GB?

[251] https://www.cs.oslomet.no/ haugerud/mem.pdf
[252] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13time1.mp4
[253] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13time2.mp4
[254] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del5.mp4
[255] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del6.mp4
[256] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del7.mp4
[257] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del8.mp4
[258] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del9.mp4
[259] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del10.mp4
[260] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del11.mp4
[261] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del12.mp4
[262] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del13.mp4
[263] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del14.mp4
[264] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del15.mp4
[265] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del16.mp4
[266] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del17.mp4
[267] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del18.mp4
[268] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del19.mp4
[269] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del20.mp4
[270] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os13del21.mp4

13.2 Internminne

Om internminnet bruker man ofte betegnelsen RAM som er en forkortelse for Random Access Memory. Det kalles 'Random' fordi hvilken som helst byte kan leses ut eller aksesseres like raskt som enhver annen byte. Men som vi skal se vil det i praksis ikke alltid stemme at det tar like lang tid å laste inn to forskjellige byte fra RAM. En årsak til dette som gjelder for de aller fleste systemer er cache som mellomlagrer data. Hvis vi henter inn verdine på en variabel og denne ligger i cache, går inntil ti ganger raskere enn om den må hentes helt fra RAM. En annen årsak som gjelder større servere er at servere med flere titalls CPU-er ofte er delt inn i såkale numanodes som kommuniserer raskere med enkelte tilordnede deler av RAM. Dette kan utgjøre en hastighetsforskjell på inntil to tre ganger.

Vi så i avsnitt 7.3 at både CPU-registre og cache er laget av SRAM (Static RAM), men det er ikke en del av internminnet. Aksess til SRAM er ekstremt hurtig og SRAM er statisk i den betydning at det ikke trenger å oppfriskes. Internminnet er laget av DRAM som står for Dynamic RAM. Mer en 10 ganger i sekundet må DRAM opplades, ellers forsvinner informasjonen. SRAM består av 6 transistorer for hver bit som lagres. Men DRAM trenger bare en transistor og en kapasitator(lagrer elektrisk ladning) for å lagre en bit. Derfor er DRAM billigere, mindre, bruker mindre effekt og kan lages i større enheter. Internminnet består derfor av DRAM eller forbedrede varianter av DRAM. DDR5 SDRAM (Double-Data Rate generation 5 Synchronus Dynamic RAM) er et av de foreløpig siste leddene av kjedene av forbedrede utgaver av DRAM.

I samme avsnitt viste Fig. 78 noen typiske størrelser og aksesstider for de sentrale lagringsmedien som finnes i en datamaskin, fra registre til harddisk. Legg spesielt merke til den store forskjellen i aksesstid mellom internminnet og harddisk, selv moderne superraske SSD-disker er tusen ganger tregere.

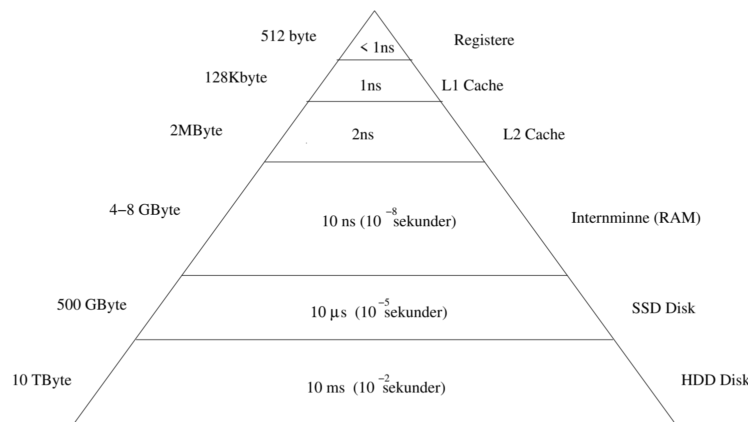
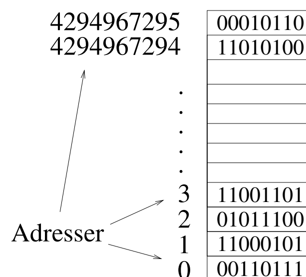


Figure 68: Minne-pyramiden. Størrelsen og tiden det tar å hente data øker nedover pyramiden.

Internminnet (RAM, Random Access Memory/arbeidsminne) er et stort array av bytes:



Alle disse adressene utgjør tilsammen det *fysiske minnet* siden det er adresser til fysisk DRAM enheter som lagrer bit.

I følgende tabell kan man se hvor mange adresser det er plass til for forskjellige størrelser av registre. Det totalet antall adresser som finnes for en gitt størrelse av et register kalles et adresserom.

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

13.3 Virtuelt adresserom

Generelt er det ikke plass til alle programmer i internminnet på en gang. Derfor gir man hvert enkelt program sitt eget virtuelle adresserom fra 0 til det programmet måtte trenge. Er adresse-registeret 32 bit, er typisk det virtuelle adresserommet opp til 4Gbyte. Det vil da virke for prosessen som den har tilgang til alt dette minnet. Men i virkeligheten er ikke nødvendigvis alt i bruk og av det som er i bruk kan noe ligge i RAM og andre deler på disk.

Disse virtuelle eller logiske adressene brukes overalt hvor programmet refererer til seg selv, for eksempel i en instruksjon som

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

som betyr last inn byte nummer 1023 i det 8 bit store registeret %al. 1023 er da den virtuelle adressen. Når programmet lastes inn i internminnet og kjøres vil det variere hvor i det fysiske minnet programmet legges. Det må derfor være mulig å oversette mellom virtuelle og fysiske adresser.

13.4 Internminnet/RAM

Et kjørbart program ligger i utgangspunktet på harddisken, men må lastes inn i internminnet før det kan kjøres. Skjematisk må kildekode gjennom prosessen i Fig. 69 før den kan kjøres.

13.5 C++ library

Hvis man lager C++-prosjekter og har metoder man ofte bruker, kan man lage sin egen library-fil som man kobler sammen med hovedprogrammet når man skal bruke det. Kompileringen av biblioteksfilen kan se slik ut:

```
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
```

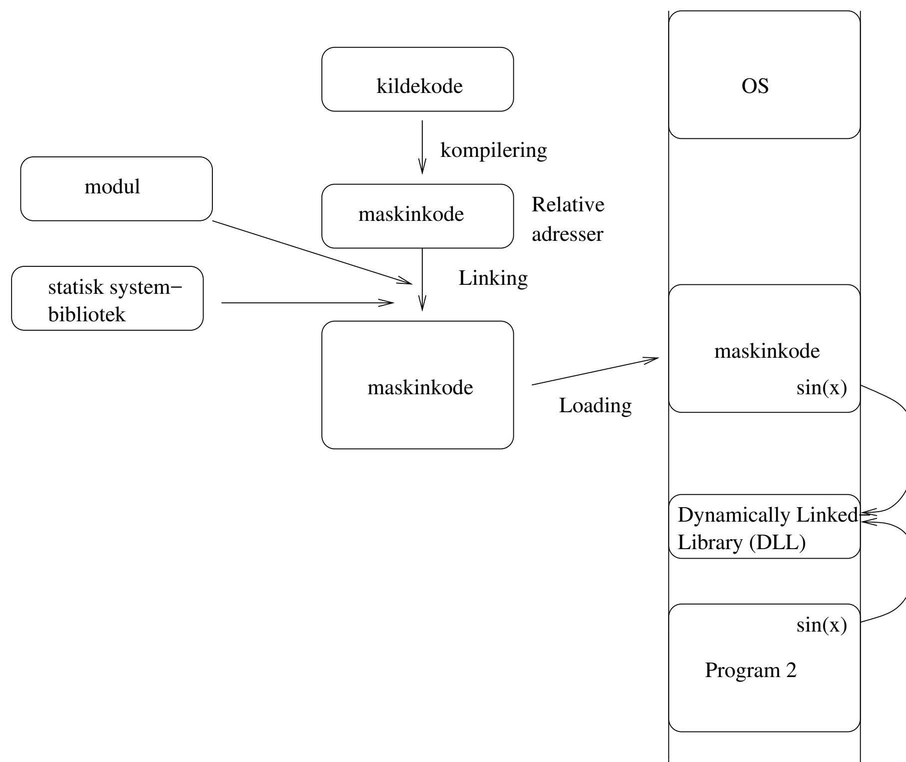


Figure 69: Loading av et brukerprogram/en prosess

```
g++ -o sim mainsim.o simulation.o user.o  -lm  -L../Tools -lTools
sim  # Kjører programmet
```

Dette linker (limer sammen) de 3 programmene med libTools.a (som loaderen finner pga -L../Tools) og andre biblioteker (-lm tar med et matte-bibliotek) og lager en kjørbar fil med navn sim. Man kan også lage et dynamisk (shared) library med filendelse so.

```
g++ -fpic -c randTools.cpp
g++ -fpic -c calcTools.cpp
g++ -shared -o libsTools.so randTools.o calcTools.o

g++ -o sim mainsim.o simulation.o user.o  -lm  -L../sTools -lsTools
export LD_LIBRARY_PATH="../sTools"
sim  # Kjører programmet
```

13.6 Layout av en prosess sitt adresserom/segmentation

Det virtuelle adresserommet til en prosess er delt opp i regioner eller segmenter og de følgende er de viktigste:

- Den statiske binære koden som prosessen kjører, text segmentet, ofte bare kalt text
- Heap'en hvor globale variabler og data som dynamisk generers lagres
- Stack'en hvor de lokale variablene lagres, brukes også til funksjonskall

- MMAP, minneavbildninger av filer(og devicer) på disk direkte i det virtuelle minnet

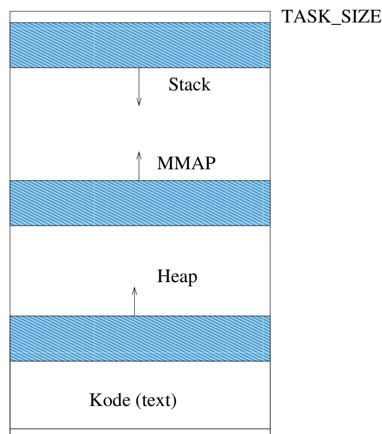


Figure 70: Layout for det virtuelle adresserommet for en Linux-prosess.

13.7 Minneadressering og MMU

Et programs virtuelle adressering til variabler, subrutiner, bibliotek, data og så videre må knyttes til fysiske adresser. Dette kunne skjedd ved loading, men ville vært svært tidkrevende og tungvint. I moderne OS gjøres dette dynamisk mens programmene kjører. Dette muligjør at programmer og biblioteker kan flyttes til og fra harddisk og bare loades når det er behov for dem. Om OS skulle oversette fysiske adresser til logiske/virtuelle, ville det belaste CPU-en for mye, så dette tar en egen enhet, MMU (Memory Managment Unit), seg av.

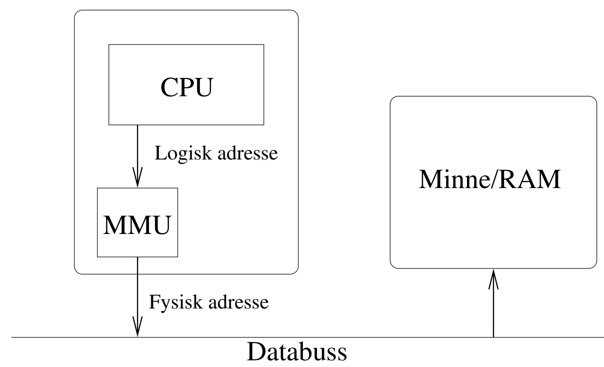


Figure 71: MMU oversetter logiske adresser fra CPU til fysiske RAM-adresser i realtime

13.8 Eksempel på MMU-tabell

Anta at de to programmene Prog1 og Prog2 skal kjøre på en maskin. Som i Fig. 72 vil adressene i de kompilerte programmene være logiske og ikke til en fastlagt adresse i minne. Dette fordi man da bare trenger å oppdatere MMU-tabellene når programmene plasseres eller omplasseres i RAM. Når CPU utfører instruksjonen 'load 32' for Prog1 som refererer til minnet sendes den logiske adressen 32 til MMU

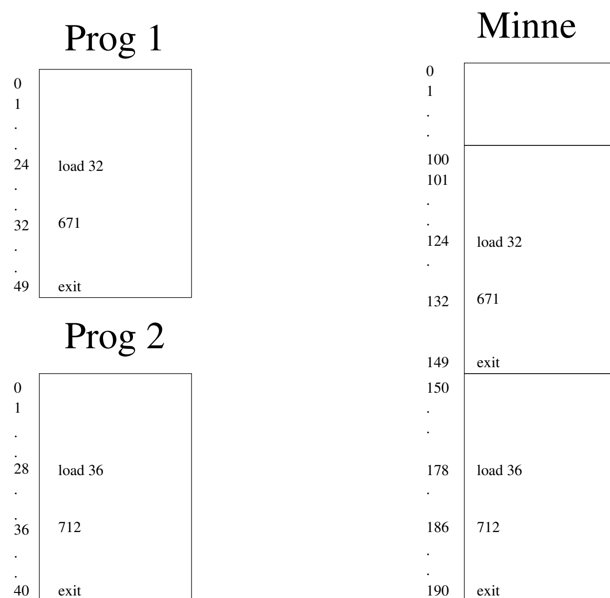


Figure 72: Den kompilerte Prog1 og Prog2 maskinkoden inneholder logiske adresser

som bruker sin tabell for Prog1-adresser til å oversette til den fysiske adressen 132, slik at riktig byte blir loadet. MMU trenger da tabeller som ser slik ut:

Prog1		Prog2	
0	100	0	150
.	.	.	.
.	.	.	.
24	124	28	178
.	.	.	.
.	.	.	.
32	132	36	186
.	.	.	.
40	140	40	190

Men det vil være alt for minnekrevende å ha en linje i tabellen for hver adresse! Det logiske minnet deles derfor opp i pages som legges i det fysiske minnet hver for seg. I eksempelet over kunne en page f. eks. utgjøre 50 adresser. MMU trengte da bare å vite at Prog1 starte på adresse 100 og at Prog2 startet på adresse 150.

13.9 Paging

Ved å dele inn minnet i like store biter (pages/sider), vil man effektivt kunne laste disse sidene inn og ut av minnet og samtidig enkelt holde oversikt over hvor hver side er i en page-tabell. Dette gjør at det er enkelt og plassbesparende å dynamisk allokere (sette av) nytt minne til en prosess. Man unngår den fragmentering som ville oppstått om vilkårlig store biter av minnet ble tildelt en prosess. Når prosessen ble avluttet, ville det da blit et hull med tilgjengelig med akkurat denne størrelsen. Ved å bruke faste sidestørrelser, unngår man slike ujevnt store hull. Inndelingen i sider gjør at deler av programmer effektivt kan lastes inn og ut av minnet og dermed gjør det enkelt å implementere virtuelt minne. En viktig fordel for OS er at med full oversikt over en prosess sitt minnet i en page-tabell, er det lett å kontrollere at

prosessen bare skriver til det minnet den er tildelt. Oppsummert har bruk av logiske eller virtuelle minneadresser og inndeling av disse i sider av samme størrelse har følgende fordeler:

- Fast sidestørrelser hindrer fragmentering
- Dynamisk flytting av deler av prosesser til og fra disk
- Full kontroll for OS over prosessers minnebruk
- Mulliggjør å bruke diskplass til å utvide minnet, virtuelt minne

13.10 Pages

En page har en størrelse 2^n bytes og typisk er n = 12 eller 13 og page-størrelsen er dermed 4 eller 8 Kbytes. Den konkrete størrelsen avhenger av prosessorens arkitektur. 4Kbytes er vanlig for X86-prosessorer. Figur 73 hviser et eksempel på hvordan pages kan fordeles i minnet og hvor enkel MMU-tabellen da blir. Ved en context switch lagres tabellen for den gamle prosessen i dens PCB og tabellen til den nye prosessen lastes inn i MMU.

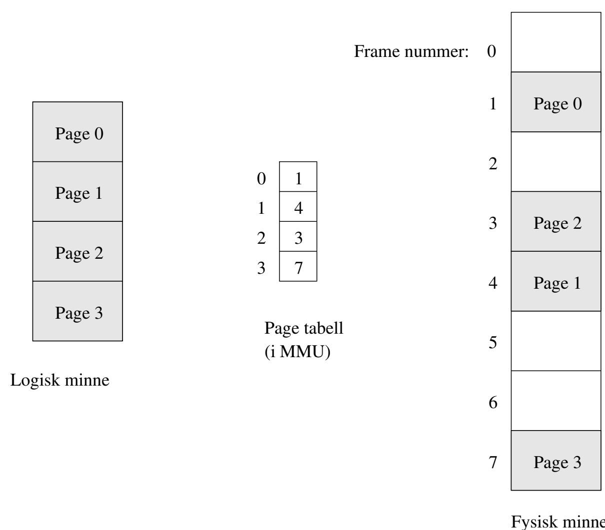


Figure 73: Logisk minne og paging

En prosess som bruker 100Mbyte minne vil med 4Kbyte page størrelse bestå av omtrent 25.000 sider men en prosess som bruker 4GByte RAM vil ha en million sider i MMU. Det er ikke plass til å lagre adressen til alle disse sidene i selve MMU og den fullstendige tabellen ligger selv i internminnet. Men MMU bruker en Translation Lookaside Buffer (TLB) som er hurtig cache minne som inneholder en del av page-tabellen. Ved oppslag på adresser til sider som ikke ligger her, hentes de fra minnet, men da tar det vesentlig lenger tid.

13.11 MMU eksempel med 4k page-størrelse

I Figur 74 som er hentet fra Tanenbaum, ser man et eksempel på en avbilding fra det virtuelle 64K store virtuelle adresserrommet til det 32K store fysiske adresserommet. Vanligvis er begge adresserom mye større. Og man må huske at hver prosess får tildelt sitt eget adresserom og dette peker på fysiske adresser i RAM. Andre prosesser vil da peke til andre steder i RAM. Hver gang det gjøres en context

switch og en annen prosess starter å kjøre på en CPU, må denne prosessens MMU-tabell lastes inn før den kan begynne å kjøre. Dette er eksempel på en slik tabell etter at den er lastet inn. Vi ser at det er åtte (0-7) virtuelle pages som peker på åtte fysiske page-frames i RAM.

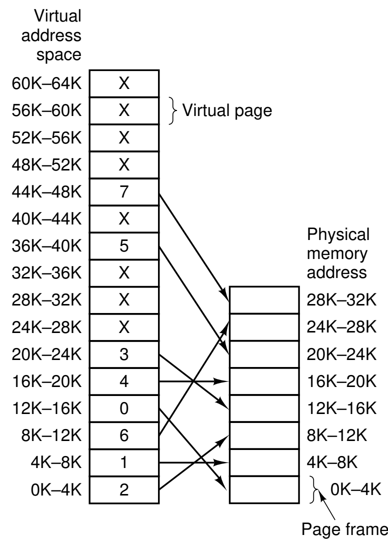


Figure 74: Figure 3-9 i Tanenbaum. Forholdet mellom virtuelle og fysiske adresser

Når MMU mottar en innkommende virtuell adresse, 8196 i eksempelet i Figur 75, må denne adressen ekstremt hurtig oversettes til en fysisk adresse. I dette tilfellet er side-størrelsen 4K og 12 bit vil da kunne brukes til å adressere hele siden. Det virtuelle adresserommet er på 64k og man trenger da 16 bit for å adressere hele dette adresserommet.

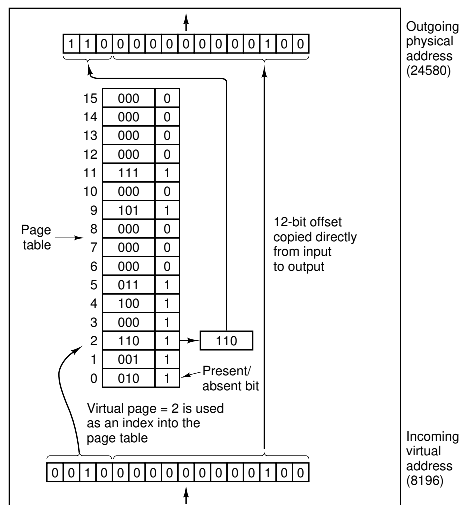


Figure 75: Figure 3-10 i Tanenbaum. Slik oversetter MMU virtuelle(logiske) adresser til fysiske.

Vi ser at de fire første bit'ene brukes til å angi hvilket nummer i rekken av 16 pages som en adresse hører til. I eksempelet er de fire første bit'ene 0010 = 2 og det betyr dermed virtuell page nr. 2. MMU-tabellen viser hvilken fysisk frame hver av de 16 virtuelle sidene peker på og vi ser at page 2 peker på fysisk frame nummer 110 = 6. Dette kan vi også se i Figur 73, der en pil viser at page nummer 2 peker på fysisk frame nummer 6 (man starter å telle på null). Oversettelsen skjer lynraskt ved at de tre bit'ene i indeks 2 i MMU-tabellen, 110, hektes på foran de 12 bit'ene som forteller hvor i den 4K store framen byte'en som

ønskes ligger. Dette gir dermed øyeblikkelig den utgående adressen 24580. Det fysiske adresserommet er på 32k og 15 bit er da nok til å dekke hele adresserommet.

Hver page har 2^12 = 4096 adresser. Page 0 begynner på 0, page 1 begynner på 4096, page 2 begynner på 8192 og så videre. Inkommende adresse er 8196, fordi den er 2x4096 (første byte page 2, 0010) + offset 4 (100) som tilsammen blir 8192 + 4 = 8196. Den utgående adressen til fysisk frame blir 24580, fordi den er 6*4096 (første byte page 6, 110) + offset 4 (100) som tilsammen blir 24576 + 4 = 24580.

13.12 Paging og swapping

Å dele in det logiske minnet i pages gjør det mulig å dynamisk laste inn og ut deler av en prosess. Dermed kan minnet til det samlede antall prosesser på en maskin være større enn det fysiske minnet, resten lagres page for page på harddisk på swap-området. Det virtuelle minnet til en prosess kan da fysisk lagres både i RAM og på disk. I Fig. 77 har fysisk minne bare plass til 3 pages og resten må lagres på disk. Det å laste pages til og fra swap-området på disken kalles "paging".

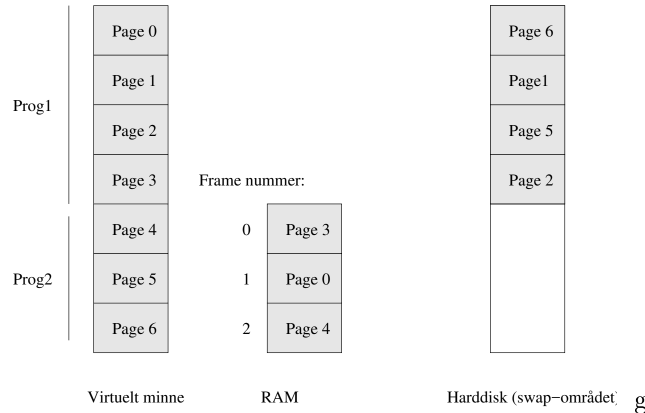


Figure 76: Virtuelt minne. Bare en page av Prog2 ligger i minne. Om data fra page 5 eller 6 blir spurt etter, må disse lastes inn.

Tidligere var swapping eneste måte å bruke disk til virtuelt minne; da blir hele prosessen lastet ut på disk. Med en disk med lesehastighet på 100MByte/s tar det 10 sekunder å swappe en prosess på 1 GByte. Swapping brukes i moderne OS oftest bare når det er ekstrem mangel på minne. Er det fysiske minnet altfor lite, vil OS bruke nesten all sin tid på å flytte sider til og fra disk; dette kalles trashing. Det er viktig å huske at det kan ta flere hundere tusen ganger så lang tid å hente pages fra en disk som å aksessere RAM, så virtuelt minne kan på ingen måte fullt ut erstatte RAM.

13.13 Page Table entry

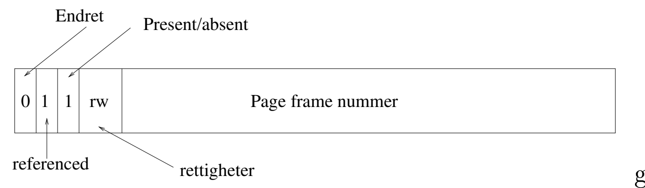


Figure 77: Page table entry.

- Page Frame nummer: Fysisk frame-nummer i RAM
- Present: Hvis 0 blir det en page-fault
- Endret: Hvis 1 er siden dirty og må skrives til disk om den fjernes
- Rettigheter: lese, skrive, kjøre
- Referenced: settes hvis brukt, brukes av paging-algoritmer

13.14 TLB - Translation Lookaside Buffer

- En prosess som bruker 100Mbyte minne vil med 4Kbyte page størrelse bestå av omtrent 25.000 sider
- 4GByte 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
- Inneholder en liten del av page-tabellen,
- Ved oppslag på adresser til sider som ikke ligger i TLB, hentes de fra RAM
- Kalles TLB-miss eller soft-miss. Tar vesentlig lenger tid enn om adressen er i TLB

13.15 Typisk TLB ytelse

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

13.16 Internminnet og Cache

Vi har tidligere sett at for å kunne fore en hurtig prosessor med instruksjoner og data raskt nok, bruker man flere nivåer av mellomlagring av data, såkalt cache-minne. Det går vesentlig raskere å hente minne fra cache-minnet enn fra internminnet. I Fig. 45 så vi noen typiske størrelser og aksesstider for de sentrale lagringsmedien som finnes i en datamaskin, fra registre til harddisk. Spesielt bemerket vi den store forskjellen i aksesstid mellom internminnet og harddisk.

Cache inneholder både data, instruksjoner og deler av MMU page-tables i TLB (Translation Lookaside Buffer). I L1 cache er ofte disse egne enheter, mens L2 cache pleier å være en enhet. I de senere årene har man klart å få plass til L2 på selve prosessorchip'en (den lille brikken som utgjør mikroprosessoren, bare noen kvadratcentimeter stor). Arkitekturen til en moderne prosessor kan da i grove trekk se ut som i Fig. 78.

Noen arkitekturer har i tillegg enda et lag i minnehierarkiet, en offchip L3 cache som sitter mellom mikroprosessoren og RAM. For Intel Core i7 og AMD Opteron K10 har også L3 cache fått plass på prosessor-chip'en.

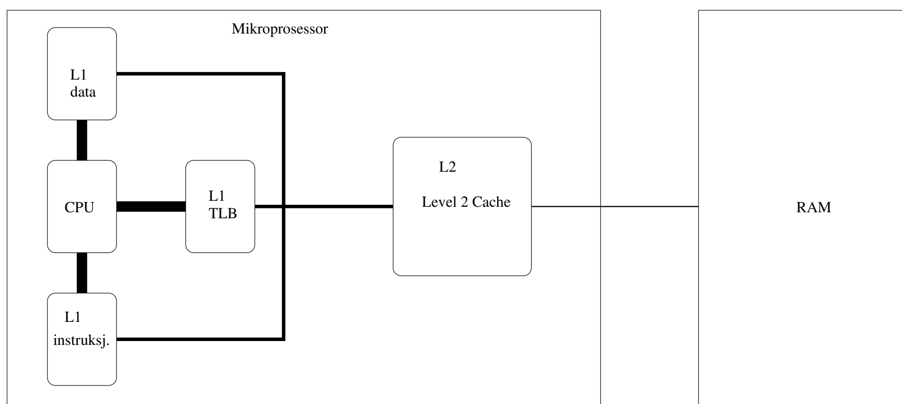


Figure 78: Level 1 cache (L1) bestående av tre deler. I AMD Athlon 64 er TLB i tillegg delt i to deler, en for adresser til instruksjoner og en for adresser til data.

13.17 Paging-algoritmer

1. Page-fault (En page mangler i minnet; ligger på swap på disk)
2. Ingen ledig frame i minnet
3. OS må velge hvilken page som skal legges på disk
4. Gjøres av paging-algoritme

14 Forelesning 23/4-24. Internminne i praksis

Slides brukt i forelesningen[271]

14.1 Forelesningsvideoer

Uredigert opptak av hele første time av forelesningen (00:46:42)[272]

Uredigert opptak av hele andre time av forelesningen (00:49:26)[273]

os14del2.mp4[274] (02:52) Oppsummering av det vi jobbet med sist, virtuelle adresser, MMU,pagetables, TLB
os14del3.mp4[275] (03:58) Slide: Dynamisk allokering
os14del4.mp4[276] (15:43) Demo: Kjøring av C-program med statisk og dynamisk allokering av minne og mointorering av RAM-bruk med top; VIRT, RES og SHR
os14del5.mp4[277] (07:03) Demo: ramsmp, test av RAM-hastighet ved lesing og skriving av blokker av forskjellig størrelse
os14del6.mp4[278] (03:04) Demo: Linux kommandoen free, fil-cache
os14del7.mp4[279] (02:34) Demo: Linux kommandoen top og RAM, VIRT, RES og SHR
os14del8.mp4[280] (03:01) Spørsmål: Hvordan kan VIRT være så stort som 4516 KByte før man i det hele tatt begynner å kjøre programmet res.c?
os14del9.mp4[281] (03:35) Tegning og forklaring: SI-enheter: K, M og GByte, binære enheter: Ki, Mi og Gi
os14del10.mp4[282] (14:41) Demo og tegning og forklaring: C-program med et array på 4 GByte, små og store hopp i RAM gir svært forskjellig tidsbruk
os14del11.mp4[283] (09:08) Demo: monitorering av kjøring av array-programmet med programmet perf, cache-references, cache-misses og minor faults
os14del12.mp4[284] (09:49) Demo og tegning og forklaring: C-program med en 20K*20K matrise, ombytte av indeks gir stor forskjell i tidsbruk, perf
os14del13.mp4[285] (02:38) Slide: Noen viktige minne-begreper
os14del14.mp4[286] (03:02) Spørsmål: Hvorfor så mange som 204 context-switches ved den matrise-kjøringen som tok lengst tid? Trolig tilfeldig
os14del15.mp4[287] (01:48) Spørsmål: Hva er filsystem-cache?

[271] https://www.cs.oslomet.no/ haugerud/memogdisk.pdf
[272] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14time1.mp4
[273] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14time2.mp4
[274] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del2.mp4
[275] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del3.mp4
[276] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del4.mp4
[277] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del5.mp4
[278] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del6.mp4
[279] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del7.mp4
[280] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del8.mp4
[281] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del9.mp4
[282] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del10.mp4
[283] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del11.mp4
[284] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del12.mp4
[285] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del13.mp4
[286] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del14.mp4
[287] https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os14del15.mp4

14.2 Dynamisk allokering

Et program kan be om at det settes av minnet til sine variabler før det starter, men det kan også be om at minne allokeres dynamisk. F. eks. vil et Java-statement

```
PCB = new process;
```

gjøre at denne plassen settes av i minnet først når programmet utfører det. Programmet tildeles page for page med minne. I C++ må man eksplisitt delete objekter som ikke er i bruk lenger for å frigjøre minne, JVM utfører dette automatisk (garbage collection). Den delen av et programs minne som inneholder variabler og data og som dynamisk kan øke og minke i størrelse, kalles ofte heap. Variabler i funskjoner/metoder som forsvinner og ikke kan brukes mer etter at kallet på funksjonen er ferdig, legges på stack.

14.3 VIRT, RES og SHR i top

Hvis man kompilerer og kjører følgendeC- program på en Linux-maskin, vil man kunne observere hvordan størrelsene VIRT, RES og SHR endrer seg.

```
#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;
     }

   printf("\nVenter på å avslutte: ");
   scanf("%d",&size);
}
```

Programmet starter med å definere et statisk array med 1 M elementer som hver er på 4 byte, altså 4MiB. Både et slikt statisk array og et dynamisk array som lages med malloc() vil legges på heap.

Størrelsen VIRT er beskrevet slik i manualsiden for top:

```
VIRT  --  Virtual Memory Size (KiB)
```

```
          The total amount of virtual memory used by the task.  It includes all code, data and shared libraries
          plus pages that have been swapped out and pages that have been mapped but not used.
```

VIRT er altså all det internminnet som prosessen kan tenkes å bruke, men som ikke nødvendigvis ligger i RAM. Det som ikke ligger i RAM, ligger i SWAP-området på disken.

Hvis vi kjører programmet med bare ett element i arrayet, får vi

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
19924 haugerud  20   0    4352    652    584 S   0,0  0,0   0:00.00 a.out
```

mens hvis vi øker størrelsen til 1024*1024 som vist i koden over, kompilerer og kjøre på nytt, viser top:

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
25448 haugerud  20   0    8448    640    572 S   0,0  0,0   0:00.00 a.out
```

Og endringen i VIRT er 8448 - 4352 = 4096 og altså nøyaktig 4MiB = 4*1024*1024. Denne størrelsen er definert når programmet starter, men man kan dynamisk legge til mer internminnet som vist i koden over med funksjonen malloc. Hvis vi dynamisk legger til et nytt array med 1 M elementer

```
rex:~/mem/a$ ./a.out

Størrelse: 1048576
Lager int  array med 1048576 elementer
```

vil vi se på top at VIRT øker

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
25448 haugerud  20   0   12548    640    572 S   0,0  0,0   0:00.00 a.out
```

og økningen er 12548 - 8448 = 4100 KiB som er som forventet ganske nøyaktig 4MiB.

Men som vi ser endres ikke RES i det hele tatt av dette og det er fordi disse array-elementene bare er allokert i det virtuelle minnerommet og ikke fysisk lastet inn i RAM. RES er definert på følgende måte:

```
 RES  --  Resident Memory Size (KiB)
          The non-swapped physical memory a task is using.
```

RES er altså den delen av prosessens internminnet som akkurat nå ligger fysisk inne i RAM. Når vi lar programmet fortsette å kjøre og tilordne verdier til alle elementene i det dynamiske arrayet, gir top dette:

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
25448 haugerud  20   0   12548   5256   1276 S   0,0  0,0   0:00.00 a.out
```

Vi ser at RES øker med litt over 4MiB som først og fremst skyldes at hele arrayet nå lastes inn i RAM.

Størrelsen SHR er i top definert som

```
SHR  --  Shared Memory Size (KiB)
         The  amount  of shared memory available to a task, not all of which is typically resident.  It simply
         reflects memory that could be potentially shared with other processes.
```

SHR er den mengden av internminnet som det kan være mulig å dele med andre prosesser; merk den ligger ikke nødvendigvis i RAM nå.

14.4 Noen minne-begreper

Soft miss page-referanse er ikke i TLB; må hentes fra internminnet

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 i 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

RES (Linux) De sider som nå er lastet inn (RESident) i fysisk RAM.

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

buffer cache Del av minnet som brukes som filsystem-cache

14.5 RAM-test

```
rex:~/mem/ramsmp-3.5.0$ ./ramsmp -b 1
RAMspeed/SMP (GENERIC) v3.5.0 by Rhett M. Hollander and Paul V. Bolotoff, 2002-09

8Gb per pass mode, 2 processes

INTEGER & WRITING         1 Kb block: 16376.46 MB/s
INTEGER & WRITING         2 Kb block: 17503.79 MB/s
INTEGER & WRITING         4 Kb block: 16018.84 MB/s
INTEGER & WRITING         8 Kb block: 17191.61 MB/s
INTEGER & WRITING        16 Kb block: 17629.32 MB/s
INTEGER & WRITING        32 Kb block: 17232.47 MB/s
INTEGER & WRITING        64 Kb block: 12087.30 MB/s
INTEGER & WRITING       128 Kb block: 10896.62 MB/s
INTEGER & WRITING       256 Kb block: 11532.74 MB/s
INTEGER & WRITING       512 Kb block: 11663.74 MB/s
INTEGER & WRITING      1024 Kb block: 12726.65 MB/s
INTEGER & WRITING      2048 Kb block: 6481.25 MB/s
INTEGER & WRITING      4096 Kb block: 2418.77 MB/s
INTEGER & WRITING      8192 Kb block: 2152.39 MB/s
INTEGER & WRITING     16384 Kb block: 2141.66 MB/s
INTEGER & WRITING     32768 Kb block: 2137.81 MB/s
```

Intel Core Duo 6600[288]

[288] https://www.cpu-world.com/CPUs/Core_2/Intel-Core%202%20Duo%20E6600%20HH80557PH0564M%20%28BX80557E6600%29.html

14.6 free

```
rex:~/www$ free -m
             total       used       free     shared    buffers     cached
Mem:          2011       1453        557          0         14        551
-/+ buffers/cache:        887       1123
Swap:         1937        683       1253
```

14.7 top

```
top - 01:50:33 up 78 days, 11:02, 35 users,  load average: 0.19, 0.51, 0.61
Tasks: 408 total,   1 running, 399 sleeping,   0 stopped,   8 zombie
Cpu(s):  0.8%us,  0.7%sy,  0.0%ni, 98.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   2059344k total,  1501056k used,   558288k free,    14572k buffers
Swap:  1983988k total,   700372k used,  1283616k free,   565164k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
23123 haugerud  20   0  2676 1428  932 R    0  0.1   0:00.21 top
    1 root      20   0  2932 1232  792 S    0  0.1   0:30.41 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.02 kthreadd
    3 root      RT   0     0    0    0 S    0  0.0   0:32.07 migration/0
    4 root      20   0     0    0    0 S    0  0.0   0:35.21 ksoftirqd/0
```

14.8 Eksempler på minnebruk

```
#include <stdio.h>

int array[200000000];
main(int argc, char *argv[]){
   int i,j;
   for(i = 0;i < 2000000;i++){
      j = i*100;
      array[i] = i;
   }
}
```

```
#include <stdio.h>

int array[10000][10000];
main(int argc, char *argv[]){
   int i,j;
   for(i = 0;i < 10000;i++){
      for(j = 0;j < 10000;j++){
 array[i][j] = 5;
      }
   }
}
```

15 Forelesning 23/4-24. Disker og filsystemer

Slides brukt i forelesningen[289]

15.1 Forelesningsvideoer

Uredigert opptak av hele første time av forelesningen (00:38:06)[290]

Uredigert opptak av hele andre time av forelesningen (01:09:50)[291]

15.2 Disker

En harddisk består av et antall plater av et magnetisk materiale. For hver plate er det et lese/skrive-hode som kan lese/skrive bits ved å måle magnetisering/magnetisere platene. De fleste disker lagrer data på begge sider av platene og har derfor lese/skrive-hode over og under.

Figure 79: Overflaten av en plate på innsiden av en harddisk. Lesehodet flyttet posisjon mens bildet ble tatt og kan derfor sees i to posisjoner.

Denne linken[292] viser en video av en åpen harddisk mens den kjører.

15.2.1 Sektor

Det området som lesehodet dekker under en rotasjon, kalles en track og en track er delt opp i sektorer. En sektor er

- grunnenhet for disker
- vanligvis på 512 bytes

[289]https://www.cs.oslomet.no/ haugerud/disk.pdf
[290]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os15time1.mp4
[291]https://www.cs.oslomet.no/ haugerud/os/Forelesning/video/2021/os15time2.mp4
[292]https://commons.wikimedia.org/wiki/File:HardDisk1.ogv

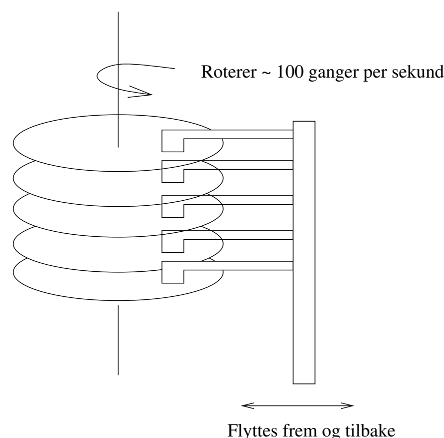


Figure 80: Tversnitt av en harddisk. En typsik rotasjonshastighet er 7200 rpm (rounds per minute).

- minste enhet som kan leses/skrives til.

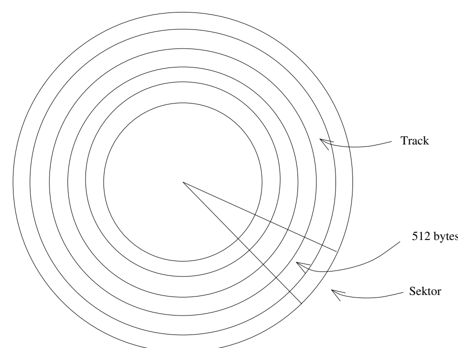


Figure 81: Hver plateoverflate er delt inn i tracks og sektorer. Den delen av en track som ligger innenfor en sektor, er den minste enheten det lagres data på og den er vanligvis på 512 bytes.

Noen ganger brukes begrepet sektor om alle tracks i en retning på disken.

15.2.2 Sylinder

En sylinder er samlingen av alle tracks fra alle platene i disken som ligger i samme avstand fra sentrum. Adressen til den minste lesbare enheteten, en sektor, er derfor gitt ved tre parametre [leshode, track, sektornummer]. Når OS vil lese noe fra disk, sendes en forespørsel med disse tre tallene.

15.3 Partisjoner

En disk-partisjon defineres som et antall sylindere som ligger fysisk samlet etter hverandre. For eksempel kan man bestemme at alle sylindere fra og med nummer 150 til og med 672 skal utgjøre en partisjon. Dette er den største enheten man deler inn en disk i. Under Windows er det vanlig ha en stor partisjon som utgjør hele disken, mens det under Linux er vanlig å dele inn disken i flere partisjoner. Monteringspunkter i filsystemet kan da tildeles bestemte partisjoner slik at for eksempel alt som ligger under `/home` legges på partisjon nummer 3 på disken.

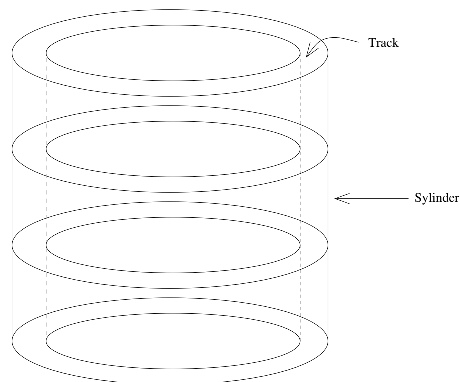


Figure 82: En sylinder defineres som samlingen av tracks på alle overflater i samme avstand fra sentrum. Sylinder nummer 39 er derfor samlingen av alle track nr. 39 på begge sider av alle platene.

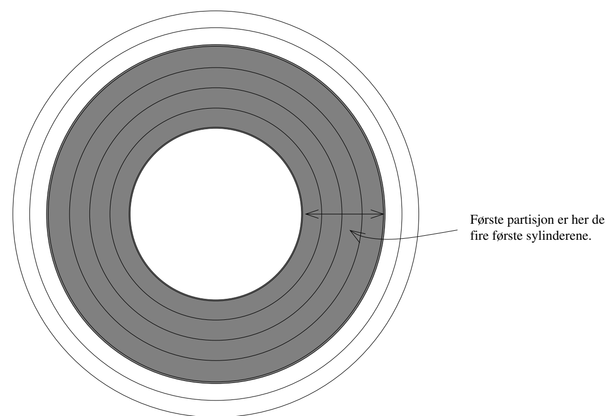


Figure 83: En partisjon består av et antall sylindre som ligger etter hverandre.

Noen av fordelene med partisjoner er:

- Hvis man har en egen partisjon for brukeres filer og partisjonen som OS ligger på blir ødelagt eller OS av andre grunner må installeres på nytt, vil man kunne beholde partisjonen med brukerfiler.
- Hvis man bare har en disk, kan man likevel ha forskjellige filsystemer og dermed forskjellige OS på den samme disken når den er delt i partisjoner.
- Mindre partisjoner og dermed mindre filsystemer er noe hurtigere enn å ha alt på en partisjon.
- Filsystemene på partisjoner kan tilpasses dataene som skal ligge der. For eksempel stor clusterstørrelse til videofilmer.

15.4 SSD (Solid State Drive)

- Basert på flash-minne som i minnepinner og har ingen bevegelige deler
- Tåler rystelser bedre og er lydløs
- Rask random aksesstid, 0.1 ms mot 5-10 ms for roterende disker
- Dyrere enn tradisjonelle disker og mindre kapasitet

15.5 Filsystemer

Før en ny disk kan tas i bruk må den formatteres. Dette er en lavnivå organisering av disken som vanligvis gjøres på fabrikken der den deles inn i sektorer, som for de fleste harddisker er på 512 byte. Når dette er gjort kan disk-controlleren lese og skrive til disse sektorene. Når man senere bruker software til å formattere en disk, er dette en høynivå formattering som setter disken tilbake til slik den var når den var ny, og i tillegg gjør operasjoner som å legge inn en boot-sektor. Før operativsystemet og applikasjoner kan ta disken i bruk må det så lages et filsystem på disken. Det finnes mange forskjellige filsystemer, NTFS er det vanligste på Windows, tidligere var FAT det vanligste. På Linux er filsystemet ext3 det vanligste. Hvis disken er inndelt i flere partisjoner, kan det lages forskjellige filsystemer på de forskjellige partisjonene. Fra Windows kan man ikke uten videre lese og skrive til partisjoner med ext3, men fra Linux kan man lese og skrive til parisjoner med FAT og NTFS.

Filsystemet tar utganspunkt i den miste enheten som kan leses fra eller skrives til, sektoren som typisk er på 512 bytes. Den viktigste oppgaven til filsystemet er å fordele mapper og filer på diskens sektorer og holde orden på hvor alt ligger. I de fleste tilfeller er en sektor for liten til å være en optimal størrelse for inndelingen av en disk og filsystemet deler derfor disken inn i større blokker (Linux: blocks, Windows: clustere). Størrelsen på blokkene må bestemmes når filsystemet lages og fordeler og ulemper ved store/små blokker må da veies mot hverandre.

- 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

15.5.1 Tabell over filenes blokker

Alle filsystemer har en oversikt over hvilke blokker enhver fil på systemet består av. Når en fil lages, vil den om mulig lagres på sammenhengende blokker. Når filene øker vil den dynamisk tildeles flere blokker, men da kan det være at det ikke er plass ved siden av de opprinnelige blokkene og filen må spres på flere områder av disken.

15.5.2 Fragmentering

En oppdeling av filer rundt om kring på disken som resultat av dynamisk allokering når filer vokser, kalles fragmentering og den blir ofte ganske omfattende på disker som er mye i bruk og hvor det meste av plassen blir brukt. Under Windows kan man defragmentere disken med Disk Defragmenter. Det må da være minst 15% ledig plass og prosessen kan ta lang tid. For Linux ext-filsystemer finnes det ikke noen innebygd defragmenterer, men det finnes slike verktøy som kan installeres.

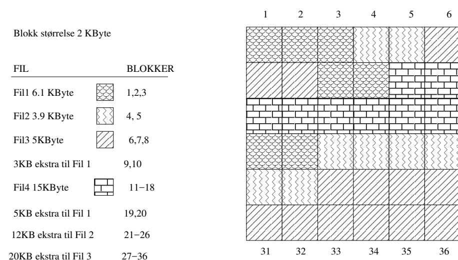


Figure 84: Filsystemet holder oversikt over hvilke blokker en fil består av. Blokkstørrelsen er 2KByte i dette eksempelet. Bare hele blokker kan allokeres til en fil, slik at all plassen ikke utnyttes når filstørrelsen ikke eksakt går opp når man deler på filstørrelsen.

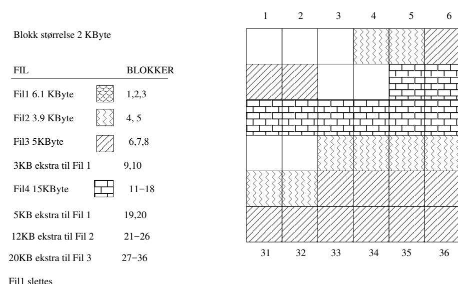


Figure 85: Når en fil slettes vil det oppstå huller på disken og dette vil føre til enda større grad av fragmentering.

15.5.3 Sletting av filer

Når en fil slettes, vil de fleste filsystemer bare slette informasjonen om filene og hvilke blokker som tilhører filene og ikke slette innholdet av blokkene. Dermed kan man med applikasjoner som autopsy eller ved å se på et disk-image direkte med en hex-editor finne igjen hele eller deler av en slettet fil. Det finnes egne applikasjoner som brukes til å slette filer bedre ved å skrive over innholdet av filene med nuller eller tilfeldige bit. Selvom man gjør en slik operasjon flere ganger, kan man med måleinstrumenter som er enda mer nøyaktige enn standard lese/skrive-hoder finne ut hva som opprinnelig var skrevet. Et eksempel på dette kan sees i Fig. 86.

Bildet er hentet fra boken Forensic Discovery av Farmer og Venema som er tilgjengelig online[293].

15.6 Lage et Linux ext3 filsystem

Først må man lage en tom fil av den størrelse man ønsker på image't.

```
haugerud@lap:~/disk/mount$ dd if=/dev/zero of=minfil bs=8M count=1
1+0 records in
1+0 records out
8388608 bytes (8,4 MB, 8,0 MiB) copied, 0,00784602 s, 1,1 GB/s
```

Dette lager en 8 MiB fil med null-tegn (ASCII tegn nummer null). Deretter kan man bygge et filsystem på denne filen:

[293] https://www.porcupine.org/forensics/forensic-discovery/

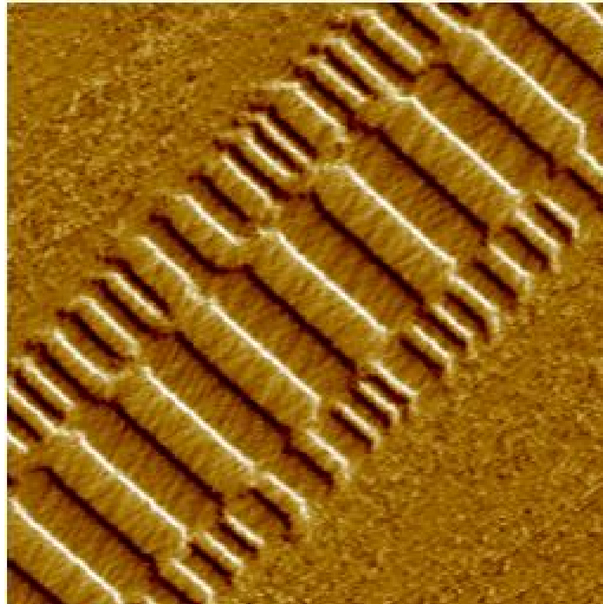


Figure 86: Rester etter data som er overskrevet på en harddisk.

```
haugerud@lap:~/disk/mount$ mkfs -t ext3 minfil
mke2fs 1.44.1 (24-Mar-2018)
Discarding device blocks: done
Creating filesystem with 8192 1k blocks and 2048 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
```

Deretter kan dette image't monteres i filsystemet på helt samme måte som om det var en disk:

```
haugerud@lap:~/disk/mount$ sudo mount minfil /mnt
[sudo] password for haugerud:
haugerud@lap:~/disk/mount$ cd /mnt/
haugerud@lap:/mnt$ ls -l
total 12
drwx------ 2 root root 12288 april 27 00:10 lost+found
```

Tidligere måtte man eksplisitt bruke opsjonen -o loop for å montere en fil, men det fungerer nå uten å spesifisere det.

15.7 NTFS

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

- Deler inn disken i clustere

- Clusterstørrelse på 512 bytes, 1 KiB, 2 KiB, 4 KiB og opp til maks 64 KiB
- 4 KiB clustere er default for disker på 2GiB eller mer
- Clusterne adresseres med 64 bits pekere
- Komprimering
- Cluster størrelse 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

15.7.1 Volum

- Et volum består av en eller flere clustere
- Kan omfatte deler(partisjoner) av en disk, en hel disk, eller flere disker
- Filsystemet defineres for dette volumet
- Maksimum antall clustere i et volum er 2^{32}, 16TiB med 4KiB clustere

15.7.2 Master File Table(MFT)

Den viktigste filen i et NTFS-volum er MFT selv.

- Filen MFT består av 16 records med metadata og deretter en record for hver fil og mappe
- Hver fil har en 1KB record som inneholder all informasjon om filen som attributter
- Eksempler på atributter: tidsstempler, filnavn, data eller peker til hvor clusterene med data ligger
- Hvis plass lagres begynnelsen av dataene i MFT record'en
- Små filer kan lagres i sin helhet i MFT record'en
- Hvis det ikke er plass til pekere til alle clusterne, lages det en peker til en ny MFT-record
- Rettigheter ble tidligere lagret i hver fil-record: hvem er eier, hvem kan lese, skrive, aksessere
- Rettigheter lagres nå i en av de 16 MFT metatdatafilene, $Secure
- OS-kjernen behandler en fil som et objekt

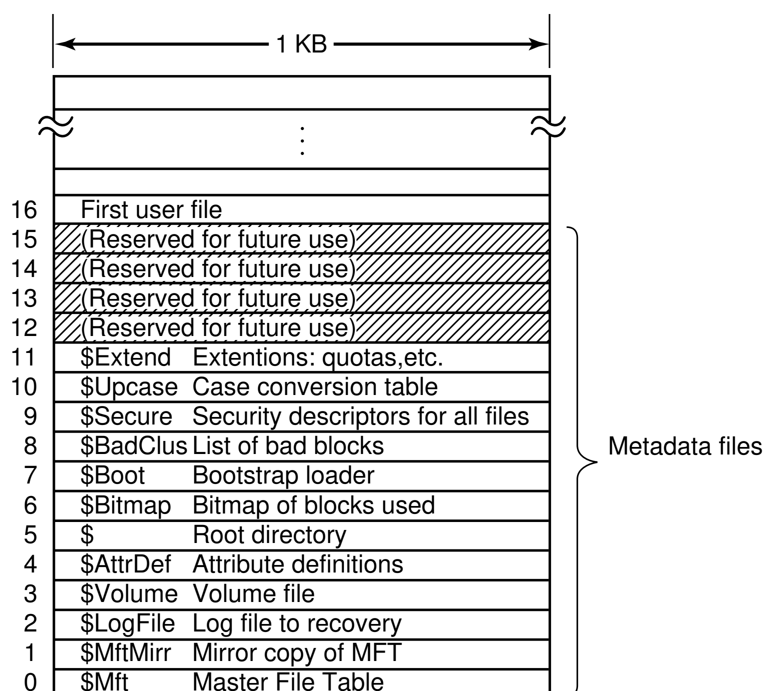


Figure 87: Figure 11-41 i Tanenbaum.

15.7.3 Linux-partisjoner

Eksempelet under er informasjon som gis når man velger p for print fra menyen etter at man som root har kjørt kommandoen `fdisk /dev/hda` på en linux PC:

```
Disk /dev/hda: 61.4 GB, 61492838400 bytes
255 heads, 63 sectors/track, 7476 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/hda1   *           1         608     4883728+  83  Linux
/dev/hda2             609         624      128520   82  Linux swap / Solaris
/dev/hda3             625        2537    15366172+  83  Linux
/dev/hda4            2538        7476    39672517+   5  Extended
/dev/hda5            2538        6500    31832766   83  Linux
/dev/hda6            6501        7476     7839688+  83  Linux
```

Den første partisjonen tildeles navnet `/dev/hda1` og består av alle sylinderne fra 1 til og med sylinder nummer 608. Hver sylinder er på 8225280 bytes og denne partisjonen er derfor på 608×8225280 bytes $= 5.0009 \cdot 10^9$ bytes = 4.66 GBytes. Alternativt sier output at denne partisjonen består av 4883728 blocks med størrelse 1024 bytes. Partisjon nr. 2 er en liten swap-partisjon på 16 sylindre og totalt 126 MByte. Den fjerde partisjonen er spesiell. Det kan bare lages fire såkalte primære partisjoner og om man skal ha flere enn fire må den fjerde lages som en extended partisjon som inneholder de resterende. `/dev/hda4` inneholder ikke data, men definerer området på disken som utgjør partisjon 5 og 6. Linux-kommandoen `df` viser hvordan filsystemet er montert på partisjonene.

for SATA og SCSI-disker heter disk-devicet vanligvis `/dev/sda` og kommandoen `fdisk /dev/sda` på en linux PC kan gi:

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

Figure 88: Figure 11-42 i Tanenbaum.

```
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  33460226    16G  5 Extended
/dev/sda5       591681536 625141759  33460224    16G 82 Linux swap / Solaris
```

og heads, tracks og cylinders er ikke lenger nevnt.

```
[root]@rex$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/hda1              4806904   4223584    339136  93% /
/dev/hda3             15124900  13790960    565632  97% /lokal
/dev/hda5             31333024  22349088   7392300  76% /mysql
/dev/hda6              7716496   2785908   4538604  39% /mln
```

Alt som ligger under `/lokal` i filsystemet vil fysisk ligge på partisjon nummer 3. Tilsvarende for partisjon 5 og 6. Resten av filsystemet, alt annent under / ligger på 1. partisjon. Neste disk vil het `/dev/hdb` og på samme måte kan denne deles inn i partisjoner og andre deler av filsystemet kan monters på disse partisjonene.

15.8 Windows-partisjoner

Om man deler opp en disk under Windows, tildeles hver partisjon bokstaver, C, D, E etc. Bokstavene A og B er tradisjonelt satt av for to diskett-stasjoner. Om man har flere disker kan hele denne eller deler av den om man lager flere partisjoner, tildels andre bokstaver. På Windows XP kan man Ved å kjøre programmet `Diskpart` se på og endre partisjoneringen av diskene:

```
C:\>Diskpart

Microsoft DiskPart version 1.0
Copyright (C) 1999-2001 Microsoft Corporation.
On computer: DIRAC

DISKPART> list disk

  Disk ###  Status      Size     Free     Dyn  Gpt
  --------  ----------  -------  -------  ---  ---
  Disk 0    Online        75 GB      0 B
  Disk 1    Online        75 GB    54 GB
```

Denne PC-en har to disker og ved å velge en av diskene kan man se hvilke partisjoner den inneholder:

```
DISKPART> select disk 0

Disk 0 is now the selected disk.

DISKPART> detail disk

WDC WD800BB-32BSA0
Disk ID: 3E423E41
Type   : IDE
Bus    : 0
Target : 0
LUN ID : 0

  Volume ###  Ltr  Label        Fs     Type        Size     Status     Info
  ----------  ---  -----------  -----  ----------  -------  ---------  --------
  Volume 2     C                NTFS   Partition     39 GB  Healthy    System
  Volume 3     F                       Partition     35 GB  Healthy
```

Vi ser at disken er delt i to omtrent like store partisjoner, at den første heter C, er systemdisken og har filsystemet NTFS. Den andre heter F og på denne er det ennå ikke lagd noe filsystem.

15.9 Disk controller og DMA

Et multitasking OS vil fortløpende ha behov for å lese data fra mange forskjellige steder på en disk. Siden disken snurrer rundt er det ikke opplagt hva som er den hurtigste måten å lese for eksempel 20 forskjellige slike forespørsler. En forespørsel består typisk av tre tall; [leshode, track, sektornummer]. Å flytte lesehodet til nærmeste neste track som skal leses er en mulighet, å la leshodet flytte seg hele veien fra innerst til ytterst og plukke opp forespøsler underveis er en annen. OS må velge en slik algoritme for å hente inn data. På moderne disker utføres slike algoritmer av mikroprosessoren som sitter på diskens egen disk controller. Tidligere tok OS seg direkte av lesingen av data og administreringen av disken, men det ordner nå disk controlleren.

Før var det også vanlig at OS detaljstyrte skrivingen av data fra disken til internminnet. Dette krevet veldig mange interrupts hver gang det komm inn data fra disken og derfor bruker moderne systemer DMA (Direct Memory Access) som avlaster denne jobben for CPU-en. Dette er vanligvis en egen chip knyttet til systembussen med en egen liten mikroprosessor og tillatelse fra CPU til å skrive direkte til minnet. CPU kan dermed be DMA om en større eller mindre lese eller skriveoperasjoner og DMA vil administrere kopieringen mellom disk og internminnet. Først når alle dataene er på plass sender DMA et interrupt til CPU som forteller at kopieringen er fullført. DMA brukes også for lesning av data fra andre enheter som CD-spiller, USB-devicer etc.

Også for I/O enheter som harddisker som er svært langsomme sammenlignet med internminnet, er det nyttig å bruke cache. I internminnet er det satt av plass til disk-cache, slik at mest mulig av det som er lest i det siste meelomlagres og kan hentes ut mye hurtigere om det kommer en ny forespørsel.

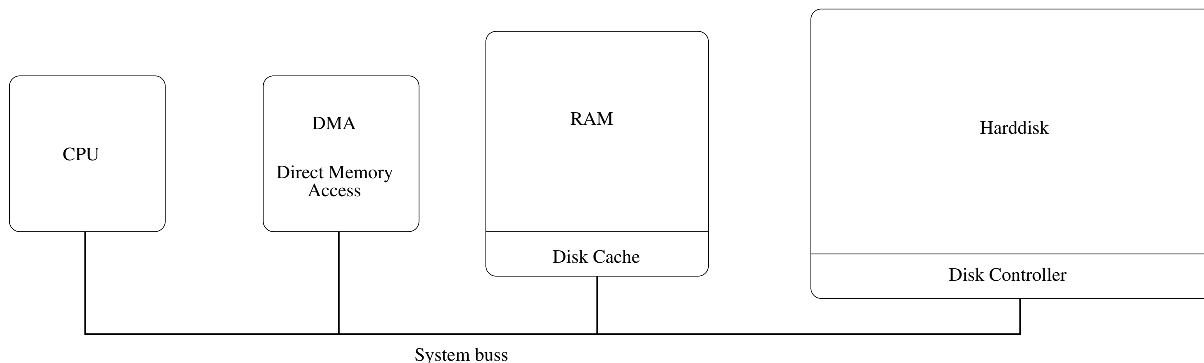


Figure 89: DMA kommuniserer med disk-controlleren og sørger for at det OS ønsker blir kopiert mellom harddisken og internminnet.

15.10 ATA/IDE, SATA og SCSI

Det finnes flere interface-standarer eller grensesnitt-typer for harddisker. De varierer i pris og ytelse. ATA/IDE er billigst og brukes til standard PC'er. SATA er ATA's arvtager, har høyere ytelse og brukes også noe av server. SCSI og SAS er dyrere og brukes av servere som krever høy ytelse.

15.10.1 ATA/IDE

Den tidligste versjonen av denne grensessnitt-standaren ble på slutten av åttitallet kjent som IDE, Integrated Drive Electronics, fordi disk-controlleren ble plassert på selve disken. Etterhvert ble det offisielle navnet på standarden ATA (Advanced Technology Attachment).

- Den billigste teknologien
- Brukes i standard desktop PC'er og laptoper
- Kalles ofte PATA (Parallell ATA) eter at SATA (Serial ATA) ble innført
- Overføringshastigheter opp til 100 MB/s
- Kan ha inntil 2 disker på samme kabel (master og slave)

15.10.2 SATA

- Serial ATA, raskere og bedre enn ATA, men omtrent samme pris
- Introdusert i 2003, har tatt over for ATA
- Må ha SATA-kontroller på hovedkortet eller egen SATA-kontroller
- Overføringshastigheter opp til 300 MB/S (SATA2/SATA-300)
- Bruker samme metode (8B/10B encoding) som ethernet til å sende data
- En disk per kabel

Figure 90: ATA/IDE kabel

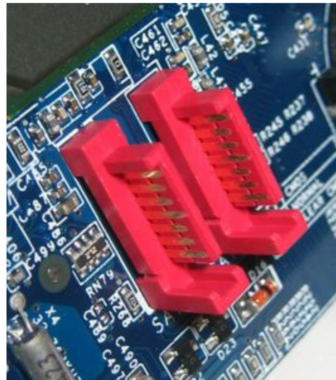


Figure 91: SATA-kontakter på hovedkort

15.10.3 SCSI

- SCSI = Small Computer Systems Interface
- Interface-standard fra 1986 for disker, CD-ROM etc.
- Mer selvstendige disker enn ATA-disker, kan ha mange disker i serie på samme kabel
- Generelt raskere, mer robuste og dyrere enn ATA
- SCSI mest brukt i servere som krever høy disk-ytelse
- Overføringshastigheter opp mot 640 MB/s (Ultra-640 SCSI)
- SAS, Serial Attached SCSI, enda hutigere, bedre og dyrere enn parallell SCSI
- SAS støtter SATA devicer

15.11 KiB, MiB og GiB

Benenvninger som KB og MB er ikke alltid entydge, KB kan bety både $2^{10} = 1024$ bytes og 1000 bytes. Den opprinnelige SI[294]-definisjonen av prefiksene er den helt korrekte og sier at K = 1000, M =

[294] International System of Units, som definerer det metriske systemet

Figure 92: SATA-kabel

Figure 93: SCSI-kabel

1000.000, G = 1000.000.000, etc. Vanlig praksis når det for eksempel gjelder RAM er at 128MB betyr $128 \cdot 1024^2 = 128 \cdot 1048576 = 134 \cdot 10^6$ bytes. Men hvis en harddisk-produsent oppgir at en disk er på 300 GB, betyr det at den er $300 \cdot 10^9 = 279.4 \cdot 1024^3$ bytes og et OS vil da typisk rapportere den som en disk med kapasitet på 279.4 GB. For å ordne opp i dette og flere lignende tilfeller av flertydighet definerte i 1999 International Electrotechnical Commission (IEC) nye binære prefikser kibi-, mebi-, gibi- og tilhørende symboler Ki, Mi, Gi. Disse prefiksene symboliserer potenser av 2 slik at Ki = 2^{10} = 1024, Mi = 2^{20} og Gi = 2^{30}. I 2005 ble dette en IEEE[295]-standard.

Navn	Symbol	Verdi	Eksempel
kilo	K	$10^3 = 1000$	
mega	M	$10^6 = 1000.000$	
giga	G	$10^9 = 1000.000.000$	
tera	T	$10^{12} = 1000.000.000.000$	
kibi	Ki	$2^{10} = 1024$	100 KB = 97.6 KiB
mebi	Mi	$2^{20} = 1.048.576$	100 MB = 95.4 MiB
gibi	Gi	$2^{30} = 1.083.741.824$	100 GB = 93.1 GiB
tebi	Ti	$2^{40} = 1.099.511.627.776$	100 TB = 90.9 TiB

Et disk-eksempel viser at produsenten Seagate bruker SI-benvening og sier at en disk er på 160 GB. På første figur ser man at Linux fdisk bruker samme benevning og 1 GB = 1 milliard bytes.
Derimot viser XP's Disk Managment at disken er på 149.05 GB og bruker altså 1 GB = 1024^3 bytes = 1.08 milliarder bytes.

Partisjoneringsverktøyet GParted som brukes under Ubuntu-installasjon, rapporterer disk-størrelser i GiB, slik at det er helt entydig hva som menes. Men fortsatt er MiB og GiB relativt sjelden i bruk.

[295] Institute of Electrical and Electronics Engineers

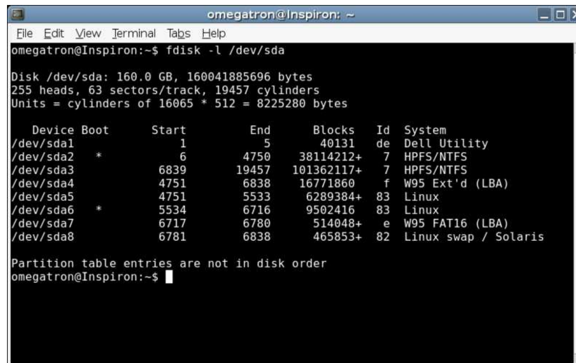


Figure 94: fdisk viser 160 GB

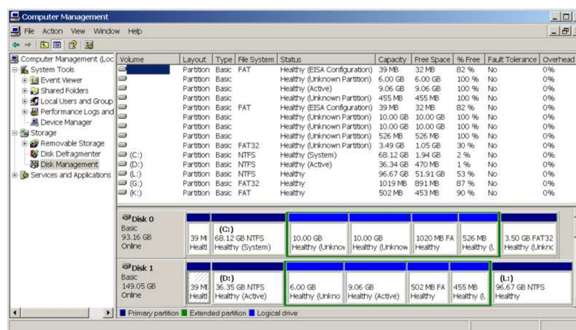


Figure 95: XP viser 149.05 GB

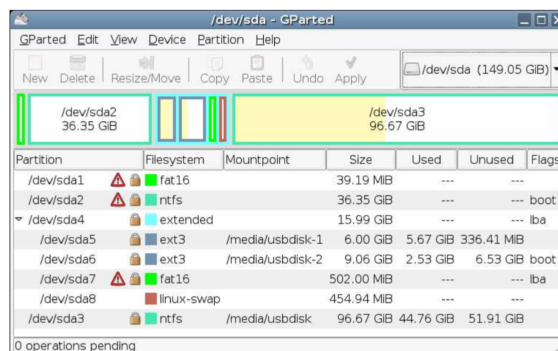


Figure 96: GParted viser at en 160 GB disk er på 149.05 GiB.

15.12 Sammenligning av overføringshastigheter på minne-enheter

enhet	Hastighet (MBit/s)
Serial Infrared (SIR)	0.115
Bluetooth 1.1	0.7
Medium Infrared (MIR)	0.5-1
CD-ROM, 1x	1.2
Bluetooth 2.0	2.1
Fast IR	4
Wireless IEEE 802.11b	5.5-11
10 MBit Ethernet	10
DVD-ROM, 1x	11.1
USB 1.0	12
Bluetooth 4.0	25
Bluetooth 5	50
Wireless IEEE 802.11g	54
CD-ROM, 52x	62.4
100 MBit Ethernet	100
Wireless IEEE 802.11n	150
DVD-ROM, 16x	177.3
FireWire IEEE 1394 400	400
Blu-ray Disk 12x	432
USB 2.0	480
Wireless IEEE 802.11ac	500
FireWire 800	800
Gigabit Ethernet	1,000
PATA 133	1,064
SATA2 300	2,400
Ultra-320 SCSI	2,560
FireWire 3,200	3,200
SATA3	4,800
USB 3.0	5,000
Ultra-640 SCSI	5,120
Wireless IEEE 802.11ad	6,750
10 Gigabit Ethernet	10,000
USB 3.1	10,000
Thunderbolt 1	10,000
SAS 3	12,000
SATA 3.2	16,000
Thunderbolt 1	20,000
Thunderbolt 3	40,000
40 Gigabit Ethernet	40,000
100 Gigabit Ethernet	100,000
InfiniBand	100,000

Til sammenligning noen typiske tall for krav til overføringshastigheter for film i forskjellige kvaliteter:

hastighet	kvalitet
2-3 Mbit/s	VHS
8-12 Mbit/s	DVD
36 Mbit/s	Blueray
4-25 Mbit/s	HD-TV/4K

I motsetning til hva som er vanlig i mange andre sammenhenger, betyr her prefikset M en million. Altså

betyr Mbit/s ikke 2^{20}bit/s = 1.048.576 bit/s. Bruken av slike prefiks er diskutert i forrige avsnitt.

15.12.1 Sammenligning av disker

Her er et eksempel på hver av disktypene med priser hentet fra komplett.no. Men husk at ytelsestallene er hentet fra produsentene.

Type	Kapasitet	hastighet	Søketid	Rotasjon	Buffer	Produsent	pris
SATA-600	1 TB	600 MBps	8.5	7200 rpm	64MB	Seagate	528
SAS	2 TB	1200 MBps	4.16	7200 rpm	128 MB	Seagate	1,675
SSD	120 GB	600 MBps				Kingston	578
SSD	240 GB	600 MBps				Corsair	1,049
SSD	2TB GB	600 MBps			2 GB	Samsung	6,699

I 2012 så det slik ut:

Type	Kapasitet	hastighet	Søketid	Rotasjon	Buffer	Produsent	pris
SATA-600	1 TB	600 MBps		7200 rpm	32MB	Seagate	795
SAS	1 TB	600 MBps		7200 rpm	64 MB	Seagate	1.395
SAS	600 MB	600 MBps		15000 rpm	16 MB	Seagate	3.695
SSD	60 GB	300 MBps				Corsair	799
SSD	240 GB	600 MBps				Corsair	1,999

Og for enda et par år tidligere så det slik ut:

Type	Kapasitet	hastighet	Søketid	Rotasjon	Buffer	Produsent	pris
ATA-133	320 GB	133 MBps	8.5 ms	7200 rpm	8 MB	Hitachi	795
SATA-300	320 GB	300 MBps	8.5 ms	7200 rpm	16 MB	Hitachi	795
Ultra320 SCSI	300 GB	320 MBps	4.7 ms	10000 rpm	8 MB	Seagate	5.750
SAS	300 GB	300 MBps	3.5 ms	15000 rpm	16 MB	Seagate	8.995

15.13 RAID

Mens utviklingen av hastigheten til prosessorer, cache og delvis RAM har gått raskt, har utviklingen av hastigheten til harddisker vært langsom. En måte å forbedre ytelsen på når det begynner å bli vanskelig å få hver enklet enhet til å gå raskere, er å bruke flere enheter i parallell. Utviklingen av multicore prosessorer er et eksempel på dette. For disker kalles teknologien som gjør dette RAID (Redundant Array of Independent Disks). Man bruker da flere like disker til å øke hastigheten man kan hente data og man kan også bruke et slikt oppsett til redundans; dobbel lagring av data slik at man ikke mister data om en disk blir ødelagt.

RAID 0 Minst to disker. Striper diskene. Ingen redundans. Hurtigere å lese.

RAID 1 Minst to disker. Dupliserer dataene. Hurtigere å lese. Kan fortsatt lese alt om en disk ryker.

RAID 3 Minst tre disker. Parallell aksess, veldig små striper, 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 disker. Paritet lagres på en ekstra disk. Store striper, 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 disker. Paritet lagres fordelt på diskene. Store striper, sektor eller blocks. Om en disk ryker kan informasjonen hentes ut fra de som er igjen.

RAID kan implementeres i software, det vil si av OS, eller i hardware ved en dedikert RAID-controller på hovedkortet. RAID 0, 1 og 5 er implementert i Windows 2003 server.

15.13.1 Ytelse

Ved såkalt striping av diskene økes ytelsen ved både lesing og skriving av filer. Dette fordi en stor fil deles i striper som fordeles på diskene. Innholdet av en stor fil vil være fordelt på alle diskene og data kan både leses fra og skrives til diskene i parallell og da går det mye raskere.

15.13.2 Paritet

Hvis man har en samling bits (for eksempel en byte, 8 bit) som har verdi en eller null, kan man telle antall enere. Hvis antall enere er like (0, 2, 4, ..., har samlingen av bits like paritet. Hvis antall enere er odde (1, 3, 5, ...), sier vi at samlingen av bits har odde paritet. Dette kan brukes til en meget enkel feilsjekking, hvis man sender et antall bit over et nettverk og pariteten har endret seg på veien, kan man konkludere med at minst ett bit må ha endret verdi. I et RAID kan man ta et bit fra hver data-disk i RAID'et, regne ut pariteten og skrive den til en egen paritetsdisk. Hvis en av diskene i RAID'et crasher og alle dataene for denne disken går tapt, kan man bruke dataene på paritetsdisken for å finne ut verdien på de tapte bit'ene. Hvis pariteten for de igjenværende diskene er den samme som den lagrede pariteten, har en null gått tapt. Hvis pariteten for de igjenværende diskene er forskjellig fra den lagrede pariteten, har en ener gått tapt. Dermed kan hver bit på den tapte disken gjenskapes. I RAID 3 er stripene små, ned til en byte. I RAID 4 og 5 er stripene et antall sektorer. I begge tilfeller er prinsippet for redundans det samme. Bergningen av paritet må gjøres når det skrives til diskene. Det kan gjøres hurtig, fordi XOR-porter kan regne ut paritet i paralell for 32 eller 64 bit av gangen, for henholdsvis 32 og 64 bits prosessorer. Det finnes også hardware-RAID, egne enheter som gjør disse paritetsberegningene og styrer RAID'et uavhengig av prosessoren.

15.13.3 Eksempel på paritetsberegning

Anta at vi i et RAID 3 striper disken bit for bit og bruker 5 disker. Disken med paritet lagrer da den samlede pariteten for bit'ene for de 4 andre diskene; en ener om antallet er odde og en null om antallet er like:

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

Hvis nå for eksempel den 2. disken crasher og alle dataene fra den blir borte, vil RAID'et se slik ut:

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

Hvordan kan man nå trylle frem igjen dataene på den ødelagte disk2 og legge dem inn på en ny disk? Vi ser da at dataene fra denne disken kan trylles frem igjen ved å bruke paritetsdisken og reglene nevnt over: Hvis pariteten for de igjenværende diskene er den samme som den lagrede pariteten, har en null gått tapt. Hvis pariteten for de igjenværende diskene er forskjellig fra den lagrede pariteten, har en ener gått tapt. Prøv selv!

16 Forelesning 14/5-24(2 timer). Prøveeksamen 2022

Prøveeksamen er identisk med konte-eksamen fra høsten 2021, men den praktiske delen med innlogging er noe endret for å tilpasses årets eksamen, våren 2022.

Prøveeksamen 2022[296]

Løsningsforslag prøveeksamen 2022[297] Løsningsforslaget er til konte-eksamen høsten 2021, merk at det er litt endringer i den praktiske delen.

16.1 Forelesningsvideoer

Uredigert opptak av hele forelesningen med gjennomgang av deler av prøveeksamen (1:42:16)[298]

17 Forelesning 14/5-24(2 timer). Prøveeksamen 2023

Dere kan logge inn på Guacamole[299] og velge os22k og dere vil kunne få opp samme filsystem som ble brukt under eksamen, slik at dere kan gjøre de praktiske oppgavene.

Prøveeksamen er identisk med konte-eksamen fra høsten 2022. Prøveeksamen 2023[300]

Løsningsforslag prøveeksamen 2023[301] Løsningsforslaget er til konte-eksamen høsten 2022.

17.1 Forelesningsvideoer

Uredigert opptak av hele forelesningen med gjennomgang av deler av prøveeksamen [302]

18 Forelesning 21/5-24(2 timer). Prøveeksamen 2024

Dere kan logge inn på Guacamole[303] og velge os23k og dere vil kunne få opp samme filsystem som ble brukt under eksamen, slik at dere kan gjøre de praktiske oppgavene.

Prøveeksamen er identisk med konte-eksamen fra høsten 2023. Prøveeksamen 2024[304]

Løsningsforslag prøveeksamen 2024[305] Løsningsforslaget er til konte-eksamen høsten 2023, svarene i den praktiske delen er ikke 100% det samme som i online prøven, men metoden å løse oppgavene på er riktige.

[296] https://os.cs.oslomet.no/os/eksamen/peksamen2022.pdf
[297] https://os.cs.oslomet.no/os/eksamen/fasitH2021.pdf
[298] https://os.cs.oslomet.no/os/Forelesning/video/2022/pex2022.mp4
[299] https://guacamole.cs.oslomet.no/guacamole
[300] https://os.cs.oslomet.no/os/eksamen/eksamenH2022.pdf
[301] https://os.cs.oslomet.no/os/eksamen/fasitH2022.pdf
[302] https://os.cs.oslomet.no/os/pex23.mp4
[303] https://guacamole.cs.oslomet.no/guacamole
[304] https://os.cs.oslomet.no/os/eksamen/eksamenH2023.pdf
[305] https://os.cs.oslomet.no/os/eksamen/fasitH2023.pdf

18.1 Forelesningsvideoer

Uredigert opptak av hele forelesningen med gjennomgang av prøveeksamen[306]

19 Forelesning 7/5-24(2 timer). Prøveeksamen 2025 (og 20 og 21)

19.1 Prøveeksamen 2025

Prøveeksamen er identisk med konte-eksamen fra høsten 2024.

Prøveeksamen 2025[307]

Løsningsforslag prøveeksamen 2025[308]

19.1.1 Forelesningsvideoer

Uredigert opptak av gjennomgang av prøveeksamen (1:47:37)[309]

19.2 Prøveeksamen 2020

Prøveeksamen er identisk med konte-eksamen fra høsten 2019.

Prøveeksamen 2020[310]

Løsningsforslag prøveeksamen 2020[311]

19.2.1 Forelesningsvideoer

Uredigert opptak av hele forelesningen (01:30:29)[312]

Opptak av forelesningen inndelt etter temaer:

ekH19del1.mp4[313] (00:44) Oppgave 1 Datamaskinarkitektur
ekH19del2.mp4[314] (01:39) Oppgave 2 Datamaskinarkitektur
ekH19del3.mp4[315] (03:27) Oppgave 3 Datamaskinarkitektur
ekH19del4.mp4[316] (02:13) Oppgave 4 Datamaskinarkitektur

[306] https://os.cs.oslomet.no/os/proveEx24.mp4
[307] https://os.cs.oslomet.no/os/eksamen/eksamenH2024.pdf
[308] https://os.cs.oslomet.no/os/eksamen/fasitH2024.pdf
[309] https://os.cs.oslomet.no/os/Forelesning/video/ekH24.mp4
[310] https://os.cs.oslomet.no/os/eksamen/eksamenH2019.pdf
[311] https://os.cs.oslomet.no/os/eksamen/fasitH2019.pdf
[312] https://os.cs.oslomet.no/os/Forelesning/video/ekH19.mp4
[313] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del1.mp4
[314] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del2.mp4
[315] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del3.mp4
[316] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del4.mp4

ekH19del5.mp4[317] (01:39) Oppgave 5 Prosesser
ekH19del6.mp4[318] (02:09) Oppgave 6 Prosesser
ekH19del7.mp4[319] (02:01) Oppgave 7 Prosesser
ekH19del8.mp4[320] (03:12) Oppgave 8 Prosesser
ekH19del9.mp4[321] (01:38) Oppgave 9 Synkronisering
ekH19del10.mp4[322] (01:54) Oppgave 10 Synkronisering
ekH19del11.mp4[323] (02:14) Oppgave 11 Synkronisering
ekH19del12.mp4[324] (03:29) Oppgave 12 Synkronisering
ekH19del13.mp4[325] (07:15) Oppgave 13 Synkronisering
ekH19del14.mp4[326] (03:47) Oppgave 14 Synkronisering
ekH19del15.mp4[327] (02:40) Oppgave 15-17 Linux kommandolinje
ekH19del16.mp4[328] (01:18) Oppgave 18 Linux kommandolinje
ekH19del17.mp4[329] (04:10) Oppgave 19 Linux kommandolinje
ekH19del18.mp4[330] (08:42) Oppgave 20 Bash-scripting
ekH19del19.mp4[331] (02:02) Oppgave 21 PowerShell
ekH19del20.mp4[332] (05:57) Oppgave 22 PowerShell-script
ekH19del21.mp4[333] (02:01) Oppgave 23 Interminne
ekH19del22.mp4[334] (01:41) Oppgave 24 og 25 Interminne
ekH19del23.mp4[335] (02:53) Spørsmål om segmenter til oppgave 24
ekH19del24.mp4[336] (07:42) Spørsmål om praktiske problemer ved Inspera-eksamen hjemme

19.3 Prøveeksamen 2021

Prøveeksamen er identisk med konte-eksamen fra høsten 2020, bortsett fra de 4 siste oppgavene som kom i tillegg.

Prøveeksamen 2021[337]

Løsningsforslag prøveeksamen 2021[338]

19.3.1 Forelesningsvideoer

Uredigert opptak av første del av forelesningen (0:42:16)[339]

[317] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del5.mp4
[318] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del6.mp4
[319] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del7.mp4
[320] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del8.mp4
[321] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del9.mp4
[322] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del10.mp4
[323] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del11.mp4
[324] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del12.mp4
[325] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del13.mp4
[326] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del14.mp4
[327] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del15.mp4
[328] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del16.mp4
[329] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del17.mp4
[330] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del18.mp4
[331] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del19.mp4
[332] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del20.mp4
[333] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del21.mp4
[334] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del22.mp4
[335] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del23.mp4
[336] https://os.cs.oslomet.no/os/Forelesning/video/ekH19del24.mp4
[337] https://os.cs.oslomet.no/os/eksamen/eksamenH2020.pdf
[338] https://os.cs.oslomet.no/os/eksamen/fasitH2020.pdf
[339] https://os.cs.oslomet.no/os/Forelesning/video/2021/pex21del1.mp4

Uredigert opptak av andre del av forelesningen (1:50:53)[340]

[340] https://os.cs.oslomet.no/os/Forelesning/video/2021/pex21del2.mp4