

# Operating Systems

*Lecture Notes, Theory Questions and Lab Exercises*

Erik Hjelmås

January 7, 2014



---

## About this Compendium

This is NOT A TEXTBOOK. This is just a collection of lecture notes together with weekly exercises and labs. The text and figures in English are the slides I will show in the lectures, while the text in between in Norwegian are my notes on what I have to remember to cover during that lecture. Almost all the figures are taken directly from the textbook resources:

[http://www.cs.vu.nl/~ast/books/book\\_software.html](http://www.cs.vu.nl/~ast/books/book_software.html)

<http://www.pearsonhighered.com/tanenbaum/>

The textbook for this course is the latest addition of Tanenbaum's "Modern Operating Systems". All credits to Andrew S. Tanenbaum for writing such great books. Some of the content in this compendium is also inspired by two other great books on operating systems: William Stallings' "Operating Systems" and Abraham Silberschatz, Peter B. Galvin and Greg Gagne's "Operating System Concepts".

You should spend time reading the textbook, but you can spend most of your time on the parts of the textbook which are covered in this compendium since those parts are the ones most relevant for the exam and your learning outcome.

Much of the material in this compendium is inspired by Vegar Johansen, Hårek Haugerud and Dag F. Langmyhr.



# Contents

About this Compendium . . . . .	i
<b>1 Om emnet</b>	<b>1</b>
1.1 About the course . . . . .	1
1.2 Outcome . . . . .	2
1.3 Introduction . . . . .	2
1.3.1 What's an OS? . . . . .	3
1.4 History . . . . .	5
1.4.1 1.-4. Generation . . . . .	5
1.4.2 Windows/Unix/Mac . . . . .	8
1.5 Closing Remarks . . . . .	10
1.6 Theory questions . . . . .	11
1.7 Lab exercises . . . . .	12
<b>2 Datamaskinarkitektur</b>	<b>15</b>
2.1 Outcome . . . . .	15
2.2 ISA . . . . .	16
2.2.1 Registers . . . . .	16
2.3 Assembly . . . . .	17
2.3.1 gcc . . . . .	18
2.3.2 32 vs 64 bit . . . . .	19
2.3.3 Syntax . . . . .	19
2.3.4 Stack . . . . .	21
2.4 Execution . . . . .	22

2.5	Protection . . . . .	24
2.6	Multicore/ Hyperthreading . . . . .	24
2.7	Cache . . . . .	25
2.7.1	Set Associative Cache . . . . .	27
2.7.2	Write Policy . . . . .	28
2.8	I/O . . . . .	29
2.9	Interrupts . . . . .	30
2.9.1	DMA . . . . .	31
2.10	Architecture . . . . .	32
2.11	Booting . . . . .	33
2.12	Theory questions . . . . .	34
2.13	Lab exercises . . . . .	35
<b>3</b>	<b>Introduksjon til operativsystemer</b>	<b>37</b>
3.1	Outcome . . . . .	37
3.2	OS Zoo . . . . .	37
3.3	Concepts . . . . .	38
3.3.1	Process . . . . .	38
3.3.2	File . . . . .	39
3.3.3	Pipe . . . . .	41
3.4	System Calls . . . . .	41
3.4.1	POSIX . . . . .	43
3.4.2	WinAPI . . . . .	45
3.5	OS Structure . . . . .	46
3.5.1	Monolithic . . . . .	46
3.5.2	Layered . . . . .	48
3.5.3	Microkernel . . . . .	48
3.5.4	Client-server . . . . .	49
3.5.5	Virtual machines . . . . .	49
3.6	Theory questions . . . . .	51
3.7	Lab exercises . . . . .	52

<b>4 Prosesser og tråder</b>	<b>53</b>
4.1 Outcome . . . . .	53
4.2 Processes . . . . .	53
4.2.1 Model . . . . .	53
4.2.2 Implementation . . . . .	56
4.3 Threads . . . . .	59
4.3.1 Usage . . . . .	59
4.3.2 Model . . . . .	61
4.3.3 POSIX . . . . .	62
4.3.4 User-level vs Kernel-level . . . . .	63
4.4 Theory questions . . . . .	66
4.5 Lab exercises . . . . .	67
<b>5 Prosesskommunikasjon, samtidighet og synkronisering</b>	<b>71</b>
5.1 Outcome . . . . .	71
5.2 Introduction . . . . .	71
5.2.1 Atomicity . . . . .	72
5.2.2 Deadlock . . . . .	72
5.2.3 Starvation . . . . .	72
5.2.4 Race Conditions . . . . .	72
5.2.5 Critical Region/ Mutual Exclusion . . . . .	74
5.3 "Theory Solutions" . . . . .	75
5.3.1 Non-Solutions . . . . .	75
5.3.2 Peterson's . . . . .	77
5.4 "Practical Solutions" . . . . .	79
5.4.1 HW: TSL . . . . .	79
5.4.2 HW: XCHG . . . . .	80
5.4.3 Busy Waiting Problem . . . . .	80
5.4.4 "Sleep and Wakeup" . . . . .	81
5.4.5 OS: Semaphores . . . . .	82
5.4.6 OS: Mutex . . . . .	84

5.4.7	OS: Condition Variable . . . . .	85
5.4.8	ProgLang: Monitor . . . . .	86
5.5	Theory questions . . . . .	89
5.6	Lab exercises . . . . .	91
<b>6</b>	<b>Scheduling</b>	<b>93</b>
6.1	Outcome . . . . .	93
6.2	Introduction . . . . .	93
6.3	Batch Systems . . . . .	96
6.3.1	FCFS (FIFO) . . . . .	96
6.3.2	SJF/SPN . . . . .	98
6.3.3	SRTN . . . . .	99
6.4	Interactive Systems . . . . .	100
6.4.1	RR . . . . .	100
6.4.2	Priority . . . . .	102
6.4.3	SPN with Aging . . . . .	103
6.4.4	Guaranteed . . . . .	104
6.4.5	Lottery . . . . .	104
6.4.6	Fair-Share . . . . .	104
6.5	Real-Time Systems . . . . .	104
6.6	Thread Scheduling . . . . .	105
6.7	Multiproc . . . . .	105
6.7.1	Affinity . . . . .	106
6.7.2	Gang-/Co-scheduling . . . . .	106
6.8	OS Implementation . . . . .	108
6.9	Theory questions . . . . .	109
6.10	Lab exercises . . . . .	110

<b>7 Virtuelt minne, paging og segmentering</b>	<b>111</b>
<b>7.1 Introduction</b>	<b>111</b>
<b>7.1.1 Relocation</b>	<b>112</b>
<b>7.1.2 Swapping</b>	<b>113</b>
<b>7.1.3 Free space</b>	<b>114</b>
<b>7.2 Virtual Mem.</b>	<b>115</b>
<b>7.2.1 Paging</b>	<b>116</b>
<b>7.2.2 TLB</b>	<b>118</b>
<b>7.2.3 Multilevel PT</b>	<b>119</b>
<b>7.2.4 Inverted PT</b>	<b>120</b>
<b>7.3 Segmentation</b>	<b>121</b>
<b>7.4 MULTICS</b>	<b>123</b>
<b>7.5 Theory questions</b>	<b>125</b>
<b>7.6 Lab exercises</b>	<b>126</b>
<b>8 Page replacement algoritmer, design og implementering</b>	<b>127</b>
<b>8.1 Page Replace</b>	<b>127</b>
<b>8.1.1 Optimal</b>	<b>127</b>
<b>8.1.2 FIFO-based</b>	<b>127</b>
<b>8.1.3 LRU-based</b>	<b>129</b>
<b>8.1.4 Working set-based</b>	<b>132</b>
<b>8.2 Design Issues</b>	<b>134</b>
<b>8.2.1 Page faults</b>	<b>135</b>
<b>8.2.2 Sharing</b>	<b>136</b>
<b>8.3 Implementation</b>	<b>137</b>
<b>8.3.1 OS involvement</b>	<b>137</b>
<b>8.3.2 PF handling</b>	<b>137</b>
<b>8.4 Summary</b>	<b>138</b>
<b>8.5 Theory questions</b>	<b>139</b>
<b>8.6 Lab exercises</b>	<b>140</b>

<b>9 Filsystemimplementasjon, EXTFS</b>	<b>143</b>
9.1 Files & Dirs . . . . .	143
9.1.1 Attributes . . . . .	144
9.1.2 Operations . . . . .	145
9.2 File System Implementation . . . . .	148
9.2.1 Layout . . . . .	148
9.2.2 Allocation . . . . .	149
9.2.3 Ext2fs . . . . .	151
9.2.4 Links . . . . .	157
9.2.5 LFS . . . . .	158
9.2.6 JFS . . . . .	159
9.2.7 VFS . . . . .	160
9.3 Theory questions . . . . .	162
9.4 Lab exercises . . . . .	163
<b>10 Filsystemhåndtering og ytelse, FAT og NTFS</b>	<b>165</b>
10.1 File System Management and Performance . . . . .	165
10.1.1 Blocksize . . . . .	166
10.1.2 Consistency . . . . .	167
10.1.3 Caching . . . . .	168
10.1.4 Cyl. Groups . . . . .	168
10.2 FAT12/16/32 . . . . .	169
10.2.1 Dir. entry . . . . .	170
10.2.2 The FAT . . . . .	170
10.3 NTFS . . . . .	172
10.3.1 MFT . . . . .	173
10.3.2 Records . . . . .	173
10.4 Comparison . . . . .	176
10.5 Theory questions . . . . .	177
10.6 Lab exercises . . . . .	178

<b>11 I/O</b>	<b>179</b>
11.1 I/O Hardware . . . . .	179
11.1.1 Isolated vs Memory-mapped . . . . .	180
11.1.2 DMA . . . . .	180
11.1.3 Precise vs Imprecise Interrupts . . . . .	181
11.2 I/O Software . . . . .	182
11.2.1 Three ways . . . . .	182
11.3 I/O Software Layers . . . . .	184
11.4 Clocks/Timers . . . . .	184
11.5 Disks . . . . .	185
11.5.1 HDD . . . . .	185
11.5.2 SSD . . . . .	187
11.5.3 RAID . . . . .	191
11.6 Theory questions . . . . .	192
11.7 Lab exercises . . . . .	193
<b>12 Deadlock</b>	<b>195</b>
12.1 Intro . . . . .	195
12.1.1 Resources . . . . .	195
12.1.2 Semaphores . . . . .	196
12.1.3 Deadlock def . . . . .	196
12.1.4 Four conditions . . . . .	197
12.1.5 Modelling . . . . .	197
12.2 Dealing with Deadlocks . . . . .	198
12.2.1 Ostrich alg . . . . .	199
12.2.2 Detect/Recover . . . . .	199
12.2.3 Avoidance . . . . .	200
12.2.4 Prevention . . . . .	202
12.3 Theory questions . . . . .	206
12.4 Lab exercises . . . . .	207

<b>13 Virtualisering</b>	<b>209</b>
13.1 Introduction . . . . .	209
13.1.1 Requirements . . . . .	210
13.2 Hypervisors . . . . .	210
13.3 CPU . . . . .	211
13.3.1 Binary translation . . . . .	211
13.3.2 Paravirtualization . . . . .	212
13.3.3 HW virtualization . . . . .	213
13.4 Memory . . . . .	214
13.4.1 Shadow page tables . . . . .	217
13.4.2 Nested page tables . . . . .	218
13.5 I/O . . . . .	220
13.5.1 IOMMU . . . . .	221
13.6 Theory questions . . . . .	223
13.7 Lab exercises . . . . .	224
<b>14 Objektsikkerhet</b>	<b>225</b>
14.1 Introduction . . . . .	225
14.2 Protection Mechanisms . . . . .	229
14.2.1 Protection domain . . . . .	229
14.2.2 Protection matrix . . . . .	229
14.2.3 ACL . . . . .	230
14.2.4 Capabilities . . . . .	231
14.2.5 Reference monitor . . . . .	232
14.2.6 Authorized States . . . . .	233
14.2.7 Multilevel security . . . . .	234
14.2.8 Covert channels . . . . .	235
14.3 Windows . . . . .	236
14.3.1 Fundamentals . . . . .	236
14.3.2 System calls . . . . .	237
14.3.3 Implementation . . . . .	238

14.4 Linux . . . . .	240
14.4.1 Fundamentals . . . . .	240
14.4.2 System calls . . . . .	240
14.4.3 Implementation . . . . .	241
14.5 Theory questions . . . . .	242
14.6 Lab exercises . . . . .	243
<b>15 Malware og minnesikkerhet</b>	<b>245</b>
15.1 Insider Attacks . . . . .	245
15.2 Exploiting Code Bugs . . . . .	246
15.2.1 Buffer Overflow . . . . .	247
15.2.2 Format Strings . . . . .	248
15.2.3 Return to Libc . . . . .	248
15.2.4 Integer Overflow . . . . .	250
15.2.5 Code Injection . . . . .	250
15.2.6 Privilege Escalation . . . . .	251
15.3 Malware . . . . .	251
15.3.1 Trojan Horses . . . . .	252
15.3.2 Viruses . . . . .	253
15.3.3 Worms . . . . .	254
15.3.4 Spyware . . . . .	255
15.3.5 Rootkits . . . . .	256
15.4 Defenses . . . . .	257
15.4.1 Firewalls . . . . .	257
15.4.2 Antivirus . . . . .	258
15.4.3 Code Signing . . . . .	259
15.4.4 Jailing . . . . .	260
15.4.5 Host-based IDS . . . . .	260
15.4.6 Encapsulating Mobile Code . . . . .	261
15.5 Human in the Loop . . . . .	262
15.6 Theory questions . . . . .	263
15.7 Lab exercises . . . . .	264

<b>A Bash</b>	<b>265</b>
A.1 Variables . . . . .	267
A.1.1 Arrays . . . . .	268
A.1.2 Structures/Classes . . . . .	269
A.1.3 Command-line args . . . . .	269
A.2 Input . . . . .	269
A.2.1 Input . . . . .	269
A.2.2 System commands . . . . .	270
A.3 Conditions . . . . .	270
A.3.1 if/else . . . . .	270
A.3.2 Operators . . . . .	271
A.3.3 Switch/case . . . . .	273
A.4 Iteration . . . . .	274
A.4.1 For . . . . .	274
A.4.2 While . . . . .	274
A.5 Math . . . . .	276
A.6 Functions . . . . .	277
A.7 RegExp . . . . .	277
A.7.1 Bash example . . . . .	279
A.8 Bash only . . . . .	279
A.9 Credits . . . . .	280
 <b>B PowerShell</b>	 <b>281</b>
B.1 Variables . . . . .	284
B.1.1 Arrays . . . . .	285
B.1.2 Structures/Classes . . . . .	285
B.1.3 Command-line args . . . . .	286
B.2 Input . . . . .	287
B.2.1 Input . . . . .	287
B.2.2 System commands . . . . .	288
B.3 Conditions . . . . .	288

B.3.1 if/else . . . . .	288
B.3.2 Operators . . . . .	288
B.3.3 Switch/case . . . . .	290
B.3.4 Where . . . . .	290
B.4 Iteration . . . . .	291
B.4.1 For . . . . .	291
B.4.2 While . . . . .	291
B.4.3 Foreach . . . . .	292
B.5 Math . . . . .	293
B.6 Functions . . . . .	294
B.7 RegExp . . . . .	294
B.7.1 PowerShell example . . . . .	295
B.8 PowerShell only . . . . .	296
B.9 Credits . . . . .	296



# Chapter 1

## Om emnet

### 1.1 About the course

#### Teacher and Teaching assistants

- Teacher is Erik Hjelmås  
[www.hig.no/~erikh](http://www.hig.no/~erikh)
- Lectures and Lab
- Web page: Fronter/It's Learning/Public web page
- Erik's office is K210 (Gjøvik :)
- Erik's email is erikh@hig.no
- Erik's phone/sms is 93034446
- *Teaching assistants*

#### Workload, requirements and evaluation

- 10 credits ~ 15 hrs per week (~ 300 hrs total),  
I recommend
  - 4 hrs lecture
  - 5 hrs lab
  - 2 hrs theory questions
  - 4 hrs textbook/articles/wiki\*
- Evaluation based on:

- Final exam
- Lectures, lab and theory questions
- *Mandatory work!* see web page

### **Howto Learn Operating Systems**

1. Book and theory
2. *Theory and Lab*
3. Lab: write an operating system

Vi bruker metode 2. Det viktigste er prinsippene og forståelsen for rollen operativsystemer spiller, og vi tror vi får best læring iif tidsbruk ved hjelp av mye praktiske oppgaver sammen med teorien.

Som Lab benytter vi forskjellig OS kommandoer og hjelpeapplikasjoner for å studere hva som skjer, sammen med C-programmering for dypere forståelse av enkelttemaer (vi oppnår mest læring og forståelse når vi faktisk utvikler selv det samme som vi studerer). Vi skal også innarbeide ferdigheter i bruk og drift av operativsystemer via Bash/UNIX og PowerShell/Windows.

Men skal man virkelig grave seg ned i operativsystemer, så er metode tre sterkt anbefalt (benyttes ved UiT og UiO), men farene er at man bruker for mye tid på 'knouting' og for mye tid på det helt grunnleggende (dvs man rekker bare bygge et helt enkelt system, veldig annerledes fra dagens store OSer som Windows og Linux).

## **1.2 Outcome**

### **Today's Learning Outcome**

- Layering/Abstraction
- Conceptual framework

## **1.3 Introduction**

Operativsystemer er et emne som vil være litt plagsomt for alle "som bare vil at datamaskiner skal virke". Dvs hele dette emnet dreier seg om å lære seg hvordan vi får en datamaskin til å virke best mulig, å kikke under den behagelig overflaten vi forholder oss vanligvis til. La oss innføre de begrepene som dominerer operativsystemer:

Abstraksjon:	PROSESS	ADRESSEROM	FIL
<hr/>			
Fysisk enhet:	CPU	RAM	DISK

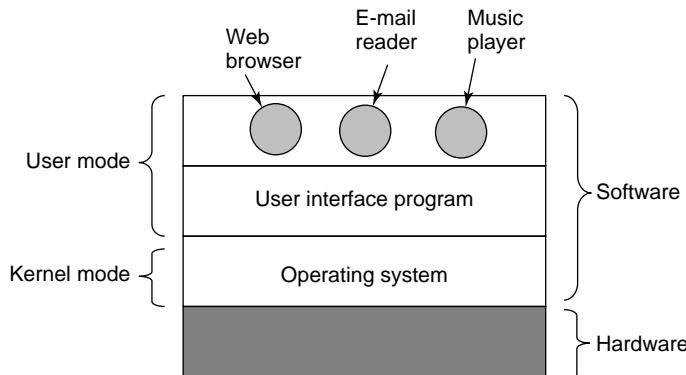
De klassiske temaene som alltid har vært en del av operativsystemer er

- Prosesshåndtering (Process management)
- Minnehåndtering (Memory management)
- Input/Output (I/O)
- Filsystemer

og vi skal i tillegg ta for oss påbygningstemaene Virtualisering og Sikkerhet.

### 1.3.1 What's an OS?

#### Where's the OS?

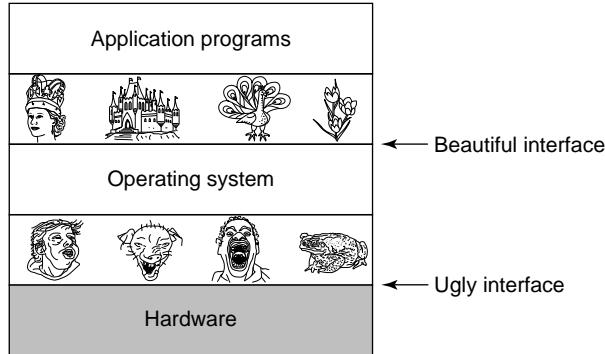


TAVLE:

- *kernel mode* og *user mode*
- skifte mellom disse kalles *mode switch/mode transition*

I all hovedsak kjøres operativsystemet i kernel mode, og mens brukerprogrammene kjøres i user mode. I kernel mode tillates alle instruksjoner (dvs alle assemblyinstruksjonene som datamaskinarkitekturen tilbyr) mens i user mode tillates ikke instruksjoner som har med direkte kontroll av maskinen eller instruksjoner som utfører I/O.

## The OS Hides the Ugly Details



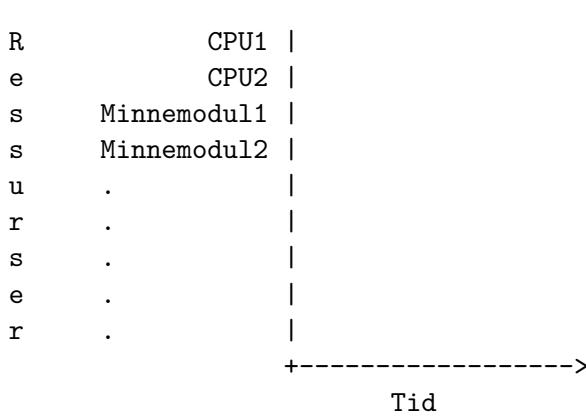
Så hva er det et OS gjør? Det gjør i all hovedsak to hovedoppgaver (TAVLE):

- Gir brukeren et ryddig grensesnitt mot maskinvaren.
- Administrerer ressurser, dvs sørge for at mange prosesser kan kjøres ”samtidig”, dette kalles *Multitasking* og kan sies å bestå av:
  - multiplexing i tid.
  - multiplexing i rom.

Tildeling i tid for prosesser P1, P2, P3, eks. en CPU:

CPU1 |P1--|P3|P2-----|P3--|P1----->  
            Tid

Tildeling i tid og rom:



Husk parallel til deling i Tid og Frekvens i datakommunikasjon.

Alle moderne operativsystemer benytter *Preemptive Multitasking* for å løse dette, dvs hver prosess for en liten tidsluke å kjøre på CPUn før prosessen avbrytes ("Preemptes") og en annen prosess får kjøre.

La oss se mer på grensesnittet. En typisk oppgave i en datamaskin er å lese inn noe data fra en disk som involverer gjerne flytting av diskarmen, formatinformasjon om disken, diskblokkaddresse, osv. Men siden vi har et OS å prate med behøver vi ikke bekymre oss for dette, modellen blir slik istedet (TAVLE):

```
read(fd, buffer, nBytes)
  |
  V
Operativsystemet
  |
  V
Driver
  |
  V
Diskkontroller (hardware)
```

**KEY PRINCIPLE!** Tanenbaum, page 4:

Abstraction is the key to managing complexity. Good abstractions turn a nearly impossible task into manageable ones. The first one of these is defining and implementing abstractions. The second one is using these abstractions to solve the problem at hand.

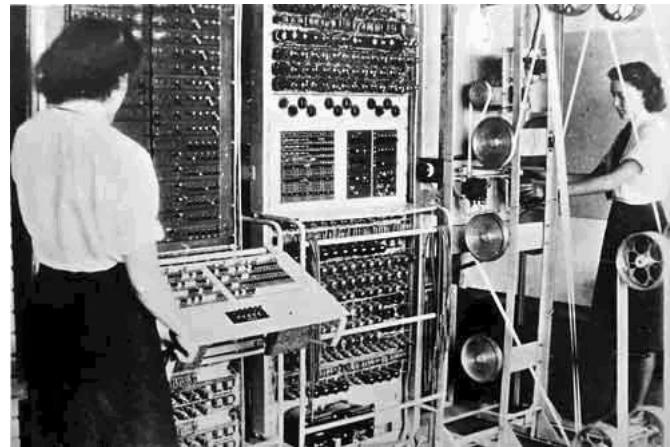
Eksempelet vi nettopp så: en fil. Hvis vi var ingeniører som jobbet med konstruksjon av en disk så hadde vi primært brydd oss om hvordan disken faktisk ser ut fysisk og den fysiske elektronikken på disken, i all hovedsak ingen abstraksjon. Mens som informatikere er vi opptatt av det mer overordnede, dvs vi bryr oss bare om at vi har en fil i et filsystem som vi skal bruke. Men denne filen finnes jo egentlig ikke! Den er en abstraksjon av det som egentlig finnes lagret fysisk på disken.

HUSK: begrepet abstraksjon kan kanskje oppfattes som noe vanskelig, men bruken av det er ikke det, hele hensikten med abstraksjon er å forenkle noe som er vanskelig.

## 1.4 History

### 1.4.1 1.-4. Generation

**First Generation: Vacuum Tubes and Machine Language**

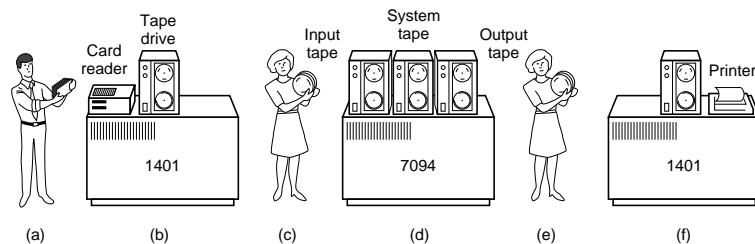


Bilde viser Colossus som ble benyttet mot slutten av andre verdenskrig i Storbritannia for å knekke koder, maskinen var rørbasert (manipulerte elektriske signaler via vaku-umrør).

Første generasjonsmaskiner ble programmert direkte i maskinkode, ikke noe assembly kode, programmert rent fysisk med flytting av ledninger og knapper.

*Ikke noe operativsystem.*

### Second Generation: Transistors and Languages

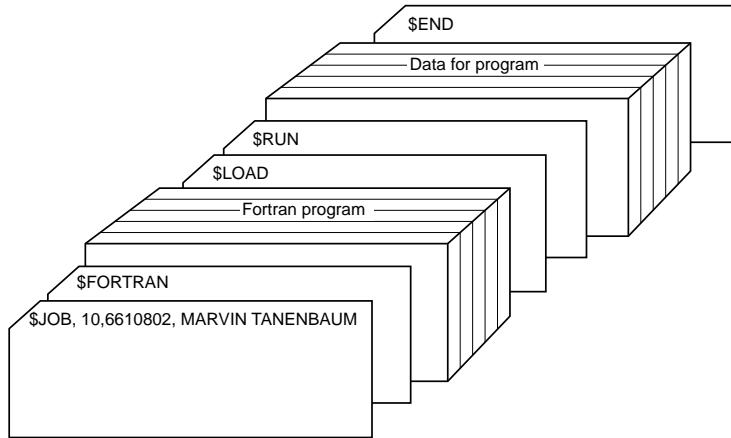


Transistoren gjorde datamaskiner pålitelige slik at de kunne produseres og selges (rik-tignok veldig dyrt!).

Programmering ble gjort typisk i FORTRAN og assemblykode, overført til hullkort og behandlet som vist i figuren: lesing og printing på IBM1401 og beregninger (real computing) på IBM 7094.

Siden mye tid gikk bort med å bære kort og taper frem og tilbake innførte man systemet med *batch* jobber, dvs en bunke med jobber som skal utføres. I dag bruker vi fortsatt begrepet batch jobber, dvs en *batch jobb er en oppgave som kan utføres selvstendig uten kommunikasjon med brukeren*. Det motsatte av en batch jobb/oppgave/prosess er en interaktiv jobb/oppgave/prosess.

## Second Generation: Typical Fortran Monitor System Job



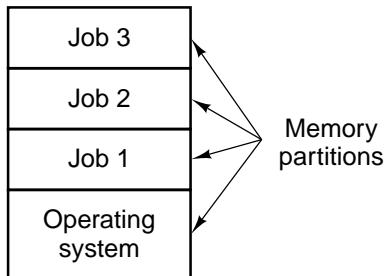
Batch håndtering var altså et tidlig operativsystemkonsept, og et primitivt grensesnitt vises her i form av kontrollkort.

Operativsystemene var stort sett FMS (Fortran Monitoring System) og IBSYS (fra IBM).

Vis tidslinjen for operativsystemer:

[http://en.wikipedia.org/wiki/Timeline\\_of\\_operating\\_systems](http://en.wikipedia.org/wiki/Timeline_of_operating_systems)

## Third Generation: ICs and Multiprogramming



Etter utviklingen av integrerte kretser (ICr) kunne man bygge mer generelle datamaskiner og via IBM System/360 kunne man ha flere programmer i minne samtidig: *Multiprogrammering*.

Disse systemene kom rett før en annen stor innovasjon på 60-tallet: *Timesharing* (og sammen med det: protection/access control/security). Opprinnelig utviklet i CTSS (Compatible Time Sharing System) ved MIT i 1962, men først skikkelig realisert med MULTICS videre utover 60-tallet. MULTICS står svært sentralt i operativsystemhistorien.

Timesharing var altså en ønsket utvikling fra batch systemer.

Se mer om Multics på <http://www.multicians.org>

### Fourth Generation: LSI

- With (Very) Large-Scale Integration (V)LSI, general-purpose small computers could be built
- Typically our current PCs (from 1980 -)

#### 1.4.2 Windows/Unix/Mac

##### Windows

- IBM sold PCs bundled with MS-DOS from beginning of 80s
- DOS/Windows from 85-95
- Win95/98/Me from 95-2000
- WinNT/2000/XP/2003/Vista/2008/7/2012/8 from 93-

IBM kunne valgt CP/M fra Gary Kildall istedet ...

Mac var først ute med GUI etter Xerox ikke syns det var interessant ...

Ingen sikkerhet på Windows før WinNT serien.

Vis Windows historikken:

[http://upload.wikimedia.org/wikipedia/commons/6/6d/Windows\\_Updated\\_Family\\_Tree.png](http://upload.wikimedia.org/wikipedia/commons/6/6d/Windows_Updated_Family_Tree.png)

og <http://www.levenez.com/windows/windows.pdf>

##### Unix/Linux

- Ken Thompson developed a stripped-down version of MULTICS on a PDP-7 he got hold of in 1969
- A large number of flavors developed (SystemV or Berkely-based)
- The GNU-project started in 1983 by Richard Stallman
- Unified with POSIX interface specification in 1985
- Minix in 1987 inspired Linus Torvalds to develop Linux (released in 1991)

Se [http://en.wikipedia.org/wiki/Tanenbaum-Torvalds\\_debate](http://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate)

og Unix historien:

<http://www.levenez.com/unix/unix.pdf>

TEHRAN, Iran, Nov. 29, 2010

## Iran Confirms Stuxnet Worm Halted Centrifuges

Country Had Previously Denied that the Computer Worm Had Affected Its Controversial Nuclear Program

[Font size](#) [Print](#) [E-mail](#) [Share](#) [Comments](#)

Like this Story? Share it:



(CBS/iStockphoto)

**(CBS/AP)** Iran's president has confirmed for the first time that a computer worm affected centrifuges in the country's uranium enrichment program.

Iran has previously denied the Stuxnet worm, which experts say is calibrated to destroy centrifuges, had caused any damage, saying they uncovered it before it could have any effect.

But President Mahmoud Ahmadinejad has said it "managed to create problems for a limited number of our centrifuges." Speaking to a press conference Monday, he said the problems were resolved.

Earlier in November, U.N. inspectors found Iran's enrichment program temporarily shut down, according to a recent report by the U.N. nuclear watchdog. The extent and cause of the shutdown were not known, but speculation fell on Stuxnet.

The finding was contained in a report from the International Atomic Energy Agency for the U.N. Security Council and the 35 IAEA board member nations.

Diplomats who spoke to the Associated Press that week said they did not know why the thousands of centrifuges stopped turning out material that Iran says it needs to fuel a future network of nuclear reactors.

I dag ...

(Stuxnet: få noen til å dytte en USB minnepinne i en PC som er på en eller annen måte i et nettverk sammen med andre PCr som kan være knyttet til en Siemens PLC kontroller, kanskje Stuxnet kommer på en laptop som tas med inn i et nettverk hvor PLC kontrollere finnes knyttet til andre PCr i samme nettverk el.l. utnytter minst

<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4250>

<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2568>

<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2729>

<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2772>

dette er altså den første kjente ormen som er rettet mot direkte industriell skade, dvs den gjør ikke noe ondsinnet mot maskina de hvis du blir infisert, dem gjør ikke noe for å samle kredittkortinfo eller for senere å kunne kreve penger for noe, den er ekstremt målrettet kun mot industriell skade!)

Det er en stor fordel for alle informatikere å kjenne til hvordan datamaskiner og operativsystemer fungerer slik at vi kan forstå hverdagen vår.

*For vår del gjelder det spesielt å forstå hvordan et dataprogram utføres i en datamaskin slik at vi kan forstå hvordan sårbarheter oppstår og utnyttes slik som f.eks. Stuxnet.*

## 1.5 Closing Remarks

### Metric Units

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
$10^{-3}$	0.001	milli	$10^3$	1,000	Kilo
$10^{-6}$	0.000001	micro	$10^6$	1,000,000	Mega
$10^{-9}$	0.000000001	nano	$10^9$	1,000,000,000	Giga
$10^{-12}$	0.00000000001	pico	$10^{12}$	1,000,000,000,000	Tera
$10^{-15}$	0.0000000000001	femto	$10^{15}$	1,000,000,000,000,000	Peta
$10^{-18}$	0.000000000000001	atto	$10^{18}$	1,000,000,000,000,000,000	Exa
$10^{-21}$	0.00000000000000001	zepto	$10^{21}$	1,000,000,000,000,000,000,000	Zetta
$10^{-24}$	0.0000000000000000001	yocto	$10^{24}$	1,000,000,000,000,000,000,000,000	Yotta

Bare pugg i vei...

**KB og KiB?** *We do not separate between KB, MB, GB, TB,... and KiB, MiB, GiB, TiB,... in this course. 1KB=1KiB is  $2^{10}$  unless otherwise stated.*

<http://en.wikipedia.org/wiki/Kibibyte>

Det har vært og er en del forsøk på å få oss til å konsekvent skrive 1KiB når vi mener  $2^{10}$  og 1KB når vi mener  $10^3$ . Dette er en god ide men desverre ikke gjennomførbar i praksis. Men det er viktig å vite at dette gjennomføres som regel hos lagringsleverandører, dvs når vi kjøper en 1TB disk og blir skuffet fordi den faktisk ikke er så stor så er det fordi vi har kjøpt  $10^{12} B$  og ikke  $2^{40} B$ , og

$$\frac{10^{12}}{2^{40}} \approx 0.909$$

mao, disken din er ca 909GB istedet for 1000GB.

**Why Study Operating Systems?** *You will never write an OS ... (5 - 50 million lines of code)*

1. Understand how you create a complex system (modules!)
2. Understand to easily troubleshoot problems
3. Understand to program optimally
4. Understand the security challenges (malware!)
5. The Concept of parallelism
6. The Concept of abstraction and virtualization
  - Sikkerhet!
  - Ytelse!
  - Ferdigheter! (kommandolinje/scripting)

## **1.6 Theory questions**

1. Forklar kort hva som er de to hovedoppgavene til et operativsystem.
2. Hva het det operativsystemet som anses som forløperen til UNIX og når omrent ble UNIX lansert første gang? Hvem skrev den første utgaven av UNIX?
3. Hva mener vi med begrepet *multiprogrammering* og hvilke fordeler innebærer dette i forhold til ren batch-kjøring.

## 1.7 Lab exercises

### 1. Windows-oppsett

*NB! Du behøver ikke gjøre denne oppgaven nå, men den er plassert her slik at du kan se hvordan lærern sin Windows-omgivelse er satt opp.*

Installer MinGW, Sysinternals Suite, GnuWIN, PowerShell Community Extensions (og gjerne Notepad++ og Putty inkludert pscp.exe), og legg alle til i PATH'en til PowerShell med

```
New-Item $pshome\profile.ps1 -type file  
notepad $pshome\profile.ps1  
$env:path += ";C:\Program files\MinGW\bin;C:\Program files\Sysinternals"  
$env:path += ";C:\Program files\gnuwin32\bin"  
$env:path += ";C:\Program files\Notepad++;C:\Program files\Putty"  
import-module pscx  
(etterhvert kanskje du vil teste ut PowerTab fra http://powertab.codeplex.com/ også)
```

### 2. Linux-oppsett

*(MERK: hvis du vil igang med oppgavene enklest mulig, må du gjerne droppe denne oppgaven og istedet logge deg inn på en linux server hvis du har tilgang til en slik)*

Sørg for at du har en kjørende linux maskin av et eller annet slag, gjerne Ubuntu (<http://www.ubuntu.com>) eller lubuntu hvis du vil ha en minst mulig ressurskrevende versjon av en Linux distribusjon, (<http://lubuntu.net/>). Hvis du bruker windows til vanlig, så installer gjerne linux på en virtuell maskin. Du kan lage virtuelle maskiner med VMware Player (<http://www.vmware.com/go/downloadplayer/>) eller VirtualBox (<http://www.virtualbox.org>). Anbefalt oppskrift er

- (a) Last ned og installer VMware Player/VirtualBox.
- (b) Lag en virtuell maskin i VMware Player/VirtualBox (velg at det skal være linux og ubuntu).
- (c) Last ned sist LTS-release (Long Term Support) CD-iso-fil av Ubuntu Desktop (<http://www.ubuntu.com>) (eller siste versjon av lubuntu).
- (d) Endre settings i den virtuelle maskinene du laget slik at CD-ROM-leseren er knyttet til iso-filen du nettopp lastet ned.
- (e) Start den virtuelle maskinen og installer Ubuntu på den.
- (f) Logg inn på den nyinstallerte virtuelle maskinen din og start programmet Ubuntu One, lag en ny bruker i denne slik at du får lagringsplass (for backup) i skyen. Eller installer Dropbox el.l. hvis du har det fra før.
- (g) Lag en mappe for operativsystemet som du velger at skal synkroniseres mot skyen (eller lag mappen direkte i Dropbox mappen hvis du endte opp med å bruke Dropbox).

3. Bli kjent med kommandolinje Unix/Linux hvis du ikke er godt kjent med den fra før. Gjennomfør tutorial en til fem på <http://www.ee.surrey.ac.uk/Teaching/Unix/index.html> (les også gjennom "introduction to the UNIX operating system" før første tutorial).
4. Bli kjent med programvarepakkesystemet til Debian-baserte distribusjoner (lærer går gjennom dette).
5. Bli kjent med en teksteditor. Hvis du ønsker å gjøre det enklest mulig, bare bruk gedit som er selvforklarende GUI. Hvis du vil lære deg en teksteditor som alle driftsfolk må kunne, så kan du bruke anledning til å lære deg vi med denne utmerkede tutorial: <http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Editors/ViIntro.pdf>.

## 6. C-programmering på Linux

Lag deg en directory hello og i denne lag en fil `hello.c` som inneholder:

```
#include <stdio.h>
int main(void) {
    printf("hello, world\n");
    return 0;
}
```

Kompiler denne til en kjørbar (eksekverbar) fil med `gcc -Wall -o hello hello.c` (-Wall betyr Warnings:All og er ikke nødvendig, men kjekk å ta med siden den hjelper oss programmere nøyere) og kjør den med `./hello`. Kjør den også ved å angi full path (dvs start kommandoen med / istedet for ./). Sjekk om environment variablen din PATH inneholder en katalog bin i hjemmeområdet ditt (`echo $PATH`). Hvis den ikke gjør det, så lag bin (`mkdir ~/bin`) hvis den ikke finnes, og legg til den i path'n med `PATH=$PATH:~/bin` (gjør evn endringen permanent ved å editere filen `~/.profile`). Kopier `hello` til bin og kjør programmet bare ved å skrive `hello`.

Hvis du ikke har programmert i C før, så les gjennom  
<http://www.loirak.com/prog/ctutor.php>



# Chapter 2

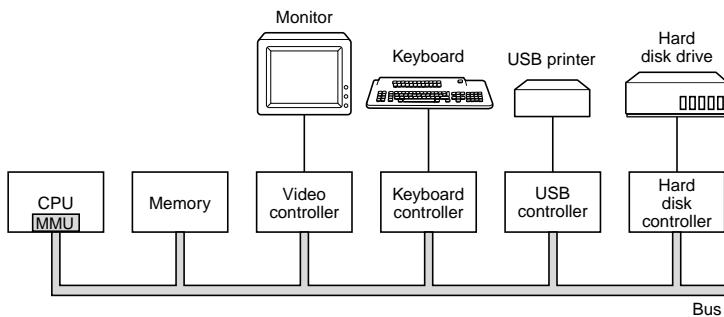
## Datamaskinarkitektur

### 2.1 Outcome

#### Learning Outcome

- Layering/Abstraction
- Performance
- Conceptual framework

#### CPU, Memory, I/O



I all hovedsak: CPU, Minne og I/O. Vi skal kikke litt mer på hvordan dette ser ut i dagens prosessorer senere i dette temaet, men i de fleste tilfeller holder det med dette bildet for at vi skal skjønne datamaskiner og operativsystemer. Legg merke til den lille MMU'n (Memory Management Unit) som befinner sammen med CPU'n, den skal vi bruke en del tid på under temaet minnehåndtering.

Men la oss bruke 20 minutter til å se en video som introduserer de komponentene/begrepene vi trenger vite om:

[http://www.youtube.com/watch?v=cNN\\_tTXABUA](http://www.youtube.com/watch?v=cNN_tTXABUA)

Merk: det gjør ikke noe om du faller fra litt innimellom, men se denne på nytt utenom forelesning for å repetere virkemåten til en datamaskin.

## 2.2 ISA

### Instruction Set Architecture

- Elec. Engineer: Microarchitecture
- Comp. Scientist: Instruction Set Architecture (ISA):
  - native data types and *instructions*
  - *registers*
  - addressing mode
  - memory architecture
  - interrupt and exception handling
  - external I/O

Hver CPU type har et bestemt sett med instruksjoner den kan utføre (instruksjonssettet, forskjellig mellom f.eks. Intel Pentium og Sun SPARC). Instruksjonssettet er det vi som programmerere ser av datamaskinarkitekturen, dvs det vi kan bruke direkte for å programmere på lavest mulige nivå. Vanligvis bruker vi den symbolske representasjonen av selve maskininstruksjonene: assembly kode.

Mens vi som informatikere som regel ikke bryr oss om lavere nivå enn instruksjonssettet, er derimot elektroingeniørene opptatt av *Mikroarkitekturen* som dreier seg om hvordan instruksjonene faktisk skal implementeres i elektroniske komponenter for å lage en fysisk CPU.

### 2.2.1 Registers

**Registers vs Memory** Register contents *belong to the current running program and the operating system*

There can be *several programs loaded in memory* (this is called multiprogramming)

Knyttet til hver CPU er et sett med registre (typisk mellom 100 og 200 stykker), hvorav et par spesielle er (TAVLE):

**Instruction Pointer/Program Counter (EIP/RIP)** instruksjonspekeren

**Stack pointer (ESP/RSP)** peker til toppen av stack

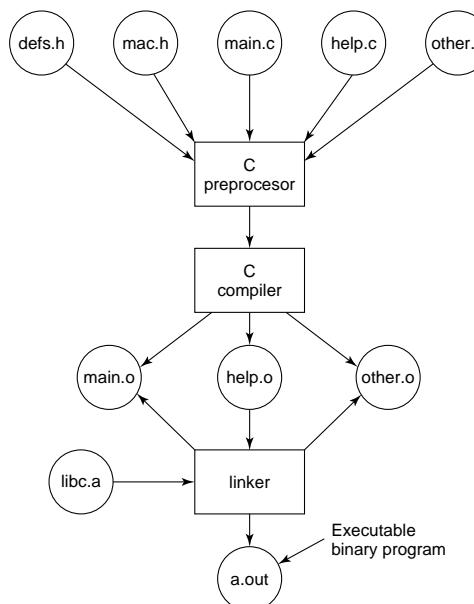
**Base pointer (EBP/RBP)** peker til starten av “current stack frame”, dvs nåværende arbeidsområde på stacken (hver gang du kaller en funksjon i koden din startes en ny stack frame)

**PSW - program status word (EFLAG/RFLAG)** kontrollbits (av og til kalt flaggregisteret), f.eks. bit 12 og 13 er user mode/kernel mode bits

Operativsystemet må være klar over alle registre siden disse må lagres unna når det svitsjes mellom prosesser (tidsmultiplexing).

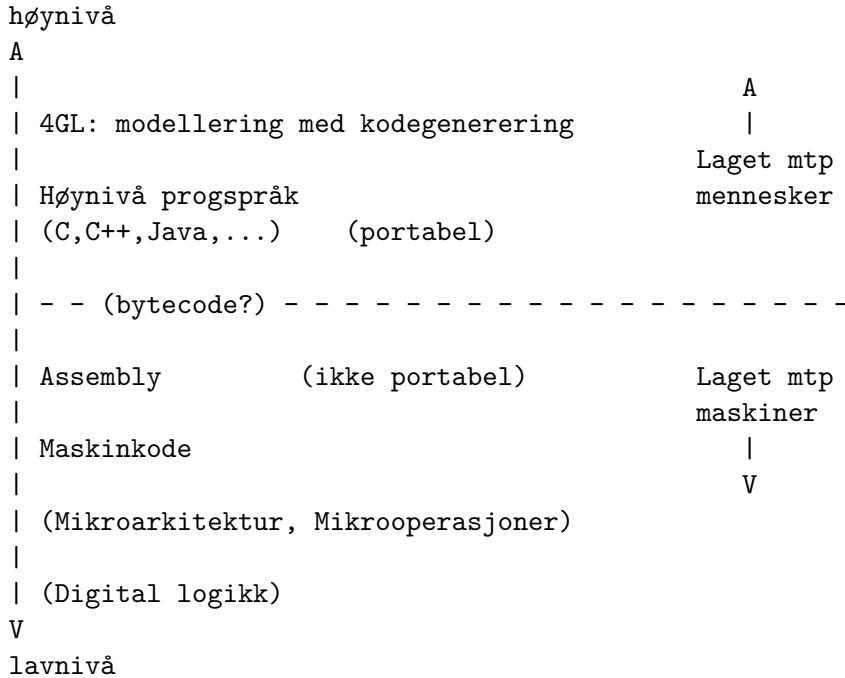
## 2.3 Assembly

### From C Code to Executable



Figuren viser hvordan et C program kompileres i flere trinn før det blir en ferdig eksekverbar fil. Hvis vi skal ha ut assemblykode så henter vi det ut etter kompileringen istedet for en objektfil (.o-fil). Objektfilene inneholder maskinkode.

Generelt kan vi betrakte abstraksjonsnivåene i en datamaskin ut fra følgende figur:



## DEMO:

```
cat asm-0.c
gcc -S asm-0.c      # lag assembly kode
gcc asm-0.c         # lag maskinkode
cat asm-0.s          # se linjen "ret"
xxd -p asm-0         # se ferdiglinket maskinkode som hex
xxd -g 0 -b asm-0 | awk '{print $2}' # evn som binær
# mapping finnes i http://ref.x86asm.net/coder64-abc.html
# eller se via innholdsfortegnelsen i
# http://download.intel.com/products/processor/manual/325383.pdf
gdb asm-0            # åpne maskinkoden i GNU Debugger
disassemble /r main # disassemble og vis tilhørende maskinkode
```

Det er lettest å disassemble opcodes som ikke har operander, dvs ret, se hvordan retq mappes til C3 i URLn over, dvs søk på C3 og etterhvert finner du en mapping til retn som er den samme, dvs en return av type near (near = innen samme minnesegment, far = return til et annet minnesegment).

### 2.3.1 gcc

## GNU Compiler Collection: gcc

```

gcc -S tmp.c      # from C to assembly
gedit tmp.s       # edit it
gcc -o tmp tmp.s # from assembly to machine code
./tmp             # run machine code

# sometimes nice to remove lines with .cfi
gcc -S -o - tmp.c | grep -v .cfi > tmp.s

```

### 2.3.2 32 vs 64 bit

**32 vs 64 bit code** Same C-code, different assembly and machine code

```

gcc -S asm-0.c      # 64-bit since my OS is 64-bit
grep push asm-0.s   # print lines containing "push"
gcc -S -m32 asm-0.c # 32-bit code
grep push asm-0.s   # print lines containing "push"

```

Difference in instructions and registers

### 2.3.3 Syntax

```

.file  "asm-0.c"    # DIRECTIVES
.text
.globl main
main:           # LABEL
    push   rbp      # INSTRUCTIONS
    mov    rsp, rbp
    mov    0, eax
    ret

```

Direktiver starter med et punkt (.) og merker (labels) avsluttes med kolon (:).

Assemblykode oversettes til maskinkode med et program som kalles en assembler (f.eks. GAS, NASM, el.l. program). Assemblykode består av instruksjoner (som oversettes som oftest en-til-en til maskinkode) og direktiver som er informasjon til assembleren. I tillegg benyttes også labels (merker) for å henvis til bestemte deler av koden.

Linje for linje betyr denne assemblykoden:

.file "asm-0.c" metainformasjon som sier hvilken kildekode som er opphavet til denne koden

.text sier at her starter kodedelen (dvs selve instruksjonene i et program kalles ofte text, i motsetning til andre deler som f.eks. kan være bare data)

**.globl main** sier at main skal være global i scope (synlig for annen kode som ikke er i akkurat denne filen, dvs kode som linkes inn, delte biblioteker o.l.)

**main:** et label (merke), det er vanlig å kalle hovedfunksjonen i et program for main

**push rbp** legger base pointer på stacken

**mov rsp, rbp** setter base pointer (registeret rbp) lik stack pointer (registeret rsp)

**mov 0, eax** setter et “general purpose register” eax til å være 0, det er vanlig å legge returverdien til et program i eax registeret

**ret** legger den gamle instruksjonspekern som befinner seg på stacken tilbake i rip registeret hvis denne finnes slik at den funksjonen som kallet meg kan fortsette der den slapp

**GNU/GAS/AT&T Syntax** [http://en.wikibooks.org/wiki/X86\\_Assembly/GAS\\_Syntax](http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax)

**instruction** mnemonic

**mnemonic suffix** b (byte), w (word), l (long), q (quadword), ...

**operand prefix** % (registers), \$ (constants)

**address op.** `movl -4(%ebp, %edx, 4), %eax`  
“load (ebp - 4 + (edx × 4)) into eax”

En mnemonic tilsvarer en maskinkode opcode. Dvs det kalles mnemonic fordi det er bare et kort navn som brukes istedet for å måtte huske den egentlig numeriske maskinkode opcode'n.

DEMO:

`asm-0.c` Eksempelet over

`asm-1.c` En ekstra kodelinje fordi 0 skrives til stacken, og deretter kopieres fra stacken til registeret (MERK: plutselig to skrivinger til minne som er mye (dvs minst 10x) tregere enn å skrive til et register)

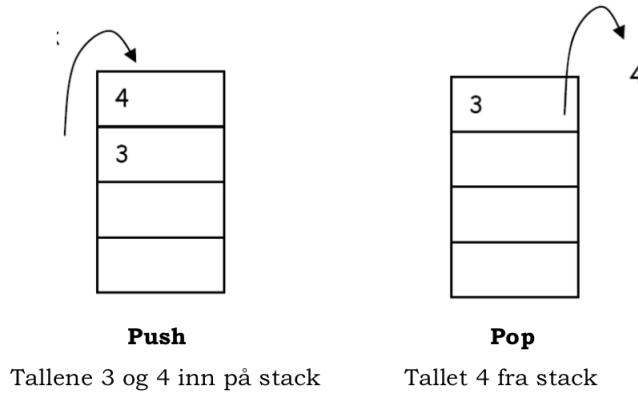
MERK: parenteser betyr altså at vi prater om en adresse i minne.

`diff asm-0.s diff asm-1.s`

MAO: DENNE STACKEN DUKKER JO OPP HELE TIDEN...den må vi repitere hva egentlig er for noe.

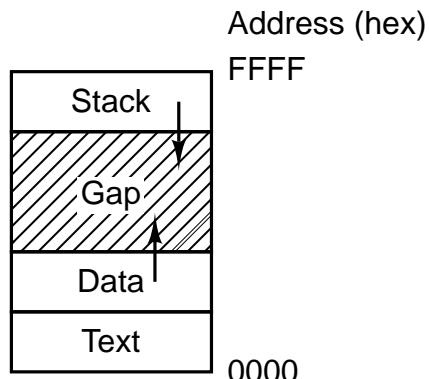
### 2.3.4 Stack

#### Stack



Stacken vokser nedover på X86-plattformen (dvs tenk på den som en bøtte som er opp ned), og brukes bl.a. til å holde lokale variable, argumenter til funksjoner og returadresser.

#### The Three Segments of Program Loaded in Memory



#### DEMO:

`asm-2.c` Lokale og globale variable, merk at det har dukket opp en section `-data` eller `.bss` (BSS betyr Block Starter by Symbol som er et vanlig navn på den ene delen av data området, dette er altså data som allokeres i data området og ikke på stack). Hele data-området er egentlig delt i flere deler, som regel data, bss og heap, se [http://en.wikipedia.org/wiki/Data\\_segment](http://en.wikipedia.org/wiki/Data_segment).

Merk at globale `j` brukes til å adressere med utgangspunkt i instruksjonspekern `rip`. Dette er et tilfelle at PC-relative adressering (<http://software.intel.com/en-us/articles/introduction-to-x64-assembly>):

RIP-relative addressing: this is new for x64 and allows accessing data tables and such in the code relative to the current instruction pointer, making position independent code easier to implement.

Se også [http://en.wikipedia.org/wiki/Addressing\\_mode#PC-relative\\_2](http://en.wikipedia.org/wiki/Addressing_mode#PC-relative_2)

Her ser vi også add instruksjonen.

Det området på stacken som en funksjon benytter kalles en *stack frame*, og er området som pekes ut mellom base pointer og stack pointer. Når en funksjon kalles lagres unna base pointer på stacken og base pointer settes lik stack pointer, dermed har vi startet en ny stack frame, og vi kan gå tilbake til forrige stack frame når funksjonen er ferdig.

Stack frame forklart: <http://www.youtube.com/watch?v=vcfQVwtoyHY>

DEMO: asm-3-stack.c

**Why Learn Assembly** From Carter (2006) *PC Assembly Language*, page 18:

- Assembly code *can be faster* and smaller than compiler generated code
- Assembly allows access to *direct hardware features*
- Deeper understanding of *how computers work*
- Understand better how *compilers* and high level languages like C work

Eksempelvis vil spillprogrammerere ønske å utnytte hardware egenskaper maksimalt, mens sikkerhetsdudes (and dudettes) ønsker å benytte assembly-nivå for effektiv implementering av lavnivå kryptooperasjoner hvor det ofte er snakk om å flytte bit i et register som en del av en kryptoalgoritme.

La oss sjekke at vi har noenlunde kontroll på assemblykode, programutførelse og stack (basert på X86 og C++ kode) og en del tilhørende kommandolinjeverktøy: <http://vimeo.com/21720824>

## 2.4 Execution

### How a CPU Works

```
while(not HALT) {  
    IR = mem[PC];      # IR = Instruction Register  
    PC++;              # PC = Program Counter (register)  
    execute(IR);  
    if(IRQ) {          # IRQ = Interrupt ReQuest
```

```

    savePC();
    loadPC(IRQ);
        # Hopper til Interrupt-rutine
    }
}

```

Demo (uten interrupts):

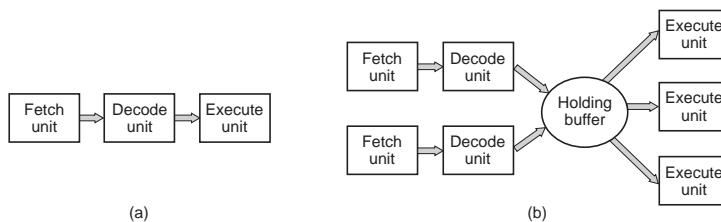
<http://www.iu.hio.no/~haugerud/os/demoer/iecycle.swf>

## Important Concepts

- *Clock speed/rate*
- *Pipeline*
- Machine instructions into *Micro operations* ( $\mu$ ops)
- *Out-of-Order* execution

Selve utførelsen av alt som skjer i en datamaskin baseres på en klokketakt, dvs hvis vi har en klokkehastighet på 1GHz så betyr det at det kommer en milliard pulser (generert av en oscillator) hvert sekund, og hver slik pulse skyver utførelsen av instruksjonene et hakk videre. Litt forenklet kan vi tenke på at CPUn da klarer å utføre en instruksjon for hver puls (klokkeperiode), som vil si at den bruker et nanosekund (et milliard'te-dels sekund) på å utføre en instruksjon (dette er ikke helt presist siden en superskalar CPU kan utføre flere mikroinstruksjoner hver klokkeperiode).

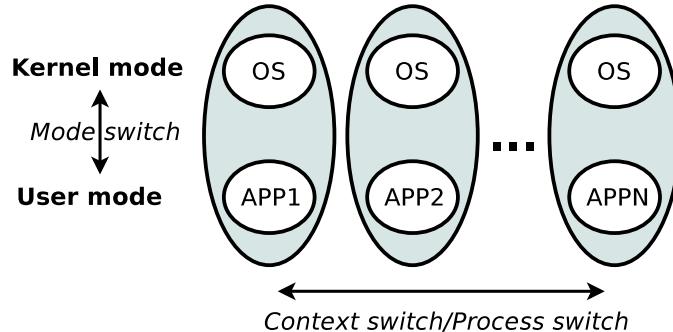
## Pipelined and Superscalar CPU



For at CPUn skal jobbe effektivt organiseres den som vist i figuren med *pipeline* av instruksjoner (slik at det alltid ligger en kø klar til å utføres), og en *superscalar* struktur (hvor det er forskjellig eksekveringsenheter i CPUn som utfører forskjellig type operasjoner, f.eks. en for heltallaritmetikk og en for flyttallsaritmetikk).

## 2.5 Protection

### PSWs Mode Bits: User/Kernel Mode



PSW registeret (X86: RFLAG/EFLAG) har to bits som styrer hvilken modes/tilstand prosessoren er i. Dette er hvordan prosessoren vet om den er i user mode eller kernel mode, og om den har lov utføre de instruksjonene den skal utføre. Programmene som kjører skifter ofte mellom og kjøre sin egen kode, og å låne tjenester fra operativsystemet (f.eks. hver gang det skal leses fra en disk eller skrives til skjerm), dette skifte kalles jo mode switch. Operativsystemet har også ansvaret for å skifte mellom alle applikasjoner som ønsker å bruke prosessoren, og dette skifte kalles context eller process switch. Mode switch er raskt, context/process switch er tregt. Merk i denne figuren at det er forskjellige applikasjoner (APP1, APP2, ..., APPN) men alle ser det samme operativsystemet (OS).

*En mode switch tar typisk rundt hundre nanosekunder (ns), mens en context switch tar noen mikrosekunder (μs).*

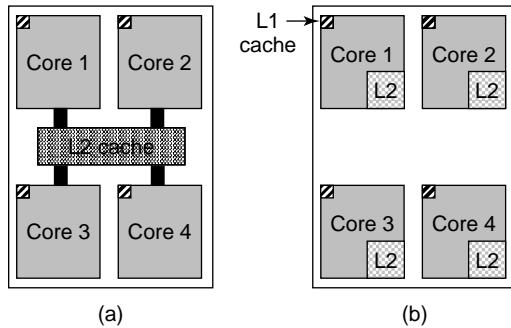
Hvor ofte gjøres en kontekst switch?

```
c1=$(grep ctxt /proc/stat | awk '{print $2}')
sleep 1
c2=$(grep ctxt /proc/stat | awk '{print $2}')
echo "Antall context switch'er siste sekund var $((c2-c1))"
```

Hvor mye tid bruker en prosess i user og kernel mode? Det avhenger i stor grad av hvor mye I/O som må gjøres, se windows perfmon (Performance monitor - Add - ProcessorMedPilOpp - Add privileged og user time, flytt musepekkern/flytt hele vinduet og se hvordan privileged grafen går opp før å håndtere interrupts siden det grafiske systemet til Windows OSet kjører i kernel mode).

## 2.6 Multicore/ Hyperthreading

### Multicore Chips and Cache Structure



Moore's lov om dobling av antall transistorer hver 18. måned.

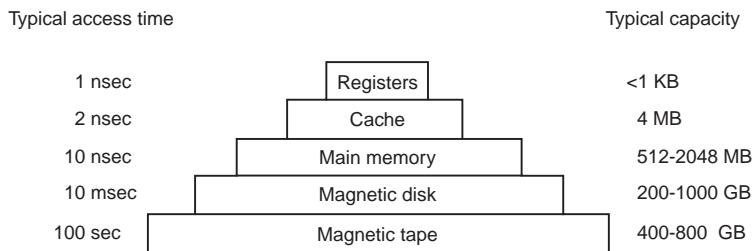
For å øke utnyttelsen i Pentium 4 ble *multithreading/hyperthreading* innført, dette betyr dublering av hardware for å holde statusen til flere tråder samtidig slik at svitsjing (tidsmultiplexing) mellom dem kan gjøres enormt fort.

[http://ark.intel.com/products/52231/Intel-Core-i7-2620M-Processor-\(4M-Cache-2\\_70-GHz\)](http://ark.intel.com/products/52231/Intel-Core-i7-2620M-Processor-(4M-Cache-2_70-GHz)) to kjerner men fire tråder som fører til at mitt OS tror jeg har fire CPUer:  
less /proc/cpuinfo eller gnome-system-monitor

Det er også vanlig med flere CPUR pr brikke, hvor L2 cache kan plasseres felles (Intel) som fører til et mer komplisert cache-kontroller eller hos hver CPU (AMD) som fører til problemer med å holde cachene konsistente.

## 2.7 Cache

### Memory Hierarchy



Tallene i figuren er veldig ca-tall.

Et annet eksempel på noen av disse tallene er

<http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait>

Flaskehalsen som oppstår mellom hastigheten på CPU og tiden det tar å gjøre minneaksess kalles ofte *von Neumann flaskehalsen*, og i praksis løses den med bl.a. cache-mekanismen.

## Why Cache Works

- Locality of reference
  - *spatial locality*
  - *temporal locality*

Cache gjør at programmer kjører raskere fordi instruksjoner gjenbrukes ofte (samlokalisert i tid) mens data ofte aksesseres blokkvis (samlokalisert i rom, har du lest en byte skal du sikkert snart ha byte'n ved siden av den også).

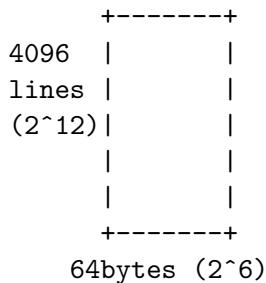
Caching står sentral i mange deler av informatikken og dukker opp flere steder i OS-verdenen. Sentrale problemstillingene knyttet til cache er nevnt på side 22 (TAVLE):

1. Når skal cache benyttes
2. Hvilken cache line skal overskrives
3. Hva gjøres med modifiserte (dirty) cache lines

Ved caching av minne behandles minne i form av *cache lines* som oftest på 64 bytes (kan være mellom 8 og 512 bytes avhengig av arkitekturen), dvs det er cache lines som caches, ikke enkeltbytes.

Et stort spørsmål knyttet til cache er hvor plasseres en ny cache line, spesielt "når cachen er full". Vi skal se mer på dette når vi ser på minnehåndtering spesielt (page replacement algoritmer kalles det da), men slik det gjerne gjøres for CPU cache er helt enkelt, Tanenbaum side 22:

For example, with 4096 cache lines of 64 bytes and 32 bits addresses, bits 6 through 17 might be used to specify the cache line, with bits 0 to 5 the byte within the cache line.



Tavle:

- Hvor mye adresseres med 32 bit?  $2^{32} = 4\text{GB}$  - Hvor mye addresses med 18 bit?  $2^{18} = 256\text{KB}$

mao, hver cache lines i minne som befinner seg med 256KB mellomrom vil plasseres samme sted i cache (overskrive hverandre), en dum men enkel "cache line replacement algoritme". Dette er nok et eksempel på en hash funksjon som vi sikkerhet husker fra databaser og fra algoritmer, se

[http://en.wikipedia.org/wiki/Hash\\_function#Caches](http://en.wikipedia.org/wiki/Hash_function#Caches)

Denne type cache kalles en *Direct mapped cache* som er en av tre typer cache.

### 2.7.1 Set Associative Cache

## Set Associative Cache

- *Direct mapped cache* (only one “way”)
  - $(N\text{-}way)$  *set-associative cache*
  - *Fully associative cache* (only one “set”)

Direct mapped og Fully associative er bare spesialtilfeller av set-associative caching.

<http://duartes.org/gustavo/blog/post/intel-cpu-caches>

Det er tre parametere som gjelder:

L antall bytes pr cache line

K hvor mange-veis assosiativ (antall "ways")

N antall set

F.eks. har Intel Nehalem (Core i7 o.l.) en 32KB L1 instruksjonscache som har  $L=64$ ,  $K=4$  og  $N=128$  og en 32KB L1 datacache som har  $L=64$ ,  $K=8$  og  $N=64$ .

$$(64 \times 4 \times 128B = 2^6 2^2 2^7 B = 2^{15} = 32KB)$$

Merk: for set associative cache og fully associative cache må vi gjøre en litt mer avansert beslutning ift hvilken cache line som skal overskrives og da benyttes som regel algoritmen *Least Recently Used (LRU)*. Da skriver man over den cache line som er minst nylig brukt.

Demo set associative cache (må kjøres i 32-bit Windows med Vivio player): <https://www.scss.tcd.ie/Jeremy.Jones/vivio/caches/cache.htm>

MERK: de heksadesimale adressene endres fordi de deles i tre og hver del er ikke nødvendigvis et multippel av 4 bit, f.eks.:

```

L=16, K=2, N=4
hex 2200 blir til 08800 på følgende måte:
      2   |   2   |   0   |   0
0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0
          tag           |set| offset
          0     8       | 0 |   0

```

## 2.7.2 Write Policy

### Write Policy

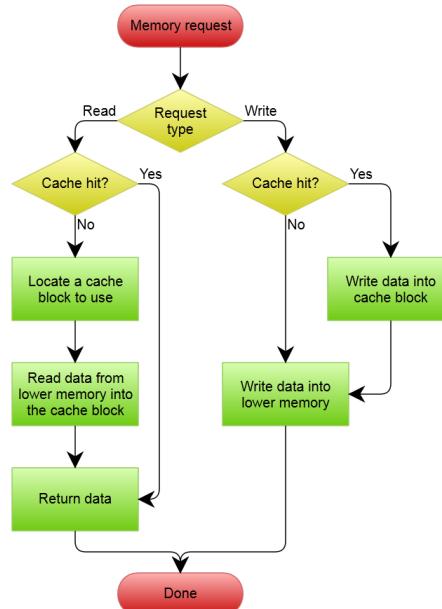
**Write-through** Write to cache line and immediately to memory

**Write-back** Write to cache line and mark cache line as *dirty*

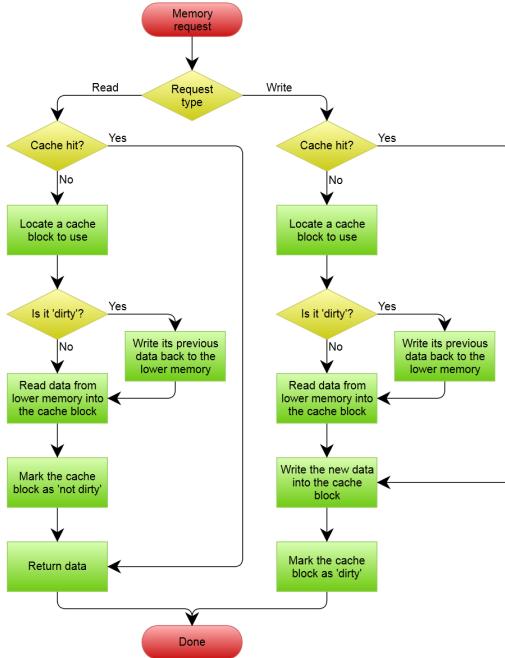
Ved write-back skrives dataene til minne først når cache line'n skal overskrives av en annen, eller i andre tilfeller som f.eks. en context switch.

Et viktig poeng er at Write-back cacher er spesielt utfordrende for når det er flere prosessorkjerner tilstede: hva hvis flere prosessorkjerner har cachet de samme dataene? Hvordan vet en prosesorkjerner at ikke en annen prosessorkjerner har skrevet til de dataene den har cachet? Dette løses selvfølgelig med en cache coherence protokoll, f.eks. MESI.

### Write-through



## Write-back



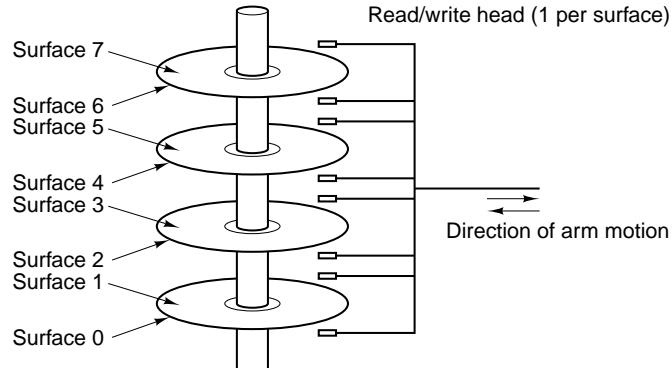
Ved Write-back caching må vi altså sjekke om cache-blokken vi ønsker cache i er dirty, dvs inneholder data som ikke er skrevet til neste nivå datalagring enda. Dette gjelder både for lese og skrive forespørsler. Write through caching er det enkleste og tryggeste (siden caching bare vil inneholde kopi av data som finnes et annet sted), men skal systemet gi god ytelse for skriveforespørsler så må man benytte write-back caching.

begge figurene fra <http://en.wikipedia.org/wiki/Cache>

I operativsystemer er det et poeng at når vi bytter om fra å jobbe på et program til et annet (context switch) så er cachen full av data fra det første og det tar tid å bytte det ut! Derav sier vi at context switch er ganske kostbart.

2.8 I/O

## Classic Disk Drive Structure



For å adressere på en klassisk disk må man vite sylinder, sektor og lesehode. En sektor har typisk 512 bytes, og det er ofte flere sektorer i de ytre sporene enn i de indre.

OSet har en device driver som prater med disk kontroller (ved å skrive til dens registre) og disk kontrollerer tar seg av alle de detaljerte lavnivå diskoperasjonene (hente ut bit fra disk, feilsjekk, osv)

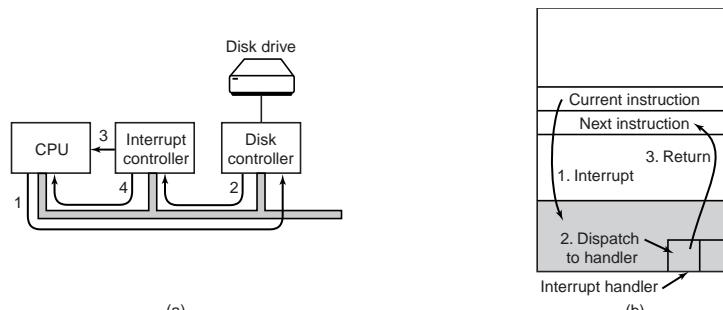
Husk at random aksess tid for disk endres dramatisk ved SSD (Solid State Disk)!

*Men Husk også at konseptet med et minnehierarki alltid vil være det (varig kunnskap!).*

<http://www.komplett.no/k/kc.aspx?bn=10088>

## 2.9 Interrupts

### I/O and Interrupts



I/O kan gjøres på tre måter (TAVLE):

1. Busy waiting/Polling
2. Interrupt
3. DMA (Direct Memory Access)

Interrupt håndteres via en egen interrupt kontroller, og som regel nederst i fysisk minne befinner det seg en *interrupt vektor tabell* hvor hver *interrupt vektor* er en adresse til koden for den tilhørende interrupt handleren/rutinen i minne.

Hvor ofte kommer et interrupt?

```
i1=$(grep intr /proc/stat | awk '{print $2}')
sleep 1
i2=$(grep intr /proc/stat | awk '{print $2}')
echo "Antall interrupts siste sekund var $((i2-i1))"
```

### Interrupts

- Hardware interrupt (asynchronous)
- Exceptions (synchronous)
- Traps/Software interrupt (synchronous)

TAVLE:

**Hardware interrupt** som regel knyttet til I/O slik via akkurat så

**Exceptions** er internt generert hardware interrupt typisk generert ved deling på null eller forsøkt tilgang til et minneområde som ikke finnes el.l.

**Traps/Software interrupt** brukes ved systemkall

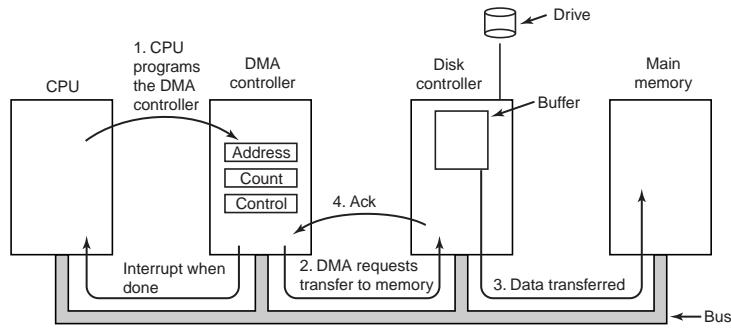
Trap brukes også av og til som betegnelse på det å overføre kontroll fra user mode til kernel mode. Vær klar over at disse begrepene brukes ofte med litt forskjellig betydning avhengig av hvem du prater med (slik det ofte er, f.eks. noen sier kanskje at trap er en type exception). Hardware interrupt er asynkrone fordi de kan oppstå når som helst, mens exceptions/traps/software interrupts er synkrone fordi de kan bare oppstå som konsekvens av en instruksjon.

Oppsummerende video med I/O kontroller m/register, memory-mapped vs porter, polling vs interrupt, interrupt vektor tabell, exceptions.

<http://www.youtube.com/watch?v=1DLmuIXzk0A>

#### 2.9.1 DMA

##### Direct Memory Access (DMA)

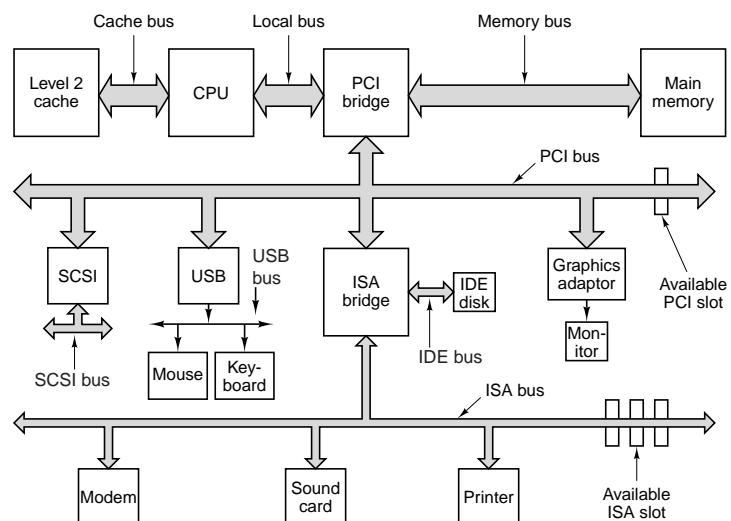


DMA gjør som regel I/O langt mer effektivt. DMA er en egen kontroller i hardware som gjør I/O på vegne av CPUn. Dvs istedet for at CPU henter byte for byte (eller word for word) fra en device, så ber den DMA kontrolleren gjøre det istedet og si ifra (dvs gi et interrupt) når den er ferdig.

På figuren skjer trinn 1 en gang, mens trinn 2-4 gjentas mange ganger avhengig av hvor mye data som skal overføres.

## 2.10 Architecture

### Structure of a Pentium System



Det er mange busser i et typisk moderne datasystem. Den opprinnelige PCI bussen kjørte på 66MHz og hadde en bredde på 32 bit som betyddet at den kunne frakte 528 MB/sek.

Se også Core og I7 på side 9 og 15 i Mikroprosessor kompendiet.

## **2.11 Booting**

### **Booting a PC**

1. BIOS (Basic Input Output System) runs and scans all devices
2. BIOS determines boot device from list in CMOS memory
3. First sector (MBR) from boot device is read and executed
4. Active partition read from partition table at end of MBR
5. Second boot loader read and executed from active partition which loads OS
6. OS queries BIOS for devices and loads device drivers
7. OS initializes tables, background process and starts login/GUI

## 2.12 Theory questions

1. Hva er et “Directive” i assemblykode?
2. Hva forbinder du med begrepene *superskalar* og *pipelining* i en prosessor?
3. Hva menes med *kernel mode* og *user mode*?
4. Hva menes med *hyperthreading/multithreading* som ble innført med Pentium 4? Hva var hensikten med dette?
5. Hva er ulempen ved en direkte mappet (“direct mapped”) cache?
6. Anta en CPU som skal sjekke om dataene på en gitt adresse i minne finnes i en sett-assosiativ cache. Hvordan finner CPU’en frem til riktig sett og riktig vei (“way”)?
7. Nevn tre måter interrupt kan oppstå på i en datamaskin.
8. Hvor store cache lines har en 2-veis sett-assosiativ cache som er 32KB stor og har 128 sett?

## 2.13 Lab exercises

1. Last ned C-kode-eksemplene fra forelesningen. Kompiler disse til assembly-kode, fjern linjer du tror er unødvendige, lag maskinkode og kjør programmene. Hvis du tror noen linjer er unødvendig, så bruk google for å finne ut hva de gjør.
2. Forklar hva hver linje i følgende assemblykode gjør:

```
01      .text
02 .globl main
03 main:
04     pushq  %rbp
05     movq  %rsp, %rbp
06     movl  $0, -4(%rbp)
07     cmpl  $0, -4(%rbp)
08     jne   .L2
09     addl  $1, -4(%rbp)
10 .L2:
11     movl  $0, %eax
12     leave
13     ret
```

Denne assemblykoden ble generert av et C-program på ca fem linjer. Hvordan så programkoden til dette C-programmet ut?

3. Forklar hva hver linje i følgende assemblykode gjør:

```
01      .text
02 .globl main
03 main:
04     pushq  %rbp
05     movq  %rsp, %rbp
06     movl  $0, -4(%rbp)
07     jmp   .L2
08 .L3:
09     addl  $1, -4(%rbp)
10     addl  $1, -4(%rbp)
11 .L2:
12     cmpl  $9, -4(%rbp)
13     jle   .L3
14     movl  $0, %eax
15     leave
16     ret
```

Denne assemblykoden ble generert av et C-program på ca fem linjer. Hvordan så programkoden til dette C-programmet ut?

## Chapter 3

# Introduksjon til operativsystemer

### 3.1 Outcome

#### Learning Outcome

- Layering/Abstraction
- Conceptual framework

### 3.2 OS Zoo

#### Types of Operating Systems

- Mainframes
- *Server/Multiprocessor/Personal/Handheld*
- Embedded
- Sensor node
- Real-time
- Smart card

Mainframes er fortsatt i stor grad levende og vil ”alltid” være det, typisk IBM System Z, se [http://en.wikipedia.org/wiki/System\\_Z](http://en.wikipedia.org/wiki/System_Z). Mainframes kan også kjøre Linux (og gjør ofte det).

Se <http://debatt.digi.no/58823/ibm-leverer-nordens-storste-linux-maskin>

For Server/Multiprocessor/Personal/Handheld type operativsystemer er det gjerne en eller annen variant av Linux eller Windows som benyttes.

Embedded systems er f.eks. en MP3-spiller eller en mikrobølgeovn, OSer som benyttes er f.eks. QNX eller VxWorks, ingen andre applikasjoner kan kjøres enn de som er laget kun for den enheten det kjører på.

Sensor nodes er små enheter som typisk brukes til overvåkning av luftkvalitet eller detektere røyk eller bevegelser/trykkendringer, typisk OS er TinyOS.

Real-time systems er systemer hvor tiden er svært viktig. *Soft real-time* er f.eks. avspilling av lyd og video hvor forsinkelse til en vis grad kan være akseptabelt, mens *hard real-time* er f.eks. styringssystemet til en robot som skjærer ut noe som passerer den med sag hvor forsinkelse ikke er akseptabelt. Et typisk eksempel er e-Cos, men merk at OSer som benyttes i embedded systems og sensor nodes også gjerne kan være real-time systems.

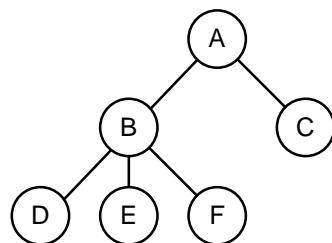
Smart kort er minidatamaskiner på kredittkort og OS som gjerne benyttes er MULTOS eller JavaCard.

Man prater ofte om preemptive og non-preemptive operativsystemer, vi kommer tilbake til dette når vi skal se på scheduling. Poenget er at real-time systemer må hvertfall være preemptive. Dvs preemptive betyr "å ta bort", dvs å kunne ta fra en prosess CPU'n uten at den egentlig selv vil det (som regel basert på klokkeinterrupt).

## 3.3 Concepts

### 3.3.1 Process

#### A Process Tree



Konsepter/Abstraksjoner:

- Prosess
- Adresserom
- Filsystem

- Pipe (rør)

Husk:

Prosess	Addresserom	Fil
CPU	RAM	DISK

*En prosess er et program under utførelse.* Et OS må håndtere flere prosesser samtidig, men er det bare en CPU kan bare en prosess kjøre om gangen.

En prosess består av innholdet i mange registre, et addresserom i minne, samt en del annen informasjon (liste over åpne filer, liste over relaterte prosesser, ventende alarmer, o.l.). *En prosess er en container for et kjørende program (kan også kalles konteksten som et program utføres innen).*

Hver prosess har en prosess control block (PCB) som er en entry i OSet's prosesstabell. En prosess som midlertidig stoppes lagrer unna all informasjon i PCBn og tar vare på addresserommet sitt.

I Unix/Linux verdenen lages alltid prosesser som barn av andre prosesser som i figuren. I Windows lages også prosesser på en måte slik, men de blir mer frittstående prosesser, dvs prosesshierarkiet i Unix/Linux dyrkes ikke i like stor grad i Windows (selv om det egentlig finnes der også).

Prosess er et konsept/en abstraksjon, og et annet relatert er *addresserom*. Addresserom er et konsept, en abstraksjon, fordi prosesser trenger å forholde seg til et virtuelt addresserom (virtual memory) som ikke er det samme som fysisk minne, f.eks. vil de fleste OS på PCr idag ha mulighet til å adressere 4GB (32-bits arkitektur gir  $2^{32}B=4GB$ ) mens fysisk minne kanskje bare er 1GB (som en typisk netbook f.eks.).

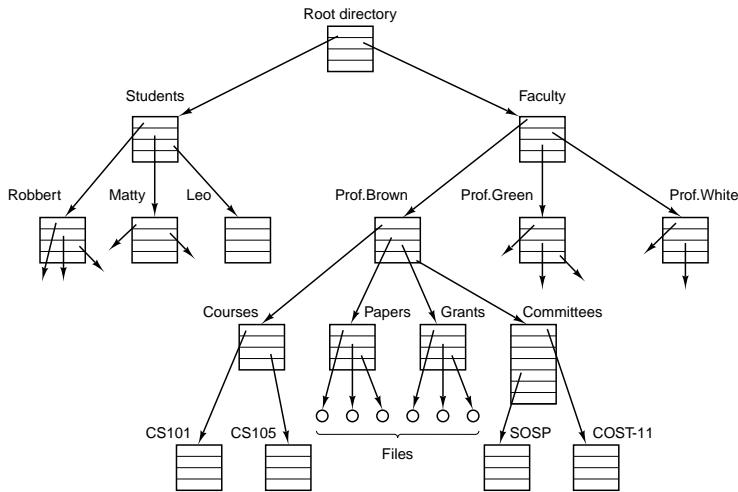
DEMO:

`top htop ps pstree` i Unix/Linux.

`taskmgr` eller helst `procexp` i Windows.

### 3.3.2 File

#### A File System



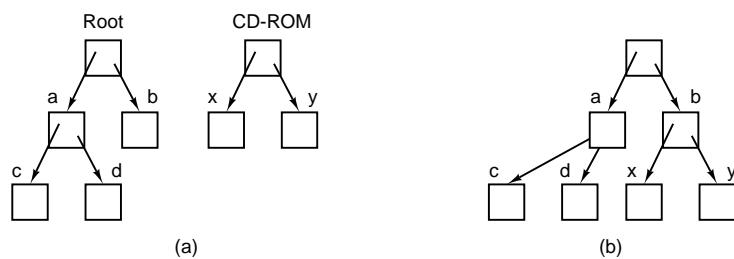
Filsystem er også et konsept/abstraksjon fordi det egentlig ikke finnes rent fysisk slik vi ser det for oss i figuren her.

I filsystemer opererer vi med konseptene *root directory* og *current working directory*.

DEMO:

```
pwd; df -h; du -sh; mount; cat /etc/fstab
```

### Mounting a File System



I Windows har vi flere filsystemer som vises adskilt knyttet til stasjonsnavn (A:, B:, C:, ..., Z:) mens i Unix/Linux knyttes alt sammen til et filsystem som vist i figuren.

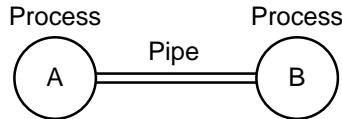
I Unix/Linux har man *block special files* og *character special files* som en del av filsystemet. Disse befinner seg i /dev directory, og representerer I/O enheter slik at man kan benytte filoperasjoner på I/O enheter. Hvis man akseresser data blokkvis (som for disker) så er det en block device mens hvis man akseresser en bytestream (characterstream) så er det en character device (f.eks. lydkort, mus).

DEMO:

```
ls -l /dev
```

### 3.3.3 Pipe

#### A Pipe



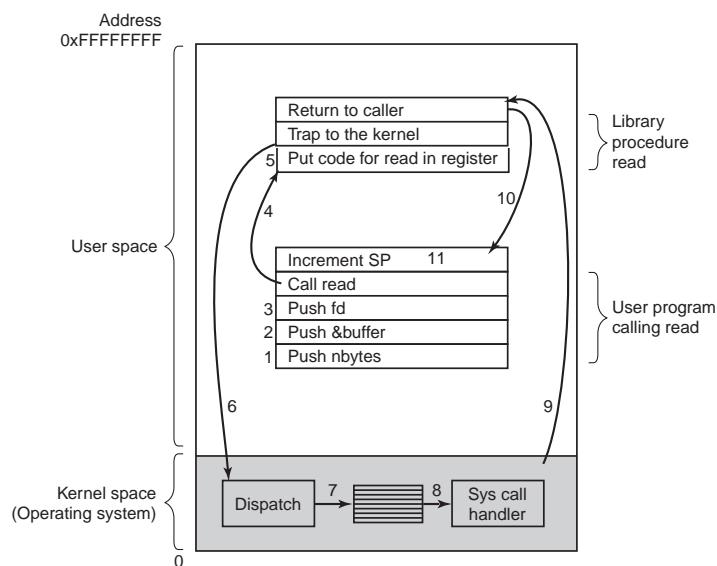
En *pipe* (rør) er en måte for to prosesser å kommunisere på, dvs sende data til hverandre. En pipe er en slags pseudofil, dvs for en prosess er å skrive til en pipe omtrent som å skrive til en fil.

DEMO:

```
ls -l /dev | wc -l
```

## 3.4 System Calls

### System Call: `read(fd,&buffer,nbytes)`



Systemkall er selve grensesnittet mot operativsystemet. Som regel benyttes ikke systemkallene helt direkte, men via et bibliotek (f.eks. libc). På dagens linux kan du få en mer direkte oversikt over systemkallene som tilbys i

`/usr/src/linux/include/linux/syscalls.h`

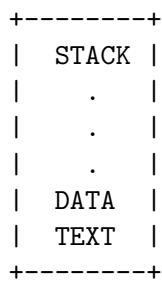
Et systemkall er altså en funksjon vi kaller når vi trenger operativsystemet til å gjøre noe, f.eks. I/O som å lese noe fra disk.

*Hvert systemkall innebærer en overgang fra user mode til kernel mode (og tilbake til user mode etterpå).*

I figuren er det vist hvordan dette skjer (dvs hvordan er trinnene i den ferdigkomplerte koden for et program som foretar et systemkall):

1. nbytes pushes på stacken
2. adressen (siden det står &) til buffer pushes på stacken
3. fd pushes på stacken
4. selve funksjonskallet `read`
5. funksjonen plasserer systemkallnummer (eller noe annet som identifiserer systemkallet) et sted hvor OSet forventer det (et register)
6. funksjonen utfører `trap` instruksjonen som forårsaker et skifte til kernel mode og et hopp til et bestemt sted i *kernel space*
7. OSet ser hvilke systemkallnummer som skal utføres og bruker dette som en index inn i en tabell med adresser til de respektive systemkallhandlene (dvs selve systemkallkoden i kjernen)
8. systemkall kjører
9. systemkall returnerer tilslutt til user mode funksjonen som kalte den opprinnelig
10. funksjonen returnerer til hovedprogrammet
11. hovedprogrammet rydder opp stacken ved å oppdatere stack pekeren (SP)

(tegn figur 1.20 hvis ikke gjort allerede)



DEMO:

```
strace ls
```

## A Simple Example

```
.data
str:
    .ascii "hello world"
.text
.globl main
main:
    movl    $4, %eax    # system call number
    movl    $1, %ebx    # file handle (stdout)
    movl    $str, %ecx # message to write
    movl    $11, %edx  # message length
    int     $0x80      # call kernel

    movl    $1, %eax    # system call number
    movl    $0, %ebx    # exit code
    int     $0x80      # call kernel
```

Se fil `asm-syscall.s` for litt mer kommentarer. Dette er et klassisk systemkall med bruk av assembly instruksjonen `int 0x80` dvs vi ”genererer et interrupt nr 80”. Idag brukes ikke nødvendigvis denne assembly instruksjonen siden det er andre liknende som er laget for å være kjappere, men prinsippene er de samme.

### 3.4.1 POSIX

#### POSIX System Calls

SEC. 1.6

SYSTEM CALLS

53

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

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

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

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

En oversikt over POSIX systemkallene. Merk at vi av og til kaller systemkall det som egentlig er c-bibliotek-wrapperne til systemkallene med samme navn.

### fork System Call

```
pid = fork( );           /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
    handle_error( );    /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
    /* parent code goes here. */
} else {
    /* child code goes here. */
}
```

Vi skal se nærmere på det systemkallet som lager prosesser på Unix/Linux: `fork`:

```
+-----+
| PARENT PROCESS      |
| PID=2313            |
|
| int p;
| p = fork();          |--- Systemkallet fork() skaper en prosess som
| if (p==0) //p=2321   |   | child av seg selv. Den nye child-prosessen
| printf("Child!");   |   | får tildelt sin egen PID (Prosess-ID),
| else                |   | f.eks. 2321. Slik at det fork() returnerer
| printf("Parent!");  |   | i parent-prosessen er PID'n til den nye
| exit(0);            |   | child-prosessen, altså 2321.
+-----+
|
|
|
+-----+
| CHILD PROCESS        |
| PID=2321,PPID=2313  |
|
| int p;               |   | Den nye child-prosessen har et minneområde
| p = fork();          |<-+ som er en eksakt kopi av parent-prosessen
| if (p==0) //p=0       |   | med et unntak: variabelen p (fork() sin
| printf("Child!");   |   | returnverdi), DENNE ER ALLTID 0 i child-
| else                |   | prosessen og er dermed den vi kan bruke
| printf("Parent!");  |   | for å skille mellom child og parent i koden
| exit(0);            |
```

```
+-----+ NB! første kodelinje som kjøres av child er
      altså if (p==0), dvs child arver verdien
      av parent sin IP (instruction pointer)
      (child arver også det meste av det andre i
      parent sin PCB (Process Control Block))
```

## DEMO

**fork0** enkel fork og de relevante PIDene

**fork1-getpid** hvordan vi kan bruke PIDene til å skille parent og child, samt systemkallet getpid

**fork2-fork-wait** fork-i-fork, hva blir utskriften, samt en wait

**fork3-sleep** kun mer fokus på wait, Merk: wait er altså en enkel form for *synkronisering*

**fork4-exec** exec erstatter altså hele minnområde til prosessen, mao en ny prosess bør lages ved en kombinasjoner av fork og exec (disse brukes i praksis nesten alltid i sammen)

## A Stripped-Down Shell

```
#define TRUE 1

while (TRUE) {
    type_prompt( );
    read_command(command, parameters);
    /* repeat forever */
    /* display prompt on the screen */
    /* read input from terminal */

    if (fork() != 0) {
        /* Parent code. */
        waitpid(-1, &status, 0);
        /* fork off child process */
        /* wait for child to exit */
    } else {
        /* Child code. */
        execve(command, parameters, 0);
        /* execute command */
    }
}
```

fork lager en eksakt kopi av seg selv, exec erstatter hele denne kopien med en nytt program.

### 3.4.2 WinAPI

#### Windows API Calls

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 umount
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

På Windows endres selve systemkallene ofte mellom ulike versjoner av windows og Microsoft anbefaler dermed at man ikke forholder seg direkte til dem, men heller til Win32 APIet som skal være stabilt. Dvs dette er tilsvarende mellomlag som vi akkurat så i Unix/Linux med et c-bibliotek (libc) som man bruker istedet for å bruke systemkallene direkte.

(Alikevel er det selvfølgelig noen som forsøker dokumentere systemkallene selv om de ikke er offisielt publisert fra Microsoft, se <http://j00ru.vexillium.org/ntapi/>)

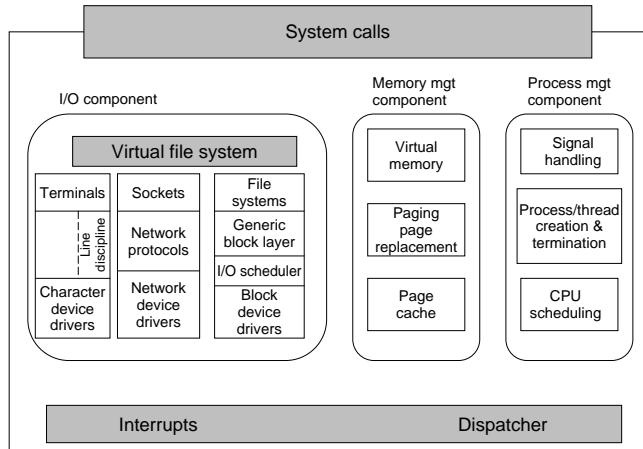
WinAPI er stort (over tusen funksjoner), og det er ofte litt uklart hva som er et systemkall (dvs utføres i kernel mode) og hva som ikke er det (WinAPI funksjoner som tidligere ble utført i kernel mode kan i ny windows versjon kanskje være implementert i user mode).

Figuren (og vi skal) dermed fokusere på de WinAPI funksjonene som noenlunde tilsvarer de vi skal se på i Unix/Linux verdenen.

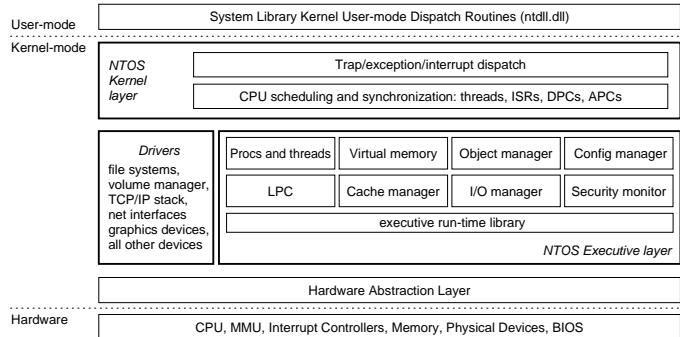
## 3.5 OS Structure

### 3.5.1 Monolithic

#### Linux

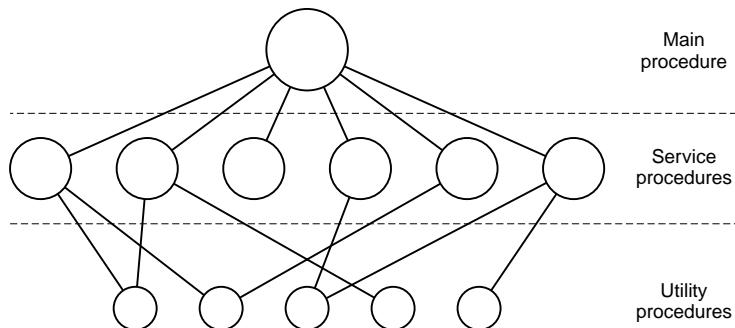


## Windows



Windows og Linux er begge i all hovedsak monolittiske operativsystemkjerner. Dette betyr at alle metoder i kjernen har tilgang til alle datastrukturer slik at kjernen kan kjøre mest mulig effektivt (dvs uten å måtte skifte mellom user mode/kernel mode hele tiden).

## Structure for a Monolithic System



Hvis alle deler av operativsystemet kompileres og linkes sammen til en stor eksekverbar binærfil, kalles dette en monolittisk struktur. Linux (OS-kjernen befinner seg i en fil som heter `mlinuz`) er et eksempel på dette. Windows (OS-kjernel befinner som i en fil som heter `ntoskrnl.exe`) er også i praksis dette, selv om den også kan kalles en hybrid mellom monolittisk og mikrokjerne.

I en monolittisk struktur kan det fortsatt være (og er selvfølgelig) en indre struktur som vi f.eks. kan se for oss som vist i figuren, hvor 'service procedures' er det kodesnuttene som utfører de respektive systemkallene.

Effektivt, men fare for rotete struktur og vanskelig å få "100%" stabilt.

### 3.5.2 Layered

#### Structure of the THE OS

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Lagdeling er et mye brukt konsept i informatikken (kanskje mest kjent fra nettverk med TCP/IP modellens 5 lag), og ble foreslått for operativsystemer av Dijkstra i 1968 i OSet fra THE (Technische Hogeshcool Eindhoven).

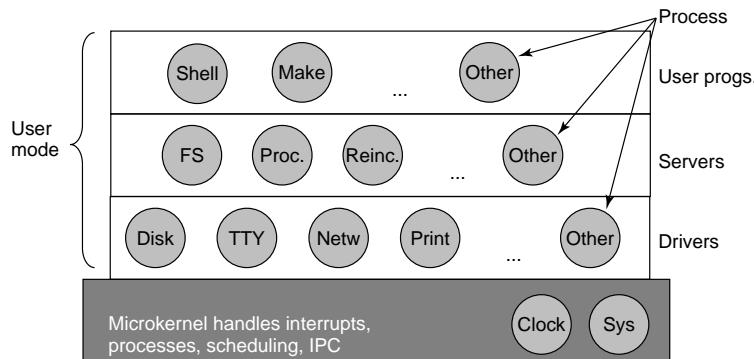
Laginndeling var først og fremst rettet mot et ryddig design av operativsystemet.

Hensikten med laginndeling er jo at hvert lag skal kunne kodes uavhengig av hverandre og at kommunikasjon mellom lagene bare skjer oppover eller nedover til nærmeste lag. I hvilken grad denne kommunikasjonen ble overholdt i THE OSet er noe uklart.

Men dette konseptet likner litt på ring-konseptet innført med MULTICS og vanlig på dagens x86 arkitektur.

### 3.5.3 Microkernel

#### Structure of MINIX 3



Hvis hele OSet kjører i kernel mode så kan en feil i en driver (f.eks. feil minneadressering i lydkortdriveren) føre til at OSet kræsjer. Studier viser at gjennomsnittlig antall feil pr 1000 linjer kode er ca 10. Dette betyr at et monolittisk OS med 5 millioner linje kode har ca 50.000 feil. Ikke alle disse er nødvendigvis alvorlige, de fleste er sikkert harmløse, men en god del vil kunne føre til systemkræsj.

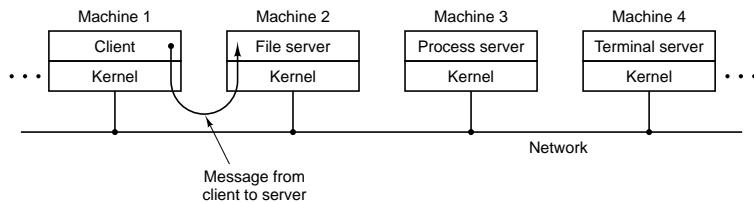
Poenget med Microkernel er å dele OSet opp i små veldefinerte deler hvorav kun kernel kjører i kernel mode.

Mikrokjernen i MINIX 3 er omrent 3200 linjer C kode og 800 linjer assembly kode. Figuren viser hvordan MINIX 3 er strukturert.

Mikrokjerne er spesielt aktuelt sikkerhets-kritiske systemer (rakettstyringssystemer o.l.).

### 3.5.4 Client-server

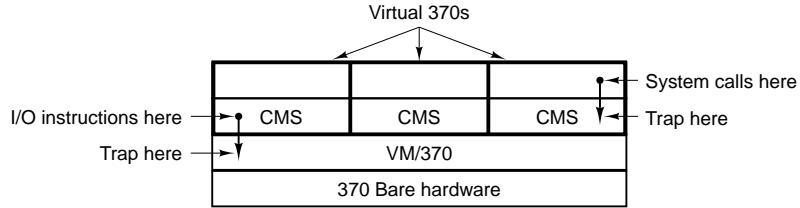
#### Client-Server Model



Modulene i et OS kan også arrangeres som en klient/tjener modell som vist i figuren, og dermed muliggjør et distribuert OS.

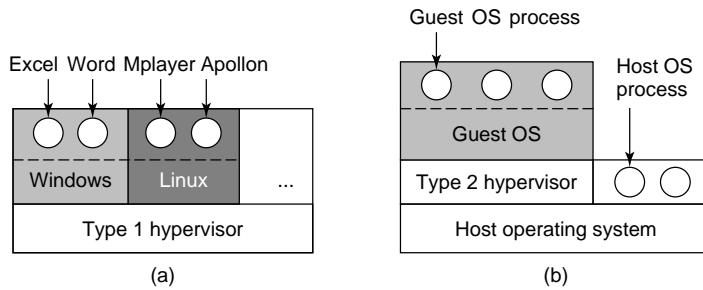
### 3.5.5 Virtual machines

#### Structure of VM/370 with CMS



IBMs VM/370 (fra 1972) var første populære hypervisor (opprinnelig kalt virtual machine monitor) og gjorde at det for gjesteOSene (CMS i figuren) så ut som de hadde hver sin fullblods System 370 maskin.

### Type 1 and Type 2 Hypervisors



På x86 arkitekturen har virtualisering vært uinteressant inntil sent på 90-tallet, fordi ytelsen har vært for dårlig og arkitekturen har ikke støttet virtualisering, dvs hvis et OS forsøker gjøre en instruksjon i user mode som den ikke skal gjøre så skal det forårsake en trap, dette skjer ikke på Intel og AMD arkitekturen. Men i Intel og AMDs hardwarevirtualiseringstøtte som kom i 2005 løses dette problemet ved å innføre et enda mer privilegert nivå enn kernel mode, dvs et 'kernel kernel mode' som en type 1 hypervisor kan kjøre i (slik at OS og brukerapplikasjoner kan kjøre som før i kernel mode og user mode).

(i praksis er dette femte nivået egentlig slik at alle de fire ringene/nivåene som vi er vant til (hvor kernel mode er ring 0 og user mode ring 3) kalles non-root mode, og det innføres en egen root mode som da en hypervisor kan kjøre i).

Derav utviklet vmware teknologien seg på slutten av 90-tallet, en type 2 hypervisor, mens klassisk type 1 hypervisor er blitt populært på x86 arkitekturen etter 2005.

Et annet alternativ er å modifisere gjesteOSet slik at det tilpasses hypervisoren, det kalles paravirtualisering og ble populært med Xen.

## 3.6 Theory questions

1. Hva menes med *preemptive* og *non-preemptive* operativsystemer?
2. Hva mener vi når vi sier at et OS er et *monolittisk system*? Nevn eksempel på OS med en slik design.
3. Hva mener vi når vi sier at en trend innen OS-design har gått i retning av *micro-kernel*? Hvilke fordeler gir ett slik OS-design?
4. Forklar sammenhengen mellom systemkall, kommandoer og instruksjoner. Gi eksempler.
5. Beskriv mest mulig detaljert hva som skjer når systemkallet `read(fd, &buffer, nbytes)` utføres.
6. Vi har gitt følgende c-program:

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

Hvilke utskrifter kan vi få når denne koden kjøres om vi forutsetter at `fork()` systemkallet ikke feiler?

### 3.7 Lab exercises

1. Last ned C-kode eksemplene fra forelesningen (fork eksemplene) og studer de nøyne.

2. **Kommandolinjeargumenter og systemkall**

Lag deg en directory execdemo og i denne lag en fil execdemo.c som inneholder:

```
#include <stdio.h>          /* printf */
#include <unistd.h>          /* execve */
int main (int argc, char *argv[]) {
    printf("Et program som lett tryner, men enkel demo av exec\n");
    execve(argv[argc-1], NULL, NULL);
    printf("Denne blir aldri skrevet ut med mindre execve feilet");
    return 0;
}
```

Kompiler (som i forrige punkt) og kjør med tre forskjellige kommandolinjeargumenter og tenk nøyne etter hva som skjer:

```
./execdemo ~/bin/hello
./execdemo /bin/ps ~/bin/hello
./execdemo ~/bin/hello /bin/ps
```

3. (Nå må vi også nevne at det er naturlig å lære seg verktøyet *make*, men det er opp til den enkelte av dere og vurdere om dere vil investere tid i)

# Chapter 4

## Prossesser og tråder

### 4.1 Outcome

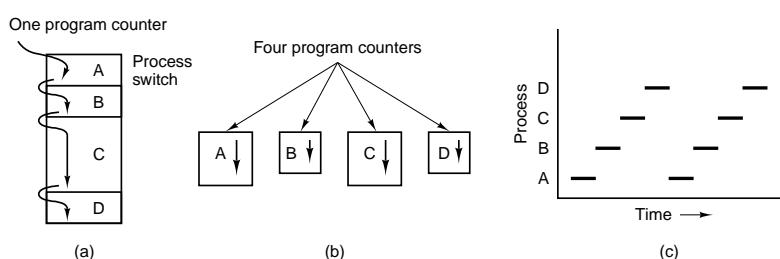
#### Today's Learning Outcome

- Parallel programming
- Troubleshooting
- Conceptual framework

### 4.2 Processes

#### 4.2.1 Model

##### Multiprogramming/Multitasking



Intro: Nå skal vi fokusere på hvordan vi deler CPU'n, men husk at det egentlig skjer mange oppgaver parallelt i en datamaskin. En datamaskin er egentlig et nettverk, det finnes mange prosessorer (stort sett en i hver kontroller, en DMA kontroller, GPU'n osv) men nå fokuserer vi mest på CPU'n på hovedkortet.

En prosess er et program under utførelse, tenk over følgende analogi: Du skal bake en kake og har en oppskrift og et sett med ingredienser (mel, egg, osv). A bake kaken er prosessen, du er CPUn, oppskriften er programkoden og ingrediensene er input-dataene til programmet. Mens du baker kommer plutselig en god venn av deg løpende inn på kjøkkenet med en nese som blør som en foss. Du må da skynde deg å stoppe kjøkkenmaskina di, sette unna ingrediensene og notere deg hvor langt du var kommet i oppskriften, slik at du kan fortsette å bake litt senere. Det å skifte ditt fokus over til å hjelpe å stappe papir opp i nesa på vennen din, er det samme som en *context switch* (kalle også av og til en *process switch*). Du må lagre unna alt du trenger for å starte på igjen senere, for å jobbe med noe annet (starte en annen prosess).

Å skifte mellom mange prosesser, dvs kjøre mange programmer "samtidig" kalles *multiprogramming/multitasking*. Dvs, strengt tatt kalles det å ha flere programmer i minnet samtidig (og disse får kjøre hver for seg inntil en av de blokkerer på I/O) for multiprogramming, mens det å fordele tiden mellom disse programmene (enten ved at disse frivillig gir bort CPU'en til hverandre eller operativsystemet aktivt fordeler tiden basert på tidsluker) kalles multitasking.

Figuren viser forskjellige måter å se for seg hva som skjer i prosessmodellen. Figur a viser hvordan instruksjonspekern (IP, også kalt Program Counter - PC) kan behandle programmene i fysisk minne, mens figur b viser hvordan vi kan tenke på dette som fire logiske programtellere (og det finnes en for hver prosess men bare en brukes om gangen). Figur c viser hvordan bare en prosess får CPU tid om gangen (siden det finnes bare en fysisk CPU i dette tilfellet).

Prosesser i Unix/Linux verdenen er organisert i et hierarki, se DEMO  
`pstree -p` og `procexp` i windows (se hvordan `procexp` er under powershell som er under GUI-shellet).

Dette er mye av historiske årsaker, dvs at den eneste måten å skape en prosess på er for en prosess å fork'e (som regel kombinert med exec). I Windows bruker man ikke dette hierarkiet (selv om det egentlig finnes der også), og fork/exec er slått sammen i `CreateProcess`.

TAVLE:

To typer prosesser:

- Brukerprosesser (vanlige brukerprogram)
- Service prosesser (daemons/services i Unix/Linux, kalles bare system services i Windows), *de prosessene som kan kjøre uten at en bruker er innlogget* (men de er ikke like batch prosesser fordi batch prosesser som regel er kortvarige prosesser som skal kjøres uten interaktivitet, mens service prosesser er langvarige prosesser som ikke har mye å gjøre, men skal være der hele tiden).

Vi husker `fork` demoene fra sist, parallel i windows er `win32CreateProcess.c`

MERK: en context switch er skifte mellom prosesser og tar mye tid, mens en *mode switch/transition* er når en prosess benytter seg av operativsystemet via systemkall (evn et interrupt/exception forekommer). Selv om det da er OSet som kjører, regnes det som prosessen kjører i kernel mode, se DEMO perfmon (Performance monitor - Add - ProcessorMedPilOpp - Add privileged og user time, flytt musepekern/flytt hele vinduet og se hvordan privileged grafen går opp for å håndtere interrupts siden det grafiske systemet til Windows OSet kjører i kernel mode).

### Process Creation

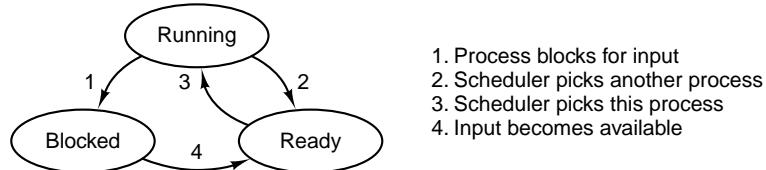
1. System initialization
  2. Execution of a process creation system call by a running process
  3. A user request to create a new process
  4. Initiation of a batch job
- 
1. mange prosesser startes ved oppstart, de fleste av disse kalles services, f.eks. en prosess som kjører periodiske jobber (cron i Unix/Linux, Task Scheduler i Windows)
  2. en prosess starter gjerne en annen prosess (`fork/execve` eller `CreateProcess`) hvis det er oppgaver den skal gjøre som kan skiller ut (f.eks. hente store mengder data fra nettet)
  3. brukere starter prosesser via kommandoer eller museklikk
  4. på batch-systemer starter prosesser også når nye batchjobber lastes

### Process Termination

1. Normal exit (voluntary)
  2. Error exit (voluntary)
  3. Fatal error (involuntary)
  4. Killed by another (involuntary)
- 
1. naturlig ferdig
  2. program velger å avslutte fordi f.eks. en fil som skal åpnes finnes ikke
  3. program avsluttet pga av en bug, f.eks. dele på 0 eller feil minneaksess

4. prosessen får et signal om å avslutte (kill i Unix/Linux, TerminateProcess i Windows), dette signalet kommer fra operativsystemet, men kan være iverksatt av en annen prosess. En prosess får som regel lov til å sende signal til en annen prosess hvis prosessen har samme eier (eller har eier tilsv administrator/root)

### Process States



Overgang 1 skyldes i all hovedsak at en prosess venter på I/O.

Overgang 4 skyldes i all hovedsak at en prosess ikke lenger venter på I/O.

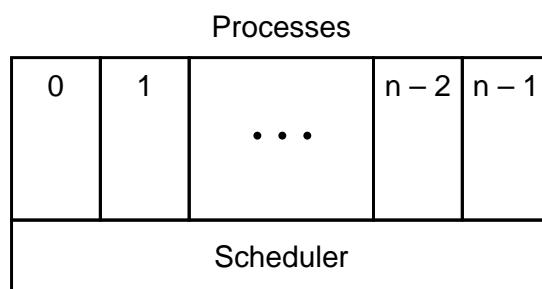
Overgang 2 og 3 skyldes scheduleren som bestemmer hvilken prosess som skal kjøres på CPUn.

DEMO: se PROCESS STATE CODES i `man ps` (Ready og Running er 'R', Blocked er 'D' eller 'S').

(i top shift-O og w enter for å sortere på status, deretter shift-r for å reversere å få de kjørende prosesser øverst, de med status 'R')

#### 4.2.2 Implementation

##### Simplified OS View



Dette bildet er viktig å tenke og reflektere litt over, det er slik et OS egentlig fungerer rent teoretisk, scheduleren sørger for å dele CPUn mellom prosessene. Det eneste som trengs er en mekanisme for at scheduleren får kjøre selv også, mellom hvert prosessbytte.

## Process Control Block (PCB)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Operativsystemet har en tabell med oversikt over alle prosesser (prosess tabellen), hvert entry i denne tabellen representerer en prosess og en slik entry kalles ofte en *process control block (PCB)*.

I PCBn finnes all informasjon om en prosess, figuren viser typisk hvilke entries vi finner i en PCB knyttet til prosesshåndtering, minnehåndtering og filhåndtering.

I praksis finnes ikke prosesstabellen som en tabell-datastruktur, men vi kan hente ut et snapshot med `top -b -n 1 > proctabell.txt`.

## Process Switch/Context Switch

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Husk figuren "Simplified OS View", her er det som skjer når det byttes mellom prosesser.

*Det viktigste å merke seg er at ved prosess bytte (også kalt context switch) så lagres prosessen unna i sin PCB og en ny prosess hentes inn fra sin PCB. En context switch tar typisk noen mikrosekunder.*

## Two Types of Processes

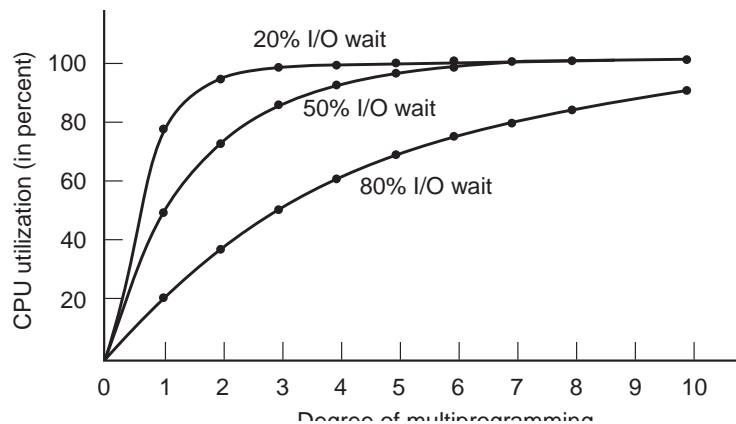
**CPU-bound** computing (scientific apps, multimedia) *Remember: hyperthreading doesn't help CPU-bound processes*

**I/O-bound** not much to do, mostly wait for I/O

Demo: `time regn.bash io.bash`

Hyperthreading gjør at det ser ut som vi har fire CPU'er men vi har fortsatt bare to ALU'er dermed kan bare to prosesser regne om gangen. Men har vi mange I/O-bound prosesser så får vi et mer effektivt system siden vi kan kontekstsvitsje mye raskere mellom dem med hyperthreadede CPU-kjerner.

## Multiprogramming/Multitasking Model



Her har vi en statistisk modell som kan brukes til å studere CPU-utnyttelse.

Anta at alle prosessene utnytter CPUn 20% av tiden, har vi da 100% CPU utnyttelse med 5 prosesser? NEI, fordi flere prosesser kan jo vente på I/O samtidig! Dette kan vi modellere slik som i figuren:

$n$  er antall prosesser i minne.

$p$  er tiden en prosess venter på I/O (i gjennomsnitt for alle prosesser).

Dette gir oss: CPU-utnyttelse =  $1 - p^n$

Denne modellen forutsetter at prosessene er uavhengige. Den illustrerer hovedpoenget og gir oss innsikt, men er ikke helt presis siden en helt presis modell ville vært for komplisert å studere her.

Når vi tenker på at en prosess som venter på at du skal taste inn noe er i I/O wait tilstand, så ser vi at 80% I/O wait eller mer ikke er unormalt.

Vi kan knytte denne modellen mot praktiske vurderinger som følger:

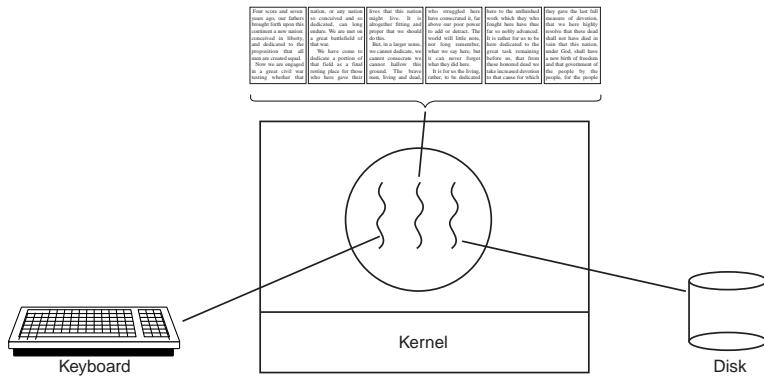
Gitt en datamaskin med 512MB minne, hvor hver prosess (inkl OSet) tar 128MB minne. Da er det i teorien plass til 3 brukerprosesser samtidig i minne og hvis vi antar at hver prosess venter 80% på I/O så vil CPU-utnyttelse være  $1 - 0,8^3 = 0,488 \sim 49\%$ , mens ved å øke minne til 1024MB blir det plass til 4 brukerprosesser til og vi får en CPU-utnyttelse på  $1 - 0,8^7 \sim 79\%$  som er en økning i utnyttelse på 30 prosentpoeng mens hvis vi legger til enda 512MB minne slik at vi får 1536MB minne økner utnyttelsen bare med 12 prosentpoeng (regn ut og sjekk!). Dermed kan vi kanksje si at det er økonomisk fornuftig å gå fra 512MB til 1024MB minne men ikke helt opp til 1536MB (selvfølgelig veldig avhengig av situasjonen og priser).

*Men i praksis ønsker du mest mulig RAM (internminne) uansett pga at ledig RAM benyttes som cache mot disk.*

## 4.3 Threads

### 4.3.1 Usage

#### Word Processor with Three Threads



Tråder kalles ofte lett-vekts prosesser og har samme hensikt som prosesser: muligheten til å skape parallelitet. Årsakene til at vi har tråder i tillegg til prosesser er:

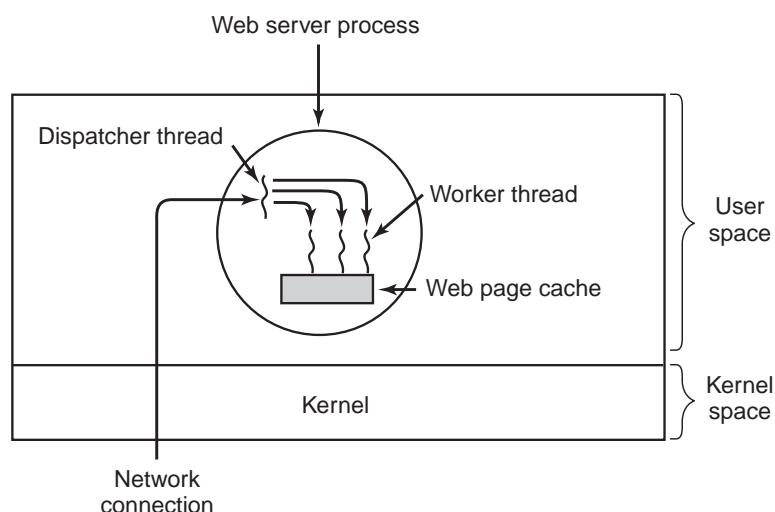
- trådene deler samme adresserom siden de tilhører samme prosess
- å lage en tråd er 10-100 ganger så raskt som å lage en prosess
- (en prosess med flere tråder kan utnytte multiprosessorsystemer)

Eksempelet i figuren er en tekstbehandler med tre tråder:

1. En tråd tar imot input fra brukeren
2. En tråd reformaterer dokumentet så fort det trengs
3. En tråd tar periodisk automatisk backup

Det hadde ikke vært holdbart om brukeren måtte vente på reformatering eller automatisk backup hver gang det skulle skje.

### Multithreaded Web Server



En webserver bruker flertrådmodellen for å håndtere innkommende forespørsler om websider. En "hovedtråd" (dispatcher thread) tar imot henvendelser og starter en working thread for hver henvendelse, working thread'ene må rett som det her hente noe fra disk og dermed blokkere på I/O, som gjør at andre tråder bør kjøre. Dermed blir det en effektiv webserver.

DEMO: pstree -p (trådene har klammeparenteser, prosessene hakeparenteser). Tilsvarende i Windows med procexp

### Code for Multithreaded Web Server

```

while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}

while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}

```

(a) (b)

Websverven kan kodes omrent slik, med dispatcher tråden til venstre og working thread til høyre, merk at kode er lik for alle working threads.

*Hvis vi har tilgjengelig en ikke-blokkerende read systemkall, så kan vi bruke endelig-tilstands maskiner istedet for tråder, men dette er veldig vanskelig å programmere.*

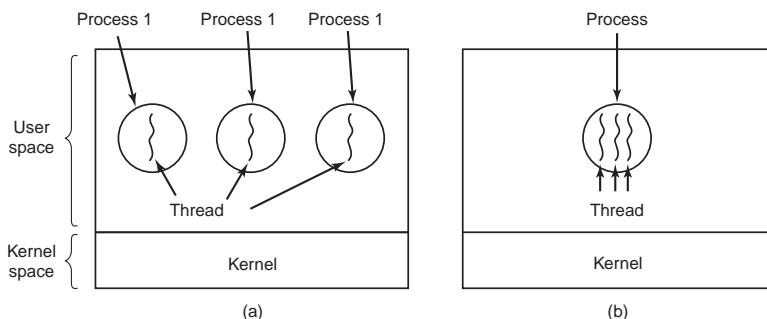
### Three Ways to Construct a Server

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Med endelig-tilstands maskiner måtte altså tilstanden lagres for hver gang man kjør et ikke-blokkerende systemkall, og man måtte håndtere signaler for når dette systemkallet var fullført, det er enklere med tråder.

#### 4.3.2 Model

##### Thread Model



Merk forskjellen mellom prosesser og tråder: her er det altså til venstre tre en-tråd prosesser med hvert sitt adresserom, mens til høyre en prosess med tre tråder som deler ett adresserom.

Scheduling vil da foregå på to nivåer: en mellom prosesser og en på et lavere nivå mellom trådene i prosessen. Dette er den klassiske modellen, i praksis gjøres det litt annerledes som vi snart skal se i diskusjonen om kernel-level vs user-level tråder.

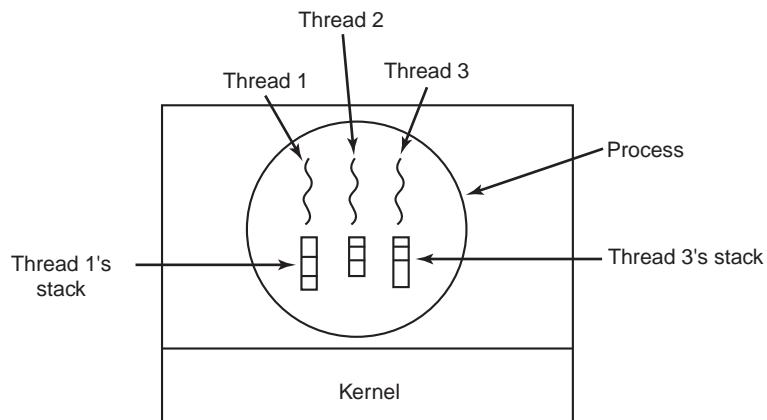
### Thread Items

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Stacken til en tråd inneholder da lokale variable og returadresser for funksjoner i tråden.

Merk: det er ingen beskyttelse mellom tråder siden de deler samme adresserom, en tråd kan overskrive minne til en annen tråd i samme prosess. Men dette er jo naturlig siden trådene er laget av samme prosess og da skal samarbeide.

### Each Thread Has Its Own Stack



Tenk over hvorfor: tråder er innenfor samme prosess og har derfor samme globale variable, men tråder gjør gjerne forskjellige oppgaver og det de gjør forskjellig er å bruke forskjellig funksjoner noe som innebærer at de har egne lokale variable og returadresser som nettopp er den informasjonen som ivaretas på en stack: derav har tråder egen stack.

#### 4.3.3 POSIX

##### Pthread Function Calls

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

POSIX standard biblioteket for tråder er Pthread.

Merk det spesielle kallet `pthread_yield`: dette finnes fordi tråder samarbeider og programmeren kan bestemme at en tråd skal gi fra seg CPU i bestemte kodesnutter. Mellom prosesser finnes ikke dette siden prosesser antas å ville ha så mye CPU som mulig.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread %d, tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d, i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

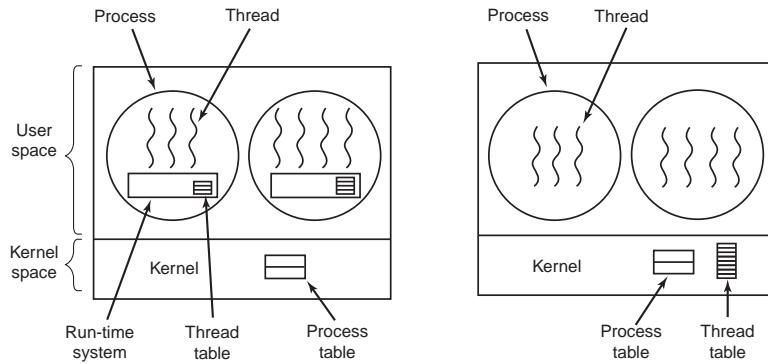
        if (status != 0) {
            printf("Oops. pthread_create returned error code %d, status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

DEMO av denne koden:

`thread0.c` (hvorfor blir utskriften forskjellig hver gang?)

#### 4.3.4 User-level vs Kernel-level

##### User-level vs Kernel-level



Skål tråder være synlig for OSet? eller skal vi la tråder bare være en ren applikasjonsgreie i userspace som OSet ikke bekymrer seg for?

Ved user-level tråder kan vi bruke tråder på et OS som ikke støtter tråder. Hver prosess må da ha sitt eget *run-time-system* som har en tråd-tabell tilsvarende OSet sin prosesstabell run-time-systemet er da inne i bilde ved scheduling av tråder: når en tråd frivillig har gitt opp CPU'n. User-level tråder har følgende fordeler/ulemper:

- + svitsje mellom tråder er kanskje 10x raskere enn ved kernel-level fordi man unngår overgang til kernel-level, dvs man unngår mode switch/transition)
- + prosesser kan ha sin egen schedulingalgoritme (samtid benytte tråder på OSer som ikke støtter det som nevnt over)
- – ved blokkerende I/O kall blokkeres hele prosessen
- – ved hver page fault blokkeres prosessen
- – tråder må gi opp CPU'n frivillig siden det ikke er noen naturlig klokkeinterrupt på user-level
- – tråder benyttes som regel mest for applikasjoner hvor det er mye blokking basert på I/O

Det siste argumentet er tungveiende og gjør at user-level tråder er uvanlig.

Ved kernel-level tråder (som er det vanlige) betraktes trådene i all hovedsak som prosesser og OSet har det run-time-systemet som vi måtte ha i hver prosess ved user-level tråder. OSet har da trådtabellen (og hver tråd har en TrådID akkurat som prosesser har en PID). Fordeler og ulemper er:

- + når en tråd blokkerer kan OSet kjøre en annen tråd fra samme prosess (dvs ingen blokking av andre tråder i samme prosess)
- – tar lengre tid å skifte mellom tråder

- – tar lengre tid å opprette/slette tråder (derav forsøker en del systemer å re-sirkulere tråder, dvs merke de som 'not-runnable' uten å slette datastrukturen)

POSIX biblioteket Pthread som vi nettopp nevnte kan benyttes til både user-level og kernel-level tråder men som regel kjører vi det alltid på et OS som støtter kernel-level tråder (slik som Windows og Unix/Linux) og da bruker vi automatisk kernel-level tråder.

Til slutt: bildet er litt mer komplisert enn bare prosesser og tråder, og skillet mellom dem er mørk uklart i praksis selv om prinsippene er slik vi nettopp har gjennomgått, se i boka side 860-861 for hvordan dette er på Windows og les fjerde paragraf på side 746 "In 2000, ..." for systemkallet `clone` på Linux.

## 4.4 Theory questions

1. Hva er en prosess i et operativsystem?
2. Beskriv tre viktige tilstander en prosess kan befinne seg i.
3. Hva er en prosesskontrollblokk (PCB) og hva inneholder den?
4. Lag en liste over egenskaper som deles av tråder i en prosess og egenskaper som er unike for hver tråd.
5. Hva skjer om vi får et blokkerende I/O-kall i en tråd på user level?
6. Forklar hvilke (og hvor mange) trådtabeller som finnes når:
  - tråder kjøres på user level?
  - tråder kjøres på kernel level?

## 4.5 Lab exercises

### 1. Lage nye prosesser og enkel synkronisering av disse.

Lag et C-program som starter seks prosesser i henhold til følgende tidsskjema (S betyr start, T betyr terminer/exit):

```

Prosess-
nummer
^
5 |           S-----T
4 |   S-----T
3 |       S----T
2 S-----T
1 |   S----T
0 S--T
+-----> tid i sekunder
0  1  2  3  4  5  6  7

```

Det eneste hver prosess skal gjøre er å kjøre følgende funksjon:

```

void process(int number, int time) {
    printf("Prosess %d kjører\n", number);
    sleep(time);
    printf("Prosess %d kjørte i %d sekunder\n", number, time);
}

```

Bruk systemkallet waitpid for å synkronisere (dvs vente med å starte en prosess til en annen er terminert).

### 2. To prosesser som skal inkrementere en global variabel.

Kjør følgende program:

```

#include <stdio.h>      /* printf */
#include <stdlib.h>      /* exit */
#include <unistd.h>      /* fork */
#include <sys/wait.h>     /* waitpid */
#include <sys/types.h>    /* pid_t */

int g_ant = 0;          /* global declaration */

void writeloop(char *text) {
    long i = 0;
    while (g_ant < 30) {
        /* print and increment g_ant only each 100000 iteration */

```

```
if (++i % 100000 == 0) /* % is the same as modulo */
    printf("%s: %d\n", text, ++g_ant);
}
}

/* Note: the following code does not do proper error checking
   of return values from functions, so this is not robust code,
   but coded this way to be easy to read */

int main(void)
{
    pid_t pid;

    pid = fork();
    if (pid == 0) {           /* child */
        writeloop("Child");
        exit(0);
    }
    writeloop("Parent");    /* parent */
    waitpid(pid, NULL, 0);
    return 0;
}
```

Hvordan telles variabelen g\_ant? Forklar kort hva som skjer.

### 3. To tråder som skal inkrementere en global variabel.

Kjør følgende program (merk: når du kompilerer programmer som benytter pthreads-tråder så må du angi pthreads-library når du kompilerer: gcc -Wall -pthread -o prog prog.c):

```
#include <stdio.h>    /* printf */
#include <stdlib.h>   /* exit */
#include <pthread.h>  /* pthread_t pthread_create pthread_join */

int g_ant = 0;          /* global declaration */

void *writeloop(void *arg) {
    long i = 0;
    while (g_ant < 30) {
        if (++i % 1000000 == 0)
            printf("%s: %d\n", (char*) arg, ++g_ant);
    }
    exit(0);
}
```

```
int main(void)
{
    pthread_t tid;
    pthread_create(&tid, NULL, writeloop, "2nd thread");
    writeloop("1st thread");
    pthread_join(tid, NULL);
    return 0;
}
```

Hvordan telles variabelen g\_ant? Forklar kort hva som skjer.



# **Chapter 5**

## **Prosesskommunikasjon, samtidighet og synkronisering**

### **5.1 Outcome**

#### **Today's Learning Outcome**

- Parallel programming
- Performance
- Security
- Conceptual framework

### **5.2 Introduction**

#### **Three Issues**

1. How can one process/thread pass information to another?
2. How can multiple processes/threads avoid getting in each others way?
3. How can we make sure multiple processes/threads run in the proper sequence (with respect to each other)?

*Most of what chapter 2.3 states about processes also applies to threads and vice versa.*

Merk: vi skal altså se på mekanismer for å håndtere disse problemstillingene og disse samme mekanismene kan anvendes som regel enten på tråder eller på prosesser.

Det du lærer i dette temaet er viktig i to sammenhenger: 1. Dette er noen av de vanskeligste problemene operativsystemet må løse (altså hvordan funker et operativsystem), 2. Dette er problemstillinger du vil møte på i en lang rekke andre sammenhenger som programmerer. Med andre ord, disse problemene finnes både i operativsystemet og i applikasjonene.

Men la oss bare nevne kort litt terminologi først.

### 5.2.1 Atomicity

#### Atomicity/Atomic Operation

- *Indivisible*
- Certain low-level operations cannot be intercepted, they must be performed as one uninteruptable instruction, these are called atomic operations

Atomisk betyr altså udelelig.

### 5.2.2 Deadlock

**Deadlock** When no processes/threads can proceed because everyone is waiting for an event that only one of them can trigger.

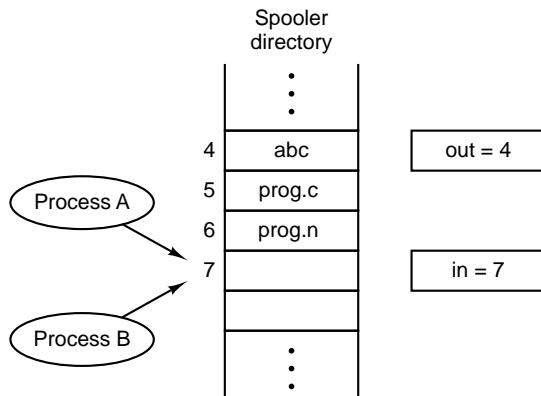
Deadlock (vranglås) betyr at vi har en låst situasjon hvor alle venter på hverandre, tenk et trafikkryss hvor alle har vikeplikt for hverandre og alle bare står og venter på hverandre.

### 5.2.3 Starvation

**Starvation** A process that is ready to run but is never given the CPU due to scheduling mechanisms and/or priority ("an overlooked process").

### 5.2.4 Race Conditions

#### Race Conditions



Problemet med delte variable i intern minnet: variablene in og out.

Prosess A og B ønsker begge å skrive ut mens en daemonprosess jobber i bakgrunnen med å foreta selve utskriftene slik at A og B bare behøver legge sine utskriftsjobber i køen. out indikerer hvilken jobb daemonprosessen skal printe ut som neste jobb. in indikerer neste ledige plass i køen (dvs der prosess kan legge sin utskriftsjobb).

Følgende skjer:

1. A leser in lik 7
2. Scheduleren skifter til B
3. B leser in lik 7, lagrer sin utskriftsjobb i plass 7, inkrementerer in, og fortsetter med andre instruksjoner inntil den blir avbrutt
4. A får etterhvert tilbake CPUn, lagrer sin utskriftsjobb i plass 7 (og skriver dermed over B's utskriftsjobb!), inkrementerer in, og fortsetter med andre instruksjoner inntil den blir avbrutt

*Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when are called **race conditions**.*

(debugging av slike programmer er et mareritt...)

(kommenter sammenheng med sikkerhet)

**A Common Example 1/2** What really happens when you run the code `i++;` ?

```
register1 = i;
register1 = register1 + 1;
i = register1;
```

*Assume i=5 is a global variable, let's look at what can happen when thread T1 does `i++;` and thread T2 does `i--;`*

i er en variabel som befinner seg i minne, før CPUn kan gjøre noe med den må den leses av og plasseres i et register. Så behandles den, før den skrives tilbake til minne.

### A Common Example 2/2

```

T1: register1 = i;           [register1 = 5]
T1: register1 = register1 + 1; [register1 = 6]

<scheduler switches thread>

T2: register2 = i;           [register2 = 5]
T2: register2 = register2 - 1; [register2 = 4]

<scheduler switches thread>

T1: i = register1;          [i = 6]

<scheduler switches thread>

T2: i = register2;          [i = 4]

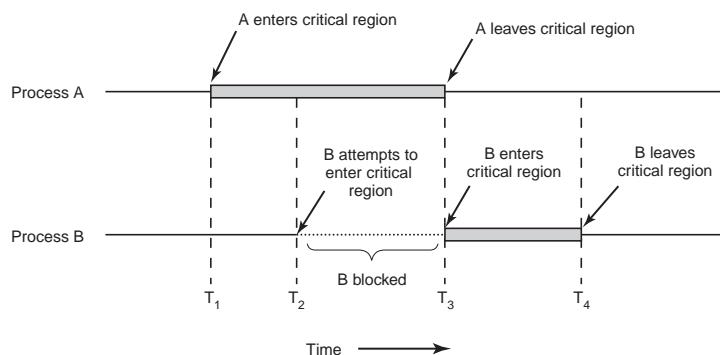
```

$i$  ender opp med verdien 4, den skulle blitt 5, men kunne altså blitt både 4, 5 og 6 avhengig av rekkefølgen trådene T1 og T2 får kjøre.

Demo: incdec.c

### 5.2.5 Critical Region/ Mutual Exclusion

#### Critical Region/Mutual Exclusion



Hvordan unngå race conditions? Jo vi må sørge for at bare en prosess kan benytte en delt variabel om gangen. Vi kan løse dette ved å dele inn koden vår i *kritiske sektorer* (der hvor felles variable behandles og race conditions kan oppstå).

## Four Conditions

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

En mulig løsning er å disable interrupts når man går inn i kritisk sektor, men dette er skummelt av to årsaker:

1. OSet kan disable interrupts, men en brukerprosess kan ikke få lov til det siden vi da blir avhengig av at den slår på igjen interrupts
2. disabling interrupts påvirker bare en CPU så dette vil ikke funke på fler-CPUs, mao brudd på betingelse 2 over

OSet benytter seg av av og til av disabling av interrupt selv når det skal oppdatere egne variable, men dette gjøres i stor grad bare på embedded systemer hvor det er single-CPU.

I praksis låser man minnebussen istedet for å disable interrupts når man ønsker eksklusiv tilgang til minne i en kort periode.

## 5.3 "Theory Solutions"

### 5.3.1 Non-Solutions

#### Solution with Strict Alternation?

```
while (TRUE) {  
    while (turn != 0) /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1) /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)

Vi kan løse problemet med eksklusiv tilgang til kritisk sektor ved å bruke en variabel *turn* (0 eller 1) som sier hvem sin tur det er til å gå inn i kritisk sektor. Den som det ikke er sin tur må vente og konstant teste innholdet i variablene *turn* inntil den endrer verdi, det kalles *busy waiting* og variablene *turn* kalles en *spin lock*.

Alternativet til busy waiting er å blokkere (og dermed context switch'e), og valget mellom busy waiting eller å blokkere er relevant på multiprosessor/multicore CPU'er hvor man ofte kaller denne problemstillingen *spin or switch*. Dvs busy waiting/spinning er helt ok hvis ventetiden er kort siden blokering medfører context switch som også er kostbart.

Dette fungerer mtp ekslusiv tilgang til kritisk sektor, men det er et problem: Det er streng alternering, dvs de to prosessene må få hver sin tur. Hvis den ene prosessen er mye raskere enn den andre, kan den bli utestengt fra kritisk sektor fordi den andre er treg i ikke-kritisk sektor, og dette er brudd på betingelse 3.

Mao, hvis vi går tilbake til eksempelet med spooling av filer for printing, så kan ikke en prosess printe to filer etter hverandre ...

### Solution with Array Instead?

```
/* Process 0 */

while(TRUE) {
    interested[0]=TRUE;
    while(interested[other]==TRUE) /* loop */;
    critical_region();
    interested[0]=FALSE;
    noncritical_region();
}
```

La oss løse problemet med streng alternering ved å ikke kreve streng alternering, vi bytter ut turn variabelen med et to-dimensjonalt array hvor hver prosess kan ha sitt element hvor de sier om de er interessert i å gå inn i kritisk sektor eller ei. Denne løsningen garanterer også ekslusiv tilgang (betingelse 1). La oss se på et tilfelle (TAVLE):

1. prosess 0 setter sin interesse til TRUE
2. scheduleren flytter CPUn til prosess 1
3. prosess 1 setter sin interesse til TRUE og går inn i spinlock
4. scheduleren flytter CPUn til prosess 0
5. prosess 0 går inn i spinlock
6. oisann ...

Vi får en deadlock, to prosesser som venter på hverandre, dette kommer vi tilbake til senere i emnet.

Men kanksje vi skulle prøve å kombinere disse? dvs turn og interested slik at vi kan unngå streng alternering men også unngå deadlock?

### 5.3.2 Peterson's

#### Peterson's Solution

```
#define FALSE 0
#define TRUE 1
#define N      2           /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process);    /* process is 0 or 1 */
{
    int other;                /* number of the other process */

    other = 1 - process;      /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;           /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process)     /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Peterson's løsning fra 1981 (som er en forenkling av Dekker's løsning fra 1965) er en løsning som ikke krever streng alternering, dvs dette er en løsning som oppfyller de fire betingelsene nevnt tidligere (altså en godkjent løsning).

Hvis både prosess 0 og prosess 1 kaller `enter_region` samtidig så vil ved worst-case f.eks. prosess 0 rekke å sette `turn` lik 0, før prosess 1 får CPUen og skriver over `turn` med 1. Da vil prosess 0 gå inn i kritisk sektor (virker kanskje litt unaturlig siden `turn` er lik 1 men det slik det funker) mens prosess 1 må vente inntil prosess 0 har kjørt `leave_region`.

#### Peterson's (fra Silberschatz)

```

#define FALSE 0
#define TRUE 1
#define N    2           /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;             /* number of the other process */

    other = 1 - process;   /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process; other; /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}

```

Peterson's er egentlig litt lettere å forklare hvis vi ser på den slik som enkelte andre lærebøker gjør (e.g. Silberschatz et al.) hvor vi bytter bare om behandlingen av turn variabelen som vist i figuren.

Vi kan prøve forklare det slik:

1. begge har satt interested til TRUE (begge sier "jeg er interessert")
2. prosess 0 setter turn til 1 ("jeg vil la kameraten min kjøre først")
3. scheduleren flytter CPU til prosess 1
4. prosess 1 setter turn til 1 ("jeg vil la kameraten min kjøre først") og går inn i spinlock ("jeg venter til det blir min tur eller kameraten min ikke er interessert lenger")
5. scheduleren flytter CPU til prosess 0
6. prosess 0 går inn i kritisk sektor ("det er blitt min tur jo, da gir jeg ikke vente")
7. prosess 0 blir ferdig med kritisk sektor og setter sin interesse til FALSE dermed kjører prosess 1

og prosess 1 kan kjøre flere ganger ("printe flere filer etter hverandre") den siden prosess 0 må sette sin interesse til TRUE for at den skal bli med i alterneringen.

Dermed har vi en løsning som funker.

## 5.4 “Practical Solutions”

### 5.4.1 HW: TSL

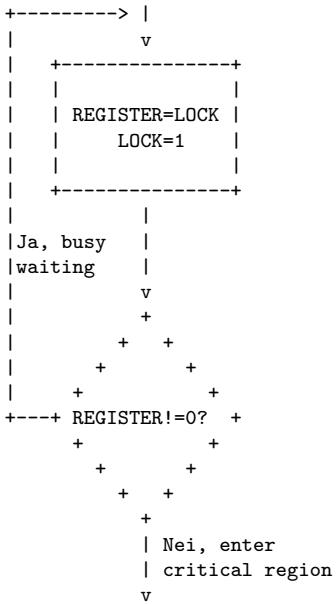
#### Critical Region with TSL

```
enter_region:  
    TSL REGISTER,LOCK  
    CMP REGISTER,#0  
    JNE enter_region  
    RET  
          | copy lock to register and set lock to 1  
          | was lock zero?  
          | if it was nonzero, lock was set, so loop  
          | return to caller; critical region entered  
  
leave_region:  
    MOVE LOCK,#0  
    RET  
          | store a 0 in lock  
          | return to caller
```

Et problem som kan forekomme med software-løsninger som Peterson’s er at måten moderne datamaskinarkitekturen realiserer maskinkode på gjør at instruksjonene ikke nødvendigvis er de samme enhetene som de vi behandler i software, dermed bør vi tilstrebe en hardware-støttet løsning istedet.

Vi kan gjøre det enklere enn Peterson’s løsning ved å bruke hardware-støtte (dvs bruke en instruksjon som finnes i instruksjonssettet til datamaskinarkitekturen), siden det vi egentlig ønsker oss er bare en lock variabel som vi kan sette/”låse” i en atomisk operasjon (dvs uavbrutt operasjon, se tilbake på eksempelet med register1 og register2).

Et alternativ til Peterson’s er å bruke den hardware-støttede instruksjon TSL (Test and Set Lock) istedet. *Husk at minne kan være delt, mens registerinnholdet er spesifikt for hver prosess/tråd.* Nøkkelen med TSL er at den er atomisk, dvs lesing av innholdet i en variabel i minne, kopiering av denne til et register, og overskriving av denne i minne, disse tre operasjonene vil alltid skje som en uavbrutt operasjon (fordi minnebussen låses i hardware under denne instruksjonen). Og dermed kan enter\_region og leave\_region implementeres som vist i figuren.



### 5.4.2 HW: XCHG

#### Critical Region with XCHG

<pre> enter_region:     MOVE REGISTER,#1     XCHG REGISTER,LOCK     CMP REGISTER,#0     JNE enter_region     RET </pre>	put a 1 in the register   swap the contents of the register and lock variable   was lock zero?   if it was non zero, lock was set, so loop   return to caller; critical region entered
<pre> leave_region:     MOVE LOCK,#0     RET </pre>	store a 0 in lock   return to caller

XCHG er bare en alternativ måte å bruke en lock variabel på iif TSL, hensikten er den samme.

### 5.4.3 Busy Waiting Problem

**Priority Inversion Problem** Consider two processes H (high priority) and L (low priority).

The scheduling rules are such that H is run whenever its in ready state.

While L in its critical section, H becomes ready (e.g. I/O completed).

*H starts busy waiting and loops forever ...*

Dette kan være en konsekvens av å velge busy waiting som strategi, dvs uante konsekvenser som denne kan forekomme. For et eksempel fra virkeligheten se

[http://research.microsoft.com/en-us/um/people/mbj/Mars\\_Pathfinder/Authoritative\\_Account.html](http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html) (se på andre avsnitt i "The Failure").

Busy waiting bør som regel unngås siden det er bortkastet CPU tid, la oss heller se på muligheten for å blokkere en prosess istedet for å la den bedrive busy waiting.

#### 5.4.4 "Sleep and Wakeup"

```
#define N 100           /* number of slots in the buffer */
int count = 0;          /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();           /* repeat forever */
        if (count == N) sleep();         /* generate next item */
        insert_item(item);              /* if buffer is full, go to sleep */
        count = count + 1;              /* put item in buffer */
        if (count == 1) wakeup(consumer); /* increment count of items in buffer */
    }
}

void consumer(void)
{
    int item;   context switch
    while (TRUE) {           /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, got to sleep */
        item = remove_item();  /* take item out of buffer */
        count = count - 1;    /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);   /* print item */
    }
}
```

Istedet for busy waiting ønsker vi ofte bare å blokkere prosesser (*switch* istedet for *spin*), dette kan vi se for oss at kan gjøres med et systemkall `sleep()` hvor blokeringen fjernes ved at en annen prosess sender en `wakeup(PID)`.

Producer-Consumer er et litt generalisert problem som likner på printer-spooler problemet vi så på tidligere. To prosesser deler et buffer (hvor nøkkelen er den delte variablen `count`) hvor den ene prosessen (producern) legger til enheter og den andre (consumern) fjerner enheter. Hvis bufferet er full må producern vente, og hvis det er tomt så må consumern vente. Tenk på enhetene som utskriftsjobber, producern som en MSWord-tråd og consumern som en printerdaemon.

Figuren over viser nok et tilfelle av race condition ved at tilgang til `count`-variabelen er ikke beskyttet med noe kritisk sektor. Hvis `count` er 0 og consumern kjører og det blir context switch (schedulern flytter CPU til producern) der som det er markert i figuren, vil producern etterhvert sende et `wakeup` signal som consumern ikke vil ta

imot fordi den har ikke kjørt `sleep()` enda (consumer vil sove, og snart vil producer sove også). Dette er altså en *race condition*, siden variablen `count` kan endres mellom Time-Of-Check (if-statementet) og Time-Of-Use (funksjonskallet `sleep()`).

(det er også mange andre race conditions her siden ingen kritisk sektor er beskyttet: det kan forekomme at `count` har feil verdi hvis det skjer en context switch mellom `insert_item` og den påfølgende inkrementeringen av `count`, eller innenfor inkrementeringen av `count` som i eksempelet med `register1` og `register2`, og tilsvarende i consumer.)

Dette kan løses ved å innføre en variabel som tar vare på et wakeup-waiting-bit som `sleep` sjekker før den blokkerer, men dette er en lite generell løsning (dvs, løsningen funker ikke for mer enn to prosesser, den *skaleres ikke* til flere prosesser). Hva hvis vi bruker en spesiell type variabel som teller antall wakeup-waiting-bit stedet? Jo, det er løsningen og det kalles et *semafør*. La oss se på hva som er så spesielt med det.

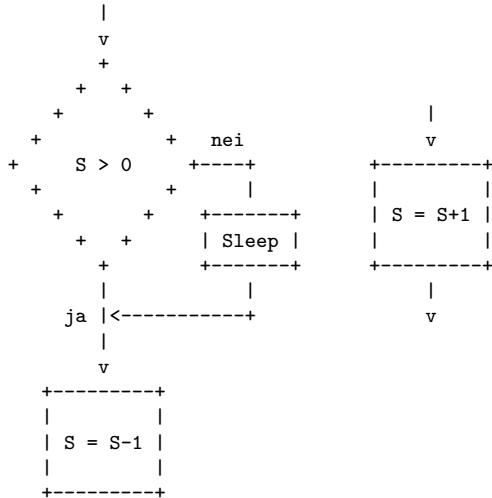
#### 5.4.5 OS: Semaphores

**Semaphore** “A special kind of an `int`”:

- Counts *up* and *down* atomically
- If a process/thread does a *down* on a semaphore which is 0, it is blocked (placed in a waiting queue)
- If a process/thread does an *up* on a semaphore which is 0, one of the processes/threads is removed from the waiting queue (becomes unblocked)

Semaforer har to tilhørende funksjoner down og up:

down virker slik:



up virker slik:



(Merk: avhengig av definisjonen av semafor så kan en semafor ha en negativ verdi, den negative verdien vil da indikere hvor mange prosesser/tråder som sover på den.)

Hvis en eller flere prosesser/tråder sover på en semafor, så vil en av de vekkes og fullføre sin down operasjon. Etter en up kan semaforen stadig være 0, men det er en færre prosess som sover (dvs blokkerer) på den.

```

#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        /* generate something to put in buffer */
        down(&empty);
        /* decrement empty count */
        down(&mutex);
        /* enter critical region */
        insert_item(item);
        /* put new item in buffer */
        up(&mutex);
        /* leave critical region */
        up(&full);
        /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        /* decrement full count */
        down(&mutex);
        /* enter critical region */
        item = remove_item();
        /* take item from buffer */
        up(&mutex);
        /* leave critical region */
        up(&empty);
        /* increment count of empty slots */
        consume_item(item);
        /* do something with the item */
    }
}
  
```

I figuren er Producer-Consumer løst med semaforer. Her brukes en binær semafor til å kontrollere tilgang til kritisk sektor. I dette kodeeksempelet kalles den variabelen `mutex` (Merk: en binær semafor er IKKE det samme som en mutex, men de kan brukes ofte på samme måte, vi kommer straks tilbake til forskjellen mellom de). `mutex` brukes altså til eksklusiv tilgang til kritisk sektor, mens `empty` og `full` brukes til *synkronisering*, dvs både til å holde rede på antall ledige plasser i bufferet og blokkere (consumern blokkerer når `full` er lik 0, mens producern blokkerer når `empty` er lik 0, mao vi må ha to semaphorer siden et semaphore blokkerer bare når det blir 0, ikke når det blir 100 :).

Virker helt kurant ikke sant? vel, hva hvis du bytter om rekkefølgen på de to to down-kallene i producern, da kan producern blokkere hvis bufferet er fullt mens `mutex`'n er 0, og neste gang consumern skal aksessere bufferet vil den gjøre down på `mutex`'n og blokkere den og, og vi har en deadlock ...

(mao, det negative med semaforer/mutex'er er at det er vanskelig å kode riktig med dem).

DEMO: 1-en-producer-consumer-semafor.c

(vis koden, kjør, kommenter bort random wait på hvert sitt sted for å se at det funker selv om producer er mye raskere enn consumer eller motsatt)

I POSIX er altså en down på en semafor `sem_wait` mens en up er `sem_post`.

#### 5.4.6 OS: Mutex

##### Mutex Implementation with TSL

```

mutex_lock:
    TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
    CMP REGISTER,#0             | was mutex zero?
    JZE ok                     | if it was zero, mutex was unlocked, so return
    CALL thread_yield           | mutex is busy; schedule another thread
    JMP mutex_lock              | try again
ok:   RET                      | return to caller; critical region entered

mutex_unlock:
    MOVE MUTEX,#0               | store a 0 in mutex
    RET                          | return to caller

```

Mutex kan implementeres i user mode som vist i figuren. Nøkkelen her er `thread_yield`. Husk at en tråd som gjør busy waiting i userspace vil låse hele prosessen fordi en tråd i userspace må gi fra seg CPUen frivillig, det finnes ikke noe klokkeinterrupt for tråder i userspace.

Mutex er altså en enkel lock-variabel som enten er LOCKED (1 evn !0) eller UNLOCKED (0).

Merk: noen lærebøker og tekster skiller ikke mellom mutex og binær semafor, men i de flestes oppfatning er forskjellen den at en mutex har en eier (dvs den som låser mutex'n må låse den

*opp), mens en semafor har ingen eier (det behøver ikke være sammen prosess/tråd som gjør down og up på en binær semafor).*

### Mutexes in Pthread

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

DEMO: 2-en-producer-consumer-semafor-og-mutex.c

(vis koden, kjør, kommenter bort random wait på hvert sitt sted for å se at det funker selv om producer er mye raskere enn consumer eller motsatt, denne koden viser at en mutex og en binær semafor kan som regel ha samme funksjon)

### 5.4.7 OS: Condition Variable

#### Condition Variables in Pthread

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Betingelsesvariable er et alternativ til semaforer, dvs det er en type variabel som gjør at en tråd kan vente basert på en betingelse, og den kan sende signal til andre som venter på variabelen. Brukes som regel alltid sammen med en mutex, la oss se på et eksempel.

Merk: *det spesielle med en condition variable (betingelsesvariabel) er at den har ingen verdi slik som en semafor. Den er kun en variabel som kan ta imot et signal.*

```

#include <stdio.h>
#include <pthread.h>
#define MAX 100000000
pthread_mutex_t the_mutex;           /* how many numbers to produce */
pthread_cond_t condc, condp;
int buffer = 0;                      /* buffer used between producer and consumer */
void *producer(void *ptr)            /* produce data */
{
    int i;
    for (i=1; i<=MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i;                  /* put item in buffer */
        pthread_cond_signal(&condc);   /* wake up consumer */
        pthread_mutex_unlock(&the_mutex);/* release access to buffer */
    }
    pthread_exit(0);
}
void *consumer(void *ptr)             /* consume data */
{
    int i;
    for (i=1; i<=MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0;                  /* take item out of buffer */
        pthread_cond_signal(&condp);   /* wake up producer */
        pthread_mutex_unlock(&the_mutex);/* release access to buffer */
    }
    pthread_exit(0);
}
int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}

```

Producer-Consumer solved with mutex and condition variable.

DEMO: 3-en-producer-consumer-mutex-og-condvar.c

(vis koden (Merk: her er tellesemaforene erstattet med en bufferindex variabel som behandles i kritisk sektor siden den testes etter mutex er låst), kjør, kommenter bort random wait på hvert sitt sted for å se at det funker selv om producer er mye raskere enn consumer eller motsatt)

#### 5.4.8 ProgLang: Monitor

##### A Monitor

```

monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    end;

    procedure consumer( );
    .
    .
    end;
end monitor;

```

Siden programmering med semaforer kan være krevende, *kan monitor* være et alternativ. En monitor er høyere nivå, dvs det er en konstruksjon som finnes i *enkelte programmeringsspråk* og dermed overlater vi til kompilatoren og sørge for mye av det vanskeligste med å få håndtering av kritiske sektorer riktig.

Hovedpoenget er at prosedyrer (dvs de kritiske sektorene som regel), variable og datastrukturer plasseres i en monitor. Variablene og datastrukturene i en monitor kan bare håndteres via monitoren prosedyrer. Prosesser kaller da prosedyrene i monitoren *men bare en prosess kan være aktiv i monitoren om gangen*.

I tillegg til gjensidig eksklusjon trengs mulighet for at en prosess kan blokkere basert på betingelser som vi har sett før. Derav har man også betingelsesvariable (condition variables) tilgjengelig.

```

monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;

procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end;
procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
  end;
end;

```

Producer Consumer i Pidgin Pascal, et pseudokodespråk.

## CHAPTER 5. PROSESSKOMMUNIKASJON, SAMTIDIGHET OG SYNKRONISERING

---

```
public class ProducerConsumer {
    static final int N = 100; // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor
    public static void main(String args[]) {
        p.start(); // start the producer thread
        c.start(); // start the consumer thread
    }
    static class producer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... } // actually produce
    }
    static class consumer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // consumer loop
                item = mon.remove();
                consume_item(item);
            }
        }
        private void consume_item(int item) { ... } // actually consume
    }
    static class our_monitor { // this is a monitor
        private int buffer[] = new int[N];
        private int count = 0, lo = 0, hi = 0; // counters and indices
        public synchronized void insert(int val) {
            if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
            buffer [hi] = val; // insert an item into the buffer
            hi = (hi + 1) % N; // slot to place next item in
            count = count + 1; // one more item in the buffer now
            if (count == 1) notify(); // if consumer was sleeping, wake it up
        }
        public synchronized int remove() {
            int val;
            if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
            val = buffer [lo]; // fetch an item from the buffer
            lo = (lo + 1) % N; // slot to fetch next item from
            count = count - 1; // one few items in the buffer
            if (count == N - 1) notify(); // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {}}
    }
}
```

Producer Consumer løst i java, merk hvor monitoren er!

DEMO: ProducerConsumer . java  
(vis koden, kjør)

DEMO: løs incdec . c med både semafor og med mutex

## 5.5 Theory questions

1. I læreboka figur 2.21 finnes et eksempel med to prosesser som ønsker utskrift. Hvilken prosess sørger for å hente utskriftfilene fra spooler directory? Hvorfor kan det gå galt med denne løsningen?
2. Når brukes *busy waiting* for å synkronisere prosesser?
3. De fleste CPUer har en assemblyinstruksjon som heter TSL eller XCHG. Forklar hvordan en av disse virker, og hva som er gjørt at den er nyttig ved synkronisering av prosesser.
4. Gitt følgende trådkode:

```

void *consumer(void *arg)
{ int i;
  for (i=0;i<5;i++){
    pthread_mutex_lock(&mutex);
    if (state == EMPTY)
      pthread_cond_wait(&signals, &mutex);

    -----
    pthread_cond_signal(&signals);
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(NULL);
}

```

Forklar hva wait- og signal-operasjonene i ovenstående programlinjer har som funksjon. Hva er hensikten med variabelen `mutex` og hvorfor forekommer den i wait-operasjonen?

5. Gitt følgende program:

```

01 int g_ant = 0;          /* global declaration */
02
03 void *writeloop(void *arg) {
04   while (g_ant < 10) {
05     g_ant++;
06     usleep(rand()%10);
07     printf("%d\n", g_ant);
08   }
09   exit(0);
10 }

```

```
11
12 int main(void)
13 {
14     pthread_t tid;
15     pthread_create(&tid, NULL, writeloop, NULL);
16     writeloop(NULL);
17     pthread_join(tid, NULL);
18     return 0;
19 }
```

Forklar hvorfor utskriften fra dette programmet ikke nødvendigvis blir slik vi ønsker at den skal bli:

```
1
2
3
4
5
6
7
8
9
10
```

Legg inn de synkroniserings-mekanismene du ønsker (semforer, mutexer eller betingelsesvariable) for å garantere at utskriften blir slik vi ønsker. Forklar hvorfor du plasserer de slik du gjør. (du behøver ikke skrive opp igjen hele programmet, bare svar ved å angi mellom hvilke kodelinjer du vil innføre ny kode, og hvorfor.)

## 5.6 Lab exercises

### 1. Lage nye tråder og enkel semafor-synkronisering av disse.

Skriv om programmet fra forrige ukes lab (oppgave 1 "Lage nye prosesser og enkel synkronisering av disse") til å være trådbasert i stedet for prosessbasert, og bruk semaforer istedet for prosessventing for å synkronisere de. Det kan hende du da vil møte på begrensningen i å overføre argumenter til en trådfunksjonen siden en trådfunksjonen under pthreads vil at argumentet sitt skal være en void-peker (dvs, en peker som kan peke til hva-som-helst). Slik du kan løse dette er at du definerer en struct med argumentene dine, oppretter og allokerer plass til den i main, og bare sender med en peker til den i det tråden opprettes:

```
struct threadargs {
    int id;          /* thread number */
    int sec;         /* how many seconds to sleep */
    int signal[6];  /* which threads to signal when done */
};

.

.

void *tfunc(void *arg) {
    struct threadargs *targs=arg;
.

.

int main(void)
{
    int i,j;
    struct threadargs *targs[6];
    /* allocate memory for threadargs and zero out semaphore signals */
    for (i=0;i<6;i++) {
        targs[i] = (struct threadargs*) malloc(sizeof(struct threadargs));
        for (j=0;j<6;j++) {
            targs[i]->signal[j]=0;
        }
    }
.

.
```

(hmmm, google gjerne for å finne ut om malloc allokkerer plass i data-området eller stack-området)

### 2. Flere Producere og Consumere.

Ta utgangspunkt i en av producer-consumer eksemplene fra forelesningen. Disse eksemplene hadde bare en producer og en consumer. Skriv om koden slik at programmet tar tallet  $N$  som kommandolinjeargument og det fører til at det blir  $N$  producere og  $N$  consumere (som alle sammen fortsatt jobber mot det samme bufferet).



# Chapter 6

## Scheduling

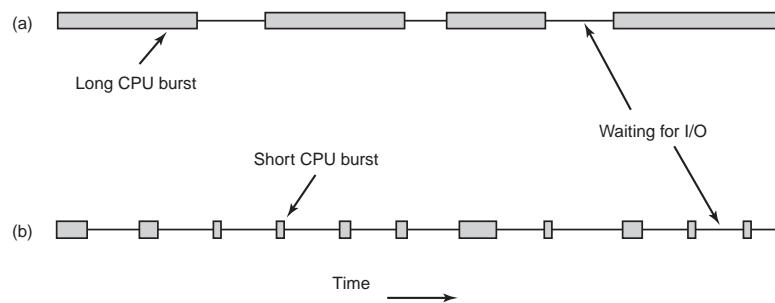
### 6.1 Outcome

#### Today's Learning Outcome

- Performance
- Conceptual framework

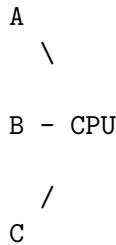
### 6.2 Introduction

#### Process Behaviour



(Scheduling var viktig før, og er viktig nå, men har sjeldent vært viktig på PCer siden det stort sett bare er en bruker på de)

Skal CPU kjøre prosess A, B eller C? Scheduleren bestemmer.



I utgpkt betyr ikke scheduling så mye på vanlige PCr fordi de er så raske at de begrenses mest av hvor fort brukeren klarer å gi CPUen oppgaver. Men hvis brukeren benytter PCn til videoprosessering, eller å kjøre en lang rekke virtuelle maskiner får CPUn fort nok å gjøre.

(TAVLE:) Hva scheduleren overordnet må sørge for:

- Velge riktig prosess (en interaktiv brukerprosess er som regel viktigere enn en periodisk statistikkinnssamlingsprosess)
- Ikke skifte prosess for ofte siden context switch er kostbart

Prosesser er enten (se figuren):

**CPU-bound** krever mye CPU-kraft (f.eks. videoprosesserings-prosesser)

**I/O-bound** bruker mest tid på å vente på I/O (de fleste vanlige prosesser)

Merk: poenget er lengden på CPU-burst, IKKE lengden på I/O-wait, dvs I/O-bound prosesser er ikke I/O-bound fordi I/O tar så lang tid, men fordi de har lite å gjøre mellom hver I/O request.

### Ideal Process Mix

- *It is good to have a mix of CPU-bound and I/O-bound processes*

Slik at ikke alle enten sloss om CPUn eller alle venter på I/O. Har vi en miks så kan scheduleren velge en CPU-bound prosess mens en I/O-bound prosess ikke har noe å gjøre.

### Many Schedulers

- I/O block writes, Printer jobs, Routers, ...
- CPU scheduling
  - Long-term/Admission scheduler

- Medium-term/Memory scheduler
- *Short-term/CPU scheduler*
- (Dispatcher)

Long-term/Admission scheduler bestemmer om en prosess får komme inn i ready-køen, dvs den kan stoppe et system fra å bli for fullt, “beklager, jeg har ikke plass til flere prosesser nå...”.

Medium-term/Memory scheduler kjører litt oftere enn long-term og vurderer om noen prosesser skal tas ut av (swappes ut midlertidig) eller tas inn igjen i køen.

Short-term/CPU scheduler kjører oftest (derav “Short-term”) og er den vi skal prate mest om, dvs den som bestemmer hvilke prosesser som slipper til når på CPU(ene).

Når Short-term/CPU scheduler har gjort sin beslutning, er det Dispatcheren som gjør jobben med å bytte ut kjørende prosess men den som skal ta over.

### Preemptive vs Nonpreemptive

- *Preemptive scheduling/multitasking* means that the OS can take the CPU away from a running process
- *Nonpreemptive/Cooperative scheduling/multitasking* means that the process has to voluntarily give up the CPU

[http://en.wikipedia.org/wiki/Preemption\\_\(computing\)](http://en.wikipedia.org/wiki/Preemption_(computing))

### When to Schedule?

- When a process is created
- When a process exits
- When a process blocks on I/O
- When an I/O interrupt occurs
- At each single or every  $k$ th clock interrupt, commonly:
  - *preemptive* scheduling uses clock interrupts
  - *nonpreemptive* does not

Når en prosess skapes, skal den nye kjøres eller den som skapte prosessen (parent)? Scheduleren bestemmer.

Preempt betyr å avbryte, som regel sier vi at vi har preemptive scheduling hvis en prosess som har CPUn ikke får lov til å beholde CPU'en til den er ferdigkjørt (eller selv blokkerer på I/O eller gir fra seg CPU'en frivillig). Dette betyr vanligvis at scheduleren bruker klokkeinterrupt for å avbryte kjørende prosesser. Men vi sier også av og til at alle tilfeller som fører til at scheduleren avbryter prosesser er preemptive scheduling (som vi snart skal se for f.eks. Shortest Remaining Time Next algortimen).

## Scheduling Goals

### All systems

- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

### Batch systems

- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

### Interactive systems

- Response time - respond to requests quickly
- Proportionality - meet users' expectations

### Real-time systems

- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Ift throughput og turnaround: det er avveininger, hvis man prioriterer korte prosesser for høy throughput så vil en lang prosess aldri slippe til og føre til at man får uendelig høy gjennomsnitts-turnarounds tid.

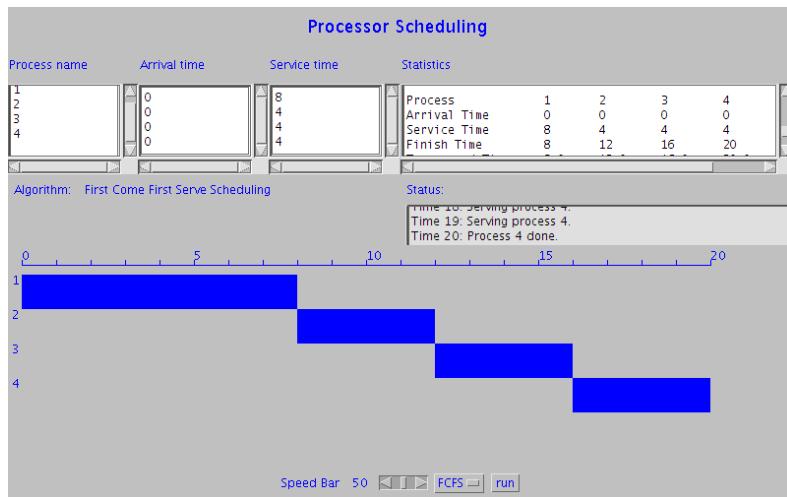
Ift proportionality: hvis en bruker klikker på et ikon for å sende en fax forventer hun at det tar tid, men klikker hun på krysset i hjørne av vindu forventer hun at det skal lukkes med en gang.

Ift real-time systems, så gjelder predictability (forutsigbarhet) mest for soft real-time, dvs multimedia, og det er viktigere at lyd spilles av riktig enn at bilde ikke hakker (øyet vårt godtar litt dårlig bilde, men øre vårt godtar ikke hakkete lyd).

## 6.3 Batch Systems

### 6.3.1 FCFS (FIFO)

#### First-Come First-Served, Same Arrival Time



(også kalt First-In First-Out, FIFO)

Dette er en non-preemptive schedulingalgoritme.

Vi antar at vi har en *ready kø*, en kø med prosesser klare til å kjøres.

Prossesser plasseres i alltid bakerst i ready køen og schedulern velger den første i køen hver gang og lar den kjøre ferdig (dette er da en nonpreemptive algoritme).

Enkel å forstå, enkel å implementere.

Langt unna optimal i mange tilfeller, f.eks. hvis det er en CPU-bound prosess og mange I/O-bound prosesser som ønsker gjøre mange I/O-requests så vil disse ta veldig lang tid siden den CPU-bound prosessen alltid vil kjøre imellom. F.eks. hvis prosess A trenger å gjøre 1000 separate diskreads, mens prosess B er CPU-bound med 1 sek CPU-arbeid mellom hver I/O request, så vil A ta 1000 sek å fullføre (i motsetning til en scheduler med preemption hvert 10. msec, som ville fullført A på 10 sek).

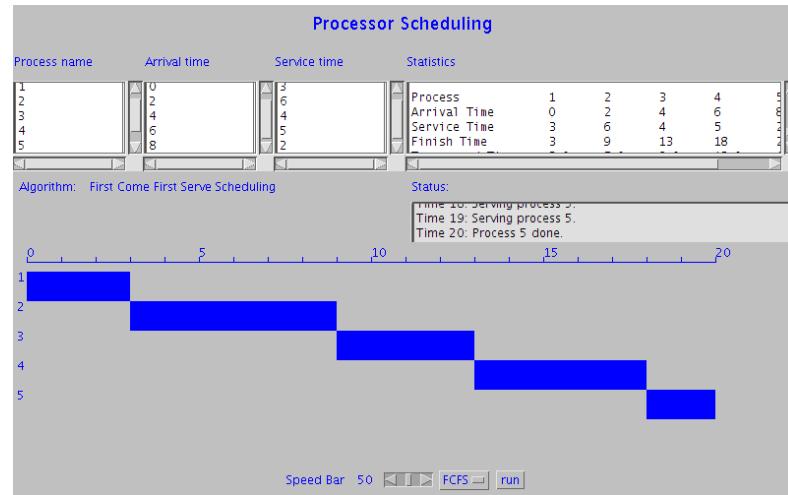
*FCFS er fair i betydning at den som venter lengst får slippe til, men det er spesielt fordelaktig for lange CPU-bound prosesser.*

Gjennomsnittlig turnaround tid FCFS-SAT (Same Arrival Time):

$$\frac{8 + 12 + 16 + 20}{4} = \mathbf{14}$$

(TAVLE: kolonne for SAT og DAT, rader for FCFS, SJF, SRTN)

## First-Come First-Served, Different Arrival Time

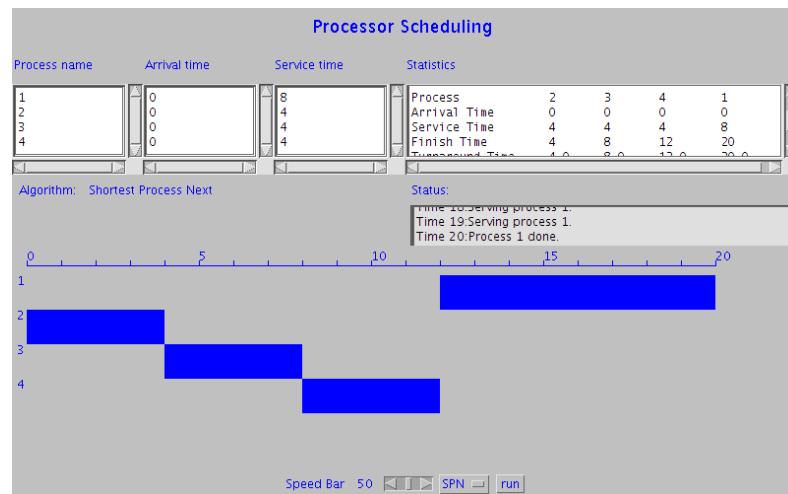


Gjennomsnittlig turnaround tid FCFS-DAT (Different Arrival Time):

$$\frac{3 + 7 + 9 + 12 + 12}{5} = \frac{43}{5} = 8.6$$

### 6.3.2 SJF/SPN

#### Shortest Job First, Same Arrival Time



(i interaktive systemer så kalles Shortest Job First for Shortest Process Next).

En nonpreemptive algoritme som antar at kjøretiden pr prosess er kjent på forhånd.

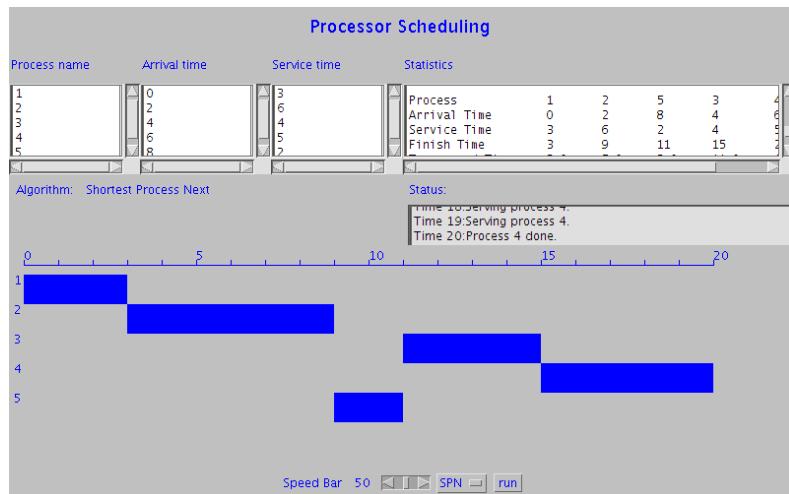
Denne er optimal mtp turnaround tid gitt at det ikke ankommer nye prosesser.

*Problematisk hvis de store prosessene alltid må vente til slutt. Hvis nye korte prosesser ankommer hele tiden, vil de store aldri slippe til ... (starvation)*

Gjennomsnittlig turnaround tid SJF-SAT:

$$\frac{4 + 8 + 12 + 20}{4} = \mathbf{11}$$

### Shortest Job First, Different Arrival Time

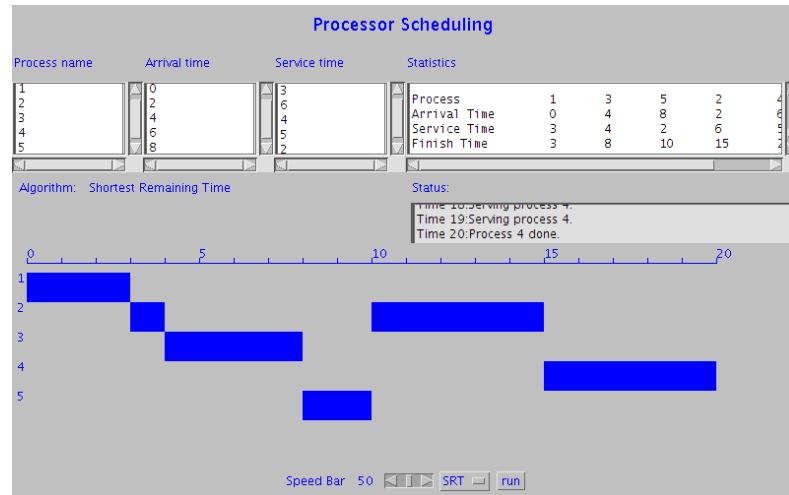


Gjennomsnittlig turnaround tid SJF-DAT:

$$\frac{3 + 7 + 11 + 14 + 3}{5} = \frac{38}{5} = \mathbf{7.6}$$

### 6.3.3 SRTN

#### Shortest Remaining Time Next



En preemptive versjon av SJF. Scheduleren velger alltid den prosess som har igjen kortest kjøretid. Samme ulempe som SJF (lange prosesser kan ende opp ventende lenge). Dvs her kjører scheduleren hver gang en ny prosess ankommer køen og scheduleren har makt til å avbryte en lang prosess med en ny kortere en. Men scheduleren gjør denne vurderingen altså bare hver gang prosesser avsluttes eller nye prosesser ankommer køen.

Gjennomsnittlig turnaround tid SRTN-DAT (SRTN-SAT er lik SJF-SAT):

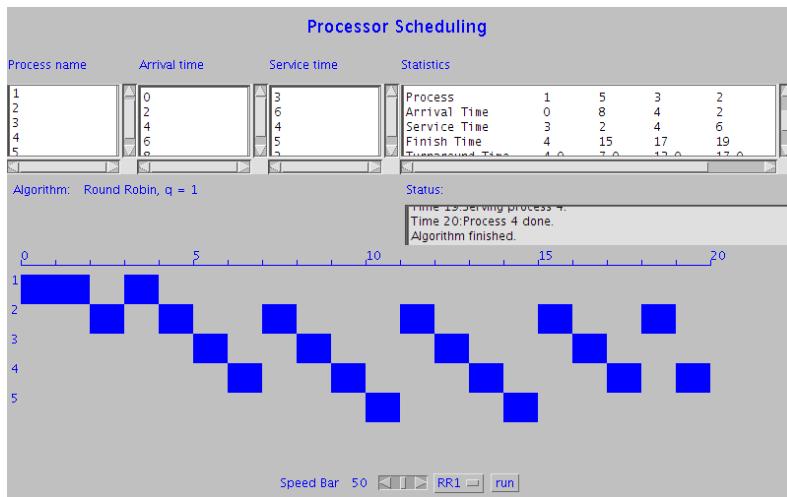
$$\frac{3 + 13 + 4 + 14 + 2}{5} = \frac{36}{5} = 7.2$$

(men husk: SJF og SRTN antar at man kjenner kjøretid til prosessen, og det gjør man ikke alltid)

## 6.4 Interactive Systems

### 6.4.1 RR

#### Round-Robin Scheduling, Quantum 1



Round-Robin er en preemptive versjon av First-Come First-Served.

Prosessene får kjøre i et tidskvantum hver og legges så bakerst i køen.

Eneste interessante spm er hvor stort tidskvantum skal være, hvis context switch tar 1msec og tidskvantum 4msec så er 20% av CPUn bortkastet på overhead, men hvis tidskvantum er 100msec så kan brukere oppleve altfor lang responstid hvis det er mange samtidige prosesser fordi alle får altfor lang tid om gangen.

Gjennomsnittlig turnaround tid RR1-DAT:

$$\frac{4 + 17 + 13 + 14 + 7}{5} = \frac{55}{5} = 11$$

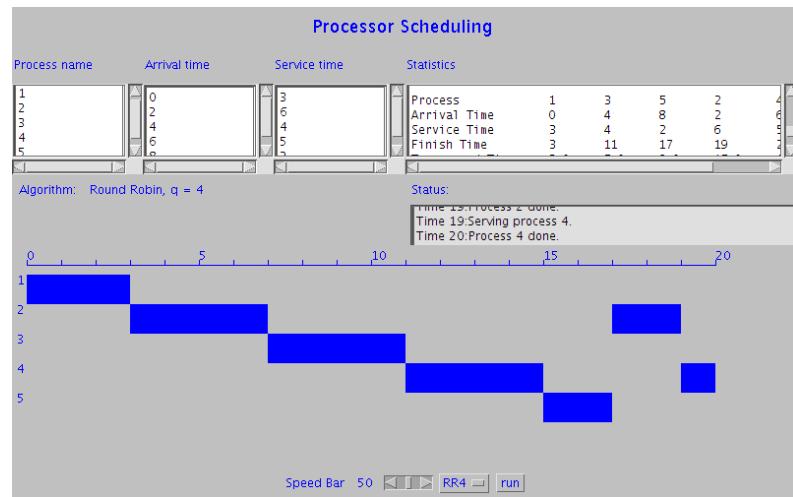
Demo: Linux, les om jiffies på man 7 time

Demo: Windows, clockres gir "jiffie" intervallet, 2x for desktop, 12x for server, kan endres med

SystemPropertiesAdvanced, Advanced, Performance, Advanced og se hva som skjer med i

`hkml:\System\CurrentControlSet\control\PriorityControl`

### Round-Robin Scheduling, Quantum 4

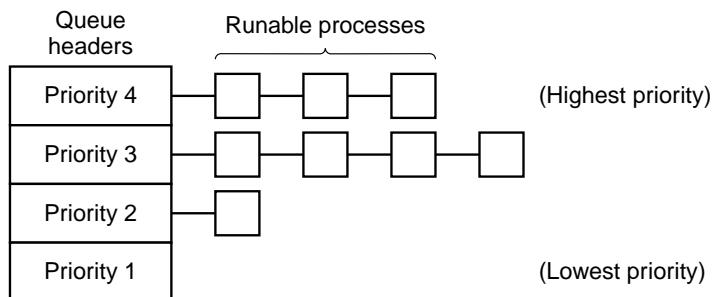


Gjennomsnittlig turnaround tid RR4-DAT:

$$\frac{3 + 17 + 7 + 14 + 9}{5} = \frac{50}{5} = 10$$

### 6.4.2 Priority

#### Priority Scheduling with Multiple Queues



Det er også vanlig å gi prosesser prioritet og kanskje prioritetsklasser som vist i figuren. En video som spilles av må kjøre jevnt, mens en mail som kan sendes kan bruke 1 sek ekstra slik at videoavspillingen prioriteres.

Prioritet må som regel være dynamisk (hvis ikke vil lavt prioriterte prosesser kanskje aldri slippe til, starvation ...).

I/O-bound prosesser bør gis høy prioritet (f.eks. bør de få prioritet  $1/f$  hvor  $f$  er andel av siste kvantum brukt).

Round-robin benyttes gjerne innenfor hver klasse.

### 6.4.3 SPN with Aging

**Shortest Process Next with Aging** How to estimate the next use of a specific process (e.g. command-line commands)?

$$\bar{T}_n = (1 - a)T_n + a\bar{T}_{n-1}$$

$$(\text{Def. } \bar{T}_0 = (1 - a)T_0)$$

Merk: SJF=SJN=SPN, så denne kalles av og til Shortest Estimated Process Next, siden det spesielle med denne er bare hvordan remaining time estimeres. Og her er det aging kommer inn.

Vi har et sett med kjøretider

$$T_0 T_1 T_2 T_3$$

mellan disse gjør vi estimerer:

$$T_0 \bar{T}_0 T_1 \bar{T}_1 T_2 \bar{T}_2 T_3 \bar{T}_3$$

hvor  $\bar{T}_n$  er estimatet som gjøres umiddelbart etter siste registrerte kjøretid  $T_n$ .

$T_n$  representerer da siste nøyaktige måling, mens  $\bar{T}_n$  representerer historikken (et estimat).

Med  $a = 1/2$ , vil estimatet for 4. gangs kjøring bli:

$$\bar{T}_2 = \frac{1}{2}T_2 + \frac{1}{2}\bar{T}_1 = \frac{1}{2}T_2 + \frac{1}{4}T_1 + \frac{1}{8}T_0$$

Eksempel: Agingalgoritmen med  $a = \frac{1}{2}$  brukes til å predikere (spå) kjøretider. Fire tidligere kjøringer fra den eldste til den siste er 20ms, 15ms, 40ms og 10ms. Hva blir estimatet for neste kjøring?

Her er altså  $T_0 = 20, T_1 = 16, T_2 = 40, T_3 = 10$  og vi skal finne estimatet for neste kjøring  $\bar{T}_3$ :

$$\bar{T}_3 = \frac{1}{2}T_3 + \frac{1}{2}\bar{T}_2$$

vi setter inn for  $\bar{T}_2$  og deretter for  $\bar{T}_1$  og  $\bar{T}_0$  slik at vi får

$$\bar{T}_3 = \frac{1}{2}T_3 + \frac{1}{4}T_2 + \frac{1}{8}T_1 + \frac{1}{16}T_0$$

som da gir oss

$$\bar{T}_3 = \frac{10}{2} + \frac{40}{4} + \frac{16}{8} + \frac{20}{16} = 5 + 10 + 2 + 1.25 = \mathbf{18.25}$$

#### 6.4.4 Guaranteed

##### Guaranteed Scheduling

- With  $n$  processes each process get  $1/n$  of CPU time
- Must keep track of CPU time for each process
- Always run the process with the least consumed CPU time in relation to its guaranteed time

#### 6.4.5 Lottery

##### Lottery Scheduling

- Basically achieves the same as guaranteed scheduling, but much simpler
- Each process gets one (or more if higher priority) lottery ticket
- Scheduling by a lottery draw

#### 6.4.6 Fair-Share

##### Fair-Share Scheduling

- Guaranteed scheduling, but with respect to users
- If user A have process  $a_1, a_2, a_3$  and  $a_4$ , and user B have only process  $b_1$ , the scheduling might be:  
 $a_1 b_1 a_2 b_1 a_3 b_1 \dots$

### 6.5 Real-Time Systems

#### Scheduling in Real-Time Systems

- $m$  periodic events, event  $i$  occurs with period  $P_i$  and requires  $C_i$  CPU time for each event, the load can only be handled if

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

TAVLE:

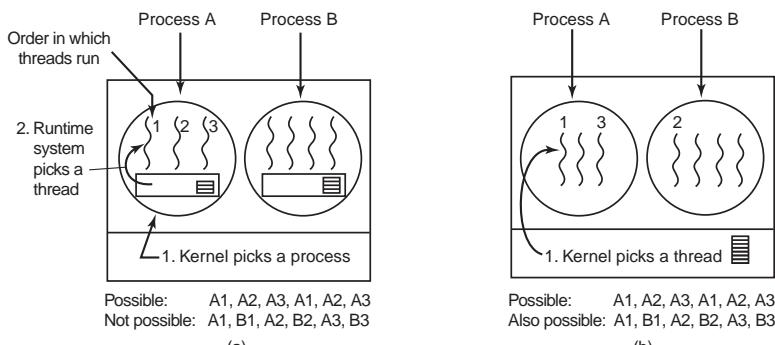
3 prosesser, periodiske hendelser hvert 100, 200 og 500 msec. Hver hendelse krever henholdsvis 50, 30 og 100 msec. Er dette mulig å schedule'e ?

$$\frac{50}{100} + \frac{30}{200} + \frac{100}{500} = 0.5 + 0.15 + 0.2 = 0.85$$

Ja, dette lar seg schedule'e slik at real-time systemet vil funke siden  $0.85 \leq 1$ .

## 6.6 Thread Scheduling

### Thread Scheduling



Husk at her får vi scheduling på to nivåer ved user-level tråder, først må prosessen schedules og innenfor denne prosessen schedules trådene. Hvis en tråd på user-level gjør et blokkerende I/O kall blokkeres hele prosessen.

Figuren viser situasjonen hvis vi har round-robin scheduling med tidskvantum 50msec, men hver tråd har bare 5msec med jobb å gjøre, figur a viser situasjonen for user-level, mens figur b viser situasjonen for kernel level.

Det er viktig å huske at å svitsje mellom tråder er ti ganger raskere enn å gjøre en full context switch.

MEN user-level scheduling kan være nyttig for applikasjonstilpasning: f.eks. kan en user-level scheduler prioritere at dispatcher-tråden alltid får kjøre når den er klar i web-server eksempelet vi så på ved introen til tråder. En kernel scheduler vil ikke nødvendigvis være klar over slike applikasjonsspesifikke forhold.

## 6.7 Multiproc

### How to Make Faster Computers

- According to Einstein's special theory of relativity, no electrical signal can propagate faster than the speed of light, which is about 30 cm/nsec in vacuum and about 20 cm/nsec in copper wire or optical fiber.
- *How does this relate to the speed of a computer?*

En prosessor med

- 1GHz klokke kan signalet teoretisk maksimalt forplante seg 200mm pr klokkeperiode, og tilsvarende:
  - 10GHz - 20mm
  - 100GHz - 2mm
  - 1THz - 0.2mm

Jo mindre kretsene blir, jo større varmeutvikling og desto vanskeligere å få bort varmen.

### 6.7.1 Affinity

#### Processor Affinity/CPU Pinning

- A process/thread can be restricted to run on only one or a set of CPUs

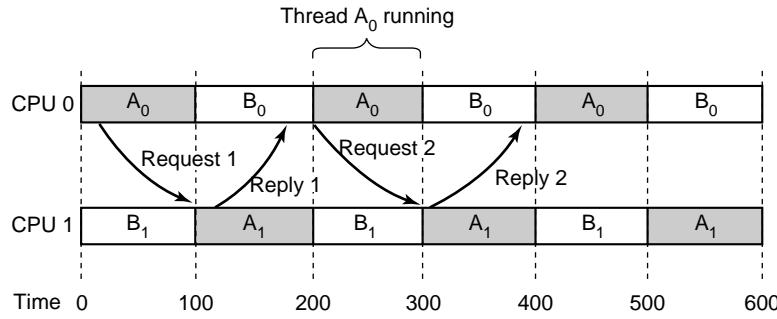
Et problem er at prosesser/tråder kanskje ikke bør hoppe random mellom CPU'r siden det er mye prosess/trådspesifikk caching av data knyttet til en CPU der tråden har kjørt. Scheduling som tar høyde for dette kalles *affinity scheduling*. Dvs scheduling som forsøker å la en prosess/tråd kjøre på samme CPU som den kjørte sist i håp om at det er igjen mye relevant data for den prosess/tråden i den CPU'n sin cache.

Demo: `taskset -c 0 ./regn.bash`  
`sudo htop`, a for set affinity

Scheduleren gjør naturlig affinity så det er normalt lite behov for oss å kunstig manipulere dette, men det kan være spesielle situasjoner som f.eks. noe programvare som har lisenskostnader pr antall CPU'er i bruk (Oracle-databaser kanskje).

### 6.7.2 Gang-/Co-scheduling

#### Communication Between Two Threads



Mange tråder kommuniserer mye, derav trengs en løsning med både time og space sharing.

Figuren viser hvilket problem som kan oppstå når tråden  $A_0$  og  $A_1$  skal kommunisere, en toveis-kommunikasjon kan ta 200msec noe som er uholdbart.

Løsningen kalles *Gang scheduling*.

### Gang-/Co-Scheduling

	CPU						
	0	1	2	3	4	5	
Time slot	0	$A_0$	$A_1$	$A_2$	$A_3$	$A_4$	$A_5$
	1	$B_0$	$B_1$	$B_2$	$C_0$	$C_1$	$C_2$
	2	$D_0$	$D_1$	$D_2$	$D_3$	$D_4$	$E_0$
	3	$E_1$	$E_2$	$E_3$	$E_4$	$E_5$	$E_6$
	4	$A_0$	$A_1$	$A_2$	$A_3$	$A_4$	$A_5$
	5	$B_0$	$B_1$	$B_2$	$C_0$	$C_1$	$C_2$
	6	$D_0$	$D_1$	$D_2$	$D_3$	$D_4$	$E_0$
	7	$E_1$	$E_2$	$E_3$	$E_4$	$E_5$	$E_6$

### Gang scheduling

1. Grupper av relaterte tråder schedules som en enhet (en gjeng).
2. Alle medlemmene i gjengen kjører samtidig på ulike tidsdelte CPU'r.
3. Alle medlemmene i gjengen starter og ender sine tidsluker sammen

Alle CPU'r schedules synkront.

Figuren viser et eksempelt på dette med 6 CPU'r og 5 prosesser med ulikt antall tråder.

Poenget er altså at alle tråder i samme prosess bør kjøre samtidig i samme tidsluke.

Dette praktiseres delvis i hypervisore som vmware, xen og linux/kvm, men man er forsiktig fordi det kan også redusere ytelsen. Men hvis man vet det er mye kommunisering mellom tråder ved bruk av spinlocks så er gang/co-scheduling en fordel.

## 6.8 OS Implementation

**Current Use of Scheduling** [http://en.wikipedia.org/wiki/Scheduling\\_\(computing\)](http://en.wikipedia.org/wiki/Scheduling_(computing))

see Windows and GNU/Linux paragraphs, then see

[http://en.wikipedia.org/wiki/Multilevel\\_feedback\\_queue](http://en.wikipedia.org/wiki/Multilevel_feedback_queue)

Merk at en multilevel feedback kø er en kombinasjon av FIFO (FCFS), prioritetskøer, pre-emption, med round-robin i laveste nivå kø.

Demo: Windows, perfmon, se alle chrome-trådenes dynamiske prioritet

## 6.9 Theory questions

1. Hva mener vi når vi sier at schedulingen er non-preemptive? Hvilke systemer bruker preemptive og non-preemptive scheduling?
2. Hva ligger i begrepet "starvation" ifb med Shortest Job First (SJF) scheduling algoritmen?
3. Anta Round Robin scheduling. Vurder konsekvensene ved korte og lange tidskvantum.
4. Nevn noen fordeler vi har ved prioritert scheduling.
5. Anta ett sett med fire prosesser med ankomst-tid og burst/CPU-tid (i ms) gitt i tabell:

Prosess	Burst/CPU-tid (ms)	Ankomst-tid
P1	7	0
P2	4	2
P3	1	4
P4	4	5

Tegn et Ganttskjema (tidstabell) som illustrerer eksekveringen av prosessene for hver av følgende schedulingsalgoritmer:

- First-Come First-Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time Next (SRTN)
- Round Robin med tidskvantum på 2ms

Hva blir gjennomsnittlig turnaroundtid for hver av schedulingsalgoritmene?

## 6.10 Lab exercises

### 1. Dining philosophers.

Dining philosophers er et klassisk problem innen operativsystemer, og er beskrevet i læreboka i kapittel 2.5. Første gang man leser om det virker det kanskje som et snodig teoretisk problem, men det addreserer en del sentrale problemstillinger:

**Kritisk sektor/gjensidig ekslusjon** Filosofene må dele gafler/spisepinner.

**Deadlock (vranglås)** Siden gaflene/spisepinnene er eksclusive ressurser og filosofene må ha to av dem for å spise, innebærer det en fare for deadlock.

**Starvation (utsulting)** Alle filosofene må få mulighet til å spise.

**Parallellitet** Det må være mulig for flest mulig filosofer å spise om gangen.

**Rettferdighet** Alle filosofene bør få mulighet til å spise like mye.

I denne omgang er vi primært opptatt av de to første. Programmer et forslag til løsning av dining philosophers, bruk gjerne tråder som filosofer. *Du kan gjerne ta utgangspunkt i figur 2.46 i læreboka og implementere den koden komplett i C.*

# Chapter 7

## Virtuelt minne, paging og segmentering

### 7.1 Introduction

**Question** *What is the multi-gigabyte file `pagefile.sys` hidden in my root directory?*

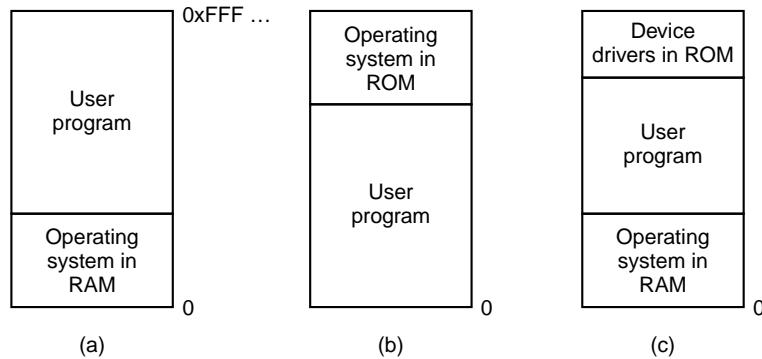
```
Get-ChildItem C:\ -force
```

**In the Ideal World** “Hello, I’m a process, I would like memory that is

- *Private*
- *Infinitely large*
- *Infinitely fast*
- *Nonvolatile* (does not lose data when powered off)
- *(Inexpensive)”*

Vi kan desverre ikke tilby dette, men vi kan tilby minnehierarkiet fra første forelesning (HW Review - Memory).

### No Memory Abstraction (and Monoprogramming)



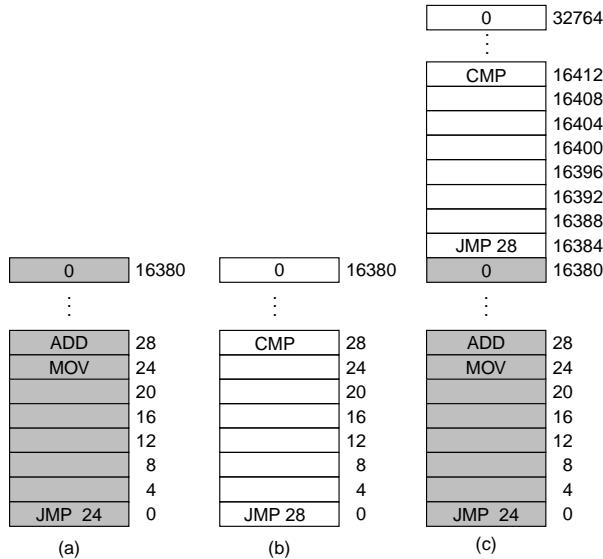
Historisk: en prosess i minne om gangen. Dette er altså monoprogrammering (i motsetning til multiprogrammering hvor flere prosesser er i minne om gangen).

Ingen abstraksjon: direkte addressering av fysisk minne.

(a) og (c) er primært historiske. (b) er i bruk på noen PDAr og embedded systems. Problemet med (a) og (c) er at en bug i programmet kan overskrive operativsystemet. En måte å oppnå en hvis parallellitet i slike systemer er via tråder.

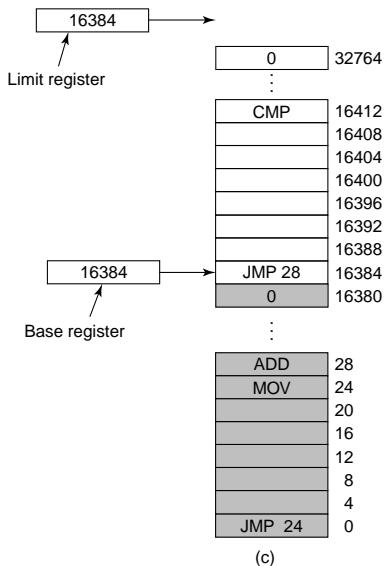
### 7.1.1 Relocation

## Multiple Programs: the Relocation Problem



Første instruksjon i det øverste programmet gjør JMP 28 som hopper inn i det andre programmet. Problemet er at begge programmene addresser fysisk minne direkte. En løsning er *statisk relokalisering*: ved at alle minneaddresser endres i det programmet lastes. Ingen god løsning siden lasting blir tregere og det kan være vanskelig å finne alle minneaddresser i programmet.

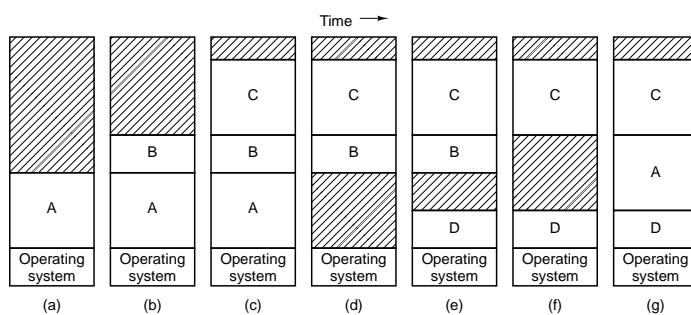
### Base and Limit Registers



Dynamisk Relokalisering kan gjøres ved at to spesielle registre benyttes: base og limit registerne. Verdien i base register legges til alle minneadresseringer, og limit sjekkes for at programmet ikke forsøker addressere utenfor sitt minneområde. Det negative ved dette er behovet for en addisjon og sammelikning ved hver eneste minneadressering.

#### 7.1.2 Swapping

##### Swapping

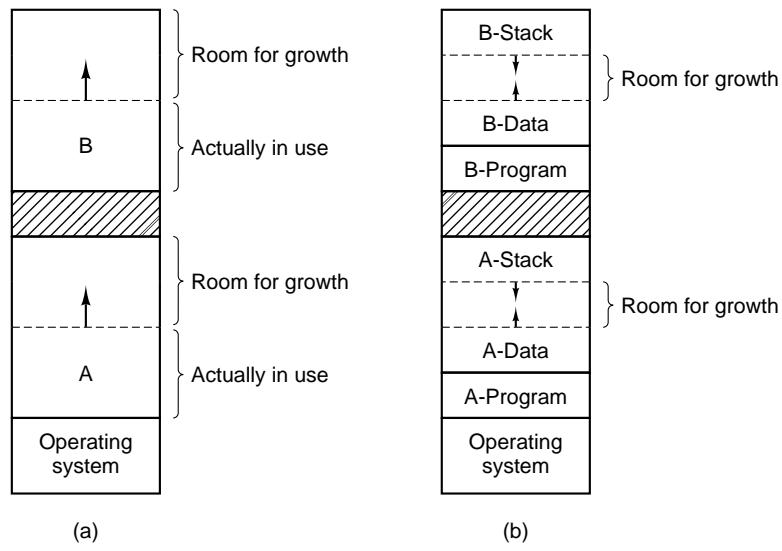


Flere prosesser i minne om gangen. (Her er det altså fortsatt bare relokalisering med base/limit registerne.)

Tradisjonell swapping betyr altså at hele prosesser er enten i minne eller "swappet" ut på disk. Swapping skaper hull i minnet (ekstern fragmentering).

Begrepet swap brukes idag ofte om diskområder som benyttes til andre formål (paging) enn tradisjonell swapping. Dvs swapping idag brukes ofte om noen av de samme oppgavene som paging, dvs å flytte page'r mellom RAM og disk.

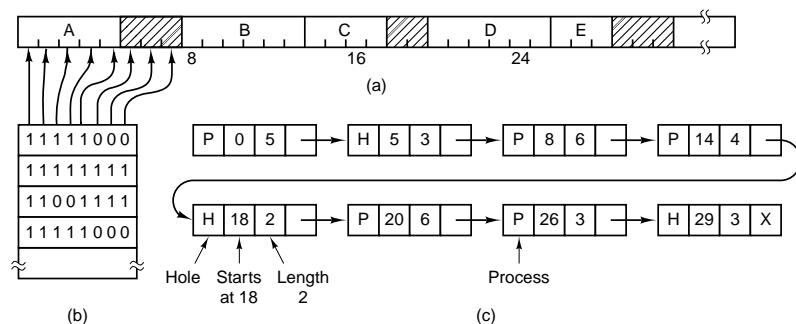
### Growing Processes



Ved swapping må OS'et bestemme hvor mye plass det skal avsette til prosessen i minne. Det er veldig vanskelig å beregne størrelsen til et program før det lastes og kjøres, stort sett bare programkoden som er statisk størrelse.

#### 7.1.3 Free space

##### Memory Management with Bitmaps and Linked Lists

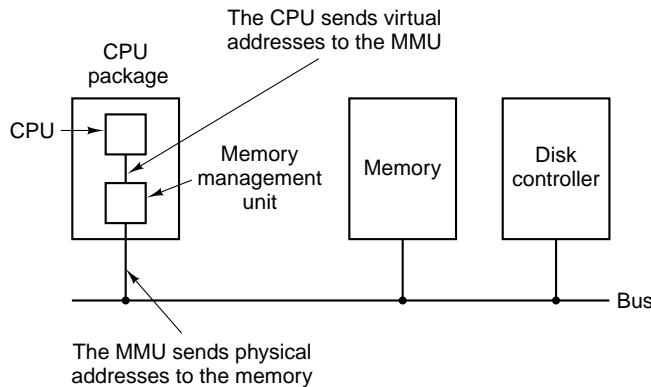


Det er veldig tregt å søke gjennom en bitmap etter en gitt størrelse ledig plass.

Benyttes i mange sammenhenger for å holde rede på ledig plass i minne eller på disk, fordi en bit i en bitmap representerer gjerne en page på 4KB og dermed blir aldri bitmap'n nevneverdig stor.

## 7.2 Virtual Mem.

### The Memory Management Unit



MMU oversetter *virtuelle addresser* til *fysiske addresser*.

Det store poenget er at vi kan kjøre programmer som bare delvis er i minne, fordi vi innfører et virtuelt addresserom i tillegg til det fysiske addresserommet.

VIS virtuelt addresserom med `vmmmap.exe` og `procexp.exe` på Windows og Ubuntu System Monitor. Kjør `hello-stop.c`.

VIS inndeling i kernel space / user space :

<http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>

Merk: 64-bits adresser betyr i praksis 48-bit på X86, som blir til 44-bit på Windows, derav kernel/user space inndeling 8TB/8TB, mens Linux bruker 48-bit som gir kernel/user space inndeling 128TB/128TB.

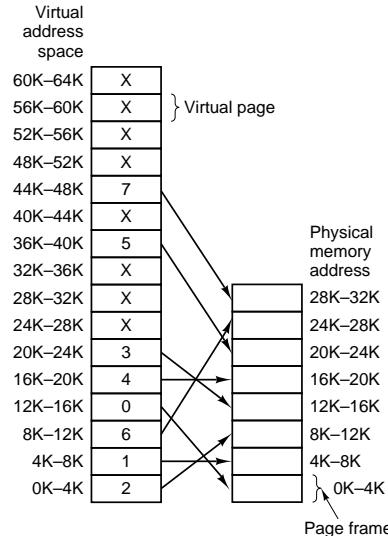
VIS Norsk historikk :

[http://en.wikipedia.org/wiki/Virtual\\_memory](http://en.wikipedia.org/wiki/Virtual_memory)

TEGN: prosesser som ser hvert sitt virtuelle addresserom (ofte kalt memory map), og disse virtuelle addresserommene benytter mange eller få page'r i fysisk minne.

### 7.2.1 Paging

#### Virtual and Physical Addresses



Eksempel for 16-bits addresser (64KB) og 32KB fysisk minne (dvs 15-bits fysiske addresser).

Det virtuelle addressrommet er delt inn i *page'er* (av og til kalt *virtuelle page'er*). De tilsvarende enhetene i det fysiske minne kalles *page frames*. Hvis et program forsøker addressere en page som ikke er mappet til en page frame, blir det et interrupt som kalles en *page fault* (dette er et interrupt av type *exception*, dvs i samme kategori som det interrupt vi får hvis vi forsøker dele på null i et program). Operativsystemet skriver da en lite-brukt page frame tilbake til disk og laster inn den ønskede page frame istedet. Overføringer til og fra disk skjer alltid i hele page'er.

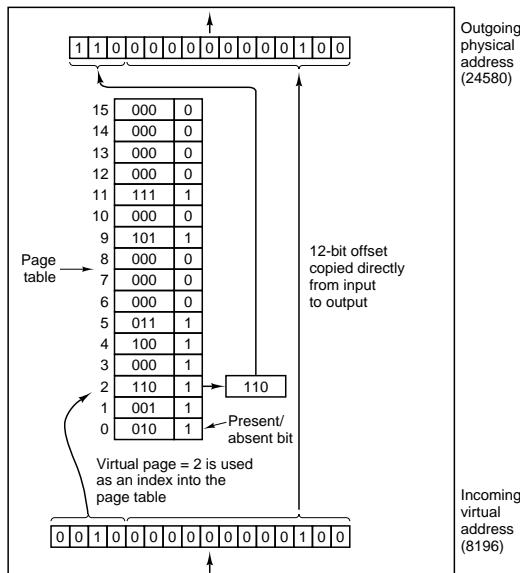
(NB! Interrupt kategoriseres litt ulikt, vi forsøker følge kategoriseringen intel benytter med hardware interrupt, exceptions, og trap (software interrupt)).

TAVLE:

- page
- page frame
- page fault

*Det å holde page tabellen oppdatert (dvs administrere page tabellen) og å frakte page'er mellom disk og RAM, kalles ofte paging.*

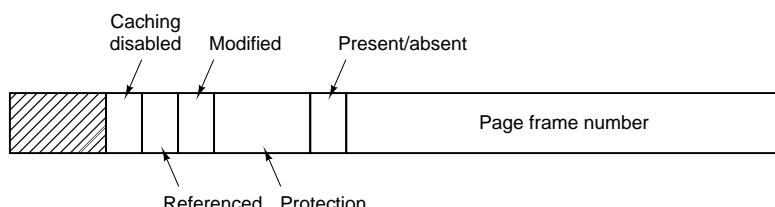
### Internal Operation of the MMU



Figuren viser hvorfor vi må ha en page størrelse  $2^n$ . Fysisk adresse blir page frame nummer med påhekta offset (hekta på som de minst signifikante bit'ene).

Virtuelt minne/virtuelt adresserom finnes altså IKKE, det er en abstraksjon, et adresserom som prosessene forholder seg til. Vi kan tenke på at som regel (dvs ved de fleste typer implementasjoner) så er virtuelt minne representert via page tabellen.

### A Page Table Entry



Present/Absent bit, Protection bits, Modified bit (*Dirty bit*), Referenced bit, Caching disabled bit (viktig ved memory-mapped I/O)

32 bit entries er vanlig.

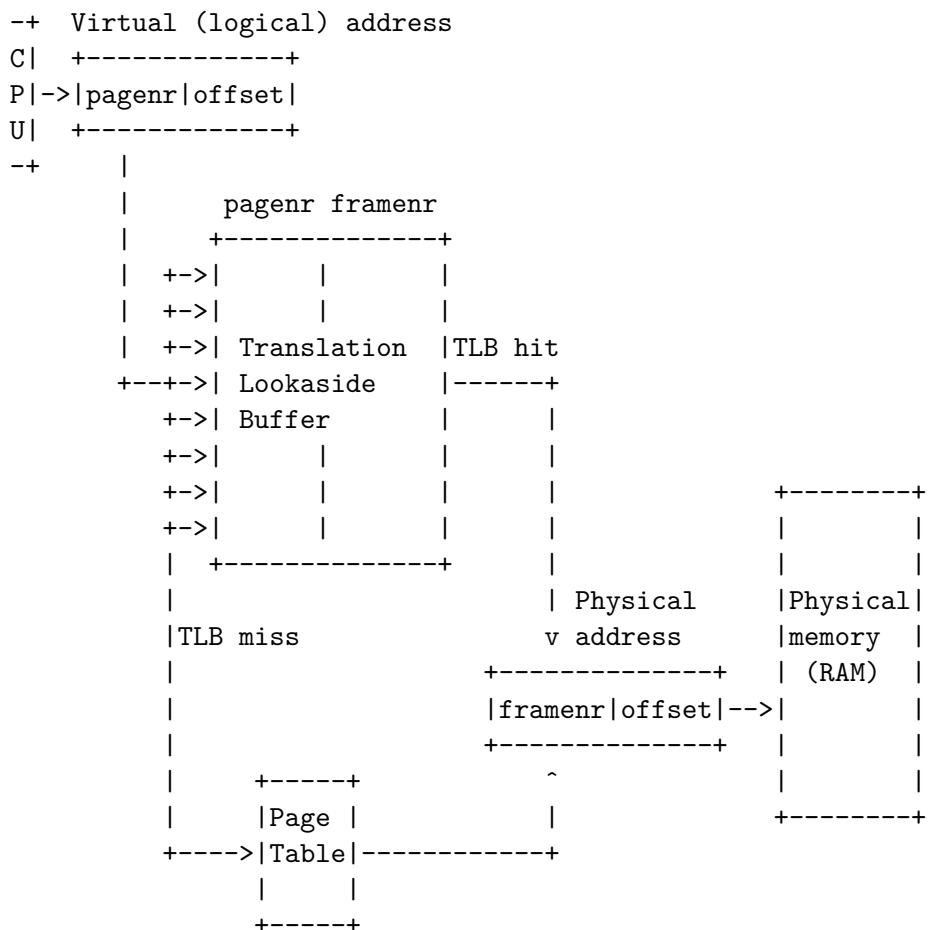
TAVLE:

- mapping virtuell til fysisk adresse må være kjapt (hvis Page table er i RAM, vil være henting av instruks medføre en ekstra RAM-referanse som medfører av programmet vil bruke dobbelt så lang tid i praksis...)
- page tabellen må ikke være for store

### 7.2.2 TLB

#### Translation Lookaside Buffer (TLB)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75



TLB er en slags cache. En hardware device som kan søke på alle sine entries parallelt. En-til-en med feltene i page tabell entriene. Hva slags typisk situasjon viser figuren? (svar: kanksje en for løkke som har sin kode i page 19-21, med data i 129 og 130, med data-index'er i 140 og stack i 860 og 861)

Ved programsekvering vil et stort antall minnereferanser være til et fåtall page'er.

Eksempel:

Anta 10ns for å aksessere TLB, 100ns for å aksessere minnet.

Å hente en byte hvis page er i TLB blir da  $10\text{ns} + 100\text{ns} = 110\text{ns}$ .

Ved en TLB miss får vi  $10\text{ns} + 100\text{ns} + 100\text{ns} = 210\text{ns}$  (altså et tillegg på 100ns siden vi må via page table)

Ved 98% hit rate blir effektiv aksesstid:

$$0.98 \times 110\text{ns} + 0.02 \times 210\text{ns} = \mathbf{112\text{ns}}$$

En TLB har typisk 64 entries.

HVA VIL DET EGENTLIG SI AT TLB ER EN TYPE CACHE?

se fjerde paragraf på [http://en.wikipedia.org/wiki/CPU\\_cache](http://en.wikipedia.org/wiki/CPU_cache)  
(se forhold til L1,L2,L3 og delen om "Example: the K8")

### Page Fault terminology

**TLB miss / Soft miss** Page Table Entry (PTE) is not TLB.

**Minor page fault / Soft miss / Soft (page) fault** Page is in memory but not marked as present in PTE (e.g. a shared page brought into memory by another process)

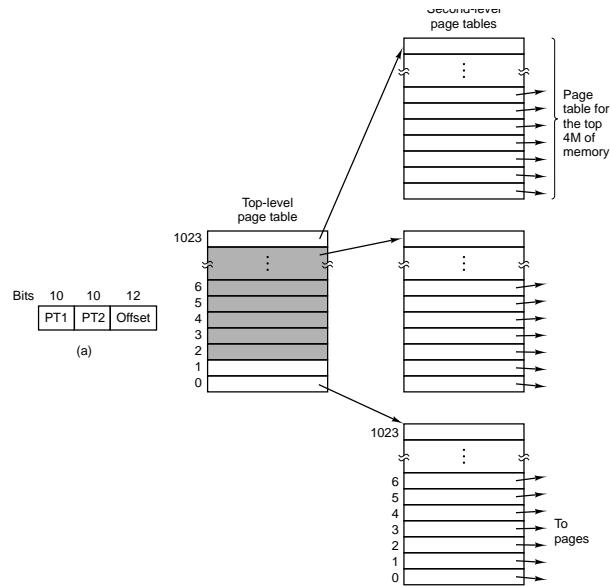
**Major page fault / Hard miss / Hard (page) fault** Page is not in memory, I/O required.

DEMO:

```
\time -v gimp
\time -v gimp
sync ; echo 3 | sudo tee /proc/sys/vm/drop_caches
\time -v gimp
```

### 7.2.3 Multilevel PT

#### Multilevel Page Tables



Gjør at ikke alle page tabellene må være i minnet samtidig.

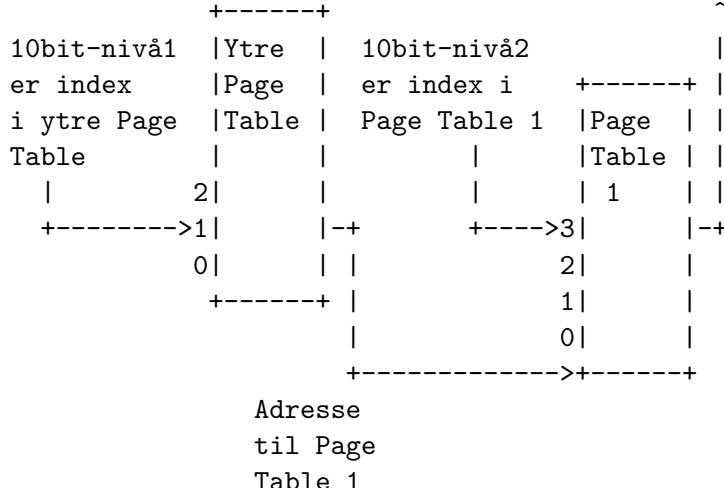
Binary address (0x00403004):

0000 0000 0100 0000 0011 0000 0000 0100

10bit-nivå1| 10bit-nivå2| 12bit-offset -----+  
V

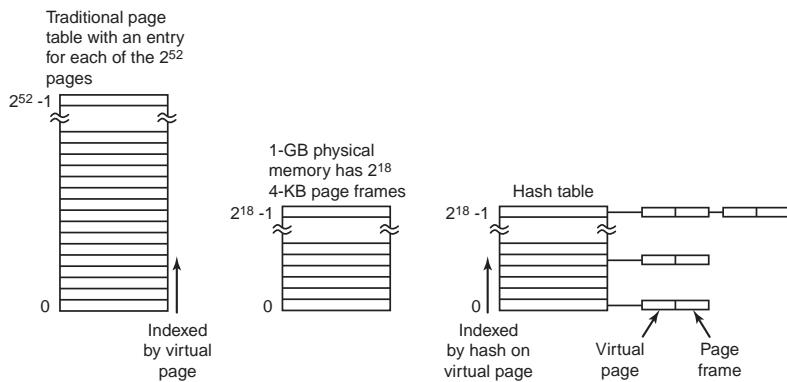
20bit-framenr|12bit-offset -> Fysisk

minne  
(RAM)



#### 7.2.4 Inverted PT

##### Inverted Page Table



Ved invertert page table har page tabellen en entry pr page frame istedet for en entry pr page (page frames pr prosess søkes da opp i page tabellen basert på PID gjerne). Gjør det vanskelig å foreta mapping virtuell til fysisk adresse. *Løses i praksis med TLB!* Løses også med en ekstra hash tabell for page tabellen istedet for lineært søker direkte i page tabellen.

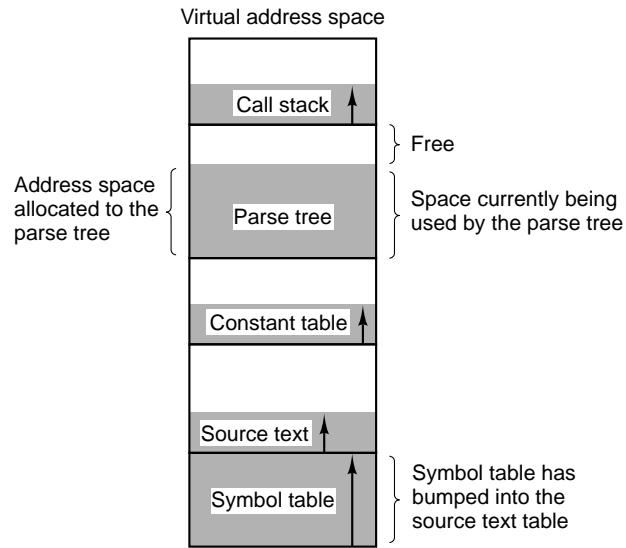
Gå gjennom hat-ipt eksempel (primært punchline tilslutt).

Trenger invertert page table eller multi-level page tabel fordi:

- Ved vanlig page table og 32-bits system, 4KB page størrelse blir det 1M entries som gir en 4MB page table størrelse.
- Ved vanlig page table og 64-bits system, 4KB page størrelse blir det  $\frac{2^{64}}{2^{12}} = 2^{52}$  entries, hver entry på 8 byte gir ca 32 PetaByte page table størrelse !

## 7.3 Segmentation

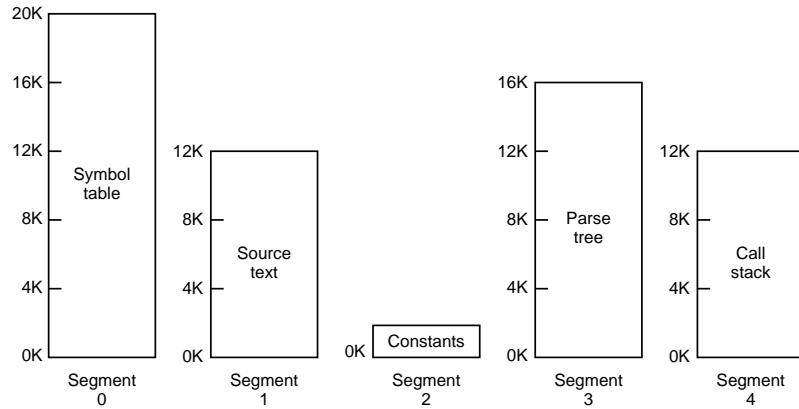
### Growing Tables in One Address Space



I situasjoner hvor mange dataområder vokser (er veldig dynamiske), kan det være fornuftig å dele opp i flere adresserom.

Å dele opp i flere adresserom gjør det også enklere mtp deling av segmenter og beskyttelse av de.

### Segmented Memory



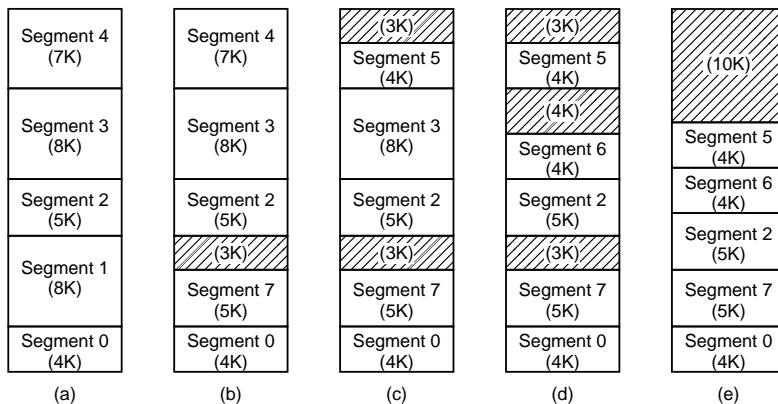
Fem segment i figuren. Et segment er altså et adskilt virtuelt addresserom (en generalisering av det tidligere nevnte skille i I-space og D-space i figur 3.25).

### Paging vs. Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Page'r er av en bestemt størrelse mens segment kan variere i størrelse og kan endre størrelse under kjøring.

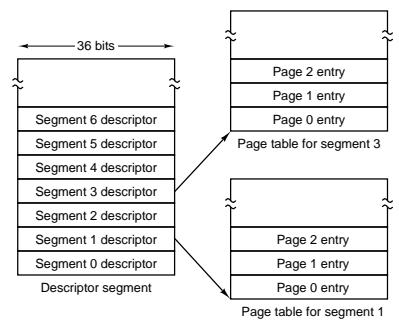
### Pure Segmentation



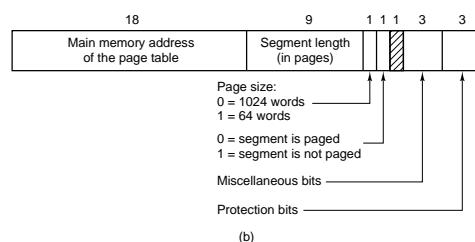
Fenomenet med hullene som utvikles mellom segmentene kaller vi *ekstern fragmentering*. Ekstern fragmentering løses med paging i minne og med en noe liknende mekanisme på harddisker (men på disker er fragmentering fortsatt et stort problem siden en fil tar tid å lese hvis den er delt utover hele disken, vi kommer tilbake til dette i neste tema).

## 7.4 MULTICS

### MULTICS Virtual Memory



(a)



(b)

Physical addresses are 24 bits but pages are aligned on 64-byte boundaries (lower order 6 bits are 000000), only 18 bits are needed to store a page table address.

Page size is either 64 or 1024 words, words are 36 bits

## 7.5 Theory questions

1. Hva brukes en bitmap til i forbindelse med minnehåndtering i et page-basert minne?
2. Hvilket felter finnes i en pagetable entry?
3. Hvilke prinsipper kan vi benytte for å redusere størrelsen på en pagetable i internminnet?
4. Affinity scheduling (“CPU pinning”) reduserer antall cache misser. Reduseres også antall TLB misser? Reduseres også antall page faults? Begrunn svaret.
5. Hvor stor blir en bitmap i et page-inndelt minnesystem med pagestørrelse 4KB og internminne på 512MB?
6. Tanenbaum oppgave 3.5  
For each of the following decimal virtual addresses, compute the virtual page number and offset for a 4K page and for an 8K page: 20000, 32768, 60000.
7. Tanenbaum oppgave 3.10  
Suppose that a machine has 48-bit virtual addresses and 32-bit physical address.
  - (a) If pages are 4K, how many entries are in the page table if it has only a single level? Explain.
  - (b) TLB with 32 entries. Suppose that a program contains instructions that fit into one page and it sequentially reads long integer elements from an array that spans thousands of pages. How effective will the TLB be for this case?

## 7.6 Lab exercises

### 1. Prosesser og tråder.

Lag et script myprocinfo.bash som gir brukeren følgende meny med tilhørende funksjonalitet:

- 1 - Hvem er jeg og hva er navnet på dette scriptet?
- 2 - Hvor lenge er det siden siste boot?
- 3 - Hvor mange prosesser og tråder finnes?
- 4 - Hvor mange context switch'er fant sted siste sekund?
- 5 - Hvor stor andel av CPU-tiden ble benyttet i kernelmode og i usermode siste sekund?
- 6 - Hvor mange interrupts fant sted siste sekund?
- 9 - Avslutt dette scriptet

Velg en funksjon:

Hint: bruk en switch case for å håndtere menyen, og hvert menyvalg bør føre til en echo hvor svaret settes inn via \$(), og for punkt 4-6 kan du hente ut info fra /proc/stat, se <http://www.linuxhowtos.org/System/procstat.htm>

### 2. Skriv om myprocinfo.bash scriptet til å bruke whiptail til å vise meny. Hint: man whiptail og

<http://stackoverflow.com/questions/1562666/bash-scripts-whiptail-file-select>.

# Chapter 8

## Page replacement algoritmer, design og implementering

### 8.1 Page Replace

#### 8.1.1 Optimal

##### Optimal Page Replacement Algorithm

- Label all pages with the number of instructions that will be executed before that page is referenced
- Evict the page with the highest label

Se Virtual Memory på Wikipedia, figur, Hva skjer når RAM blir fyllt opp?

Page faults fremvinger valget av hvilken page frame som skal erstattes (hvis minnet er fullt eller prosessen har nådd sine max page frames den får lov å bruke).

Denne optimale algoritmen kan simuleres slik at de foreslalte reelle algoritmen kan sammenlikne seg med den. Hvis man klarer en algoritme som er innenfor 1-2% av den optimale er ikke det så verst...

#### 8.1.2 FIFO-based

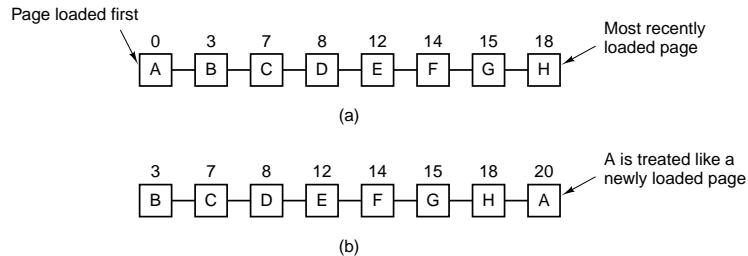
##### FIFO

###### First-In First-Out (FIFO)

- First-in first-out

## Second-chance

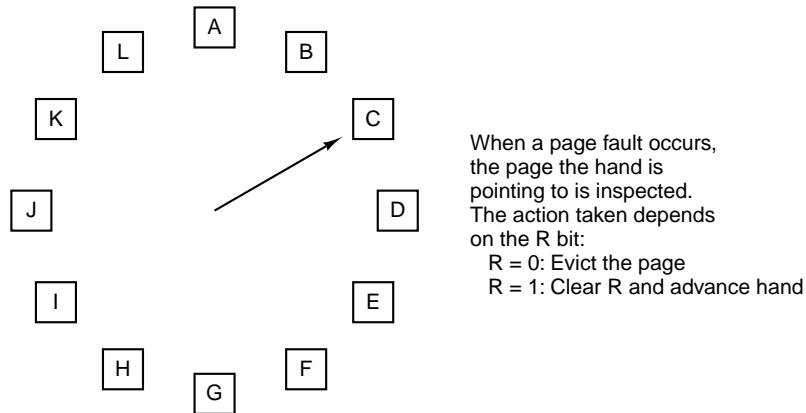
### Second-Chance Page Replacement Algorithm



Er FIFO men benytter R-bit'n i tillegg for å fjerne svakheten ved FIFO.

## Clock

### Clock Page Replacement Algorithm



Clock Page Replacement Algorithm er bare Second-Chance organisert som en ring istedet for en liste for å øke effektiviteten.

ANIMASJON (fjern M-bit) med sekvens:

R17 R1 R2 R7 R4 R9 R6 R3 R8 R4 R5 R10 R11 R12 R4 R13 R14 R15 R9 R5 R16  
 ^ ^ ^ ^ ^ ^ ^

<http://gaia.ecs.csus.edu/~zhangd/oscal/ClockFiles/Clock.htm>

### 8.1.3 LRU-based

De to siste algoritmene, Second-Chance og Clock kan også egentlig sies å være en veldig grov tilnærming til LRU.

#### NRU

##### Not Recently Used (NRU)

- Separate page frames into four classes
  0. not referenced, not modified
  1. not referenced, modified
  2. referenced, not modified
  3. referenced, modified
- Evict a random page from lowest class

M-bitet nullstilles ikke ved hvert klokketikk. Dette er en veldig enkel algoritme, men kanskje god nok. En veldig grov tilnærming til Least Recently Used.

#### LRU

##### LRU Hardware Implementation

	Page			
	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

	Page			
	0	1	2	3
0	0	0	1	1
1	0	1	1	1
2	0	0	0	0
3	0	0	0	0

(b)

	Page			
	0	1	2	3
0	0	0	0	1
1	0	0	1	1
2	1	0	0	1
3	1	1	0	1

(c)

	Page			
	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	1	0	0	0
3	1	1	1	0

(d)

	Page			
	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	1	0	0	0
3	1	1	0	1

(e)

	Page			
	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	1	1	0	1
3	1	1	0	0

(f)

	Page			
	0	1	2	3
0	0	1	1	1
1	0	0	1	1
2	1	0	0	1
3	1	0	0	0

(g)

	Page			
	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	0
3	0	0	0	0

(h)

	Page			
	0	1	2	3
0	0	1	0	0
1	1	1	0	1
2	1	1	1	0
3	1	1	0	0

(i)

	Page			
	0	1	2	3
0	0	1	0	0
1	1	1	0	0
2	1	1	1	0
3	1	1	1	0

(j)

En direkte realisering av LRU er en lenka liste over alle page frames hvor hver gang en adresseres flyttes den først i lista slik at de som der er lengst siden ble brukt er bakerst i lista og kan fjernes. Dette er svært upraktisk, så de trengs en annen tilnærming, enten hardware støtte eller software-liknende tilnærming.

Poenget med figuren er at matrisen er  $n \times n$  hvor  $n$  er antall page frames.

*Least Recently Used (LRU)* holder rede på hvor lenge siden det er siden hver page ble addressert. Hardware støtte sjeldent tilgjengelig, simulering i software kan gjøres via *Not Frequently Used (NFU)* (hvor en teller for hver page inkrementeres ved hvert klokketikk hvis page'n har sin R=1 da, i animasjonen er denne kalt MFU men er feil implementert så ikke se på den!) men den glemmer aldri noe, så *aging* er bedre.

ANIMASJON (LRU med 3 frames og sekvens nedenfor) og tavle:

		h			h			h			h		
2	3	2	1	5	2	4	5	3	2	5	2	5	2
		^-----^			^-----^			^-----^			^-----^		

Fordi LRU teller for frame 0, 1 og 2 blir:

0	0	1	0	1	2	0	1	2	0	1	2	3
1		0	1	2	0	1	2	0	1	2	0	1
2			0	1	2	0	1	2	0	1	0	

(LRU erstatter alltid den frame med minst verdi)

(MERK: se animasjonen samtidig, der ser du hvordan page'ene faktisk blir plassert i de respektive page frames, mens tabellen jeg har laget her bare viser verdien av LRU-telleren for hver page frame)

(repeter med FIFO først som gir flere page faults)

(kommenter at 3 erstattter 5 og 5 erstatter 2 selv om 5 og 2 nettopp har blitt brukt, derfor kan ikke denne brukes i praksis, men enkel å forstå!)

<http://gaia.ecs.csus.edu/~zhangd/oscal/PagingApplet.html>

## NFU

### Not Frequently Used (NFU)

- A counter for each page frame which is incremented at every clock tick if its R-bit is 1 (clears the R-bit after incrementing of course)
- The page frame with the lowest counter is replaced at a page fault
- *never forgets anything...*

(egentlig MFU i animasjonen, men er feil implementert der)

## Aging

### Aging Page Replacement Algorithm

	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
Page	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10010000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

Merk: trykkfeil for page 2 mellom (c) og (d), bit'et blir ikke forskjøvet...

Aging skiller seg fra LRU fordi aging benytter hele klokkesyklus som enheter og ikke hver enkelt minneaddressering som LRU, samt at aging har langt færre bit til å ta vare på historikken (som regel 8-bit men det har vist seg å være nok).

TAVLE:

*Demand paging* betyr at page frames allokeres ved behov istedet for at noen allokeres på forhånd. Hvis page frames allokeres før prosessen kjøres kalles det *prepaging*. Det settet med page'r som en prosess holder på å bruke kalles *working set*. Hvis det ikke er nok tilgjengelig minne for en prosess til å ha hele prosessens working set i fysisk minne, så blir det veldig hyppige page fault, og programmet er i en tilstand vi kaller *thrashing*.

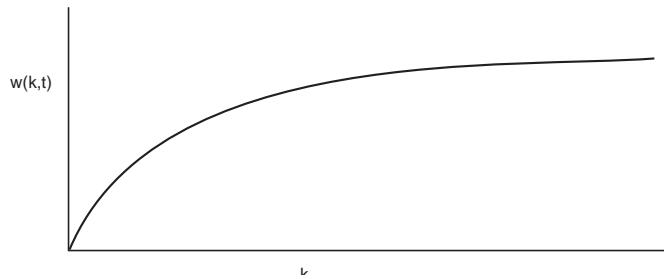
Sjekk gjerne Microsoft teknologiene Superfetch og ReadyBoost.

```
# Linux
top
# Windows
vmmmap
refresh
# høyreklikk på powershell prosessen
empty-ws
ls # i powershell
refresh
```

### 8.1.4 Working set-based

#### Working set

##### Size of Working Set



Merk: på Unix/Linux kalles working set for *Resident Set Size (RSS)*.

$w(k, t)$  er størrelsen på working set i ett gitt tidspkt  $t$  ut fra de  $k$  siste minneaddresseringene. En viktig egenskap ved working set er at det endres sakte med tiden. Som regel måler man ikke working set ut fra antall minneaddresseringer tilbake i tid, men bruker CPU-tid istedet.

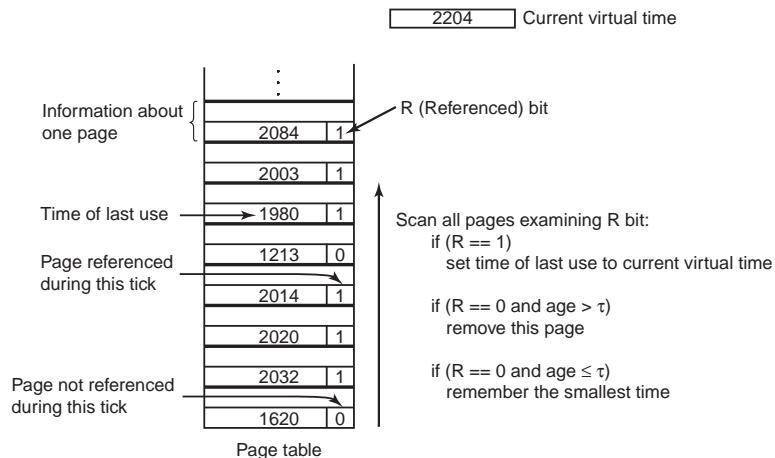
#### Remember Why Cache Works

- Locality of reference
  - *spatial locality*
  - *temporal locality*

Cache gjør at programmer kjører raskere fordi instruksjoner gjenbrukes ofte (samlokalisert i tid) mens data ofte aksesseres blokkvis (samlokalisert i rom, har du lest en byte skal du sikkert snart ha byte'n ved siden av den også). Dette er logikken bak working set også: kode og data gjenbrukes i veldig stor grad, og noen deler brukes mye oftere enn andre.

#### Working Set Page Replacement Algorithm

## 8.1. PAGE REPLACE

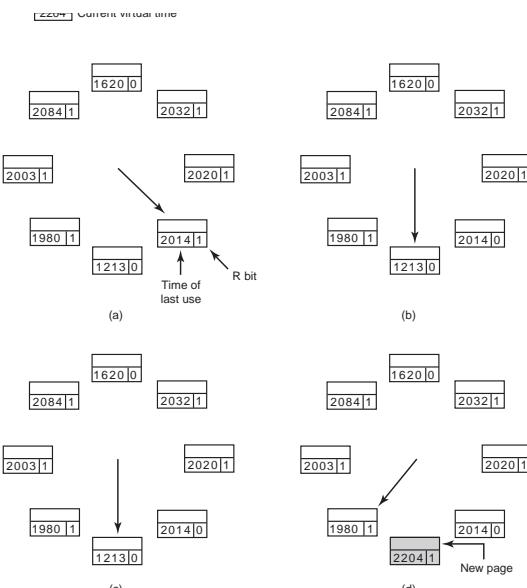


Periodisk klokkeinterrupt setter alle R-bit til 0. Current virtual time er CPU-tid for denne prosessen.

Ved page fault oppdateres altså current virtual time for alle som har sin R-bit satt for indikere at disse var nylig brukt, dvs en del av working set i det page fault'n forekom.

### WSClock

#### WSClock Page Replacement Algorithm



Basert på Clock algoritmen men benytter working set informasjon i tillegg.

Hvis en frame er dirty schedules en write og man hopper midlertidig videre.

*Hovedpoenget med WSClock fremfor LRU er at det holder at en page frame er eldre enn  $\tau$ , man må ikke finne den som er least recently used.*

### Page Replacement Algorithms Summary

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

## 8.2 Design Issues

### Local or Global Page Replacement

Age	10	9	8	7	6	5	4	3	2	1	0
A0											
A1											
A2											
A3											
A4											
A5											
B0											
B1											
B2											
B3											
B4											
B5											
B6											
C1											
C2											
C3											

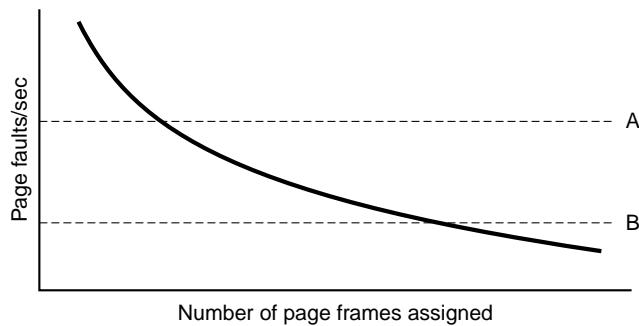
A0											
A1											
A2											
A3											
A4											
A5											
B0											
B1											
B2											
B3											
B4											
B5											
B6											
C1											
C2											
C3											

A0											
A1											
A2											
A3											
A4											
A5											
B0											
B1											
B2											
B3											
B4											
B5											
B6											
C1											
C2											
C3											

Globale algoritmer funker generelt bedre. Lokale algoritmer sliter hvis working set endrer seg mye over tid, hvis det øker kan man få thrashing selv om det totalt sett er ledige page'r, hvis det minker kan det hende det løses med minne. Uansett må det styres hvor mange page frames hver prosess skal kunne benyttet. En grei måte å gjøre det på er å se på *page fault frequency*.

### 8.2.1 Page faults

#### Page Fault Rate



A betyr for mange page faults, mens B betyr for få, dvs kanskje prosessen har fått alllokert unødvendig mye minne.

Hvordan funker dette i praksis på f.eks. Windows? <http://channel9.msdn.com/Events/TechEd/NorthAmerica/2011/WCL406> (Paging 23:30-34:00; SuperFetch 49:00-50:28; WorkingSet, Local Page Replacement og LRU med aging 05:27-07:28; Copy On Write 19:30-20:32).

**Last kontroll** Av og til er det lureste å swappe en hel prosess til disk for å frigjøre alle den page frames. Altstå god gammeldags swapping er fortsatt nyttig. Dette må sees i sammenheng med i hvilken grad prosessene i minnet er CPU-bound eller I/O-bound slik at man forsøker utnytte systemet optimalt til enhver tid.

**Page størrelse** Stor page størrelse kan føre til mye *intern fragmentering*, mens for liten page størrelse gir store page tabeller.

(Husk: forskjell på intern fragmentering, ekstern fragmentering og data fragmentering.)

Overhead ved paging kan beregnes vha gjennomsnittlig prosess størrelse  $s$ , page størrelse  $p$  og hver page entry trenger  $e$  bytes. I tillegg blir det gjennomsnittlig intern fragmentering pr prosess  $p/2$ :

$$\text{overhead} = se/p + p/2$$

deriverer mhp  $p$  og setter lik 0:

$$-se/p^2 + 1/2 = 0$$

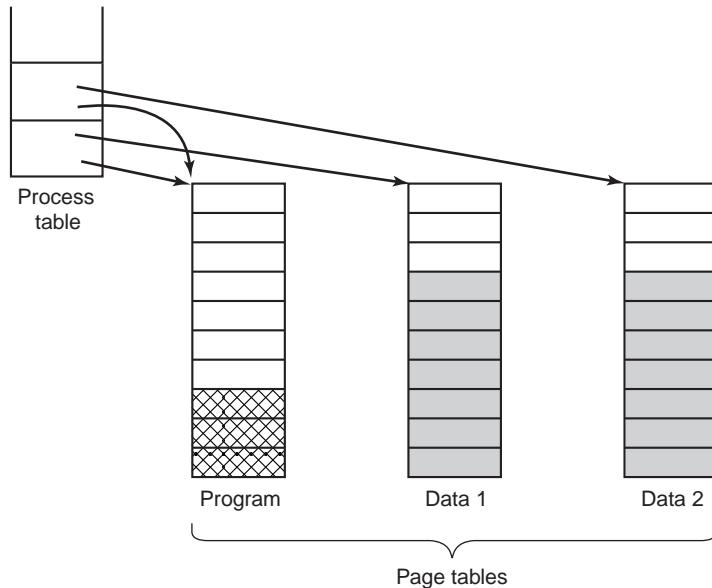
gir

$$p = \sqrt{2se}$$

som ved  $s=1\text{MB}$  og  $e=8\text{bytes}$  (som er typiske størrelser) gir  $p=4\text{K}$ .

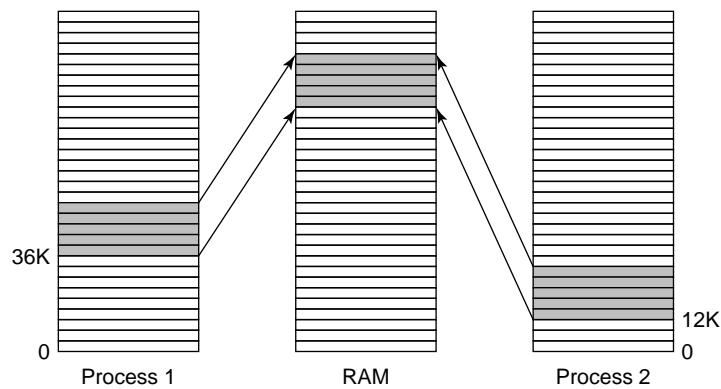
### 8.2.2 Sharing

#### Sharing Program Pages



I praksis skjer deling av page'r svært ofte, f.eks. ved fork av en ny prosess lages ingen ny kopi før en page modifiseres, dette er et viktig prinsipp som kalles *copy on write*.

#### Shared Library



Det er svært viktig å benyttet delte biblioteker fremfor å statisk kompile inn alt i hver enkelt executable fil. Det sparer enormt mye diskplass og har en stor sikkerhetsmessig fordel at et bibliotek kan oppgrades uten at alle programmene som linker til det må det. Delte bibliotek er et særlig tilfelle av det mer generelle *memory-mapped filer*.

KODEEKSEMPEL (legg inn scanf for å stoppe):

<http://www.ecst.csuchico.edu/~beej/guide/ipc/mmap.html>

## 8.3 Implementation

### 8.3.1 OS involvement

#### OS Involvement in Paging

**Process creation** Create (allocate memory and possibly also swap area) the page table dependent on the size of the program.

**Process execution** Reset MMU, flush TLB.

**Page fault** Which virtual address caused the page fault? swap page to disk and read needed page into the page frame.

**Process exit** Release page table and page frames.

### 8.3.2 PF handling

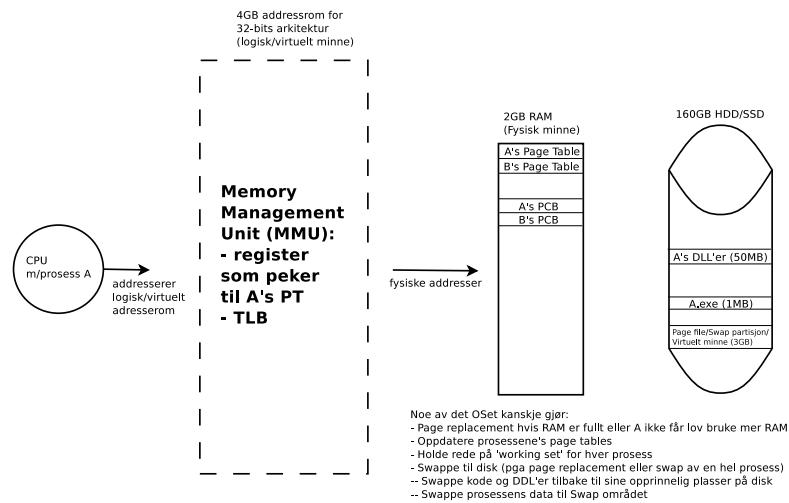
#### Page Fault Handling

1. Interrupt to kernel, general registers stored away.
2. OS finds out which virtual page is needed, checks that address is valid, and access (protection bits) is ok.
3. If page frame is dirty, the page is scheduled for transfer to disk, process suspended until write completed.
4. The wanted page is transferred from disk (process suspended while this happens).
5. Page table is updated.
6. Instruction causing the page fault is restarted.
7. The process is scheduled, registers restored and return to user space execution.

Viktig å få med seg *If page frame is dirty....*

## 8.4 Summary

### Har vi kontroll?



En oppsummerende foil: er vi sikre på at vi skjønner dette bildet? hvis ja, så har vi kontroll!

Og hvorfor har vi en stor pagefile.sys igjen?

<http://support.microsoft.com/kb/2160852>  
(skumles alt)

## 8.5 Theory questions

1. Hva menes med page replacement? Beskriv hva som skjer ved denne aktiviteten.
2. Hva menes med *working set* og *thrashing* i forbindelse med minnehåndtering?
3. Hva er hensikten med et swapområde?
4. Tanenbaum oppgave 3.28

A computer has 4 page frames. The time of loading, time of last access, and the R and M bits for each page are (the times are in clock ticks):

Page	Loaded	Last ref.	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

- (a) Which page will NRU replace?
  - (b) Which page will FIFO replace?
  - (c) Which page will LRU replace?
  - (d) Which page will second chance replace?
5. Tanenbaum oppgave 3.9
- 32-bit address space and 8K page size, the page table is entirely in hardware, with one 32-bit word pr entry. When a process starts, the page table is copied from memory to hardware, at one word every 100 nsec. Each process runs for 100 msec (incl the time to load the page table), what fraction of CPU time is devoted to loading the page table?

## 8.6 Lab exercises

### 1. Hjelp til lesbarhet: byte konvertering.

Skriv et script `human-readable-byte.bash` som leser et tall, antall bytes, fra STDIN, konverterer tallet til KB, MB, GB eller TB avhengig av størrelsen på tallet, og skriver den nye størrelsen i heltall. Scriptet bør oppføre seg omtrent slik:

```
$ echo 25000000 | bash human-readable-bytes.bash
23MB
$ echo 250 | bash human-readable-bytes.bash
250B
```

### 2. Hjelp til lesbarhet: ns konvertering.

Skriv et script `human-readable-ns.bash` som leser et tall, antall ns (nanosekunder), fra STDIN, konverterer tallet til us (microsekunder), ms (millisekunder) eller sek (sekunder) avhengig av størrelsen på tallet, og skriver den nye størrelsen i heltall. Scriptet bør oppføre seg omtrent slik:

```
$ echo 25000000 | bash human-readable-ns.bash
25ms
$ echo 250 | bash human-readable-ns.bash
250ns
```

### 3. Beregne effektiv minneaksessstid.

Skriv et script `compute-memory-access.bash` som beregner effektiv minneaksessstid gitt TLB-aksessstid `TLB=2ns`, RAM-aksessstid `RAM=10ns` og Disk-aksessstid `DISK=10ms` (med andre ord, sett disse parameterene som variable i starten av scriptet). Scriptet skal ta to kommandolinjeargumenter: andel TLB-hit og andel page faults. (Hint: det er litt knotete å få til desimalaritmetikk i bash, så google for å finne en løsning ala `SVAR=$(echo "1-0.5" | bc)`) Scriptet bør oppføre seg omtrent slik (bruk `human-readable-ns.bash` scriptet ditt til utskriften, og sikre deg at du har et heltall med noe ala `$(echo $SVAR | cut -f 1 -d .)`):

```
$ bash compute-memory-access.bash 0.7 0.1
1000016.2 nanosekunder som er
1ms
$
```

### 4. En prosess sin bruk av virtuelt og fysisk minne.

Skriv et script `procmi.bash` som tar et sett med prosess-ID'er som kommandolinjeargument og for hver av disse prosessene skriver til en fil `PID-data.meminfo` følgende info:

- Total bruk av virtuelt minne ( $VmSize$ )

(hjelp meg finne ut hvorfor  $VmSize > VmData + VmStk + VmExe + VmLib$ )

- (b) Mengde virtuelt minne som er privat ( $VmData + VmStk + VmExe$ )
- (c) Mengde virtuelt minne som er shared ( $VmLib$ )
- (d) Total bruk av fysisk minne ( $VmRSS$ )
- (e) Mengde fysisk minne som benyttes til page table ( $VmPTE$ , som da gjerne er multi-level, slik at det vil variere mye fra prosess til prosess)

Hint: `man proc` og les om filen `/proc/[pid]/status`, samt se i filen `/proc/meminfo`. Eksempel kjøring:

```
$ bash procmi.bash 1 1985 17579
$ ls 17579*
17579-20100216-15:26:31.meminfo
$ cat 17579-20100216-15\:26\:31.meminfo

***** Minne info om prosess med PID 17579 *****
Total bruk av virtuelt minne (VmSize):    3444KB
Mengde privat virtuelt minne (VmData+VmStk+VmExe):    516KB
Mengde shared virtuelt minne (VmLib):    1596KB
Total bruk av fysisk minne (VmRSS):    1112KB
Mengde fysisk minne som benyttes til page table (VmPTE):    32KB

$
```

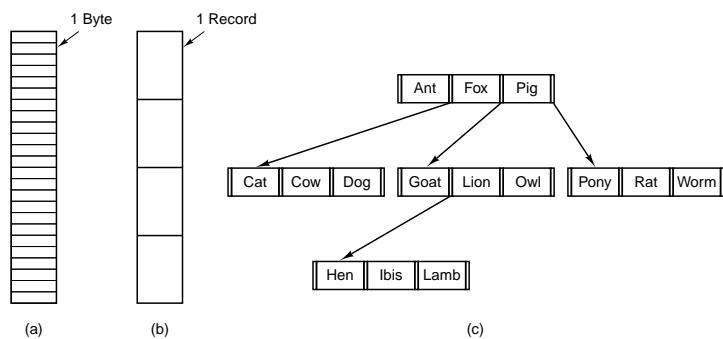


# Chapter 9

## Filsystemimplementasjon, EXTFS

### 9.1 Files & Dirs

#### File Structure



Tre typer filer.

- (a) (a) er en ustrukturert sekvens av bytes, det viktigste er at operativsystemet gir blaffen i filtyper, en fil er bare en fil. Det er applikasjonen som må håndtere filtyper. All windows og UNIX/Linux idag funker slik.
- (b) (b) benyttes ikke lenger, ble benyttet for å ha records lik "hullkort-bilder"
- (c) (c) benyttes på mange stormaskiner fortsatt (på IBM stormaskiner så kalles disse filene et dataset)

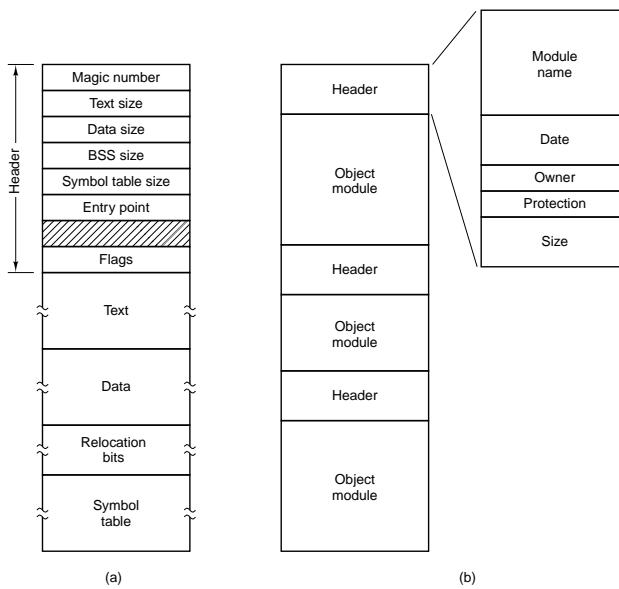
TAVLE:

1. Regular files: brukerdata, ascii eller binære (ascii benytter bare 7 av 8 bit pr tegn, nevne UTF-16: koding med 2 bytes)

2. Directories: systemfiler
3. Symbolske linker (UNIX/Linux) (Symlinks/Shortcuts in Windows)
4. Character devices (UNIX/Linux): serielle I/O enheter
5. Block devices (UNIX/Linux): diskar (alt som kan addresseres blokkvis)

Sekvensiell og random aksess (bare random aksess idag)

### Executable Files



(a) viser et typisk exe filformat (Dataområdet er initialiserte globale variable, BSS er ikke initialiserte globale variable, Symboltabellen brukes for debugging, entry point angir start for koden).

(b) viser et gammeldags arkiv (fra tiden da `ar` ble benyttet for biblioteker i motsetning til nå hvor `tar` brukes til alt).

DEMO:

```
file (/usr/share/file/magic)
```

#### 9.1.1 Attributes

##### File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Filattributter = metadata

DEMO:

`ls -ltra`

## 9.1.2 Operations

### File Operations

1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek
9. Get attributes
10. Set attributes

## 11. Rename

Dette er de mest vanlige systemkallene knyttet til filer, men eksakt hvilke som finnes varierer litt mellom de forskjellige operativsystemene.

*Hvorfor gjør vi egentlig en open? dvs trenger vi å gjøre open?* Nei, egentlig ikke, men det gjør alle påfølgende fil aksesser så mye raskere hvis filattributtene er lastet og sjekket på forhånd, og at man får henter inn diskaddresser i minne før man starter filaksessen.

```
#include <sys/types.h>                                /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);                      /* ANSI prototype */

#define BUF_SIZE 4096                                     /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                                  /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);                             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);                   /* open the source file */
    if (in_fd < 0) exit(2);                            /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);              /* create the destination file */
    if (out_fd < 0) exit(3);                            /* if it cannot be created, exit */

    while (1) { /* Copy loop */
        rd_count = read(in_fd, buffer, BUF_SIZE);      /* read a block of data */
        if (rd_count <= 0) break;                       /* if EOF or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count);     /* write data */
        if (wt_count <= 0) exit(4);                     /* wt count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                                /* no error on last read */
        exit(0);
    else
        exit(5);                                     /* error on last read */
}
```

Merk: in\_fd og out\_fd er fildeskriptorer som skapes med systemkallet open.

sjekk /usr/include/bits/fcntl.h for å se hva O\_RDONLY betyr (greit å vite når vi skal se inni fdinfo også)

merk at dette ikke er direkte systemkall men de tilhørende C-biblioteksfunksjonene

DEMO (med sleep):

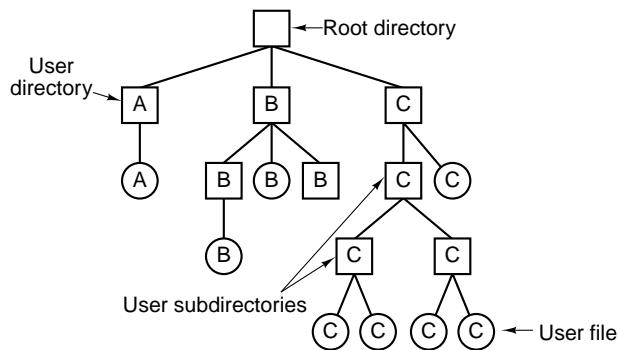
```
./cp-read-write 10MBfile a
ls -l /proc/$(pidof cp-read-write)/fd
cat /proc/$(pidof cp-read-write)/fdinfo/{3,4}
```

EXTRA DEMO:

La oss samtidig se hva som egentlig skjer når det benyttes pipes, vi så nettopp at alle prosesser har standard fildeskriptorer 0 (STDIN), 1 (STDOUT) og 2 (STDERR), se hva som skjer med disse når vi lager en pipe:

```
cat | wc -l
ls -l /proc/$(pidof cat)/fd # Merk rettigheten som er satt med 'w'
ls -l /proc/$(pidof wc)/fd   # og 'r' her
```

## Directories



Alle kataloger har entries for . og .. (Merk for / (rota) betyr de det samme).

Hver prosess har en CWD

DEMO:

```
ls -l /proc/$(pidof bash)/
```

## Directory Operations

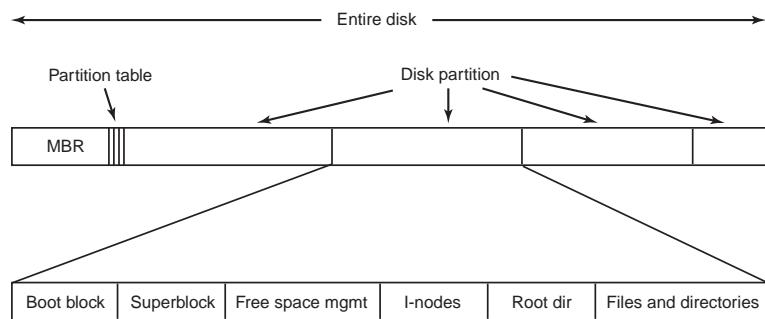
1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

Kommer tilbake til linking senere i shared files.

## 9.2 File System Implementation

### 9.2.1 Layout

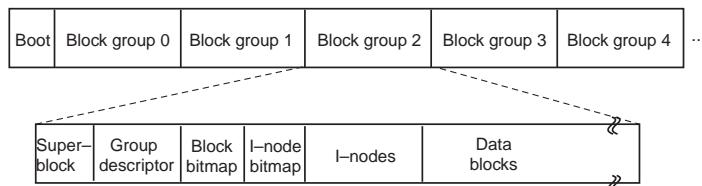
#### File System Layout



I all hovedsak er et filsystem bare en datastruktur, samt håndteringen av denne. Men det er en veldig interessant datastruktur siden den er i bruk så og si hele tiden.

Se [http://en.wikipedia.org/wiki/Partition\\_table](http://en.wikipedia.org/wiki/Partition_table) og spesielt den pågående overgangen innen teknologiene for Firmware/Partisjonstabell (fra BIOS/MBR til UEFI/GUID).

#### File System Layout: Linux ext2



ext2 filsystemet deler inn i blokkgrupper og forsøker å dele directories utover gruppene og la filer i samme directories være i samme gruppe (for å minske disk arm forflytting, dvs øke ytelsen)

DEMO:

```
sudo debugfs /dev/sda1 (gjør at man ser på filsystemdetaljer read-only)
show_super_stats ("Show superblock statistics", viser innholdet i superblokken som
er kopiert inn som start på hver gruppe, og viser alle gruppene)
```

Størrelse på partisjon bør kunne regnes ut fra:

Blokkgrupper × Blokker pr gruppe × Blokkstørrelse  
(kontroller med å se på /dev/sda1 med df -h)

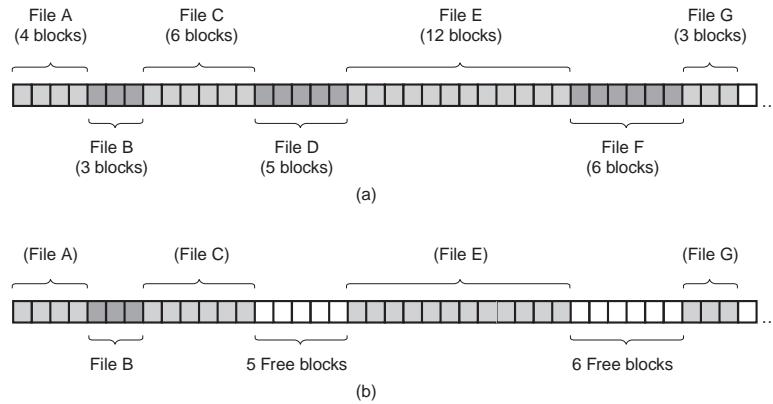
Men før vi ser mere detaljer om ext filsystemet, la oss gå tilbake til teorien et øyeblink:

Det mest interessante spørsmålet er kanskje: *Hvilke diskblokker tilhører hvilke filer?*

### 9.2.2 Allocation

#### Contiguous

##### Contiguous Disk Space Allocation



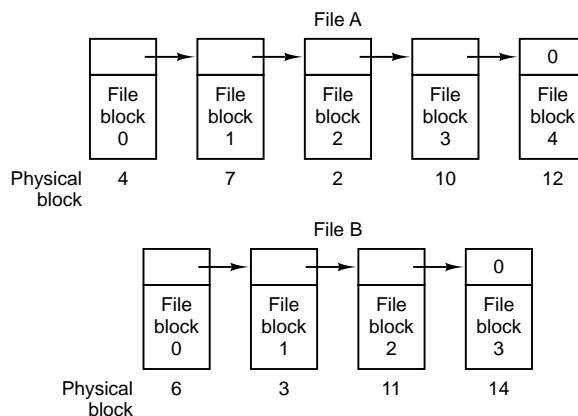
Dette er altså “sammenhengende allokering”, filer lagres som *en* sammenhengende sekvens av blokker. (Dette funker jo ikke noe særlig i praksis i read-write filsystemer, men prinsippet tilstrekkes i vanlige filsystemer *for deler av filer*, dvs en stor fil vil i de fleste filsystemer forsøkes lagret mest mulig samlet.)

Enkelt og rask sekvensiell lesing (trenger bare vite startblokk og filstørrelse), men fører til mye fragmentering og det er jo umulig å vite filstørrelser på forhånd (hvor stor er DV-fila du skal hente fra kamera?), men brukes selvfølgelig ved read-only filsystemer hvor filstørrelsene er kjente (f.eks. CD og DVD).

(“History repeats itself” sammenhengende lagring var mye benyttet på magnetbånd for lenge siden, og er kommet tilbake for CDr/DVDr).

#### Linked list

##### Linked List Allocation



Problematisk med random aksess, samt at mengden data i en blokk ikke lenger er  $2^n$  siden deler av blokka brukes til metadata (peker til neste blokk).

## FAT

### File Allocation Table

Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

Annotations indicate the start of File A at block 4 and File B at block 6, and mark block 15 as an unused block.

TAVLE:

Filene befinner seg i fysiske blokker:

A: 4-7-2-10-12

B: 6-3-11-14

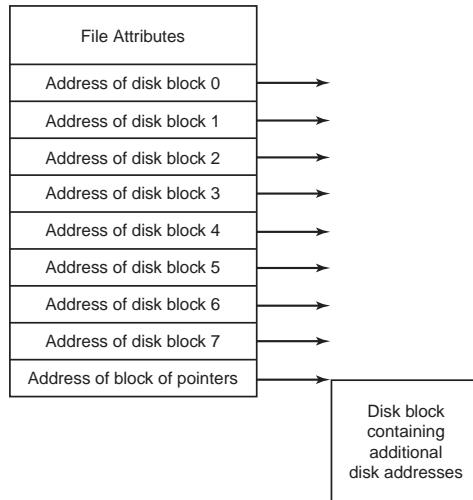
FAT løser problemet med sammenhengende lister og er jo mye brukt (FAT32).

Problemet er at størrelsen på tabellen avhenger av størrelsen på disken. Feks. med en 500GB disk eller partisjon, 2KB blokkstørrelse og 4B pr entry blir tabellen  $(500GB/2KB) \times 4B = 1000MB$  som må være i minne for at oppslag skal være effektive.

I tillegg er random aksess tregt siden man hele tiden må søke gjennom blokk-kjeden (f.eks. 4-7-2-10-12 for A over) beskrevet i FAT.

## I-node

### Example I-node



Løser FAT-problemet med “en FAT pr åpen fil”. Trenger bare å ha i-nodene tilsvarende de åpne filene i minne om gangen.

Hvis filen er større enn tilgjengelig antall blokkaddresser benyttes siste blokkadresse som en peker til et sett nye blokkaddresser (som igjen kan benytte sin siste blokkadresse som peker til enda et sett med blokkaddresser osv).

### 9.2.3 Ext2fs

#### Linux (ext2fs) I-node (128 bytes)

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	60	Address of first 12 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

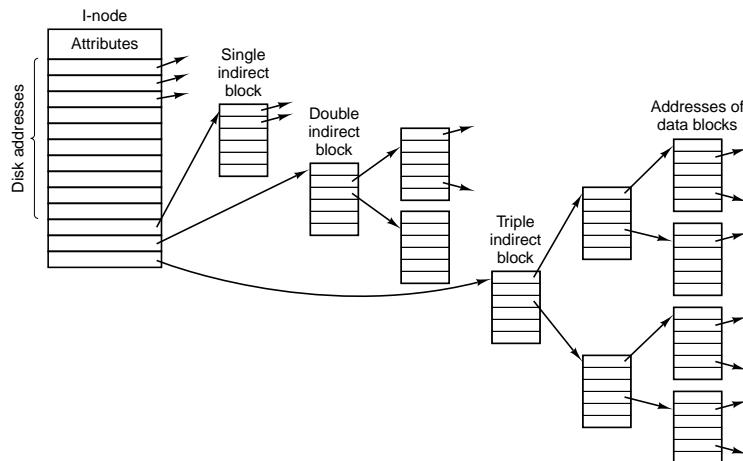
DEMO:

`stat a`

Blocks er antall blocks alllokert for filen (IKKE hvor mange er i bruk).

IO Block er “optimal blocksize for I/O”.

## UNIX System V File System



De 12 første diskadressene er lagret i i-noden. For store filer må single, double eller triple indirect block benyttes.

TAVLE:

Anta diskblokkstørrelse på 1KB (blkstr), 4bytes diskblokkadresser (blkaddr).

$12 \times 1KB = 12KB$  går i i-noden.

$256 \times 1KB + 12KB = 268KB$  med single indirect.

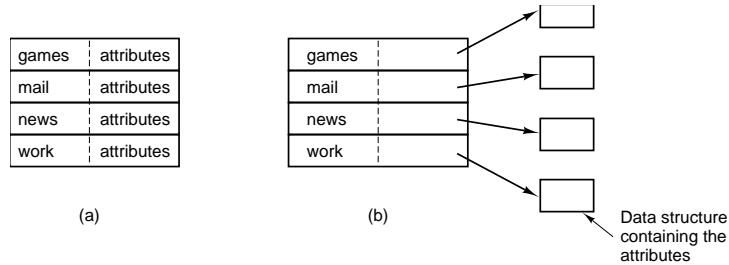
$256 \times 256 \times 1KB + 12KB = (2^{16} + 12)KB$  (64MB) med double indirect.

$256 \times 256 \times 256 \times 1KB + 12KB = (2^{24} + 12)KB$  (16GB) med triple indirect.

**Max partisjonsstørrelse** En partisjon har en 32-bits variabel "block count" som gjør at max partisjonsstørrelse blir 16TB (4KB blokkstørrelse:  $2^{12} \times 2^{32}$ ) (1EB i ext4).

**Max filstørrelse** Med 4KB blokkstørrelse kan man oppnå filstørrelser på 4TB ( $2^{10^3} \times 2^2\text{KB} = 2^{32}\text{KB} = 2^{42}\text{B} = 4\text{TB}$ ) (16TB i ext4).  
 (wikipedia sier halvparten for ext3 av en eller annen grunn?)

### Implementing Directories



En haug med filer er ikke mye nyttig, vi må kunne strukturere dem mer enn bare å ha dem i en stor haug.

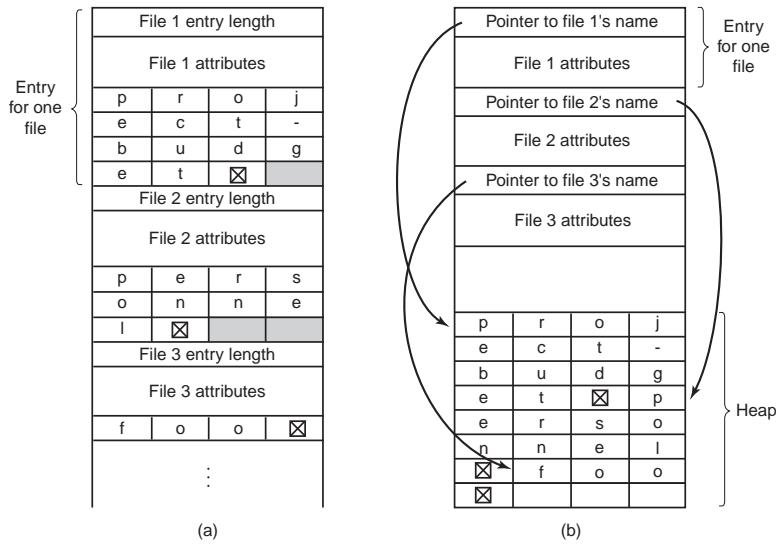
TAVLE:

To typer directories:

1. filnavn, adresse og alle attributter i selve directory entrien
2. bare filnavn og peker til i-node

TEGN 1 root dir (firkant tabell med pekere) med 2 subdir, 1-2 filer (sirkler) i hver, og tegn sirkel rundt dir tilslutt for å illustrere at de bare er filer de og.

### Two Ways of Handling Long File Names



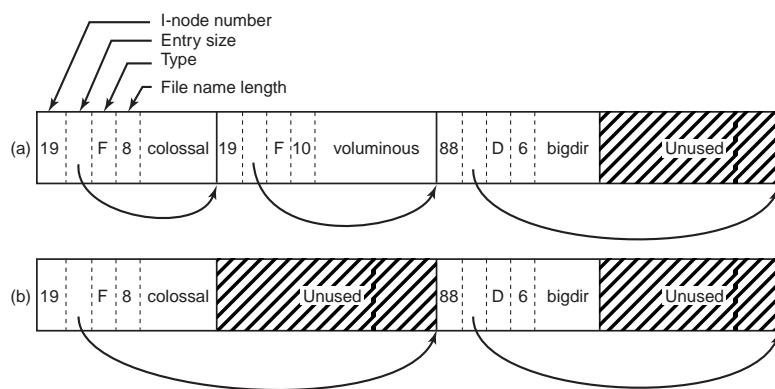
TAVLE:

To måter å håndtere lange variable filnavn på:

1. in-line, enkelt men gir variable størrelse på dir entry (f.eks. kan et entry lettere ligge i skille mellom to pages og generere page fault)
2. in-a-head, gir standard entry størrelse men litt mer komplisert med peker til filnavnet

Generelt lineært søk ikke noe problem, men store directories håndteres med "caching og hashing".

### A Linux Directory with Three Files



NB! Feil i figuren, i-node nr til voluminous skal være 42, ikke 19.

Entry size trengs hvis det er gjort noe padding (f.eks. når en fil er slettet som i eksempel, det er dette pilene illustrerer).

ext2 benytter altså in-line filnavn men ingen attributter, alt finnes i i-noden.

DEMO:

```
sudo debugfs /dev/sda1
stat / # finner blokknr som root dir er i, bruker den i skip:
sudo dd if=/dev/sda1 bs=4k count=1 skip=1283 | xxd
```

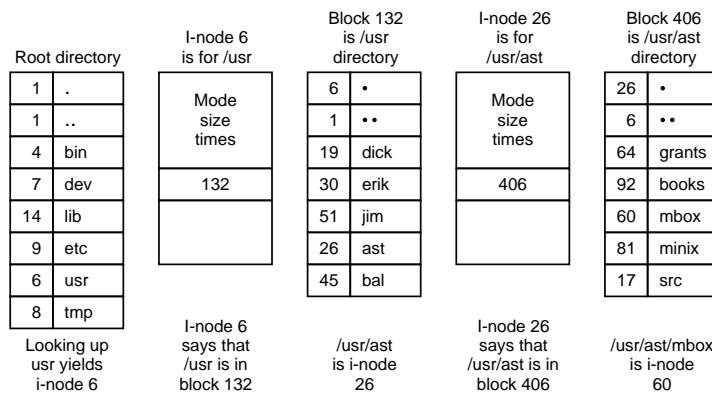
(vi finner igjen directory navn og ser at de har variabellengde entries)

### Key Issues in File Lookup

1. How to get to *root directory* (/, C:\)?
2. How are *files connected to directories*?
3. How are *data blocks connected to files*?
4. Are *directories implemented as files*?

La oss svare på disse fire spørsmålene i EXT, FAT og NTFS.

### Lookup: /usr/ast/mbox

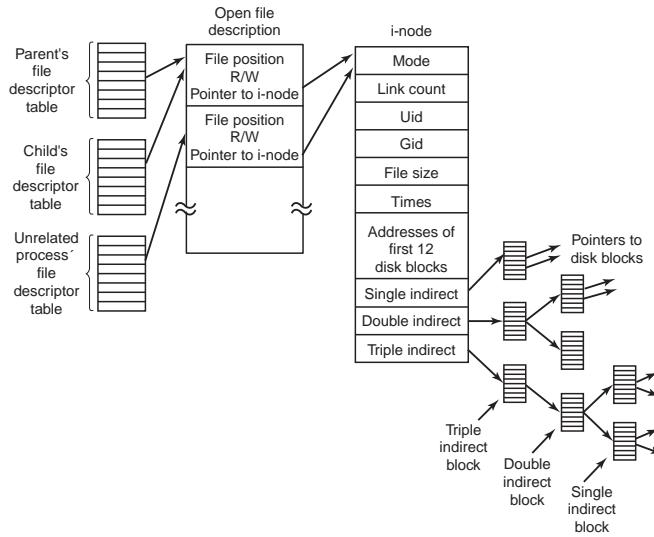


```
# ext3-image is dd of ext3 memory stick with all zeros
# except dir structure /home/erikh/{a.txt,cf3.msi}
#
```

```
##### Find root dir
#
# root dir is always in inode nr 2, and this is in block group 0, and
# the GDT gives the datablock of the inode table (which we cant read):
fsstat -f ext ext3-image
#
##### Look in root dir's inode
#
# root dirs attributes:
istat -f ext ext3-image 2
#
##### Look in root dir's datablocks
#
# hexdump of inode 2 (root's) datablock:
dd if=ext3-image bs=1k count=1 skip=510 | xxd
# find inode of /home:
ifind -f ext -n /home ext3-image
#
##### Look in home dir's inode
#
# home dirs attributes:
istat -f ext ext3-image 27889    # hex 6CF1, little endian?
#
##### Look in home dir's datablocks
#
# hexdump of inode 27889 (home's) datablock:
dd if=ext3-image bs=1k count=1 skip=117249 | xxd
# find inode of erikh:
ifind -f ext -n /home/erikh ext3-image
#
##### Look in erikh dir's inode
#
# erikh dirs attributes:
istat -f ext ext3-image 27890
#
##### Look in erikh dir's datablocks
#
# hexdump of inode 27889 (erikh's) datablock:
dd if=ext3-image bs=1k count=1 skip=117250 | xxd
# find inode of cf3.msi:
ifind -f ext -n /home/erikh/cf3.msi ext3-image
#
##### Look in cf3.msi inode
```

```
#  
# cf3.msi attributes gives me its datablocks:  
istat -f ext ext3-image 27891 | less
```

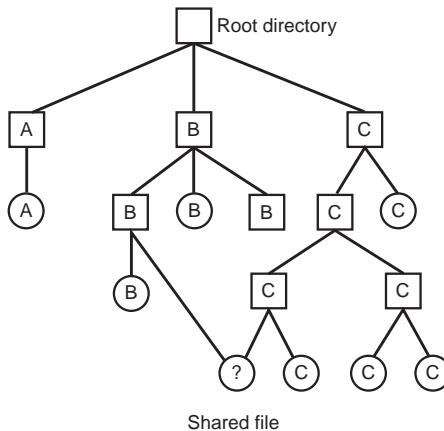
**UNIX read System Call** n=read(fd,buffer,nbytes);



DEMO fra /prod/PID/fdinfo

#### 9.2.4 Links

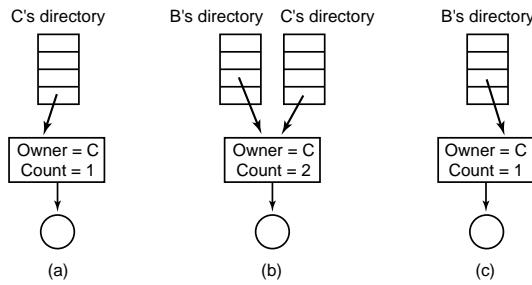
##### Shared File



Ved hjelp av en link fra en av B's kataloger til en av C's filer er trestrukturen nå blitt en rettet asyklig graf (DAG - Directed Acyclic Graph).

Både B og C kan nå dele (f.eks. editere) den samme filen.

## Shared File



Figuren viser problemet som kan oppstå ved at C ikke får slettet sin egen fil fordi C har gitt B tilgang til å hardlinke til den.

Symbolsk link (symlink) er en fil som ikke har noen datablokk men i-noden inneholder stien (path) til en annen fil.

Hard link er en directory entry som peker til en eksisterende i-node.

Generelt benyttes mest symlinker siden de er lettere å forstå, ulempen er noe ekstra overhead (jmf bruk av CNAME records i DNS istedet for multiple A records).

DEMO:

```
cat a
ls -l a # se linkteller lik 1
ln -s a sym
ls -l a # ingen endring i linkteller
ln a hard
ls -l a # linkteller er blitt 2
cat sym; cat hard # begge gir lik utskrift
stat a (debugfs) # merk inode nr
stat sym (debugfs) # merk inode nr
stat hard (debugfs) # hvilken inode nr har denne?
rm a
cat sym; cat hard # bare hard gir utskrift
stat hard (debugfs) # ingen endring, filen finnes fortsatt
```

God forklaring på tilsvarende konsepter i Windows/NTFS: <http://devtidbits.com/2009/09/07/windows-file-junctions-symbolic-links-and-hard-links/>

### 9.2.5 LFS

#### Log-Structured File Systems (LFS)

- More CPU and memory causes most reads from memory cache, thus disk access are typically small writes
- LFS is a way of *structuring the entire disk as a log* with the purpose of buffering all small writes into larger periodic segment writes appending the log
- Each segment contains a segment summary, i-nodes, directories and data-blocks, finding i-node's is a bit more difficult, but doable
- An *LFS cleaner* removes redundant files and releases segments

En diskskriving på  $50 \mu\text{s}$  er ofte foranlediget av 10ms søketid og 4ms rotasjons-delay, mao diskeffektivitet på under 1%. *Små write-operasjoner er kostbare.*

LFS forbedrer disk ytelse med en faktor 10 ved mange små writes.

LFS er ikke mye brukt i praksis, men tanken om å skape robusthet ved å være klar over hva som skal gjøres fremover i tid med filsystemet, er blitt videreført inn i eksisterende filsystemer i form av "journalling".

### 9.2.6 JFS

**Journaling File Systems** Example "removing a file":

1. Remove the file from its directory.
2. Release the i-node to the pool of free i-nodes.
3. Return all the disk blocks to the pool of free disk blocks.

*In the absence of system crashes, the order in which these steps are taken does not matter; in the presence of crashes, it does.*

f.eks. hvis bare nr 2 har rukket blitt utført, kan denne i-noden gjenbrukes og siden nr 1 ikke er blitt utført kan da en directory entry peke på feil fil

eller

f.eks. hvis bare nr 3 har rukket blitt utført, kan to filer ende opp med å dele de samme datablokkene.

**Journaling File Systems** What is the difference between:

*Add newly freed blocks from i-node K to the end of the free list*

and

*Search the list of free blocks and add newly freed blocks from i-node K to it if they are not already present*

?

**Journaling File Systems** To make journalling work, the logged operations must be *idempotent* (convergent).

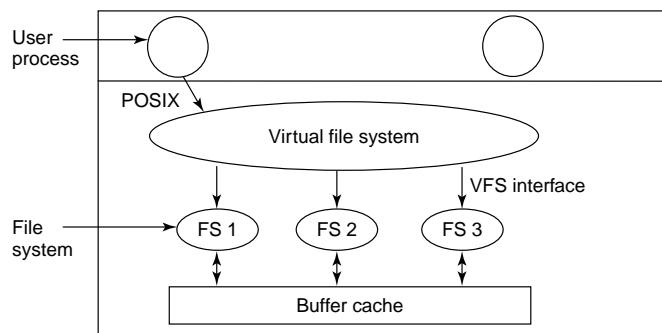
A Journalling file system keep a log of the operations it is going to do, so they can be redone in case of a system crash.

Vi ønsker altså operasjoner  $f$  som er slik at

$$f(\text{feil}) = \text{riktig} \text{ og } f(\text{riktig}) = \text{riktig}$$

### 9.2.7 VFS

#### Virtual File Systems



Et OS må håndtere mange filsystemer.

I Windows gjøres dette (inspirert fra VMS) ved at man bruker forskjellige stasjoner, hvor hver stasjon har en bokstav som navn, f.eks. en minnepinne med FAT32 filsystemet blir montert/mappet som E: mens selve operativsystemet er i NTFS på C:

Windows forsøker ikke integrere alle filsystemer inn i en helhet, det gjør UNIX/Linux.

I UNIX/Linux bruker man ikke stasjoner med bokstaver men monterer/mapper opp alle filsystemer i samme directory-struktur.

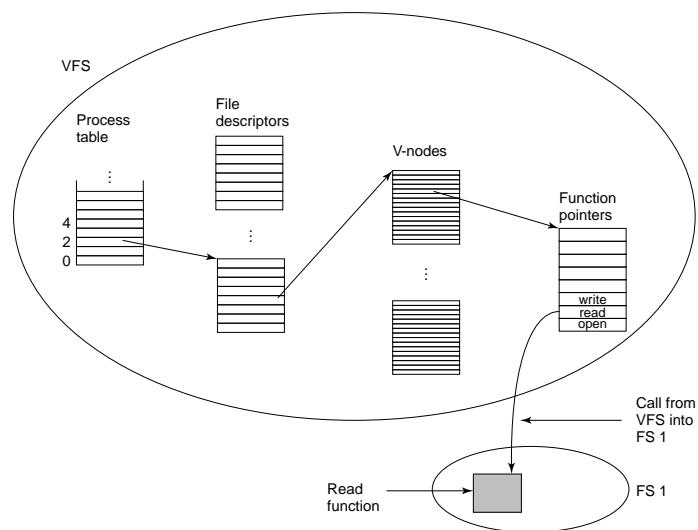
UNIX/Linux gjør dette via VFS som tar imot POSIX systemkall og oversetter de til det som filsystemet håndterer

DEMO:

dytt i minnepinne og hvis med mount (man mount og søk opp -t vfstype)

#### Virtual File Systems: read

## 9.2. FILE SYSTEM IMPLEMENTATION



Uansett hva filsystemer har av datastrukturer og metadata mappes det til v-noder.

### 9.3 Theory questions

1. Nevn minst tre eksempler på filatributter/filmetadata.
2. Hva menes med ekstern og intern fragmentering i forbindelse med en fil?
3. Tanenbaum oppgave 4.11

One way to use contiguous allocation of the disk and not to suffer from holes is to compact the disk every time a file is removed. Since all files are contiguous, copying a file requires a seek and rotational delay to read the file, followed by the transfer at full speed. Writing the file back requires the same work. Assuming a seek time of 5 msec, a rotational delay of 4 msec, a transfer rate of 8 MB/sec, and an average file size of 8 KB, how long does it take to read a file into main memory and then write it back to the disk at a new location? Using these numbers, how long would it take to compact half of a 16-GB disk?

4. Tanenbaum oppgave 4.12

In light of the answer to the previous question, does compacting the disk ever make sense?

5. Tanenbaum oppgave 4.33

How many disk operations are needed to fetch the i-node for the file /usr/ast/courses/os/handout.t? Assume that the i-node for the root directory is in memory, but nothing else along the path is in memory. Also assume that all directories fit in one disk block.

6. Et prinsipp som nyttes i forbindelse med free-space management i filsystemer er basert på bitmap prinsippet. Forklar kort hvordan dette fungerer. Anta videre et 32-bits system og en bitmap som vist under. Finn nummeret på første ledige diskblokk når følgende bitmap er gitt.

```
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0011 1111 1111 1111 1110 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000  
0000 0000 0000 0000 0000 0000 0000 0000
```

## 9.4 Lab exercises

### 1. Informasjon om deler av filsystemet.

Skriv et script `fsinfo.bash` som tar en directory som argument og skriver ut

- Hvor stor del av partisjonen directorien befinner seg på som er full
- Hvor mange filer finnes i directorien (inkl subdirectorier), gjennomsnittlig filstørrelse, og full path til den største filen
- Hvilken fil (IKKE ta med directories) har flest hardlinker til seg selv

Eksempel kjøring:

```
$ bash fsinfo.bash /opsys/  
Partisjonen /opsys/ befinner seg på er 91% full  
Det finnes 1137 filer.  
Den største er /opsys/00-X/Jan2007.doc som er 14110208 (14M) stor.  
Gjennomsnittlig filstørrelse er ca 186208 bytes.  
Filten /opsys/06-filesystems/lab/c har flest hardlinks, den har 3.  
$
```

### 2. Regulære uttrykk med grep.

Bruk grep til å finne i `/usr/share/dict/words` (eller en tilsvarende ordliste du har tilgjengelig) alle ord som:

- starter med en q
- slutter med en q
- inneholder minst to q'er
- inneholder minst to o'er og starter ikke med w eller a
- er 7 eller 8 tegn lange

### 3. Regulære uttrykk anvendt på filsystemet.

Skriv et script `fnamecheck.bash` som tar en directory som argument og skriver ut alle filnavn (inkl alle subdirectories) som inneholder norske tegn og/eller mellomrom i filnavnet.

### 4. (litt moro til slutt) Les og prøv tipsene på

<http://www.linuxjournal.com/content/tech-tip-use-gxmessage-displaying-gui-messages-scripts>



# Chapter 10

## Filsystemhåndtering og ytelse, FAT og NTFS

### 10.1 File System Management and Performance

#### How Big are Files?

Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69
Length	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

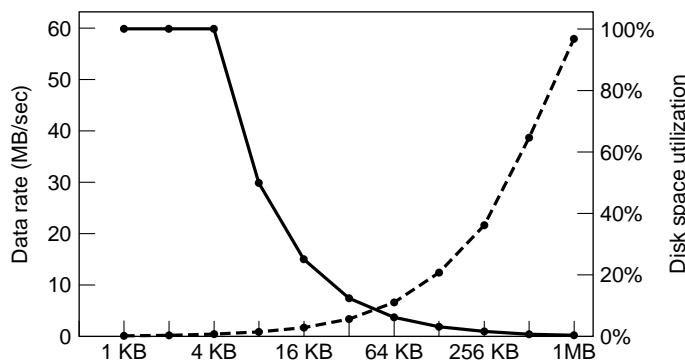
For store blokker kaster bort plass, for små blokker kaster bort tid.

4KB blokkstørrelse dekker de fleste filer, men et annet viktig poeng fra artikkelen bak disse dataene er at 93% av 4KB blokkene ble benyttet av de 10% største filene, dvs mange store filer gir lite intern fragmentering og dermed god plassutnyttelse.

UANSETT: beste blokkstørrelse vil alltid avhenge av anvendelsen/omgivelsene, siden det er stor forskjell på diskbruk i forskjellige miljøer og sammenhenger.

### 10.1.1 Blocksize

#### Rate/Space Trade-Off



NB! Bytt om stiplet og fast linje i figuren.

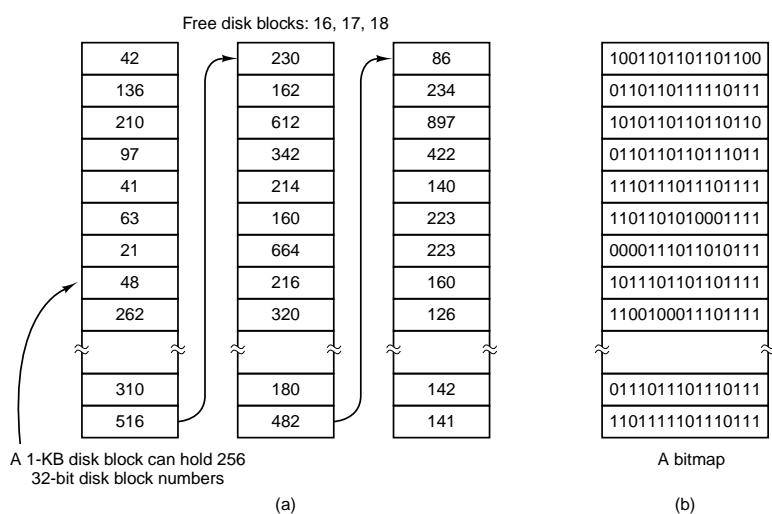
Gitt en disk med 1MB pr spor, rotasjonstid 8,33 msec (7200rpm), gjennomsnittlig seek time (flytning mellom spor) 5 msec og blokkstørrelse 4KB blir da aksessiden for en blokk:

$$5 + (8,33/2) + (4KB/1MB) \times 8,33 = 9,20$$

Mao nesten helt bestemt kun av seek time og rotasjon. Når vi først har funnet en blokk er det fint om det er mye data der.

Hvis vi vet at vi vil ha en del store filer, kanskje det lønner seg å øke blokkstørrelsen til 64KB siden det er billig med mye diskplass for tiden.

#### Free Blocks



Med 1KB blokkstørrelse og 32 bits (4 bytes) blokkadresser, er det plass til 256 blokkadresser i hver blokk (1KB/4bytes). En av de 256 brukes som peker til neste. En 16GB disk/partisjon trenger en freelist på max 65793:

Ant. diskblokker:  $2^4 \times 2^{30} / 2^{10} = 2^{24}$

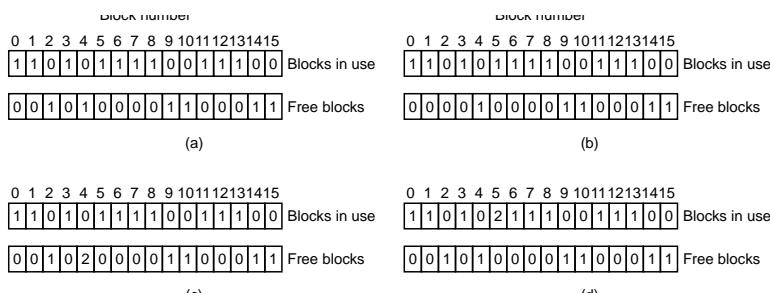
Ant. til freelist:  $2^{24} / 255 = 65793$  (mao freelist kan oppta ca 64MB)

Bitmap løsningen krever til  $2^{24}$  diskblokker  $2^{24}$  bit som er 2MB ( $2^{21}$ bytes), fordi bitmap krever kun en bit pr blokk mens freelist må ha en 32 bits adresse pr blokk.

Freelist kan kreve mindre plass ved å lagre sammenhengende sekvenser av blokker istedet for hver enkelt ledigblokkadresse, men det er ikke nødvendigvis plassbesparende (tenk over hvordan det vil se ut for en veldig fragmentert disk).

### 10.1.2 Consistency

#### File System Consistency



Ikke noe stort problem lenger pga journalling, men kan fortsatt dukke opp, konsistens sjekkes med `fsck` på Linux og `scandisk` på windows.

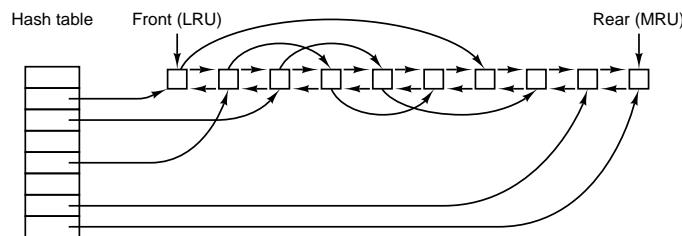
Kan sjekke konsistenthet på

1. blokker: tabell over freelist-forekomster og fil-tilhørighet for alle blokker
  - (a) er ok.
  - (b) 2 er missing block, kaster bort plass, så den legges til freelist.
  - (c) blokk 4 er registrert to ganger i free list, heller ikke noe problem, bare fjern den ene entrien i freelist.
  - (d) blokk 5 tilhører to filer (!), ikke bra, lag en kopi og gi den til den ene filen, uansett er minst en av filene tukla til
2. filer: sammenlikn antall directory entries for hver fil med linktelleren i filens i-node
  - (a) hvis flest antall i linktelleren, litt plass-sløsing, linkteller oppdateres

- (b) hvis flest directory entries, ILLE (fordi man sletter en fil i det dens linkteller blir 0, så dermed kan en fil slettes selv om en direntry fortsatt peker til den), linkteller oppdateres

### 10.1.3 Caching

#### Caching



Figuren viser hvordan buffer cache'n (jmf figur om VFS på slutten av forrige kapittel) er organisert og benytter LRU-prinsippet (jmf Page Replacement algoritmene).

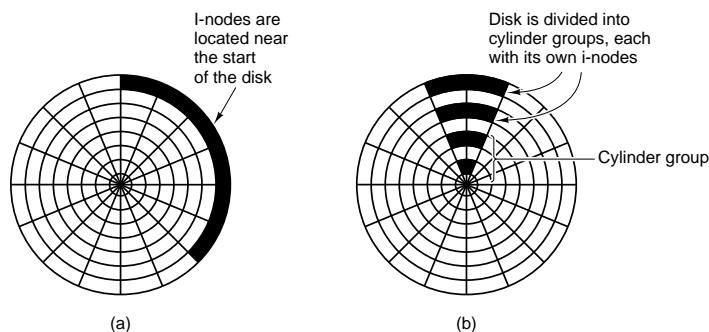
Buffer cache er altså en generell cache som legger seg rett over disken og bare bryr seg om blokker som benyttes ofte (på samme måte som page replacement algoritmene i kapittel 3). Buffer cachen bruker gjerne LRU (Least Recently Used) men tar i tillegg høyde for sannsynligheten for at blokken trengs igjen snart (f.eks. en nesten full datablokk trengs nok igjen snart), og i hvilken grad en blokk er kritisk for filsystemets konsistenshet (f.eks. en modifisert i-node).

Windows benyttet opprinnelig *write-through caches* (dirty blocks skrives alltid umiddelbart tilbake til disk) som er det som gjorde at disketter alltid var mye lettere å håndtere på windows enn på UNIX/Linux (som måtte kjøre sync først)

Block Read Ahead (OSet ser på access patterns for å finne filer som er sekvensiell i aksess og dermed kan OSet gjøre block read ahead)

### 10.1.4 Cyl. Groups

#### Cylinder Groups/Block Groups



Cylinder groups er ideen bak block groups som vi har studert på ext filsystemet på linux.

DEMO:

kopier og slett store filer på minnepinne og se på egenskapene i windows (med defragmenteringsverktøyet)

## 10.2 FAT12/16/32

### FAT12/16/32 Overview

- Probably the most widely used
- A very simple file system (simple is good! KISS!)
- No i-nodes with the files or i-node pointer in directory entry, all information in the directory entry and the FAT
- Blocks are called clusters

TAVLE:

VolumeID	Reserved sectors	FAT1	FAT2	Files and directories...
----------	------------------	------	------	--------------------------

VolumeID gir blant annet info om Bytes pr sector, Sectors pr cluster, Number of FATs (alltid 2), Sectors pr FAT, Første cluster til Root directory (som regel rett etter FAT2)

**Max partisjonsstørrelse** FAT32 bruker en 32-bits sector-count (i VolumeID, dvs i boot sektor), og siden industri-standard sektorstørrelse er 512 byte, gjør dette at max partisjonsstørrelse blir  $2^9 \times 2^{32} = 2^{41} = 2\text{TB}$ .

**Max filstørrelse** Hver direentry bruker et 32-bits felt for å lagre filstørrelse, dermed er max filstørrelse 4GB.

(merk: dette er de teoretiske størrelsene, det kan hende verktøyet man bruker for å lage en partisjon har bestemt seg for at det skal sette ytterligere begrensninger).

DEMO:

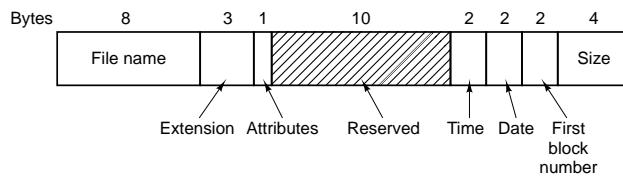
```
sudo dd if=/dev/sdb1 bs=512 count=1 skip=0 | xxd og finn ihht figur 4 http://www.pjrc.com/tech/8051/ide/fat32.html
```

bruk <http://www.statman.info/conversions/hexadecimal.html> for å vise at 200 blir 512 og at 10 blir 16, derav blokkstørrelse 8KB

vis <http://technet.microsoft.com/en-us/library/cc776720.aspx> for å bekrefte

### 10.2.1 Dir. entry

#### The Directory Entry



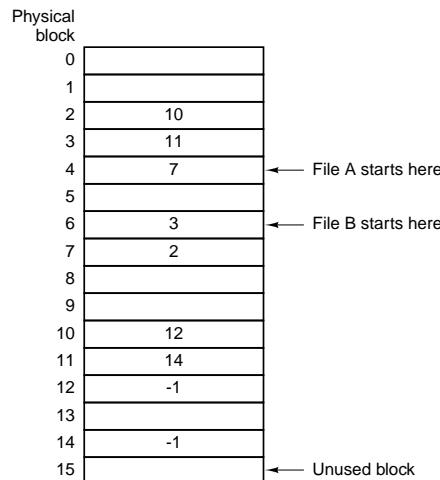
Directory entry inneholder altså noe av det samme som i-noder gjør.

Lange filnavn skjer ved at det gis korte filnavn hvor byte 7 og 8 er 1 el. høyere inntil unikt og flere directory entries benyttes til å holde lange filnavn (siden rootdir er begrenset til 512 entries (i FAT16) bør ikke lange filnavn forekomme i rootdir).

MERK: Ingen representasjon av EIER av fila, mao ingen sikkerhet!

### 10.2.2 The FAT

#### The FAT



Vi husker denne fra tidligere. Filer knyttes altså mot datablokker (clusters) gjennom directory entrien og deretter FAT-tabellen istedet for alle addressene i i-noden (og tilhørende single/double/triple indirect adresseblokker).

Nice description of FAT filesystem is

<http://www.win.tue.nl/~aeb/linux/fs/fat/fat.html>

```
# vfat-image is dd of vfat memory stick with all zeros
# except dir structure /home/erikh/{a.txt,cf3.msi}
#
##### Find root dir
#
# in the VolumeID bytes 11 and 12 gives sector size (little endian)
dd if=vfat-image bs=1 count=2 skip=11 | xxd
# hex 200 is decimal 512
# and byte 13 is number of sectors per cluster (cluster=block)
dd if=vfat-image bs=1 count=1 skip=13 | xxd
# fsstat formats all this VolumeID (boot sector) info for us:
fsstat -f fat vfat-image
# 248 blocks in FAT because: 248x512B/16b~64K (x2KB=128MB)
# note: in FAT16 root dir is in a known location so no info
# in VolumeID about this (this is different in FAT32)
#
##### Look in root dir
#
# note: root dir is not part of FAT
# root dir says home starts in FAT entry 4 (bytes 26-27 says 0300)
dd if=vfat-image bs=512 count=1 skip=497
#
##### Look in FAT for additional entries (datablocks)
#
# FAT entry 4 says no more FAT entries (ffff)
dd if=vfat-image bs=512 count=1 skip=1 | xxd
#
##### Look in datablock for directory content
#
# FAT entry 4 is sector 529+4 which says erikh is FAT entry 5
# (remember that first two FAT entries are not in use)
dd if=vfat-image bs=512 count=1 skip=533 | xxd
#
##### Look in FAT for additional entries (datablocks)
#
# FAT entry 5 says no more FAT entries (ffff)
dd if=vfat-image bs=512 count=1 skip=1 | xxd
#
##### Look in datablock for directory content
#
# FAT entry 5 is sector 529+8 which says cf3.msi is FAT entry 6
# and a.txt is FAT entry 3254 (0cb6)
dd if=vfat-image bs=512 count=1 skip=537 | xxd
```

```

#
##### Look in FAT for additional entries (datablocks)
#
# FAT entry 6 says next is entry 7, 8, 9, etc
dd if=vfat-image bs=512 count=1 skip=1 | xxd
# (we do not look in datablocks since cf3.msi a binary file)
# FAT entry 3254 says no more FAT entries (ffff)
# (3254 DIV (512B/16b=256) = 12
dd if=vfat-image bs=512 count=1 skip=13 | xxd
#
##### Look in datablock for file content
#
# FAT entry 3254 is sector 529+((3254-2)*4)=13537
dd if=vfat-image bs=512 count=1 skip=13537 | xxd

```

### Partition Sizes

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

## 10.3 NTFS

### NTFS

- Advanced file system with many features (compression and encryption)
- Solves the problem with lack of security in MS-DOS/FAT

TAVLE:

```

+-----+
| Boot sector | MFT | Systemfiler | Data...
+-----+

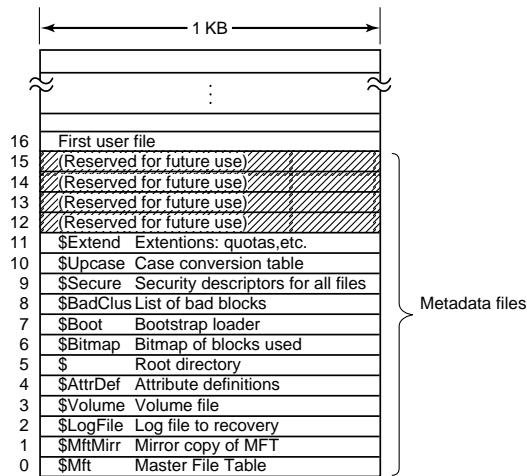
```

**Max partisjonsstørrelse**  $2^{64}$ B= 16EB

**Max filstørrelse**  $2^{64}$ B= 16EB

### 10.3.1 MFT

#### Master File Table (MFT)



MFT er en sekvens med 1KB records.

En MFT record representerer en fil eller directory.

Hver MFT record er en sekvens med (attribute header, value) par hvor attribute header sier hva slags attribute det er og hvor stor den er.

Den største attribute er som regel datastrømmen (som det kan være flere av, men det er lite brukt).

Hvis datastrømmen får plass i MFT record'n vil det si at hele fila er i MFT recorden, det kalles en *resident* eller *immediate* attribute/fil.

Demo: kjør PowerShell som Administrator: `ntfsinfo C:`

### 10.3.2 Records

#### Attributes in MFT records

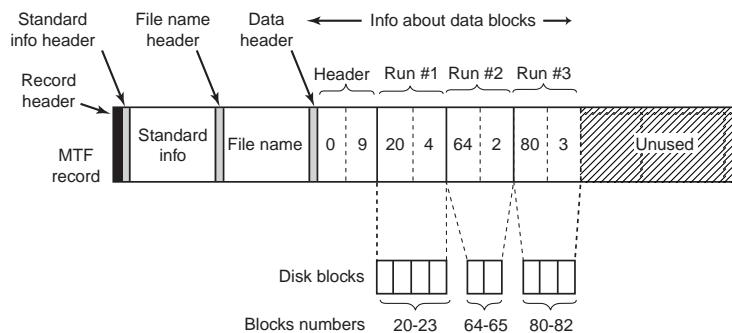
Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in \$Extend\$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier 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

NTFS definerer 13 attributes.

Standard info er alt som trengs av POSIX bl.a.

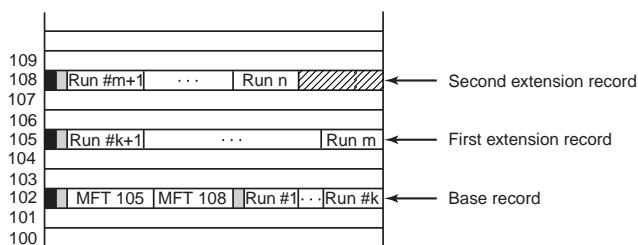
Attribute list sier hvilke andre MFT records denne filen benytter.

### A MFT Record for a Small File



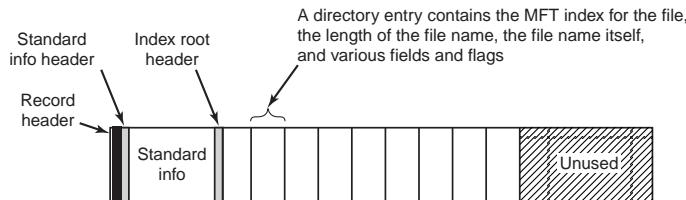
Diskblokker allokeres sammenhengende hvis mulig, og beskrives med (startblokk, lengde) par etter hverandre i MFT data attributtet.

### MFT Records for a Big File



En stor fil benytter mange MFT records for å liste opp alle diskblokker den benytter.

### MFT Record for a Small Directory



Små directories er i all hovedsak bare et sett med (filnavn,MFTindex,++) i en MFT record. Store directories derimot benytter datablokker og organiseres som et B+ tree for at søket skal være mer effektivt.

```
# ntfs-image is dd of ntfs memory stick with all zeros
# except dir structure /home/erikh/{a.txt,cf3.msi}
#
##### Find root dir
#
# the boot sector gives the first cluster (block) of the MFT file,
# and root dir is always in MFT entry nr 5:
fsstat -f ntfs ntfs-image
#
##### Look in root dir's MFT entry
#
# root dirs attributes, we want datablocks of $INDEX_ALLOCATION
istat -f ntfs ntfs-image 5
#
##### Look in root dir's datablocks
#
# root dirs datablocks gives me MFT nr of subdir home
dd if=ntfs-image bs=4k count=1 skip=3981 | xxd
# search for MFT entry of home to confirm:
ifind -f ntfs -n home ntfs-image
#
##### Look in home dir's MFT entry
#
# home dirs attributes, we want datablocks of $INDEX_ALLOCATION
istat -f ntfs ntfs-image 64
#
##### Look in home dir's MFT entry again: RESIDENT FILE (no datablocks)
#
# no, wait, there's only the resident $INDEX_ROOT attribute
icat -f ntfs ntfs-image 64-144-2 | xxd
```

```

# we can also verify this by looking in the datablocks of the MFT file
istat -f ntfs ntfs-image 0
# and since blocksize is 4KB and each entry is 1KB, entry 64 is approx
dd if=ntfs-image bs=4k count=1 skip=20 | xxd
# which gives me MFT nr of subdir erikh
ifind -f ntfs -n home/erikh ntfs-image
#
##### Look in erikh dir's MFT entry
#
# erikh dirs attributes, we want datablocks of $INDEX_ALLOCATION:
istat -f ntfs ntfs-image 65
#
##### Look in erikh dir's MFT entry again: RESIDENT FILE (no datablocks)
#
# there's only the resident $INDEX_ROOT attribute here as well:
icat -f ntfs ntfs-image 65-144-2 | xxd
# which gives me the MFT nr of files a.txt and cf3.msi:
ifind -f ntfs -n home/erikh/a.txt ntfs-image
ifind -f ntfs -n home/erikh/cf3.msi ntfs-image
#
##### Look in a.txt MFT entry (RESIDENT FILE)
#
# a.txt is a very small file which fits in the MFT
# (resident $DATA attribute, called "an immediate file")
istat -f ntfs ntfs-image 66
icat -f ntfs ntfs-image 66-128-2
#
##### Look in cf3.msi MFT entry
#
# cf3.msi attributes gives me its datablocks (stored as 'data runs'):
istat -f ntfs ntfs-image 35

```

## 10.4 Comparison

Comparison [http://en.wikipedia.org/wiki/Comparison\\_of\\_file\\_systems](http://en.wikipedia.org/wiki/Comparison_of_file_systems)

	<b>Ext3</b>	<b>FAT</b>	<b>NTFS</b>
Max file size	2TB	4GB	16EB
Max vol size	16TB	2TB	16EB
Attr location	I-node	Dir entry	MFT entry
Allocation	Table	"Linked-list"	Table
Owner/perms	Yes	No	Yes
Speed	B dir tree	Table dir	B+ dir tree

## **10.5 Theory questions**

1. Hvor store diskblokkadresser har vi i windows filsystemet NTFS?
2. Hva forståes med MFT i NTFS filsystemet?
3. Tanenbaum oppgave 4.3  
Is the open system call in UNIX absolutely essential? What would the consequences be of not having it?

## **10.6 Lab exercises**

1. Bare fortsett på forrige ukes oppgaver...

# Chapter 11

## I/O

### 11.1 I/O Hardware

#### I/O Hardware Devices

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner	400 KB/sec
Digital camcorder	3.5 MB/sec
802.11g Wireless	6.75 MB/sec
52x CD-ROM	7.8 MB/sec
Fast Ethernet	12.5 MB/sec
Compact flash card	40 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
SATA disk drive	300 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec

Det er viktig å forsøke gruppere I/O enheter så godt man kan slik at man ha færrest mulig grensesnitt mot dem, og standardisere så mye man kan. To hovedkategorier:

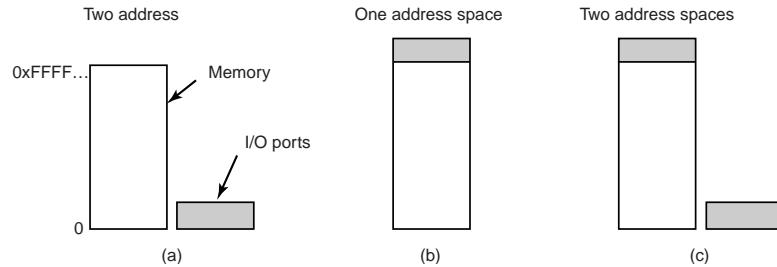
**Blokkbaserte enheter** disk, dvd, minnepinne, alt som kan addresseres som blokker av data, f.eks. 512 bytes eller 32kB

**Characterbaserte enheter** mus, skriver, lydkort, nettverkskort, bare en strøm av enkelt-characters, ikke addresserbare

Det finnes enheter som ikke passer inn, f.eks. klokker som bare genererer interrupt med jevne mellomrom.

### 11.1.1 Isolated vs Memory-mapped

#### Isolated or Memory-Mapped I/O



I/O enheter har som regel kontroll-, status- og dataregistre (databuffere).

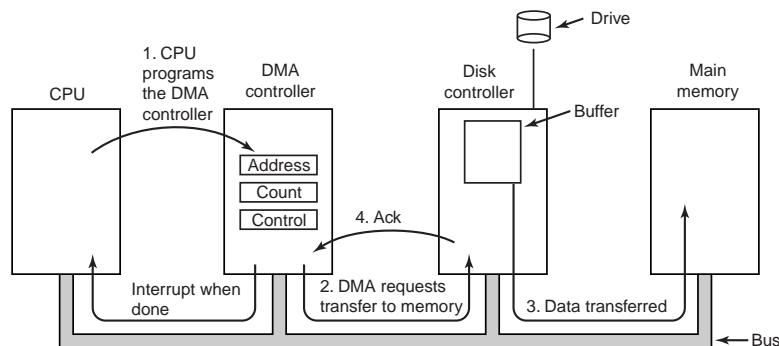
Opprinnelig hadde vi bare Isolated I/O som i figur (a), dvs man kommuniserte med I/O enheter via I/O porter (typisk et 16-bits tall slik som TCP porter).

Etterhvert begynte man benytte også memory-mapped I/O (b) som vil si at man istedet bruker en del av minne til å prate mot I/O registerne. Da slipper man egne instruksjoner for I/O-portene, man bare bruker de samme instruksjonene som man ellers bruker mot minne. Ulempen er at det spiser litt av minneområdet, samt evn problemer med caching (dvs skal en prosess lese av et statusregister på en I/O enhet blir det bare full hvis denne infoen er cachet).

Intel benytter som oftest en hybrid modell hvor databufferene er memory-mapped mens de andre registerne er har isolated I/O (dvs I/O porter).

### 11.1.2 DMA

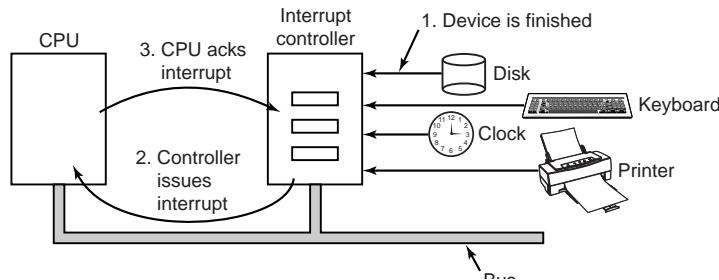
#### Direct Memory Access



Vi husker hvordan DMA funker fra tidligere.

### 11.1.3 Precise vs Imprecise Interrupts

#### Interrupts

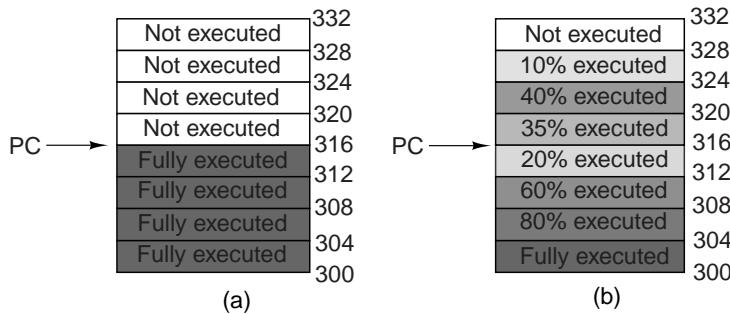


Og vi husker hvordan interrupt funker fra tidligere.

Men det fakta at vi har pipelining, superskalar prosessor og mikrooperasjoner (dvs instruksjoner dekomponert i mikrooperasjoner) kompliserer bildet vårt.

Bla tilbake til figur 1.7 side 19.

#### Precise vs Imprecise interrupts



(a) viser et presist interrupt som er definert ut fra fire kriterier:

1. PC lagres på et kjent sted
2. Alle instruksjoner før PC er ferdig
3. Ingen instruksjoner etter PC er kjørt
4. Kjent tilstand for PC-instruksjonen

(b) viser et upresist interrupt, dvs poenget her er at det er lite sammenheng mellom PC og hvilken tilstand instruksjonene rundt PC befinner seg i.

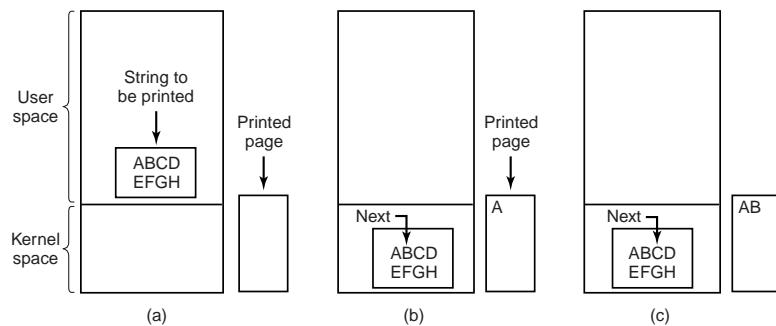
Upresise interrupt gjør livet vanskelig for operativsystemet (en masse informasjon lagres på stacken og OSet må finne ut av den), mens presise interrupt krever en komplisert interruptlogikk og CPUn må logge masse info løpende for å kunne generere presise interrupt, dette skaper overhead som reduserer ytelsen.

Arktikturen tilbyr gjerne begge deler, dvs f.eks. muligheten til å sette et kontrollbit som sørger for at presise interrupts genereres.

Uansett må vi huske at interrupts er kostbare, de spiser mye CPU-tid.

## 11.2 I/O Software

### Example: Printing a String



I/O kan gjøres på tre forskjellige metoder (TAVLE):

1. Programmert I/O
2. Interrupt-basert I/O
3. DMA

(vi husker dette fra første intro-temaet)

### 11.2.1 Three ways

Merk: dette er bare en utdyping av de tre metodene for I/O som vi introduserte i kapittelet om datamaskinarkitektur.

## Programmed I/O

```

copy_from_user(buffer, p, count);           /* p is the kernel buffer */
for (i = 0; i < count; i++) {               /* loop on every character */
    while (*printer_status_reg != READY);   /* loop until ready */
    *printer_data_register = p[i];          /* output one character */
}
return_to_user();

```

Bare les kodekommentarene til høyre.

Problemet er "loop until ready", dette er busy waiting og er dumt hvis CPUen har noe annet å gjøre (men ikke nødvendigvis dumt hvis den ikke har noe annet å gjøre som ved et embedded system, heller ikke dumt hvis skriveren har et stort buffer).

## Interrupt-Driven I/O

<pre>copy_from_user(buffer, p, count); enable_interrupts(); while (*printer_status_reg != READY); *printer_data_register = p[0]; scheduler(); </pre>	<pre>if (count == 0) {     unblock_user(); } else {     *printer_data_register = p[i];     count = count - 1;     i = i + 1; } acknowledge_interrupt(); return_from_interrupt();</pre>
--	--

(a) er koden som starter utskriften og sørger for utskrift av første tegn, men (b) er interrupt service procedure som skriver ut resten av tegnene (et tegn for hvert interrupt). Poenget er at scheduleren kan gjøre mellom hvert tegn slik at andre prosesser kan kjøre da. Når det ikke er flere tegn å skrive ut så vil brukerprosessen ikke lenger blokkeres.

## I/O Using DMA

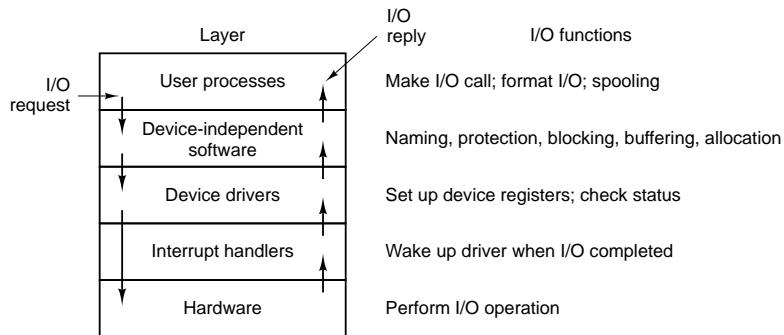
```
copy_from_user(buffer, p, count);  
set_up_DMA_controller();  
scheduler();  
  
acknowledge_interrupt();  
unblock_user();  
return_from_interrupt();
```

Med DMA så settes også jobben bort på samme måte som ved interrupt-drevet I/O men forskjellen er at nå belastes ikke CPUn for hvert tegn, DMA kontrolleren gjør alt selv.

Dvs, ved DMA så blir det bare et interrupt for utskrift av hele bufferet imotsetning til interrupt-drevet I/O som genererer et interrupt for hvert tegn som blir skrevet ut.

## 11.3 I/O Software Layers

### Layers



**Hardware** selve enheten (består typisk av kontroller og media) som utfører I/O operasjonen.

**Interrupt handler** sørger for å vække device driver som er blokkert mens I/O operasjonen pågår.

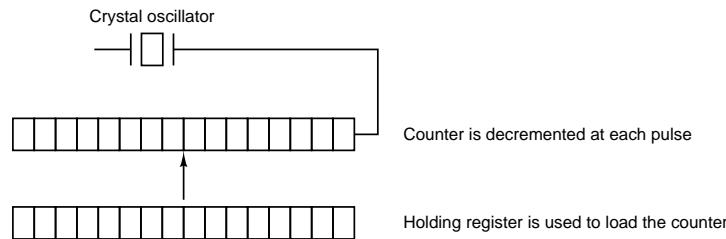
**Device driver** skriver til I/O-enhetens registrere for å styre den, samt leser av status fra disse registerne, sjekker input parameterne som den får. De fleste drivere er spesielle for den type hardware de støtter, dvs de er skrevet av leverandøren, mens andre er generelle, f.eks en driver for alle SCSI disker.

**Device-independent software** alt som skal gjøres for I/O enheter som ikke er spesifikt for bestemte enheter kan samles i uavhengig software i OSet. F.eks. navngivning: hva skal enheten hete? /dev/sda eller /dev/hda, skal lydkortet knyttes til /dev/sound osv. Aksess kontroll (dvs beskyttelse) av enhetene er også uavhengig av type enhet og som regel er oppgaver som buffring av data, feilhåndtering, alloksering og release (spesielt for alle enheter som bare kan brukes av en prosess om gangen, som f.eks. en CD-brenner), samt sørge for blokkstørrelse-uavhengighet.

**User process** den som gjør I/O forespørsel (som regel via et systemkall), formaterer I/O, spooling (f.eks. håndterer en kø av printjobber).

## 11.4 Clocks/Timers

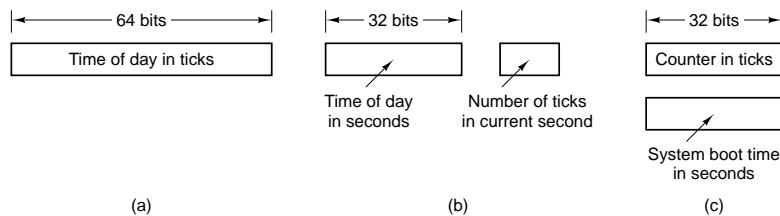
### Clock Interrupts



F.eks. vil en 500MHz krystall oscillator gi en pulse hver andre nanosekund, her kan holding registeret bestemme hvor ofte en pulse skal generere et klokkeinterrupt, og med et 32-bit holding register kan dette da styres fra 2ns til 8.6sek ( $2^{32} \times 2ns$ ) mellom hvert interrupt.

Dette skjer altså ved at verdien i holding register kopieres til countern og denne telles ned for hver pulse og genererer interrupt når den blir 0, deretter lastes verdien fra holding register på nytt og nedtellingen starter igjen.

## Real Time



Dagsriktig tid måles fra 1. jan 1970 (Unix/Linux) eller 1. jan 1980 (Windows).

Dette kan gjøres ved enten (a) telle antall ticks med en 64 bits teller (blir mange bytes i sekundet!) eller (b) antall sekunder med en 32 bits teller (med en 32 bits egen teller som teller opp antall ticks inntil det er gått 1 sekund), eller (c) telle ift system boot time.

## 11.5 Disks

### 11.5.1 HDD

#### Disk Parameters

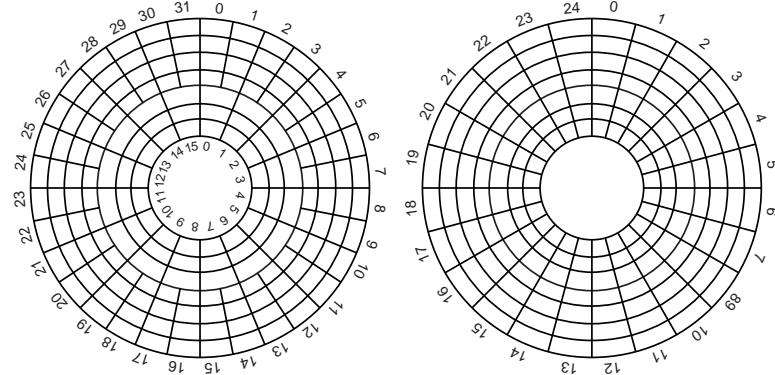
Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 $\mu$ sec

Det som er viktig å være klar over er at med flashminne eller SSD (solid state drive) lagringsmedia så er det ingen mekanisk lesearm, så søketiden blir praktisk talt borte. MEN magnetiske disker med lesearm vil være med oss i minst 10 år til og kanskje enda lenger. Så det er fortsatt viktig å kjenne til teknologien for magnetiske disker. På servere i dag benyttes stort sett SAS disker, se

[http://en.wikipedia.org/wiki/List\\_of\\_device\\_bandwidths#Storage](http://en.wikipedia.org/wiki/List_of_device_bandwidths#Storage)

Ift figuren merk at seek time er 7 ganger bedre, transfer rate er 1300 ganger bedre, mens kapasiteten er 50000 ganger bedre, mao det blir ganske tydelig at flaskehalsen er seek time.

## Disk Terminology



**Head (lese-/skrive-hode)** en disk har et antall plater/platesider som kan leses av et lese/skrivehode, eller lese/skrivearm om du vil, en disk har typisk mellom 1 og 16.

**Sylinder and Track (spor)** hver plate er delt inn i spor (en runde på plata) typisk noen ti-tusener. Alle sporene vertikalt rett ovenfor hverandre danner en sylinder.

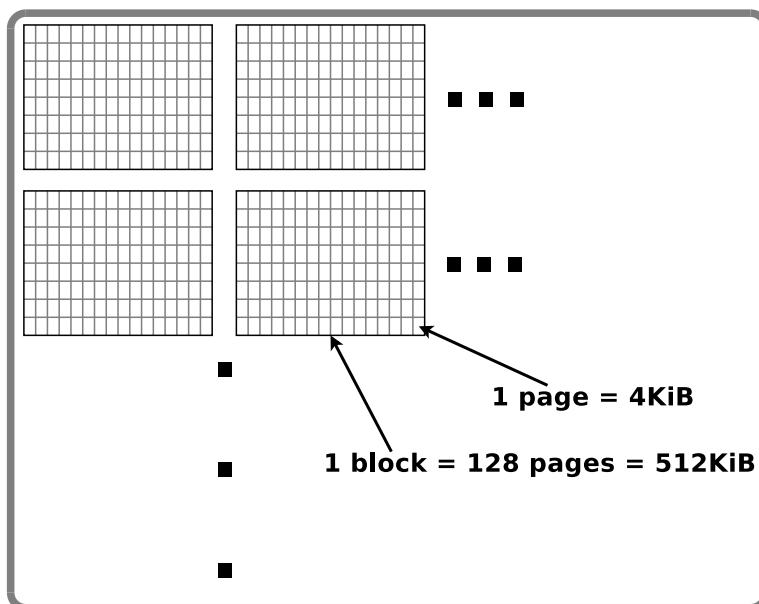
**Sektor** hvert spor er delt inn i sektorer, typisk mellom ni og noen hundre.

**Sone** siden en disk har plass til flere sektorer jo lengre ut mot kanten av platen man er så deles disken inn i soner som vist i figur a. Kontrolleren tar seg av å skjule denne fysiske geometrien på disken og presenterer heller den virtuelle geometrien vist i figur (b). Merk at det er like mange sektorer tilsammen i den virtuelle som i den fysiske.

Opprinnelig var max verdiene på IBM PC for cylinder, head og sektor 65535, 16, og 63, mens nå har man droppet å addresse slik og addresserer heller løpende lineært fra null og opptil så mange sektorer som fins. Dette kalles LBA (*Logical Block Addressing*).

### 11.5.2 SSD

#### Solid-State Drive



Figuren: "Det viktigste" å vite om SSD disker (i tillegg til å vite at de ikke har noen mekanisk bevegelige deler) er at de kan bare lese og skrive page'r (dvs man kan ikke skrive mindre enn en page), og kan bare slette blokker (dette pga de fysiske egenkapene til dette lagringsmediet, noe med at det trengs sterkere spenning for å blanke ut celler og da må dette gjøres på en større gruppe celler om gangen), og *SSD disker kan ikke overskrive page'r direkte, de må slette innholdet i en page (og dermed en hel blokk) før de kan skrive en til page.*

SSD disker er som regel laget av NAND flash IC'r som i lese/skrive hastighet er midt mellom RAM og magnetisk disk (dvs i motsetning til RAM så mister ikke NAND flash

data når strømmen blir borte, samtidig er NAND flash mye raskere enn magnetisk disk).

gå gjennom <http://www.anandtech.com/printarticle.aspx?i=3531> i pdf form ihht sidene:

**tabeller side 7** MERK: hovedpoenget med SSD er at random read/write kan bli minst en størrelsesorden bedre enn HDD grunnet at det ikke er en mekaniske lesearm som skal flyttes lenger

**øverste figur side 11** forskjell på SLC og MLC er altså bare på celle nivå (MLC celler kan overskrives minst 10.000 ganger mens SLC celler kan overskrives minst 100.000 ganger). Det finnes også celler som kan overskrives enda færre ganger i og med at de lagrer flere enn to bit pr celle.

**avsnitt side 13** "the real performance..." (parallel aksess skaper hastighet: det er dette som gjør at SSD'r kan bli så mye raskere enn vanlig minnepinner) og "Now you can read..." (du kan lese og skrive page'r MEN for å overskrive en page MÅ DU SLETT HELE BLOCK SOM DEN PAGE TILHØRER)

**tabell side 14** merk hvordan overskriving skjer på SSD, og at file delete normalt ikke kommuniserer til disken, husk det er bare noe som gjøres i filsystemet, f.eks. å fjerne en dir-entry

**figurer side 15-18** gå gjennom eksempelet som viser nettopp hvordan dette skjer i praksis og hvorfor en overskriving tar i dette tilfelle 26sec istedet for 12sec (merk: krysset over page'n som ikke er i bruk lenger, dvs "invalid" merke er ikke noe som finnes i virkeligheten på SSD'n, den vet ikke hva som kan slette før OSet gir den beskjed om at en page skal overskrives) (LES NEDERST SIDE 18 HELE og TOPP SIDE 19)

**figur og avsnitt side 23** "The TRIM command forces the block to be cleaned before our final write. There's additional overhead but it happens **after a delete and not during a critical write**"

**øverste figur side 48** her ser vi hvordan SSD fullstendig dominerer over HDD ved random read ytelse

**figur og tabell side 49** her ser vi hvor stor forskjell det kan være på random write, dvs SSD som er optimalisert for sekvensiell read/write, takler dårlig random write (NOE SOM ER ILLE FOR OSS VANLIGE BRUKERE FORDI VANLIG PC BRUK ER MEST RANDOM READ/WRITE)

**figure side 58** her ser vi hvor stor reell merkbar ytelsesforskjell vi typisk kan oppleve ved overgang til en SSD, dette er en boot med påfølgende lasting av et sett tunge applikasjoner

## Disk Actions: HDD vs SSD

Action in the OS	Reaction on a HDD	Reaction on an SSD
File Create	Write to a Sector	Write to a Page
File Overwrite	Write new data to the same Sector	Write to a Different Page if possible, else Erase Block and Write to the Same Page
File Delete	Nothing	Nothing

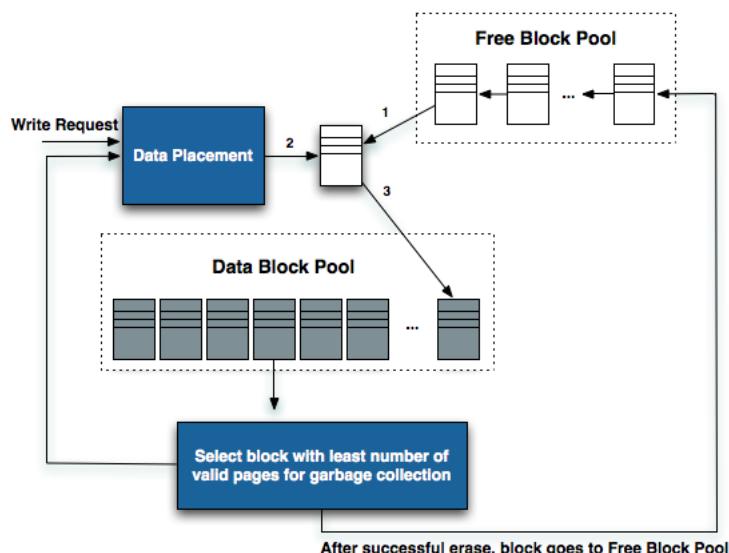
The TRIM command lets the SSD do erasures after file deletions instead of before overwrites

## Important SSD Concept

**Write amplification** read-erase-modify-write cycle causes extra writing (also due to wear levelling)

**Wear levelling** distribute erasures and re-writes evenly across the media

## Write Amp. and Wear Levelling



<http://anandtech.com/storage/showdoc.aspx?i=3631>

Vi prater ofte om en "write amplification factor" som bør være så nære 1 som mulig, dvs hvis den er 1.1 betyr det at skrivning av 1MB betyr i gjennomsnitt at 1.1MB blir skrevet til disk i praksis. Merk altså at en blokk med minst data i seg velges for å skrives sammen med den innkommende write-request slik at dataene fra denne blokka kombineres med de nye dataene i en ny blokk og den gamle blokka slettes og legges til gruppen med ledige blokker.

Merk: wear levelling kan også gjøres i software (ref vår lagdelte IO modell, dette kan gjøres enten i driveren eller på kontrollern) se siste paragraf om “special-purpose file systems” på

[http://en.wikipedia.org/wiki/Wear\\_leveling](http://en.wikipedia.org/wiki/Wear_leveling)

Merk: Filsystemer som skriver veldig ofte til et bestemt sted på diskene (f.eks. journaling eller FAT-tabellen) høres ut som et mareritt for SSD diskene siden de 10.000 gangene dette stedet på diskene kan skrives til burde bli brukt opp fort, men dette er altså ikke noe problem grunnet wear levelling.

*Husk at filsystemet forholder seg til et LBA addresserom, dvs filsystemet bare addresserer blokker lineært fra 0,1,osv mens i praksis vil altså f.eks. blokk 1 bli flyttet rundt på SSD'n av SSD kontrollern, men dette er usynlig for filsystemet. SSD kontrollern er snitt mot filsystemet og presenterer det med samme addreser hver gang. Nok en gang: ABSTRASJON OG LAGDELING LØSER ALLE PROBLEMER :)*

## SSD: OS Consequences

- Classical defragmentation destroys more than it helps
- FS-Page/Block/Sector alignment with SSD-Pages
- Write's are bad (maybe disable journaling and writing of access time?)
- *Support the TRIM command*  
[http://en.wikipedia.org/wiki/TRIM\\_\(SSD\\_command\)](http://en.wikipedia.org/wiki/TRIM_(SSD_command))

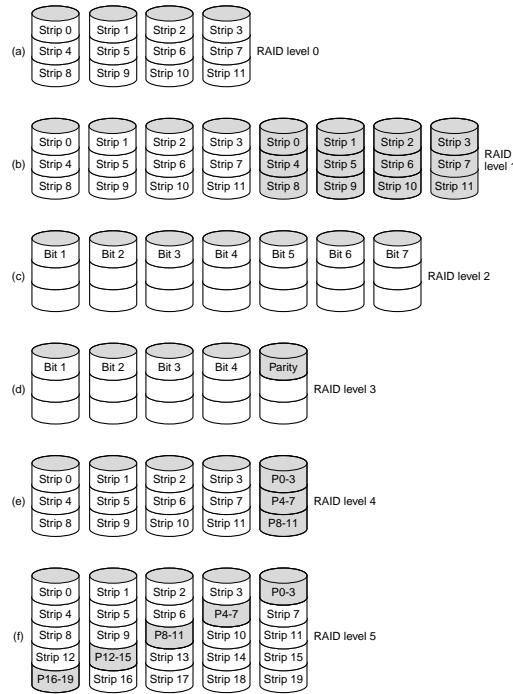
Defragmentering betyr jo å flytte data rundt om på disk for at det skal være mest mulig sammenhengende lagring, dette har ingen effekt på SSD, men derimot fører bare til mye ekstra skriving, dvs slitasje (nedkorting av en SSD's levetid).

Filsystemet bør fungere med et addresseringssystem som ikke gjør at skriving til en 4KiB filsystemblokk betyr skriving til to 4KiB SSD-page'r. Dvs filsystemets addressingsenheter bør være alignet med SSD'n sine page'r (i den grad dette er noe stort problem i praksis?).

Generelt: ikke skrive så ofte til disk: dvs skru av journaling (skriver bare til disk hvert 5. sekund) og noatime? eller lite nyttige kjerringråd? <http://tytso.livejournal.com/tag/ssd>

TRIM er støtte av Linux (siden 2.6.28) og Windows 7 (Android siden 4.3), og de fleste diskene siden 2011: <http://www.tomshardware.com/forum/252653-32-supporting-trim>

### 11.5.3 RAID



Figuren viser RAID nivå 0 til 5, merk paritetsinfo som er lagret der det er skyggede felter. RAID 6 er samme som RAID 5 men med dobbelt paritetsslagring.

Det brukes også kombinasjoner, f.eks. RAID 10 som er RAID 1 på toppen av RAID 0.

Se

<http://www.acnc.com/raid>

## **11.6 Theory questions**

1. Hva er forskjellen på memory mapped og isolated I/O? Angi fordeler og ulemper med disse to prinsippene.
2. På en harddisk, hvor mange bytes finnes som regel i en sektor? Hva er en sylinder? Hva er typisk gjennomsnittlig aksesstid for en disk i dag? Hva er overføringsraten (ca, i MB/s) mellom diskplate og buffer?
3. Hva oppnår vi med å koble diskene som RAID disker? Hvordan er diskene organisert på RAID-level 1. Forklar hvordan diskene er organisert på RAID-level 5.
4. Hvilke fire kriterier definerer et presist interrupt?
5. Forklar forskjellen mellom HDD og SSD når det gjelder lesing, skriving/overskriving og sletting av filer. Hva er poenget med TRIM kommandoen?
6. Hva betyr det at et operativsystem er tilpasset SSD disker (slik som f.eks. Windows 7 er).

## 11.7 Lab exercises

### 1. Prosesser og tråder.

Lag et script myprocinfo.ps1 som gir brukeren følgende meny med tilhørende funksjonalitet:

- 1 - Hvem er jeg og hva er navnet på dette scriptet?
- 2 - Hvor lenge er det siden siste boot?
- 3 - Hvor mange prosesser og tråder finnes?
- 4 - Hvor mange context switch'er fant sted siste sekund?
- 5 - Hvor stor andel av CPU-tiden ble benyttet i kernelmode og i usermode siste sekund?
- 6 - Hvor mange interrupts fant sted siste sekund?
- 9 - Avslutt dette scriptet

Velg en funksjon:

Hint: bruk en switch case for å håndtere menyen, og hvert menyvalg bør føre til en Write-Host hvor svaret settes inn via \$(), og du vil nok få god bruk av cmdlet'n Get-WmiObject. Installer WMI Tools, og bruk WMI CIM Studio til å søke deg frem til de properties (og de tilhørende objekter).

(Alternativt kan du nok finne mye med cmdlet'n Get-Counter også)



# Chapter 12

## Deadlock

### 12.1 Intro

#### 12.1.1 Resources

##### Resources

- Deadlocks can occur on *hardware resources* or *software resources*
- *Preemptable* and *Nonpreemptable* resources

TAVLE:

Hardware: Scanner, Brenner, Printer, Plotter, CPU, Minne

Software: filer, OS-tabeller, database-records

Preemptable: kan tas vekk fra en prosess uten problemer, f.eks. CPU eller Minne

Deadlocks dreier seg om Nonpreemptable ressurser, ofte fler av hver type (f.eks. to Brennere).

##### Resources

- Sequence of events required to use a resource:
  1. *Request* the resource
  2. *Use* the resources
  3. *Release* the resource
- If resource not available
  - Process can wait (*we assume this*)
  - Process can return error code

### 12.1.2 Semaphores

#### Protecting Resources with Semaphores

```

typedef int semaphore;
semaphore resource_1;

void process_A(void) {
    down(&resource_1);
    use_resource_1( );
    up(&resource_1);
}

(a)

```

```

typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

(b)

```

Ressurser kan beskyttes med semaforer, her er kodeeksempel for en eller to ressurser.

#### Code with Potential Deadlock

```

typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

(a)

```

```

semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}

(b)

```

Eksempel med blyant og linjal for å tegne en rett linje.

A må i (a) bare vente, mens i (b) vil en context switch etter at A har gjort down på ressurs 1 være ille siden B da vil gjøre down på ressurs 2 og tvinge frem context switch tilbake til A som heller ikke kan gjøre noe annet enn å vente: deadlock!

### 12.1.3 Deadlock def

#### Deadlocks

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
- No processes can run, release resources or wake up: this is called a *resource deadlock*.

Vi husker dining philosophers: deadlock hvis alle plukker opp sin venstre gaffel og venter på at den høyre skal bli ledig (hvis ikke venting men legger ned igjen, og fortsetter synkront å plukke opp legge ned alle samtidig har vi en relatert problemstilling: starvation).

#### 12.1.4 Four conditions

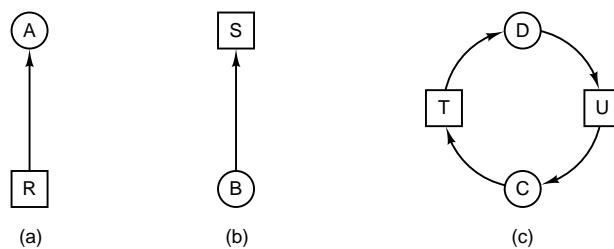
##### Conditions for Resource Deadlocks

1. *Mutual exclusion*
  - Each resource assigned to one process or available.
2. *Hold and wait*
  - Processes holding a resource can ask for other resources.
3. *No preemption*
  - Resources cannot be forcibly taken away, must be released.
4. *Circular wait*
  - Must be a circular chain of two or more processes, each waiting for a resource held by the next member of the chain.

(Note: 1-3 leads to possibility of deadlock, 1-4 guarantees deadlock)

#### 12.1.5 Modelling

##### Deadlock Modelling

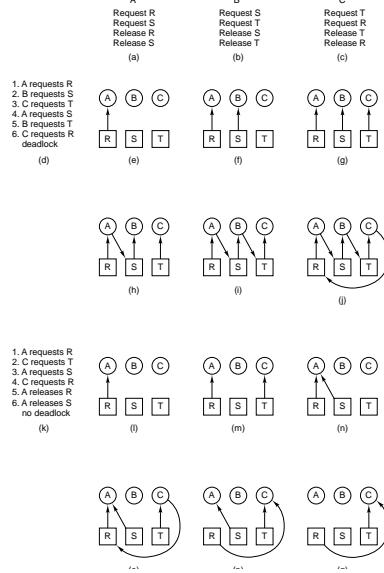


R er tildelt prosess A.

Prosess B ber om S.

Prosessene C og D er i deadlock.

### How Deadlock Can Happen/Be Avoided



(a)-(c) kjøres prosessene sekvensielt og ingen deadlock oppstår, men dette er dumt i praksis siden man ikke har noen parallellitet og en prosess som blokkerer på I/O ødelegger ytelsen.

(d)-(j) anta round-robin og blokkering på I/O, i (h) blokkerer A, i (i) blokkerer B, og i (j) blokkerer C og vi har deadlock.

Hvis OSet bestemmer seg for å vente med å kjøre B til A og C er ferdig unngår vi deadlock, men hvordan kan OSet kjenne til at det er fare for deadlock? (deadlock avoidance)

## 12.2 Dealing with Deadlocks

### Dealing with Deadlocks

1. Ignore the problem.
2. Detection and Recovery.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, negating one of the four conditions.

### 12.2.1 Ostrich alg

#### The Ostrich Algorithm

- Stick your head in the sand and pretend there is no problem at all.
- *Common approach! (with some modifications)*

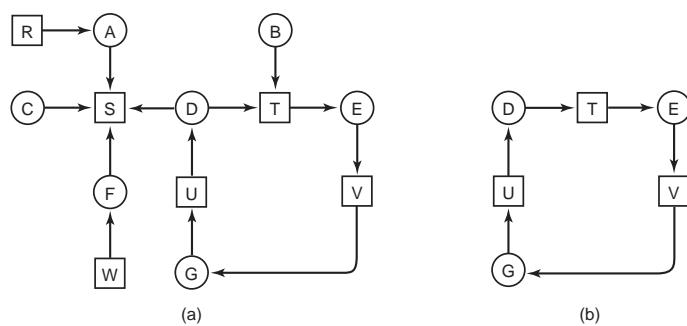
Vanlig grunnleggende tanke i UNIX og Windows, problemet oppstår så sjeldent at det er ikke verdt ytelsestapet å innføre mekanismer som håndterer deadlock.

Moderne Linux og Windows har en del forebyggende tiltak, resource ordering, dynamiske grenser på kritiske systemtabeller, osv

### 12.2.2 Detect/Recover

#### Resource graphs

##### A Resource Graph with a Cycle



Hvis vi bare har en instans av hver ressurs kan denne metoden brukes for å detektere deadlock.

*Vi lar altså deadlock inntrefte og forsøker komme oss ut av det (recovery).*

*Vi har deadlock hvis vi finner en syklus i grafen.*

Formelle algoritmer for å sjekke grafer for sykler henter vi fra diskret matematikken (kanskje fra AlgMet emnet?).

#### Recovery

##### Deadlock Recovery

- Recovery through Preemption

- Recovery through Rollback
- Recovery through Killing Processes

Preeemption er altså å ta bort en ressurs fra en prosess, veldig vanskelig i praksis (f.eks. scanner eller printer).

Prosesser som deadlocks kan restartes fra registrere sjekkpunkt (snapshots).

Drep den prosessen som frigir de nødvendige ressurser og som samtidig tar minst skade (dvs. ikke den prosessen som oppdaterer en database) (f.eks. tre prosesser som alle ønsker printer og plotter, men bare en har begge og de to andre er deadlocked).

### 12.2.3 Avoidance

#### States

##### A Safe State

	Has	Max		Has	Max		Has	Max		Has	Max		Has	Max			
A	3	9	(a)	A	3	9	(b)	A	3	B	0	—	(c)	A	3	9	
B	2	4		B	4	4		B	0	—	C	2	7		B	0	—
C	2	7		C	2	7		C	7	7		C	0	—			
	Free: 3			Free: 1			Free: 5			Free: 0			Free: 7				

TAVLE: En tilstand er safe hvis den

1. ikke er i deadlock og
2. det finnes en scheduling rekkefølge hvor alle prosessene kan kjøres ferdig selv om alle ber om sine maksimale ressurskrav

##### An Unsafe State

	Has	Max		Has	Max		Has	Max		Has	Max					
A	3	9	(a)	A	4	9	(b)	A	4	B	—	—	(c)	A	4	9
B	2	4		B	2	4		B	4	4	—	—		B	—	—
C	2	7		C	2	7		C	2	7		C	2	7		
	Free: 3			Free: 2			Free: 0			Free: 4						

A har fått en tilleggsressurs fra (a) til (b), det skulle den ikke fått fordi (b) representerer en unsafe tilstand!

Men det er ikke sikkert at det blir deadlock!

I en safe tilstand vi garantert systemet kunne kjøre ferdig, fra en unsafe tilstand finnes ingen slik garanti (men det kan hende det går bra, men vi har altså ingen garanti for det i en unsafe tilstand).

En unsafe tilstand er altså IKKE en deadlock tilstand.

## Bankers alg single

### Banker's Algorithm for a Single Resource

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7
Free:	10	
	..	

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7
Free:	2	
	..	

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7
Free:	1	
	..	

Banker's referer til en bank som gir kreditt til et sett kunder og må passe på at man ikke går tom for penger i banken.

Banker's vil sjekke hver tilstand og utsette alle forsøk på å gå til en unsafe tilstand, dvs hvis en prosess  $P_1$  kommer med en ressursforespørsel som fører systemet over i en unsafe tilstand, utsettes denne forespørselen til noen andre prosesser har frigitt de nødvendige ressurser slik at  $P_1$ 's ressursfoespørsel kan imøtekommes uten av systemet ender i en unsafe tilstand.

## Bankers alg mult

### Banker's Algorithm for Multiple Resources

	Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	
Resources assigned					

	Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	
Resources still needed					

$E = (6342)$   
 $P = (5322)$   
 $A = (1020)$

Banker's algoritme er identisk med "deadlock detection algorithm" men forskjellen er at når vi kaller den Banker's algoritme betyr det at den benyttes hele tiden til å styre unna unsafe tilstander (dvs styre unna muligheten for deadlock). Ved "deadlock detection" så lar man prosessene få sine ressurskrav så fremt det er mulig, og man bare lar deadlock oppstå, så kjører man "deadlock detection algorithm" av og til for å sjekke etter deadlock.

En annen forskjell er at ved Banker's antar vi at alle prosesser ber om sine maksimale ressurskrav, mens ved "deadlock detection algorithm" ser vi bare på hva som er ressursfoespørslene til prosessene i det algoritmen kjøres.

Altså: i det en prosess gjør en ressursforespørsel, later systemet som om denne aksepteres og kjører Banker's algoritme for å sjekke om systemet da havner i en unsafe tilstand, hvis så er tilfelle utsettes ressursfoespørselen.

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

$C - A$

	R1	R2	R3
	9	3	6

	R1	R2	R3
	0	1	1

Resource vector R Available vector V

(b) P2 runs to completion

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

$C - A$

	R1	R2	R3
	9	3	6

	R1	R2	R3
	6	2	3

Resource vector R Available vector V

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

$C - A$

	R1	R2	R3
	9	3	6

	R1	R2	R3
	7	2	3

Resource vector R Available vector V

(d) P3 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

$C - A$

	R1	R2	R3
	9	3	6

	R1	R2	R3
	9	3	4

Resource vector R Available vector V

Stalling, W. "Operating Systems: Principles and Design Principles". 7th ed.. Pearson, 2012, page 292.

Gå gjennom slik den står, kjør animasjon med default verdier

<http://gaia.ecs.csus.edu/~zhangd/oscal/Banker/Banker.html>

Se hvordan det simuleres prosesskjøring og ressurser frigjøres.

MERK: systemet mottar en forespørsel fra en prosess, så brukes Banker's for å sjekke om systemet havner i en unsafe tilstand hvis forespørselen imøtekommes, hvis så er tilfelle utsettes forespørselen.

Kjør så med Allocation matrix (5 1 1) for  $P_2$  og Claim matrix (2 1 1) for  $P_1$ , (6 3 3) for  $P_2$  og (4 2 5) for  $P_3$ . Dette er en unsafe tilstand og  $P_1$ 's forespørsel kan ikke imøtekommes, den må utsettes.

#### 12.2.4 Prevention

**Deadlock Prevention** *Can we make sure any of the four conditions are not satisfied?*

##### 1. Mutual exclusion

- Each resource assigned to one process or available.

2. Hold and wait

- Processes holding a resource can ask for other resources.

3. No preemption

- Resources cannot be forcibly taken away, must be released.

4. Circular wait

- Must be a circular chain of two or more processes, each waiting for a resource held by the next member of the chain.

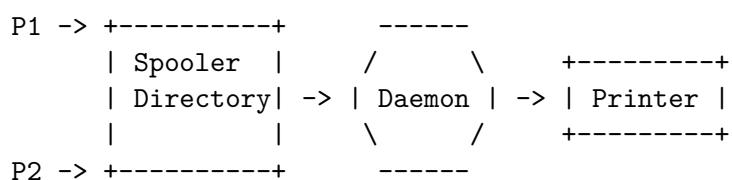
JA, dette er det som gjøres i praksis siden Banker's (og deadlock detection algoritmen) er umulig i praksis siden man ikke vet en prosess sine ressursbehov på forhånd.

### Mutual exclusion

#### Attacking Mutual Exclusion

- Can we avoid exclusive access to resources?
- Yes, for some resources we can do *spooling*.
- Good principle: *Avoid assigning a resource when that is not absolutely necessary, and try to make sure that as few processes as possible may actually claim the resource.*

TAVLE:



Deadlock kan oppstå hvis P1 og P2 fyller opp spoolerdir uten at noen av dem blir ferdige med å skrive dit, da venter begge på ledig plass som bare kan frigjøres av hver av dem siden daemon ikke vil begynne skrive ut en jobb før hele jobben er i køen (i spoolerdir).

Mao, vi har gjernet deadlock-muligheten knyttet til printern men skapt en deadlock-mulighet knyttet til diskplass (men diskplass er idag såpass tilgjengelig at en spoolerdir neppe vil fylles opp, og dermed bør sannsynligheten for deadlock være minimal).

## Hold & wait

### Attacking Hold and Wait

- Can we require all processes to state all their resource needs in advance?
- No, not possible and not practical.
- (and if we could, we could suddenly apply the Banker's algorithm)

For det første vil en prosess vente lenge på å få alle ressurser den trenger, og samtidig vil den ofte unødvendig holde på mange ressurser.

## No preempt

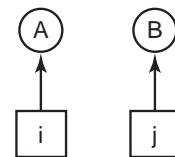
**Attacking No Preemption** In practice, only possible with preemptable resources (CPU, Memory).

Og som vi husker fra de første foilene, deadlock problematikken involverer primært nonpreemptable ressurser.

## Circular wait

### Attacking Circular Wait

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD-ROM drive



*Just order the resources!*

Sirkulær venting kan unngås ved å nummerere ressursene, og innføre regelen *Alle ressursforespørslar fra en prosess må være i riktig numerisk rekkefølge*.

F.eks. hvis  $i > j$  så kan ikke A be om j, mens hvis  $j > i$  så kan ikke B be om i, en syklus kan altså ikke oppstå.

Se Rule 1 og Rule 2 i Del 3 av

<http://www.ddj.com/hpc-high-performance-computing/204801163>

### **Deadlock Prevention Summary**

<b>Condition</b>	<b>Approach</b>
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

### 12.3 Theory questions

1. Det kreves at fire ulike betingelser alle må være oppfylt for at deadlock skal inntre. Beskriv disse kort.
2. Hva vil det si at en tilstand er “unsafe” i forbindelse med deadlock? Vil det garantert tilsi at en får en deadlock?
3. Forklar kort hva en ressursgraf er og hvordan den kan benyttes til å avsløre om vi har en deadlock situasjon.
4. Tanenbaum oppgave 6.1  
Students working at individual PCs in a computer laboratory send their files to be printed by a server which spools the files on its hard disk. Under what conditions may a deadlock occur if the disk space for the print spool is limited? How may deadlock be avoided?
5. Tanenbaum oppgave 6.18  
A computer has six tape drives, with  $n$  processes competing for them. Each process may need two drives. For which values of  $n$  is the system deadlock free?
6. Tanenbaum oppgave 6.20  
A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	Maximum	Available
ProcA	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
ProcB	2 0 1 1 0	2 2 2 1 0	
ProcC	1 1 0 1 0	2 1 3 1 0	
ProcD	1 1 1 1 0	1 1 2 2 1	

What is the smallest value of  $x$  for which this is a safe state?

## 12.4 Lab exercises

### 1. Hjelp til lesbarhet: byte konvertering.

Skriv et script `human-readable-byte.ps1` som leser et tall, antall bytes, fra STDIN, konverter tallet til KB, MB, GB eller TB avhengig av størrelsen på tallet, og skriver den nye størrelsen i heltall. Scriptet bør oppføre seg omtrent slik:

```
$ Write-Output 25000000 | human-readable-bytes.ps1
23.8418579101563MB
$ Write-Output 250 | human-readable-bytes.ps1
250B
```

### 2. Hjelp til lesbarhet: ns konvertering.

Skriv et script `human-readable-ns.ps1` som leser et tall, antall ns (nanosekunder), fra STDIN, konverter tallet til us (microsekunder), ms (millisekunder) eller sek (sekunder) avhengig av størrelsen på tallet, og skriver den nye størrelsen i heltall. Scriptet bør oppføre seg omtrent slik:

```
$ Write-Output 25000000 | human-readable-ns.ps1
25ms
$ Write-Output 250 | human-readable-ns.ps1
250ns
```

### 3. Beregne effektiv minneaksessstid.

Skriv et script `compute-memory-access.ps1` som beregner effektiv minneaksessstid gitt TLB-aksessstid TLB=2ns, RAM-aksessstid RAM=10ns og Disk-aksessstid DISK=10ms (med andre ord, sett disse parameterene som variable i starten av scriptet). Scriptet skal ta to kommandolinjeargumenter: andel TLB-hit og andel page faults. Scriptet bør oppføre seg omtrent slik (bruk `human-readable-ns.ps1` scriptet ditt til utskriften):

```
$ compute-memory-access.ps1 0.7 0.1
1000016.2 nanosekunder som er
1.0000162ms
$
```

### 4. En prosess sin bruk av virtuelt og fysisk minne.

Skriv et script `procmi.ps1` som tar et sett med prosess-ID'er som kommandolinjeargument og for hver av disse prosessene skriver til en fil `PID-dato.meminfo` følgende info:

- Total bruk av virtuelt minne (*VirtualMemorySize*)
- Størrelse på Working Set

Eksempel kjøring:

```
$ procmi.ps1 420 8566
$ Get-Content 420--20100314-221523.meminfo
***** Minne info om prosess med PID 420 *****
Total bruk av virtuelt minne: 35.6875MB
Størrelse på Working Set: 816KB

$
```

# Chapter 13

## Virtualisering

### 13.1 Introduction

#### Virtualization

- *Virtualization* means allowing a single computer to host multiple virtual machines.
- 40 year old technology!

*Poenget med dette temaet er å få innblikk i teknologien som er involvert i virtualiseringen som skjer på en fysisk maskin, slik at vi lettere kan forstå all den videre debatten rundt virtualisering som vanligvis er mye mer praktisk rettet: hvilke applikasjoner som bør/bør ikke virtualiseres, hva som funker/funker ikke ved snapshots og live migrering, hvorfor sammenheng hardware og versjon av VMM betyr mye, osv*

Se

<http://en.wikipedia.org/wiki/Virtualization>

vi snakker om platform virtualisering.

Se

[http://en.wikipedia.org/wiki/Comparison\\_of\\_platform\\_virtual\\_machines](http://en.wikipedia.org/wiki/Comparison_of_platform_virtual_machines)

god oppdatert oversikt over de forskjellige variantene.

#### Why Virtualization?

- Servers can run on different virtual machines, thus maintaining the partial failure model that a multicomputer.
- It works because most service outages are not due to hardware.

- Save electricity and *space*!

Først og fremst det siste punktet sammen med mye lettere sysadm, mulighet for snapshot, live migrering, legacy systemer, prototyping osv

### 13.1.1 Requirements

#### Virtualizable Hardware

- Popek and Goldberg, 1974:
  - A machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions.
- this caused problems on X86 until 2005...

TAVLE:

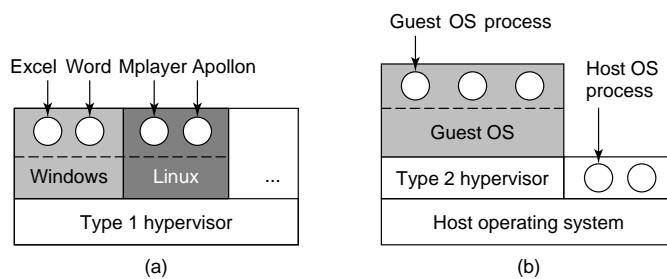
**Sensitiv instruksjon** kan bare utføres i kernel mode.

**Priviligert instruksjon** forårsaker en trap (dvs en overgang til kernel mode) hvis utført utenfor kernel mode.

Merk: dette er altså “det strenge kravet”, det må ikke være oppfylt, det har VMware bevist. Men det må være oppfylt for å få til klassisk *trap-and-emulate* virtualisering slik det alltid har blitt gjort på IBMs stormaskiner (siden ca 1970).

## 13.2 Hypervisors

#### Hypervisors



TAVLE:

Hypervisor = Virtual Machine Monitor (VMM)

**Type 1 hypervisor** direkte på hardware som et OS. Krever at Popek og Goldberg kravet er oppfylt. Alle sensitive instruksjoner som utføres av gjesteOS'et fanges opp av hypervisoren og emuleres (utføres på vegne av gjesteOS'et).

**Type 2 hypervisor** kjører på toppen av et OS (men har mye av koden sin sammen med OSet i kernel mode (i form av kjernemoduler/drivere).

(mao denne grensen kan betraktes som noe flyttende siden den er en gammel definisjon som stammer fra nettopp Popek og Goldbergs artikkel fra 1974, lenge før det fantes teknologiene binæroversetting og paravirtualisering)

## 13.3 CPU

### 13.3.1 Binary translation

#### Binary Translation

- E.g. *Binary translation* in VMware:
  - During program exec, basic blocks (ending in jump, call, trap, etc) are scanned for sensitive instructions
  - Sensitive instructions are replaced with call to vmware procedures
  - These translated blocks are cached
- Very powerful technique, can run at close to native speed because VT hardware generate many traps (which are expensive).

Dette er altså teknologien utviklet av VMware fra DISCO-prosjektet ved Stanford.

Koden til gjesteOSet granskes rett før den kjøres, og sensitive instruksjoner endres til kall til hypervisoren.

Noe tilsvarende teknologi finnes også i VirtualBox: "VirtualBox contains a Code Scanning and Analysis Manager (CSAM), which disassembles guest code, and the Patch Manager (PATM), which can replace it at runtime." fra

<http://www.virtualbox.org/manual/ch10.html#idp21833280>

**Figur side 3 "Hypervisor Architecture"** Ved software virtualisering (som i all hovedsak betyr VMware's binæroversettelse og Xen's paravirtualisering) kjører koden til gjeste OS'et i ring 1, og det er denne koden som dynamisk binæroversettes/statisk paravirtualiseres. Usermode-koden til applikasjonene i ring 3 er ikke noe problem og de kjøres direkte, men problemene oppstår når gjestekjernen i ring 1 forsøker gjøre sensitive instruksjoner som ikke er del av de privilegerte instruksjonene, det er disse instruksjonene som først og fremst må binæroversettes. Samtidig binæroversettes mye annet grunnet optimalisering.

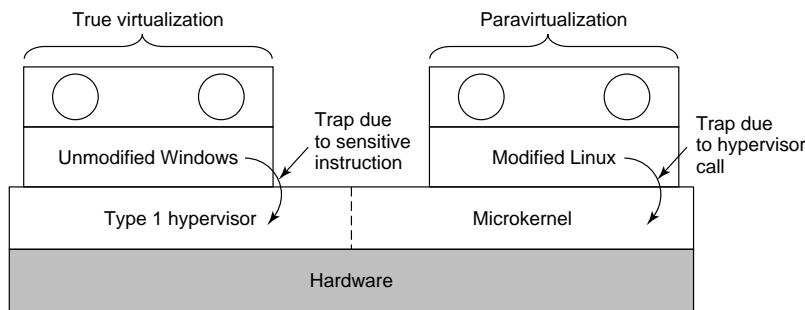
**Figur side 5** Alle systemkall havner altså hos VMM istedet for gjesteoperativsystemet, slik at VMM må videresende alle systemkall til gjesteoperativsystemet som kjører binæroversatt kode i ring 1. (Les også første to avsnitt side 7 "Much has been...")

"It is clear ..." side 6 caching er viktig (TC = translator cache), men utfordringene er fortsatt systemkall, minnehåndtering og I/O.

**Figur side 8 "Sysenter"** (sysenter og sysexit er altså enklere mode shifts enn int 0x80)  
Denne viser igjen det samme som i forrige figur men merk kommentarene under figuren som viser at et systemkall på en virtuell maskin fort kan ta opp imot 10 ganger så lang tid som på en vanlig maskin.

### 13.3.2 Paravirtualization

#### True and Paravirtualization



Mens både type 1 og type 2 hypervisor fungerer med umodifisert OS, krever paravirtualisering modifisering av OSet.

Alle sensitive instruksjoner i OSet erstattes med kall til hypervisoren.

Hypervisoren blir i praksis en mikrokjerne ved paravirtualisering.

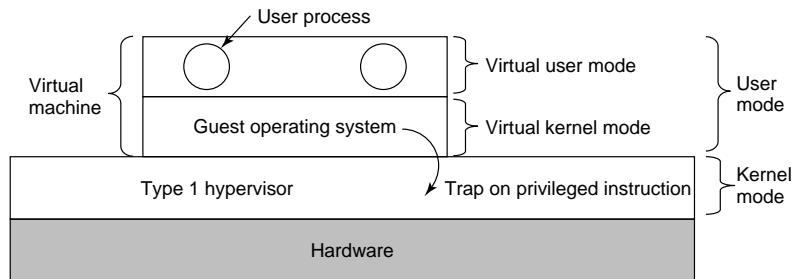
Paravirtualisering er en statisk endring av gjesteOSet slik at sensitive instruksjoner endres til kall til hypervisorn (med andre ord: samme type endring i koden som type 2 hypervisor (men statisk, dvs forhåndsendret) slik at en type 1 hypervisor kan benyttes).

*En fordel med paravirtualisering er at den tillater mye mer endring til gjesteoperativsystemet enn binæroversetting, derav mulighet for enda mer optimalisering (redusere antall overganger til VMM), men det går selvfølgelig på bekostning av fleksibilitet, dvs det er ikke alle OS du har tilgang til kildekoden til...*

**Figur side 11** All I/O gjøres av en spesielt priviligert VM (kalt Domain0 i Xen). I denne figuren kan ikke VM3 kjøres fullt ut paravirtualisert siden det er et umodifisert gjesteOS, men det er ment å illustrere en kombinasjon av binæroversetting og paravirtualisering

### 13.3.3 HW virtualization

#### Hardware Virtualization



**Figur side 13** Innføring av hardwarestøtte for virtualisering (Intel VT-x og AMD-V) på x86 betyr å innføre en ny ”Ring -1” som kalles ”VMX root mode” hvor VMM kjører. På denne måten kan gjesteoperativsystemet kjøre i sin tiltenkte Ring 0 slik at de kan oppføre seg normalt (det trengs ikke binæroversetting eller paravirtualisering). Dette kan være en fordel ved enkle systemkall siden de kan utføres uten overgang til VMM. Ulempen er at man mister optimaliseringsmulighetene man har med binæroversetting og paravirtualisering.

DEMO:

CPU flags som vi må sjekke for å undersøke i hvilken grad vi har hardware støtte for virtualisering (fra <http://virt-tools.org/learning/check-hardware-virt/>):

**vmx** Intel VT-x, basic virtualization.

**svm** AMD SVM, basic virtualization.

**ept** Extended Page Tables, an Intel feature to make emulation of guest page tables faster.

**vpid** VPID, an Intel feature to make expensive TLB flushes unnecessary when context switching between guests.

**npt** AMD Nested Page Tables, similar to EPT.

**tpr\_shadow and flexpriority** Intel feature that reduces calls into the hypervisor when accessing the Task Priority Register, which helps when running certain types of SMP guests.

**vnmi** Intel Virtual NMI feature which helps with certain sorts of interrupt events in guests.

```
egrep -o '(vmx|svm|ept|vpid|npt|tpr_shadow|flexpriority|vnmi)' \
/proc/cpuinfo | sort | uniq
```

(på Windows bruk sysinternals-verktøyet coreinfo -v)

Noen ytelsesmålinger for å forsøke illustrere forskjeller:

Ren usermode prosess: `time ./sum`

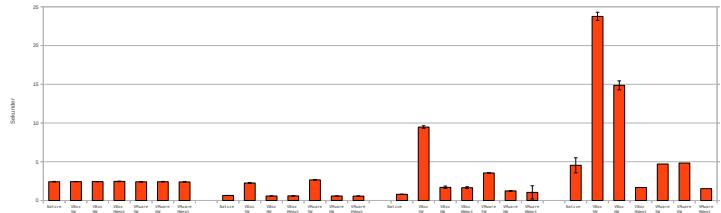
Blandet usermode/kernelmode, mange enkle systemkall: `time ./getppid` og `time ./gettimoofday`

Mye kernelmode med en del minneallokeringer: `time ./forkwait`

- getpid/gettimeofday bør gi fordel til HW virtualisering siden VMM ikke trengs (for BT så må syscall alltid passere VMM)

- forkwait bør gi fordel BT siden mange VMM enter/exit, grunnet mye jobb for kjernen spes ift opprette minneområde og pagetabeller (alle disse vil trap-and-emulate, altså trap fra gjesteOSkjernen til VMM), mao her trengs virtualiseringsstøtte i MMU.

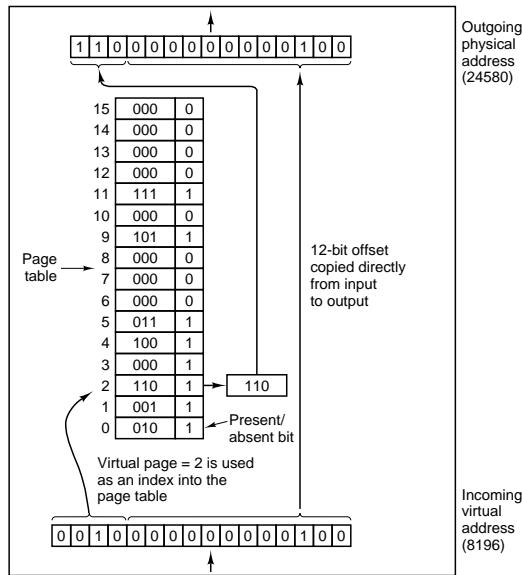
## Performance Measurements



(MERK: presise ytelsesmålinger er utrolig vanskelig siden så mange forhold spiller inn på resultatet, så ikke se deg blind på disse resultatene, de er bare en forsiktig indikator)

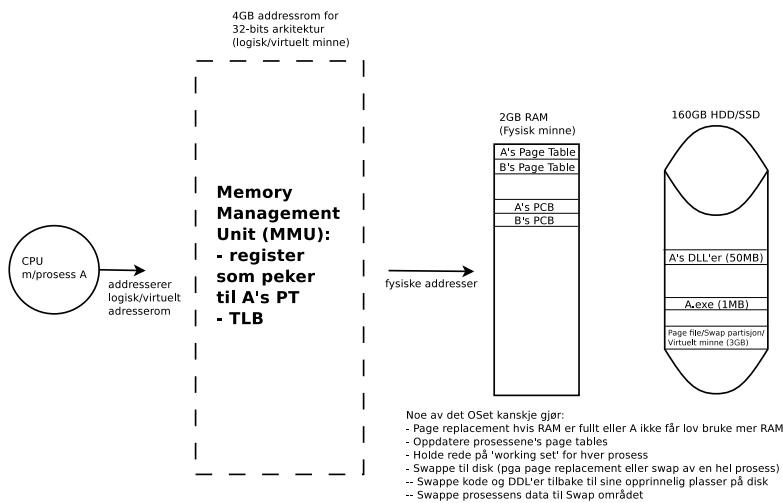
## 13.4 Memory

### Virtual and Physical Addresses



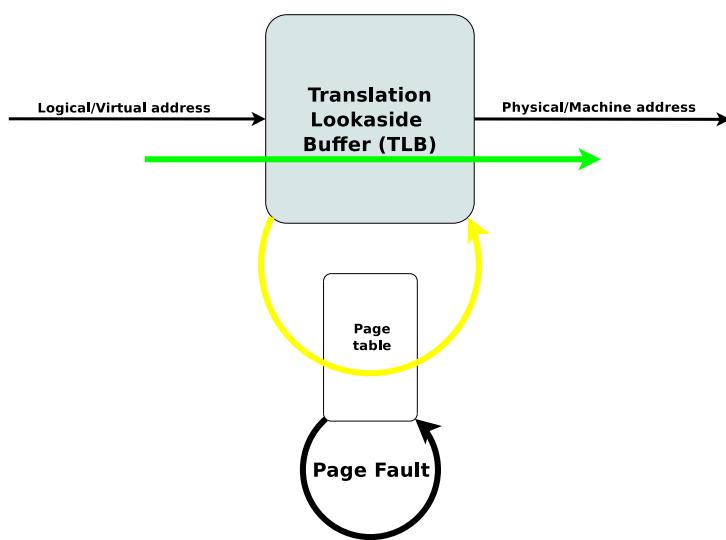
Vi husker hvorfor vi har page tabeller og hvordan adresseoversetting fra logisk/virtuelt minne til fysisk minne funker via page tabellen.

## Virtual Memory Architecture



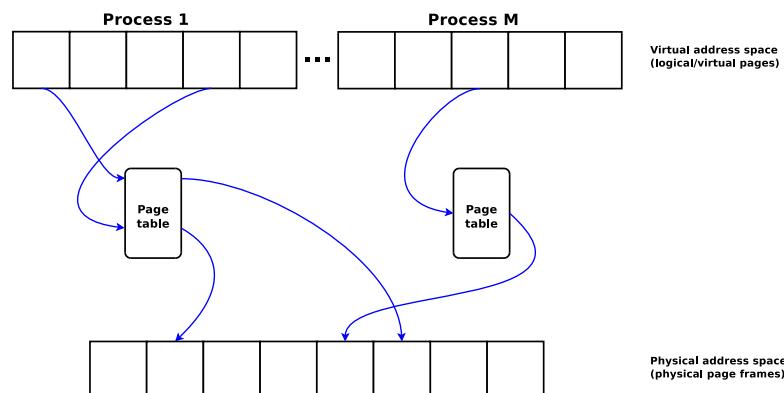
Og slik ser det overordnede bildet for virtuelt minne ut slik vi husker det.

## Hits, Misses and Page Faults



Denne sammenhengen kjenner vi til siden vi har kodet den i bash og powershell. Så lenge alle oppslag er cachet i TLB går alt veldig kjapt.

### Traditional Page Tables

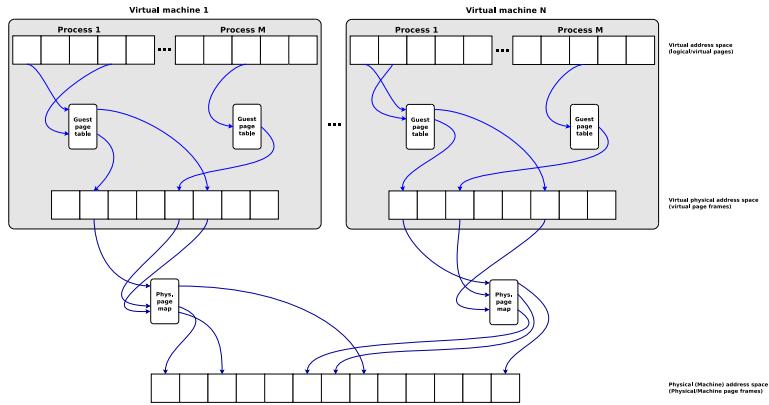


(Note: this and the following figures inspired by VMware White Paper, "Performance Evaluation of Intel EPT Hardware Assist", 2009.)

Og hver prosess har altså sin page table (i det aller fleste implementasjoner). Husk at denne som regel er en multilevel page table i praksis (to nivåer på 32-bits X86, fire nivåer på 64 bits X86 (siden bare 48 bits benyttes i praksis)).

La oss nå se på hva som skjer når vi må innføre et mellomnivå (hypervisoren) mellom hardware og OS (siden OSet nå kjører i en virtuell maskin styrt av en hypervisor).

### Page Tables in Virtual Machines

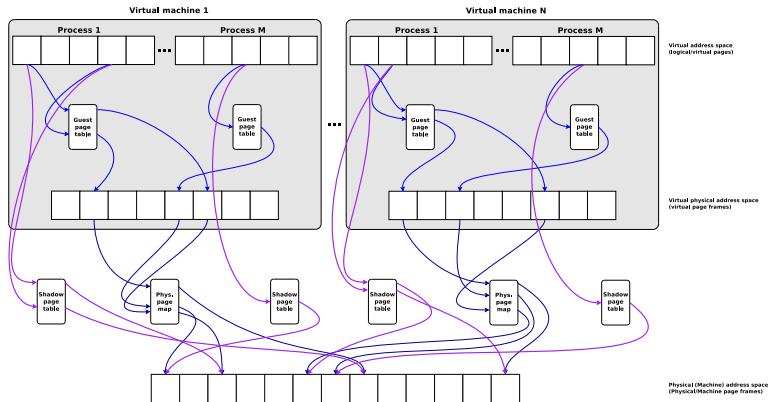


Her forholder prosessene i de virtuelle maskinene seg ikke lenger til fysisk minne (RAM), men det de tror er fysisk minne. Page tables kalles nå Guest page tables, og det som på en måte er den virkelige page table kalles en physical page map som regel.

MERK: TLB må fortsatt fylles med mappingen fra logiske/virtuelle adresser til fysiske/maskin adresser, og dette kan gjøres enten via Shadow page tables i software, eller med Nested page tables i hardware.

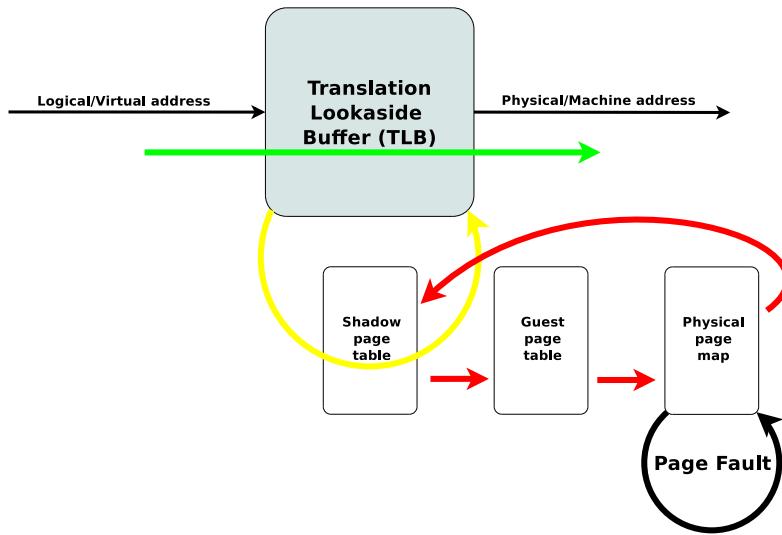
### 13.4.1 Shadow page tables

#### Shadow Page Tables (Software)



Med Shadow page tables opprettes en tredje tabell som brukes til å fylle TLB som vist i neste figur.

#### Shadow Page Tables (Software)



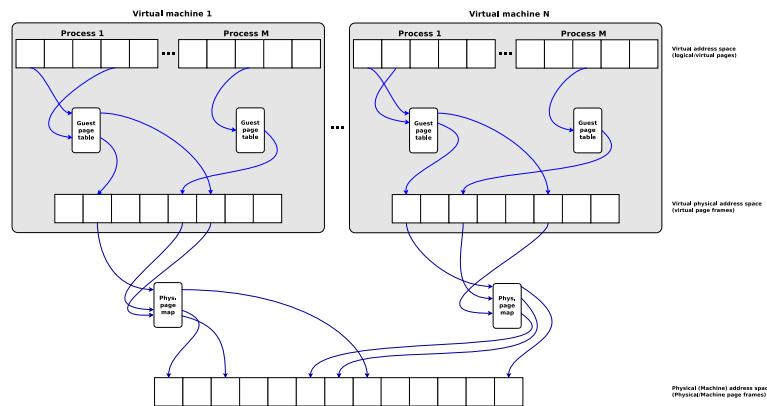
Dette fungerer bra i de fleste tilfeller, og også bedre enn hardware løsningen med nested page tables i noen tilfeller.

Problemer:

- Hver endring i guest page table må fanges opp, og det er ikke trivielt siden en prosess jo har lov til å skrive til minne (det forårsaker normalt ikke en trap).
- Hver endring i guest page table gjør at page map og shadow page table må oppdateres, noe som forårsaker overgang til hypervisor, noe som er kostbart.
- Hver endring i shadow page table (f.eks. oppdatering av references eller dirty bit i TLB, som deretter skriver til shadow page table) forårsaker også oppdateringer til page map og guest page table.

### 13.4.2 Nested page tables

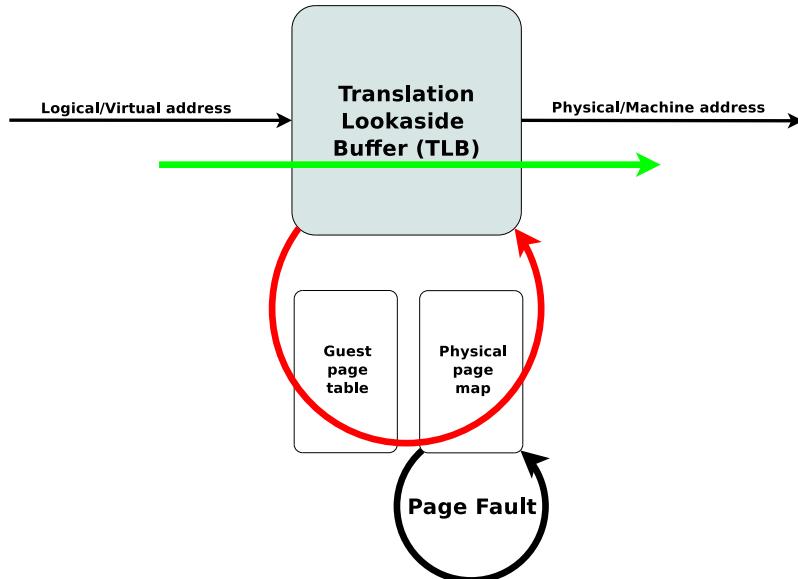
#### Nested Page Tables (Hardware)



Her bruker vi altså ikke shadow page tables, med hardware støtte så gjør vi altså ikke noe ”kunstig” i software, vi lar løsningene muligens være suboptimale og søker heller rask hardware implementasjon av disse.

Merk: vi mister altså den gule pilen, men vi får en bedre og mer optimal TLB.

### Nested Page Tables (Hardware)



**Figur side 16 “Hardware Support”** Andre generasjons hardwarestøtte for virtualisering innebærer altså en spesiell TLB som cacher guest page table og physical page map gjennomgangen (altså cacher hele 2D page walk'n) sammen med selve guest-virtuell-address til fysisk/maskin-addresse slik at TLB blir veldig effektiv (dvs man slipper vedlikeholde en shadow page table). Problemet er at en TLB miss medfører en langt mer omfattende rekke av tabelloppslag enn om man hadde en shadow page table. Istedet for  $N$  oppslag i en  $N$ -level shadow page table blir det  $N \times N$  oppslag (eller  $N \times M$  egentlig hvis man skal være presis) (siden man må via physical page maps for hver guest page table oppslag).

**AMD paper: Figur 1 side 7** Dette er altså slik multi-level page tabeller fungerer, i dette tilfelle på dagens 64-bits arkitektur, mao fire nivåer med page tabeller før man havner i RAM).

**AMD paper: Figur 5 side 14** Og slik blir det da med VMer, for hvert oppslag en VM gjør i sitt virtuelle fysiske minne, må tilsvarende ”page walk” gjøres i de virkelige page tabellene (altså de som vi har kalt physical page map). *Cluet er altså at et oppslag i ett av nivåene i en page tabell gir adressen til starten av neste nivå page tabell, og denne adressen må oversettes til korrekt fysisk/maskin adresse via en page walk.*

## Implementations

- Intel EPT
- AMD RVI (NPT)

*These also include an ASID (Address Space IDentifier) field in the TLB entries*

ASID gjør at hver TLB entry kan tilhøre en bestemt VM, derav behøver ikke TLB flushes ved context switch. Dette gjør TLB langt mer effektiv (gitt at den er stor nok). Man kombinerer gjerne dette med større page størrelser (f.eks. 2MB istedet for 4KB).

## 13.5 I/O

### I/O Virtualization

- It is easy to add more CPUs/CPU-cores
- It is easy to add more memory
- *It is not easy to add more I/O capacity...*

Det er selvfølgelig lett å legge til mer diskplass, men vi tenker litt mer generelt, I/O er mange enheter med en bestemt busstruktur.

*I/O er nok ofte den største flaskehalsen for virtualisering.*

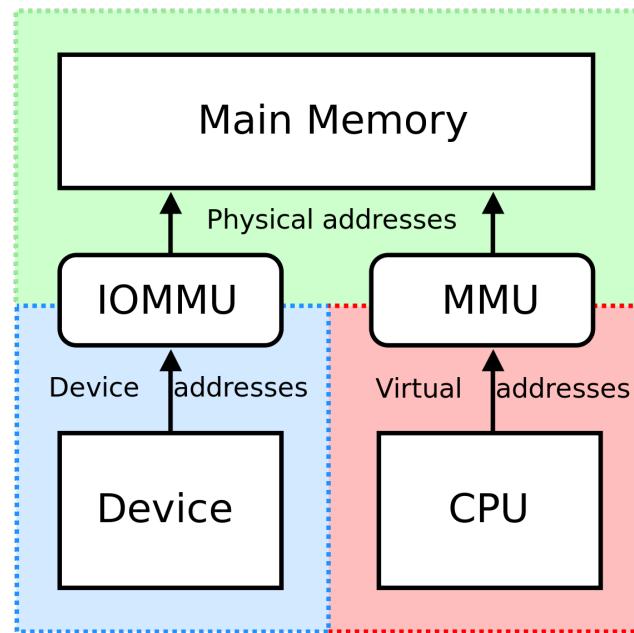
### I/O Virtualization: The DMA problem

- VM1 is mapped to 1-2GB of RAM, VM2 is mapped to 2-3GB of RAM
- VM1 is given direct access to a DMA-capable I/O-device
- VM1 programs the DMA controller to write to the area 2-3GB of RAM, *overwrites VM2's memory...*

Løsningen er å innføre en IOMMU enhet som virker mye på samme måte som den MMU vi kjenner fra før.

IOMMU er for øvrig ikke noe helt nytt på x86-arkitekturen, men en generalisering av de teknologiene som tidligere er kjent som GART (Graphics Address Remapping Table) og DEV (Device Exclusion Vector).

### 13.5.1 IOMMU



IOMMU virtualiserer addressene for I/O devicene på samme måte som addressene som kommer fra CPUen, og har TLB akkurat som vanlig MMU.

Dette gjør at IOMMU tillater direkte tilordning av I/O-enheter til VM'er.

Mao, en IOMMU baserer seg på:

- I/O page tables som gjør adresseoversetting og *akcess kontroll*
- En Device table som tilordner I/O-enheter direkte til VM'er
- En interrupt remapping table for å mappe I/O fra riktig enhet tilbake til riktig gjesteoperativsystemet (dvs tilbake til riktig VM)

MERK: innføring av IOMMU er ikke uten kostnad, dvs den medfører forsinkelse grunnet med overhead (og på samme måte som med MMU blir ved veldig avhengig av at TLB har cachet de fleste entries).

### Implementations

- Intel VT-d
- AMD-Vi

Mao, husk denne tabellen for hardware-støtte for virtualisering:

	CPU (1st gen)	Memory (2nd gen)	I/O
AMD	AMD-V	RVI/NPT	AMD-Vi
Intel	VT-x	EPT	VT-d

## **13.6 Theory questions**

1. Forklar påstanden til Popek og Goldberg fra 1974: *A machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions.*
2. Tanenbaum oppgave 8.28  
VMware does binary translation one basic block at a time, then it executes the block and starts translating the next one. Could it translate the entire program in advance and then execute it? If so, what are the advantages and disadvantages of each technique?
3. Forklar hvordan datamaskinarkitekturens beskyttelsesringer (*protection rings*) benyttes ved virtualisering når virtualiseringsteknikken er binæroversetting (VMware's teknologi).
4. Hva karakteriserer en applikasjon som vil være utfordrende/problematisk å kjøre på en virtuell maskin?.
5. Forklar kort fordeler og ulemper med *shadow page tables* i forhold til *nested/extended page tables*.
6. Forklar kort hvordan *IOMMU* kan være nyttig hardwarestøtte for virtualisering.

## 13.7 Lab exercises

### 1. Informasjon om deler av filsystemet.

Skriv et script `fsinfo.ps1` som tar en directory som argument og skriver ut

- Hvor stor del av partisjonen directorien befinner seg på som er full
- Hvor mange filer (utelat alle directories!) finnes i directorien (inkl subdirectories), gjennomsnittlig filstørrelse, og full path og størrelse til den største filen

Eksempel kjøring:

```
$ fsinfo.ps1 cf3
Partisjonen cf3 befinner seg på er 81.4456135970535% full
Det finnes 4 filer.
Den største er C:\Users\erikh\scripts\cf3\helloworld2.cf
som er 302B stor.
Gjennomsnittlig filstørrelse er 245.25B.
$ pwd
```

```
Path
-----
C:\Users\erikh\scripts
```

```
$ fsinfo.ps1 $(pwd)
Partisjonen C:\Users\erikh\scripts befinner seg på er
81.4456135970535% full
Det finnes 479 filer.
Den største er C:\Users\erikh\scripts\Lessmsierables-20050611
\LessMSIerables\wix.dll som er 992KB stor.
Gjennomsnittlig filstørrelse er 12.4778509916493KB.
$
```

### 2. Regulære uttrykk anvendt på filsystemet.

Skriv et script `fnamecheck.ps1` som tar en directory som argument og skriver ut alle filnavn (inkl alle subdirectories) som inneholder norske tegn og/eller mellomrom i filnavnet.

# Chapter 14

## Objektsikkerhet

### 14.1 Introduction

#### Introduction

- Security is challenging.
- Security is usually in conflict with *user-friendliness*.

Vi skal adressere sikkerhet knyttet til operativsystemer.

Opprinnelig (60-70 tallet) var ikke datamaskiner knyttet til nettverk så sikkerhet dreide seg om å sørge for at brukerne ikke kunne få tilgang til hverandres filer, og det er i stor grad dette operativsystemsikkerhet dreier seg om.

I det datamaskiner kom på nett og internet utviklet seg kom de store nettverkssikkerhet-utfordringene og disse henger tett sammen (er uadskillelige) med operativsystemsikkerhet.

Kapittel 9 har derfor en del temaer som ikke er typisk operativsystemsikkerhet, vi skal ta litt lett på disse.

#### Security Goals and Threats

Goal	Threat
Data confidentiality	Exposure of data
Data integrity	Tampering with data
System availability	Denial of service
Exclusion of outsiders	System takeover by viruses

Husk CIA!

I tillegg er "Exclusion of outsiders" en ny greie og typisk eksempel på et problem for både operativsystemsikkerhet og nettverkssikkerhet. Et spesielt stort problem for Norge!

Her kunne man også nevnt *Privacy* (Personvern) som også er et mål, men som ikke er trivielt siden det ofte er i konflikt med *Forensics*, dvs dataetterforskning, ref. den store debatten om datalagringsdirektivet.

**Botnets...**

The screenshot shows the homepage of the website 'Aktuell Sikkerhet'. The main title 'Aktuell Sikkerhet' is displayed prominently at the top. Below it, there is a sidebar with various links: Nyheter, Siste nytt, Søk i artikkelarkivet, Tips oss!, Nyttig, Leverandører, Produktnyheter, Kalender, Jobbmarked, Meninger, Debatt, Kommentar, and Les bladet. The main content area features a news article with the headline 'Norge fjerde største opphavsland for it-angrep'. The article discusses a report from Symantec stating that Norway is the fourth largest source of IT attacks in Europe. To the right of the article, there is a sidebar with a speech bubble containing text from Roald Bjerklund, a professor at the University of Oslo, and a link to 'Nyheter direkte fra Aktuell Sikkerhet'.

Regional Rank	Previous Regional Rank	Country	Percentage of Regional Attacks	Previous Percentage of Regional Attacks	Percentage of Worldwide Attacks
1	3	United States	35%	33%	25%
2	3	United Kingdom	15%	13%	5%
3	2	China	14%	19%	13%
4	5	Norway	13%	3%	<1%
5	4	Germany	5%	4%	8%
6	6	Italy	3%	3%	3%
7	7	France	2%	2%	6%

Dette er ikke en statistikk å være stolte av...

### Intruders and Adversaries

1. Casual prying by nontechnical users.
2. Snooping by insiders.
3. Determined attempts to make money.
4. Commercial or military espionage.

*Adversary* er et mye brukt begrep innen informasjonssikkerhet og kan brukes om motstander, rival eller fiende.

Husk at hvor mye ressurser som settes inn i sikkerhet må svare til trusselnivået...

Og et problem er at det bare er den som rammes som er villig til å betale, f.eks. er det vanskelig å få PC-brukere til å beskytte seg tilstrekkelig mot Botnet trusselen fordi de selv ikke blir rammet men PCn dere brukes til å ramme andre.

### Estonia incident...

#### The Washington Post

NEWS | POLITICS | OPINIONS | BUSINESS | LOCAL | SPORTS | ARTS & LIVING | GOING OUT GUIDE | JOBS | CARS | REAL ESTATE | SHOPPING

#### Cyber Assaults on Estonia Typify a New Battle Tactic

By Peter Finn  
Washington Post Foreign Service  
Saturday, May 19, 2007; A01

TALLINN, Estonia, May 18 -- This small Baltic country, one of the most wired societies in Europe, has been subject in recent weeks to massive and coordinated cyber attacks on Web sites of the government, banks, telecommunications companies, Internet service providers and news organizations, according to Estonian and foreign officials here.

Computer security specialists here call it an unprecedented assault on the public and private electronic infrastructure of a state. They say it is originating in Russia, which is angry over Estonia's recent relocation of a Soviet war memorial. Russian officials deny any government involvement.

### Georgia incident...

#### Russian hackers launch Georgia offensive

Cyber blitzkrieg  
By Aharon Etengoff  
Monday, 11 August 2008, 08:58

**RUSSIAN HACKERS** have launched a massive cyber attack against a number of Georgian sites and servers.

The Caucasus Network in Tbilisi is reportedly under siege by thousands of hijacked PCs that are flooding its servers with spoofed traffic.

Russian hackers have also broken into the Georgian Foreign Affairs website and replaced official photos with pictures depicting the country's president as a Nazi.

According to the [RBN](#) website, Russian computers currently control all traffic to Georgia's key servers. Although a group of German hackers managed to temporarily re-route traffic via Deutsche Telekom, the site has requested international assistance to provide Internet routing "via neutral cyber space".

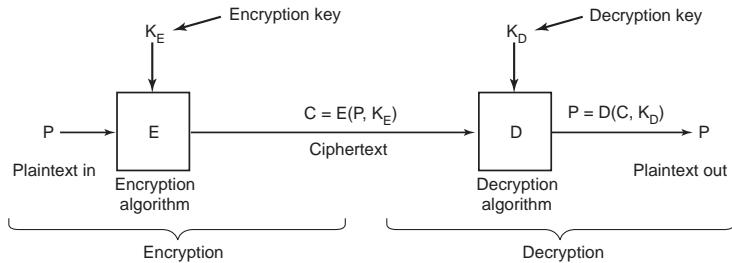
### Accidental Data Loss

1. Acts of God.
2. Hardware or software failure.
3. Human errors.

Det meste ordnes med gode backup/restore rutiner. Husk at sletting av egne filer er mest vanlig årsak til restore.

Dette er en del av det store sysadm/security management bilde og kan lett glemmes når man blir for teknisk fokuseret på ren sikkerhet, backup på egen harddisk virker høl i hue men løser altså de fleste kriser for brukerne.

## Remember Crypto?



Hvis  $K_E = K_D$  er det symmetrisk krypto, hvis i stedet vi prater om et nøkkelpar  $(K_E, K_D)$  bestående av en privat og en offentlig (public) nøkkel, så er asymmetrisk krypto (public-key crypto).

I tillegg må vi kjenne til hva en hash funksjon er, f.eks.

`echo "erikh" | sha1.`

TAVLE:

- symmetrisk krypto (`gpg -c`, `gpg -d`)
- asymmetrisk krypto
- hash/enveis funksjon (message digest)

samt

- digital signatur
- Trusted Third Party (TTP)
- sertifikat (se i browsern din)
- PKI infrastruktur

## Trusted Platform Module (TPM)

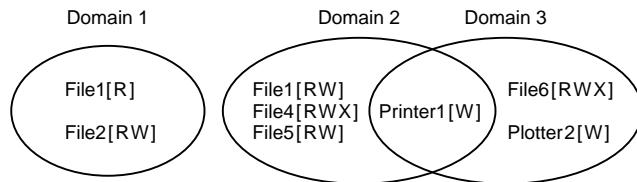
- A cryptoprocessor with nonvolatile storage.
- Allows for fast crypto and storage of keys.
- Controversial because of possible use for OS running only authorized software (e.g. only Microsoft-approved software).

TPM er altså et forslag fra industrien på en måte å oppbevare nøkler sikkert på et usikkert (untrusted) system.

## 14.2 Protection Mechanisms

### 14.2.1 Protection domain

#### Three Protection Domains



Et (beskyttelses-) domene er et sett med (objekt,rettigheter) par.

Et domene defineres ut fra hva en prosess har tilgang til, f.eks. i UNIX vil (UID,GID på Linux, SID/SID på Windows) paret definere et domene.

For best mulig sikkerhet er POLA (Principle of Least Authority, også kalt need to know) et godt prinsipp.

Alle systemkall forårsaker en domeneswitch siden operativsystemet i kernelmode har tilgang til mye mer enn en vanlig prosess i usermode.

### 14.2.2 Protection matrix

#### A Protection Matrix

		Object							
		File1	File2	File3	File4	File5	File6	Printer1	Plotter2
Domain	1	Read	Read Write						
	2			Read	Read Write Execute	Read Write		Write	
	3						Read Write Execute	Write	Write

Operativsystemet kan holde oversikt hvilke objekter som hører til hvilke domener med en matrise.

Figuren viser matrisen tilsvarende forrige figur.

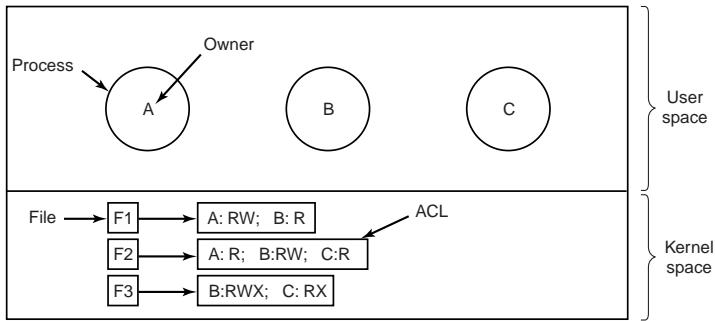
#### Adding Domains as Objects

Domain	Object										
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
1	Read	ReadWrite								Enter	
2			Read	ReadWriteExecute	ReadWrite		Write				
3						ReadWriteExecute	Write	Write			

Domenene kan tas med som objekter og en rettighet kan være å tillate domeneswitch som i figuren hvor en prosess i domene 1 kan svitsje til domene 2 men ikke tilbake.

### 14.2.3 ACL

#### Access Control Lists



Matrisen fra de to forrige figurene er lite praktisk å implementere, i praksis lagrer vi enten radene eller kolonnene (og da bare de ikke-tomme elementene).

TAVLE:

- Å lagre kolonnene kalles *Access Control List*.
- Å lagre radene kalles *Capability*.

Eierne/brukerne (som definerer domenene) kalles som regel *subjects/principals* og tilgangen kontrolleres til *objects* (som regel er objektene filer av en eller annen type, men kan være hardware devicer, semaforer, el.l.).

Merk at tilgang bestemmes utfra en bruker (med UID/GID), ikke ut fra en prosess (PID).

Man kan dele inn i generelle rettigheter (opprett eller slett objekt) og objekt-spesifikke rettigheter (sorter alfabetisk innholdet i directory).

Figuren viser inndeling i kernel space og user space for å poengtere at det er operativsystemet i kernel space som skal utføre aksess kontrollen.

*MEN MERK: aksesskontrollmekanismene som vi nå snakker om, altså det å kontrollere tilgangen fra et subjekt til et objekt, er en helt adskilt mekanisme fra usermode/kernelmode/protection rings.*

### ACLs with Groups

File	Access control list
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Som oftest har man både en bruker og en gruppe, og med gruppen innføres konseptet *rolle* (siden grupper som regel brukes til å representerer roller).

Grupper benyttes som regel ofte via wildcard:

tana,\*:RW

(dvs, i dette tilfelle er det det samme hvilken gruppe tana tilhører)

Alle utenom mysil skal ha tilgang:

mysil,:\*(none); \*,\*:RW

Entries (ACEer) i en ACL scannes i rekkefølge. DENY-entries står alltid først, og scanningen av listen avsluttes så fort en DENY-entry matcher. Hvis ikke så er tilfelle, scannes ALLOW-entriene inntil man har funnet ALLOW for alle rettigheter som søkes etter.

En annen måte å benytte grupper på er å bare angi enten bruker eller gruppe:

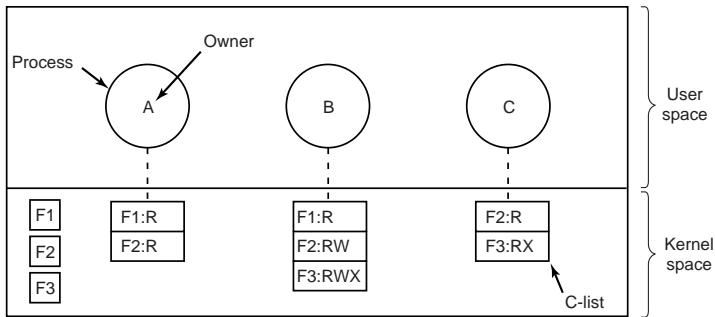
debbie:RW; phil:RW; pigfan:RW

DEMO:

```
echo mysil > a.txt
icacls a.txt
icacls /? # scroll til listen over hva entriene betyr
# fjern mine rettigheter:
icacls a.txt /deny WIN-2LTE3ENH06A\"Erik Hjelmås":` (F`)
echo mysil > a.txt
rm a.txt # hæ? hvorfor fikk jeg lov til det?
echo mysil > a.txt
icacls a.txt
#
# hvis jeg skal fjerne Inherited-rettigheter må jeg fjerne
# inheritance først:
icacls a.txt /inheritance:d
```

#### 14.2.4 Capabilities

##### Capabilities



Figuren viser tre prosesser med hver sin *capability list* som inneholder *capabilities*.

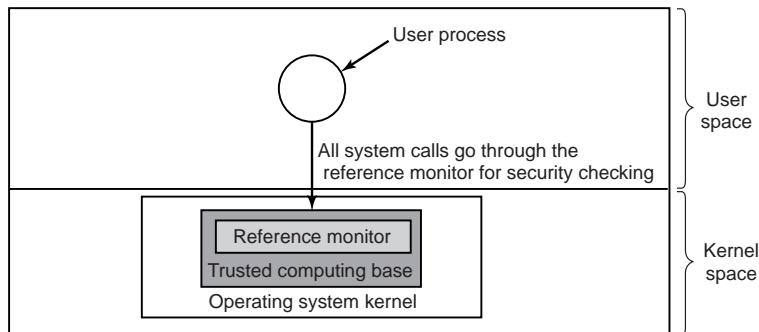
En capability består altså av en identifikator for filen (objektet) og et sett rettigheter.

Capabilities er mer effektive enn ACL fordi man slipper å søke gjennom en potensielt lang ACL liste, men et stort problem med capabilities er at det kan være tungvint å slette en fil (objekt) eller trekke tilbake rettighetene til en bestemt fil (objekt) siden det da må søkes gjennom alle subjekter (brukere).

Skumles 9.3.4 i Tanenbaum om utviklingen av email og www, *kompleksitet (for mange nye "features") er den største fienden til sikkerhet*.

#### 14.2.5 Reference monitor

##### A Reference Monitor



Det opprinnelige sikkerhetsarbeidet fra 60 og 70 tallet resulterte delvis i the Orange book:

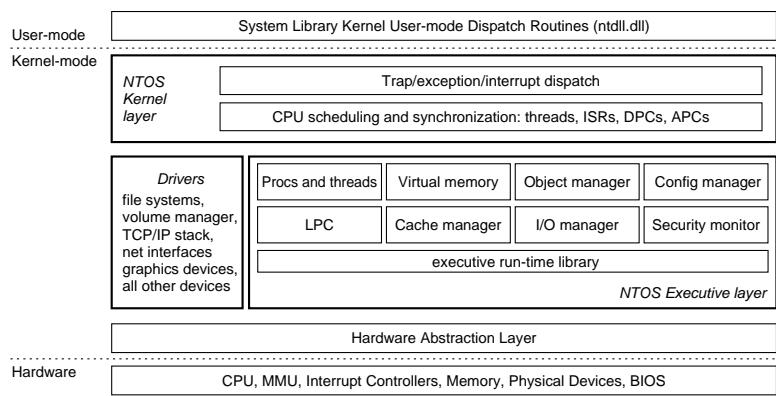
[TrustedComputerSystemEvaluationCriteria\(TCSEC\)](#)

I et trusted system skal det være en Reference monitor.

I Windows heter denne eksplisitt "Security Reference Monitor", men denne komponenten finnes i alle moderne operativsystemer som kontrollerer tilgang fra subjekt til objekt.

Samlingen av all hardware og software som er involvert i det å sørge for at sikkerhetssreglene (security policy) overholdes, kalles *trusted computing base*. Trusted computing base bør være så liten som mulig (minst mulig kompleksitet), slik at den er en grunnleggende sikkerhetsfunksjon som kan stoles på.

### Security Reference Monitor in Windows Vista



Figuren viser at Windows Vista har en egen del av operativsystemet som kalles Security Reference Monitor.

#### 14.2.6 Authorized States

##### Authorized States

Objects			
	Compiler	Mailbox 7	Secret
Eric	Read Execute		
Henry	Read Execute	Read Write	
Robert	Read Execute		Read Write

(a)

Objects			
	Compiler	Mailbox 7	Secret
Eric	Read Execute		
Henry	Read Execute	Read Write	
Robert	Read Execute	Read	Read Write

(b)

Beskyttelsesmatrisen kan betraktes som en tilstand for systemet. En *sikkerhetspolicy* kan da defineres som et sett med regler som skiller de sikre fra de usikre tilstandene.

Figuren viser et system som går fra en sikker til en usikker tilstand fordi bruker Robert har fått lesertilgang til Henry's mailboks.

### 14.2.7 Multilevel security

#### Multilevel Security

**Discretionary Access Control (DAC)** The users decide access control.

**Mandatory Access Control (MAC)** The system decide access control.

MAC er lite i bruk, men hadde stort fokus i det militære spesielt og finnes som opsjoner i mange operativsystemer (f.eks. som kjernemoduler på Linux).

#### Bell-La Padula & Biba

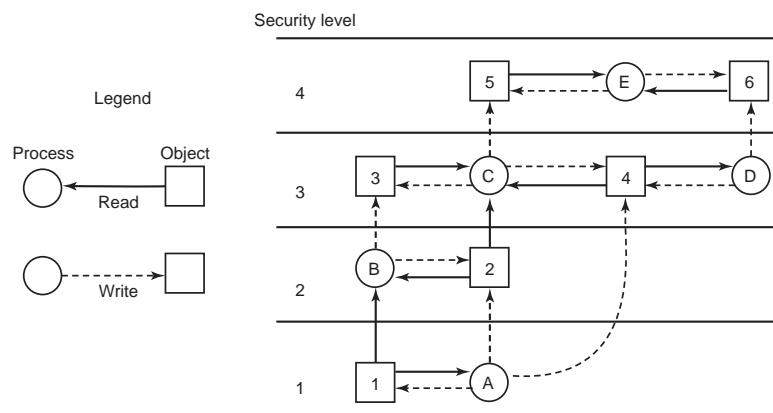
##### Bell-La Padula

**Simple security property (no-read-up)** A process running at security level  $k$  can read only objects at its level or lower.

**\* property (no-write-down)** A process running at security level  $k$  can write only objects at its level or higher.

En velkjent MAC policy er *Bell-La Padula* som sørger for å holde på hemmeligheter, og er rettet mot militære anvendelser. I denne modellen vil ikke informasjon kunne flyte nedover i systemet, dvs top secret info kan ikke skrives ned til en som ikke har top secret klarering.

##### Bell-La Padula



Når det modellers slik, kan vi se om Bell-La Padula er overholdt ved å sjekke at ingen piler peker nedover.

## Biba

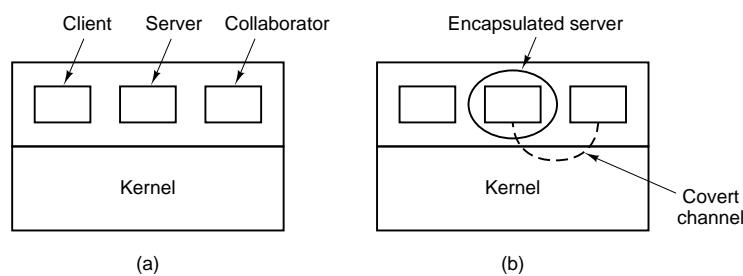
**Simple integrity property (no-write-up)** A process running at security level  $k$  can write only objects at its level or lower.

**Integrity \* property (no-read-down)** A process running at security level  $k$  can read only objects at its level or higher.

*Biba*-modellen sørger for at integritet ivaretas men er da det stikk motsatte av Bell-La Padula (!).

### 14.2.8 Covert channels

#### Covert Channel Model

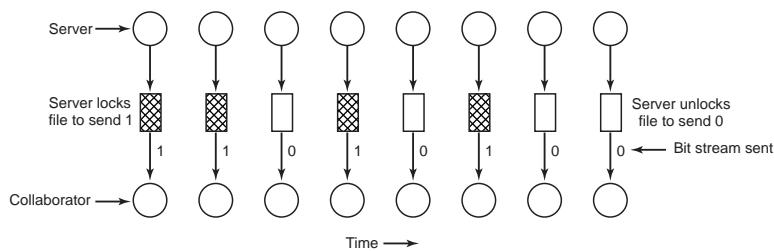


Covert Channel er altså en skjult kanal som kan lekke informasjon.

Hvordan kan vi unngå at serverprosessen gir bort informasjon til collaborator prosessen?

Vi kan hvertfall sørge for aksess kontroll og stenge for IPC osv.

#### Covert Channel Using File Locking



Men server prosessen kan alltid lekke informasjon ved alle mulige systemoppgaver som klokkes og som kan overvåkes av andre prosesser, eller ved systematisk fillåsing (som i figuren) som kan observeres av collaborator, eller sikert mange andre muligheter ...

Å finne (og unngå) alle covert channels er tilnærmet håpløst ...

## 14.3 Windows

### 14.3.1 Fundamentals

#### Access Token

Header	Expiration time	Groups	Default CACL	User SID	Group SID	Restricted SIDs	Privileges	Impersonation level	Integrity level
--------	-----------------	--------	--------------	----------	-----------	-----------------	------------	---------------------	-----------------

TAVLE:

En **SID** (Security ID) for hver bruker og gruppe, skal være globalt unike.

Mapping mellom SID og brukernavn i powershell: <http://www.microsoft.com/technet/scriptcenter/resources/pstips/feb08/pstip0201.mspx>

En prosess har en access token som vist i figuren med:

**Header** adm info.

**Expiration time** brukes ikke.

**Groups** gruppertilhørighet, benyttes av POSIX støtten (husk: primær gruppertilhørighet er et konsept om benyttes av POSIX men ikke av Windows).

**Default ACL** standard DACL (tilsvarer umask på linux) som settes på objekter prosessen oppretter hvis ikke annet angis spesielt (det finnes også en default gruppe-SID som angir hvilken gruppe som skal settes som eier av objektet).

**User SID** angir eier av prosessen.

**Group SID** angir gruppertilhørighet (SID til de gruppene som prosessen tilhører).

**Restricted SIDs** angi prosesser som kan delta i utførelsen med begrensede rettigheter.

**Privileges** Spesielle rettigheter knyttet til en bruker (*en aksesskontroll på oppgaver istedet for mot objekter*). Privileges er en måte å gi bort deler av rettigheten en administrator har (f.eks. shutdown av maskina, endre tidssone).

**Impersonation level** en access token kan brukes på vegne av noen andre (typisk hos en server på vegne av en klient), dette angis her.

**Integrity level** Low, Medium, High og System (muligens untrusted og trusted installer også), innført i Vista brukes som MAC (benyttes spesielt for å sette lav integritet på internet explorer slik at ondsinnet kode via IE ikke kan skrive over systemfiler).

Windows har altså *Privileges*:

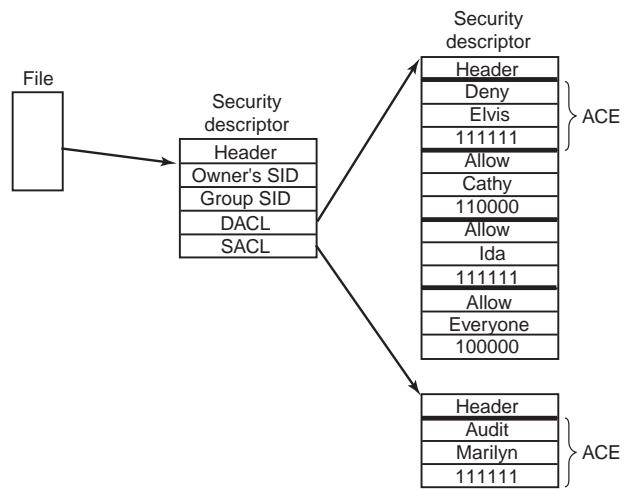
[http://msdn.microsoft.com/en-us/library/bb530716\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb530716(VS.85).aspx)

DEMO:

```
whoami /all
```

(merk at BUILTIN\Administrators gruppen har en litt rar beskrivelse, det kommer vi straks tilbake til ifb med UAC)

### Security Descriptor



En fil (objekt) har en security descriptor som angir eier, gruppelihørighet, og DACL (som scannes i rekkefølge ved aksess kontroll) og SACL som angir hva som skal logges spesielt for dette objektet. SACL angir også integritetsnivå som vi ser mer om snart.

Objekter som lagres i NTFS (filer og directories) har seks forskjellige rettigheter: read, write, execute, delete, change permissions, take ownership:

<http://www.pcguide.com/ref/hdd/file/ntfs/secPerm-c.html> NTFS rettighetene er ikke nødvendigvis lik rettighetene til andre typer objekter, la oss sammenlikne de med rettighetene knyttet til nøkler i registry:

((Get-Item filnavn).GetAccessControl()).Access og bytt ut filnavn med f.eks. HKLM:\SYSTEM\CurrentControlSet (HKLM er HKEY LOCAL MACHINE registrydelen som i powershell blir gjort tilgjengelig som stasjonen HKLM:\)

### 14.3.2 System calls

#### Principal Win32 API Functions for Security

Win32 API function	Description
InitializeSecurityDescriptor	Prepare a new security descriptor for use
LookupAccountSid	Look up the SID for a given user name
SetSecurityDescriptorOwner	Enter the owner SID in the security descriptor
SetSecurityDescriptorGroup	Enter a group SID in the security descriptor
InitializeAcl	Initialize a DACL or SACL
AddAccessAllowedAce	Add a new ACE to a DACL or SACL allowing access
AddAccessDeniedAce	Add a new ACE to a DACL or SACL denying access
DeleteAce	Remove an ACE from a DACL or SACL
SetSecurityDescriptorDacl	Attach a DACL to a security descriptor

### 14.3.3 Implementation

#### Login

1. CTRL-ALT-DEL (Secure Attention Sequence) initiate winlogon
2. Winlogon uses lsass to authenticate
3. User ends up with the GUI shell explorer process with an access token

CTRL-ALT-DEL (secure attention sequence) benyttes for å skrive at man logger på via winlogon prosessen, lsass bruker SECURITY og SAM kubene (hives) av registry for å sjekke pålogging og ved godkjent pålogging startes et grafisk shell explorer.exe med tilhørende access token.

All videre aksess kontrol (dvs hver gang en prosess forsøker bruke et objekt) gjøres av Security Reference Monitor som vist tidligere.

#### MIC

#### Mandatory Integrity Control

- Processes have an integrity level (low, medium, high, system) in their access token
- Objects have an integrity level in the SACL of their security descriptor
- *The Security Reference Monitor (SRM), before going to DACL, checks SACL and allows a process to write or delete an object only if its integrity level is greater than or equal to that of the object (no-write-up like Biba)*
- Processes cannot read process objects at a higher integrity level either (a limited no-read-up, a bit like Bell-LaPadula)

Også kalt "Windows Integrity Levels"

Hvert integritetsnivå har sin SID: Low (SID: S-1-16-4096), Medium (SID: S-1-16-8192), High (SID: S-1-16-12288), and System (SID: S-1-16-16384).

(Man kan ikke dynamisk endre integritetsnivå, dvs forsøk å endre integritetsnivå på internet explorer, og du må restarte prosessen for at endringen skal ta effekt)

Merk: Privileges og Integrity levels kan begge benyttes da til å overstyre ACL.

DEMO (jeg har lov i ACL, men blir overstyrt):

```
cd /Users/erikh
powershell
whoami /all          # jeg er på medium
Write-Output mysil > mysil.txt
exit
psexec -l powershell      # kjør powershell på lav
whoami /all          # jeg er på lav
Write-Output solan > solan.txt # ikke lov fordi:
accesschk -d -v .       # min homedir er på medium
```

## UAC

**User Account Control (UAC)** The problem: *Software developers assume their application will run as administrators on Windows.* UAC tries to promote change:

- All admin accounts are launched with standard user privileges
  - Membership in admin group marked DENY
  - Privilege set reduced to standard user set
- File system and registry namespace virtualization used for legacy application

God artikkel som forklarer dette i detalj:

<http://blogs.technet.com/markrussinovich/archive/2007/02/12/638372.aspx>

Når man logger på og startet prosesser som administrator, startes disse med en aksess token som har begrensede rettigheter. Prosessen kan få økte rettigheter via "Run as administrator" eller ved at det er kodet i applikasjon ("trustinfo" -tag som sier noe om "requestExecutionLevel" i "application manifest").

Gammeldags applikasjoner som antar at de har admin-rettigheter og ikke sier noe om "elevation" trenger file system og registry namespace virtualisering for å fungere skikkelig med begrensede rettigheter.

DEMO: - whoami /all og accesschk -f -p powershell i powershell startet som adm og uten, sammenlikne like SID (merk forskjellen på BUILTIN\Administrators) og privilege lista.

```
taskmgr
cd c:\windows
Write-Output tiger > woods.txt (access denied)
# høyreklikk i taskmgr, UAC virtualization på powershell.exe
Write-Output tiger > woods.txt
Get-Content woods.txt
# UAC virtualization på cmd.exe OFF
Get-Content woods.txt
cd $env:LocalAppData
cd VirtualStore\Windows
Get-Content woods.txt
```

## 14.4 Linux

### 14.4.1 Fundamentals

#### File Permissions

Binary	Symbolic	Allowed file accesses
111000000	rwx-----	Owner can read, write, and execute
111111000	rwxrwx---	Owner and group can read, write, and execute
110100000	rw-r-----	Owner can read and write; group can read
110100100	rw-r--r--	Owner can read and write; all others can read
111101101	rwxr-xr-x	Owner can do everything, rest can read and execute
000000000	-----	Nobody has any access
000000111	-----rwx	Only outsiders have access (strange, but legal)

Samme for filer og directories men for directories betyr x bit'n søk i stedet for execute.

I tillegg har vi SetUID, SetGID og sticky bit.

SetUID er mye benyttes f.eks. passwd som må kunne skrive til /etc/passwd men det har egentlig bare root lov til. Via disse innføres konseptet *effective UID*.

Sticky bit benyttes for å ha felles kataloger hvor alle kan skrive til, men man ikke kan slette hverandres filer (f.eks. /tmp).

Merk: *I Linux er dette altså en fast ACL med tre entries*. Det er støtte for POSIX ACL i Linux også men ikke så mye brukt foreløpig (dvs den er nok nok mye brukt i litt større sammenheng, men ikke på personlige PCr typisk).

### 14.4.2 System calls

#### Security System Calls

System call	Description
s = chmod(path, mode)	Change a file's protection mode
s = access(path, mode)	Check access using the real UID and GID
uid = getuid( )	Get the real UID
uid = geteuid( )	Get the effective UID
gid = getgid( )	Get the real GID
gid = getegid( )	Get the effective GID
s = chown(path, owner, group)	Change owner and group
s = setuid(uid)	Set the UID
s = setgid(gid)	Set the GID

`chmod` mest benyttet.

De tre siste kan bare root utføre.

### 14.4.3 Implementation

#### Implementation

1. Login checks username/password and groups
  - `/etc/passwd`, `/etc/shadow`
  - `/etc/groups`

2. Starts shell with users UID, GID

3. `sudo` similar to UAC

Linux har altså en mye enklere sikkerhetsmodell enn Windows i utgangspunktet, men det er fullt mulig å utvide linux med sikkerhetsmoduler i kjernen som skaper mere avanserte sikkerhetsmodeller (søk gjerne på SELinux).

## 14.5 Theory questions

1. Nevn fordeler og ulemper med aksesskontroll-lister i forhold til capability lister.
2. Forklar kort hva *DAC* (Discretionary Access Control) og *MAC* (Mandatory Access Control). Gi eksempler.
3. Forklar kort innholdet i minst fem av feltene i en *Windows Access Token*.
4. Forklar kort hvordan *User Account Control* på Windows fungerer.
5. Forklar kort forskjellen på filrettigheter i Unix/Linux og ACL'er knyttet til filer i Windows.
6. Gitt følgende seanse i Bash-kommmandolinje:

```
$ ls -l mypw
----- 1 root root 129824 mai 11 10:16 mypw
$ XXXXXXXXXXXXXXXX
$ ls -l mypw
-rwsr-xr-x 1 root root 129824 mai 11 10:16 mypw
```

Hvilken kommando har blitt gitt der det står 'XXXXXXXXXXXXXX'? Begrunn svaret.

## 14.6 Lab exercises

Utfør denne lab'n både i Linux/Bash og i Windows/PowerShell.

(Hint: net user, net localgroup, icacls i powershell (husk icacls /?), useradd, addgroup, chmod og chown i bash (bruk man)).

1. Du er systemadministrator for en felles filserver som benyttes av studenter og ansatte. Du skal lage brukerne ola, eli, ina og ali, samt gruppene studenter med ola og eli, og gruppen ansatte med ina og ali. Anta at det er mange flere brukere i disse gruppene enn de som nevnes her. Gjør dette ved å skrive et script (PowerShell (gjerne med cmdlet Import-CSV) for Windows og Bash for Linux) som utfører oppgaven ved å lese inndata fra filen

<http://www.ansatt.hig.no/erikh/opsys/users.csv>

(Hint: bruk

useradd -p KRYPTERTPASSORD -G GRUPPE BRUKERNAVN

for å opprette en bruker på Linux, og for å lage det krypterte passordet kan du bruke

openssl passwd -1 PASSORD )

(NB! hva slags sikkerhetsproblem kan du ha hvis en annen bruker lister ut alle detaljer om alle kjørende prosesser/tråder mens du gjør kommandoene over?)

2. Lag directory /emner med følgende rettigheter:

- ina har alle rettigheter
- studenter har kun lese (dvs kan liste opp innholdet i directorien) rettigheter
- ingen andre har noen rettigheter

3. Lag directory /prosjekt med følgende rettigheter.

- ina har alle rettigheter
- studenter har alle rettigheter utenom å slette
- ingen andre har noen rettigheter

4. Lag directory /parlament med følgende rettigheter.

- ola og eli har alle rettigheter
- alle studenter har lese rettigheter
- ali har lese rettigheter
- ingen andre har noen rettigheter

5. Hvilke av oppgavene ovenfor fikk du ikke til på Linux/Bash (med tradisjonell Unix/Linux sikkerhetsmodell)?

6. Benyttes fil-eier aktivt i aksesskontrollen på samme måte i Unix/Linux og i Windows?



# Chapter 15

## Malware og minnesikkerhet

### 15.1 Insider Attacks

#### Insider Attack: Trap Doors /Backdoors

```
while (TRUE) {  
    printf("login: ");  
    get_string(name);  
    disable_echoing();  
    printf("password: ");  
    get_string(password);  
    enable_echoing();  
    v = check_validity(name, password);  
    if (v) break;  
}  
execute_shell(name);  
  
(a)
```

```
while (TRUE) {  
    printf("login: ");  
    get_string(name);  
    disable_echoing();  
    printf("password: ");  
    get_string(password);  
    enable_echoing();  
    v = check_validity(name, password);  
    if (v || strcmp(name, "zzzzz") == 0) break;  
}  
execute_shell(name);  
  
(b)
```

Bakdør, her vist hvordan en bruker med brukernavn zzzzz alltid vil slippe gjennom autentiseringsprosessen.

TAVLE:

- Logisk bombe
- Tidsbombe
- Bakdør
- Login spoofing

(Punchline idag er å se litt på alle mulige måter sikkerhet kan forbigås)

### Insider Attack: Login Spoofing



Hvem som helst bruker kan jo skrive et program som kjører i full skjerm og ser ut som et login program, som ved forsøk på innlogging dreper brukerns kjørende shell og sender systemet tilbake til opprinnelig innloggings vindu.

Eneste måte å unngå dette på er å starte login med en tastekombinasjon som er brukerprogram ikke kan fange, CTRL-ALT-DEL er en slik kombinasjon.

## 15.2 Exploiting Code Bugs

### General Attack Script

1. Run an automated port scan to find machines that accept telnet connections.
2. Try to log in by guessing login name and password combinations.
3. Once in, run the flawed program with input that triggers the bug.
4. If the buggy program is SetUID root, create a SetUID root shell.
5. Fetch and start a zombie program that listens to an IP port for commands.
6. Arrange that the zombie program is always started when the system reboots.

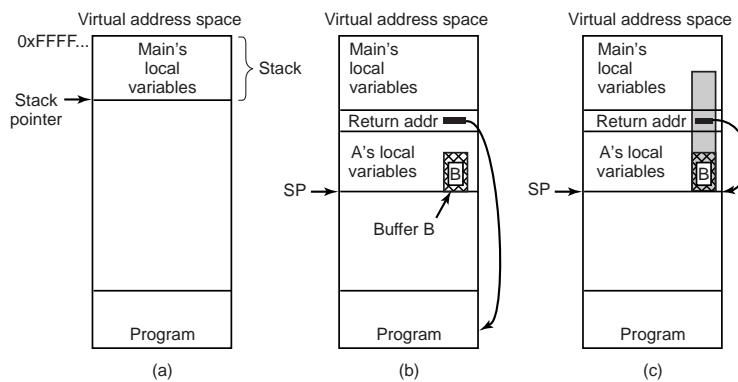
Slike rene Attack scripts som vist her er ikke så vanlige lenger, det er mer fokus på brukeren og lure brukeren til å kjøre malware, men som oftest er det en kombinasjon, dvs lure brukeren til å besøke et nettsted som inneholder kode som utnytter en svakhet i nettleseren.

Vi si at vi har en *ondsinnet hacker* eller *motstander (adversary)* generelt som ønsker å få kjøre kode på vår datamaskin med mest mulig rettigheter, ved å:

- få oss til å kjøre kode ("klikk på vedlegg")
- få oss til å besøke et nettsted som utnytter en sårbarhet i nettleseren vår og kode kjøres uten at vi er klar over det
- helautomatisk angrep på datamaskina vår uten at vi gjør noe selv, slik som i eksempelet "General attack script"

### 15.2.1 Buffer Overflow

#### Buffer Overflow Attacks



Veldig mange angrep skyldes dårlig koding i programmeringsspråket C. C-kompilatorer sjekker ikke array-grenser, og det er derfor lett å skrive over andre deler av minne enn det man hadde tenkt.

Figuren viser:

(a) et vanlig program

(b) en funksjon er blitt kallet og alllokert plass til en lokal variabel B i funksjonen, B er f.eks. et buffer på 256 char for å holde et filnavn.

(c) brukern angir filnavnet med 1500 tegn, hvorav filnavnet er meget spesielt, det inneholder i starten ondsinnet kode og eksakt så langt ut i filnavnet at B og evn andre lokal variable samt EBP (forrige stack base pointer) er passert, så inneholder den returaddresse til starten av den ondsinnede koden som overskriver den opprinnelige returaddressen. Mao et buffer overskrives slik at man tar kontroll på returaddressen og da altså endrer programflyten (istedet for å returnere til main hvor funksjonen ble kallet fra returnerer man til den ondsinnede koden).

Dette er spesielt ille hvis programmet som har denne feilen er SetUID root (dette er det klassiske unix exploit fra 80 og 90 tallet) på Linux eller hvis brukeren kjører med administrator rettigheter på Windows (husk mekanismene MIC og UAC på Windows fra forrige tema).

Hvordan beskytte seg mot dette???

-> gjøre stacken ikke-eksekverbar (vanlig idag, hardware støttet på INTEL via NX bit'et).

I tillegg advarer de fleste kompilatorer deg mot bruk av de funksjonene som typisk fører til buffer overflow muligheter (gets er kanskje den mest klassiske). Noen kompilatorer legger også på beskyttelsesmekanismer (`man gcc` og søk etter `stack-protector`).

Det beste er hvis man kan unngå å bruke de funksjonene i C/C++ som kan føre til buffer overflow, en god oversikt finnes på "Security Development Lifecycle (SDL) Banned Function Calls": <http://msdn.microsoft.com/en-us/library/bb288454.aspx>

### 15.2.2 Format Strings

#### Format String Attacks

- printf, fprintf, snprintf, ...
- syslog, ...
- printf("%.50d%n", 1, &i) prints 1 but sets i=50 !
- what if the format string is supplied by the user and the user also knows about %m\$ (man 3 printf).

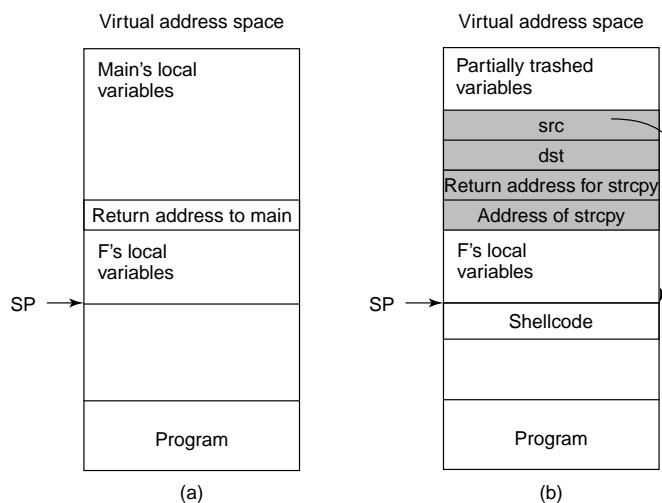
En 'Format string' (formatteringsstreng) er altså en char string som inneholder formateringsinformasjon, dvs hvordan den skal se ut når den skrives ut. Formateringsinformasjonen angis som spesialtegnsekvenser (f.eks. %.2f som sier 'rund av til to desimaler etter komma').

Hvis en bruker har anledning til å skrive inn en slik format string direkte, så kan vedkommende misbruke disse siden en av dem (%n) kan brukes til å skrive til minne, og med %m\$ kan man skrive til variabel nummer m.

Og med ytterligere finurlighet kan de kontrolleres hvor i minne man skriver, og det er åpnet for angrep liknende buffer overflow som vi nettopp så på.

### 15.2.3 Return to Libc

#### Return to Libc Attacks



```
# Følg Return_to_libc.pdf og gjør task 1
# vær i dir med findshell.c, retlib.c, exploit_1.c
$ su
$ /sbin/sysctl -w kernel.randomize_va_space=0
$ gcc -o retlib retlib.c
$ ls -l retlib # evn chown root:root retlib
$ chmod 4755 retlib # gjør det sårbare programmet SetUID root
$ rm /bin/sh
$ ln -s /bin/zsh /bin/sh # bruker zsh istedet for bash, nødvendig?
$ ls -l /bin/sh
$ exit
$ export MYSHELL="/bin/sh" # få "/bin/sh" string i minne et sted
$ gcc -o findshell findshell.c
$ ./findshell # sjekk hvor "/bin/sh" er
bfffff74
$ gdb retlib # sjekk hvor libc system og exit er
b main
r
p system
0xb893df
p exit
0xb7eba4
quit
$ echo -n 123456789012 > badfile
$ ./retlib # sjekk at leser ok uten overflow
$ echo -n 1234567890123 > badfile
$ ./retlib # her ble badfile* delvis overskrevet: fclose segfaulter
$ echo -n 1234567890123456 > badfile
$ ./retlib # samme
$ echo -n 12345678901234567 > badfile
$ ./retlib # og her begynner vi skrive over returadr til main...
$ vi exploit.c # og sett inn i linjene etter *(long *) &buf
[24]=0xbfffff7a; // "/bin/sh"
[16]=0x00b893df; // system()
[20]=0x00b7eba4; // exit()
$ gcc -o exploit_1 exploit_1.c
$ ./exploit_1 # lager badfile
$ ./retlib
# ooops funker ikke, er "/bin/sh" på rett sted?
$ gdb retlib
b main
r
x/s 0xbfffff74 # x=examine memory, s=string
```

```
x/s 0xbfffff77
quit
$ vi exploit_1.c
$ gcc -o exploit_1 exploit_1.c
$ ./exploit_1 # lager badfile
$ ./retlib
# ooops funker fortsatt ikke, vi må jukse...
$ # sett inn findshell koden i retlib.c
```

### 15.2.4 Integer Overflow

#### Integer Overflow Attacks

1. Multiplying integers can easily overflow ('wrap-around').
2. Typical attack can be providing specific height and width for an image to be opened:
  - height × width results in overflow and allocation of not enough memory.
  - image overwrites memory and of course *the user decides the content of the image*
  - ...

som den står.

### 15.2.5 Code Injection

#### Code Injection Attacks

```
int main(int argc, char *argv[])
{
    char src[100], dst[100], cmd[205] = "cp ";
    /* declare 3 strings */
    /* ask for source file */
    gets(src);
    /* get input from the keyboard */
    strcat(cmd, src);
    /* concatenate src after cp */
    strcat(cmd, " ");
    /* add a space to the end of cmd */
    printf("Please enter name of destination file: ");
    /* ask for output file name */
    gets(dst);
    /* get input from the keyboard */
    strcat(cmd, dst);
    /* complete the commands string */
    system(cmd);
    /* execute the cp command */
}
```

Innlysende som den står, hvis en bruker kan fylle inn det han/hun vil inn i et kall til system så kan vilkårlig kode kjøres.

Nøkkelen er at flere shell kommandoer kan kjøres med ; imellom.

### 15.2.6 Privilege Escalation

**Privilege Escalation Attacks** Example:

1. Program set CWD to cron daemons dir.
2. Program then crash to force core dump, because core dumps happen in CWD *by the system*.
3. This way access control on the cron daemons dir were bypassed and the memory image of the program was structured to be a valid set of commands to cron daemon who would execute them as root.

som den står.

## 15.3 Malware

### Malware

1. Trojan horses
2. Viruses
3. Worms
4. Spyware
5. Rootkits

Andre relevante begreper:

TAVLE:

**Botnet** Robot Network (zombie backdoors)

**Keylogger** Periodisk rapportering eller søk etter bestemte data (kredittkortnr)

**ID tyveri** Misbruk av IDn din selv om ingen grove sikkerhetsbrudd har skjedd, fødselsnummeret må ALDRI betraktes som en hemmelighet men bør samtidig beskyttes!

Ift kredittkort har selvfølgelig kredittkortfirmaene avviksprogramvare, men den prøver jo de ondsinnede hackerne og holde seg akkurat under terskelen for, det er alltid en kamp om hvem som ligger foran ...

Generelt: mangfold er ofte en god ting for sikkerhet (single-sign on kan være skummelt selv om det er bra).

Og igjen: husk brukervennlighet vs. sikkerhet dilemma: hvem har passord for å ringe fra mobilen sin? Dette hadde økt sikkerheten ikke sant?

### 15.3.1 Trojan Horses

#### Trojan Horses



En trojansk hest er altså noe som er noe annet enn det utgir seg for å være, eller som gjør noe annet enn vi tror det skal gjøre.

En trojansk hest er altså ikke noe programvare som bryter seg inn på maskinen din, det er noe brukeren selv installerer ... (fordi den følger med noe annet, f.eks. skjermbeskyttere)

DEMO: bruk av PATH environment variabelen for å kjøre et annet program enn det jeg tror jeg kjører...

```
echo $PATH # ser at ~/bin er før de andre bin
which ls
ls
cat /tmp/skummelt
cp ls ~/bin
ls
cat /tmp/skummelt
rm ~/bin/ls
```

Alternativt: lag et skummelt program som har et navn som tilsvarer en vanlig feiltast-ing, f.eks. ls-ltr

```
ls-ltr  
cat /tmp/skummelt  
cp ls-ltr ~/bin  
ls-ltr  
cat /tmp/skummelt  
rm ~/bin/ls-ltr
```

### 15.3.2 Viruses

#### Viruses

1. (Companion viruses)
2. Overwriting viruses
3. Parasitic viruses
4. Memory-resident viruses
5. Boot sector viruses
6. Device driver viruses
7. Macro viruses
8. Source code viruses

Et virus reproducerer seg selv.

**(Companion Virus)** ala trojaner eksempelet, men prog av samme navn, f.eks. prog.com og prog.exe (kjører disse i sekvens så brukern ikke er klar over at prog.com ble kjørt først).

**Overwriting virus** skriver over programmer.

**Parasitic virus** hekter seg på programmer, men lar sitt host-program ellers fungere normalt.

**Memory-resident virus** befinner seg i minne til enhver tid.

**Boot sector virus** befinner seg typisk i MBR eller et annet sted knyttet til bootloader for å laste seg selv sammen med OS'et.

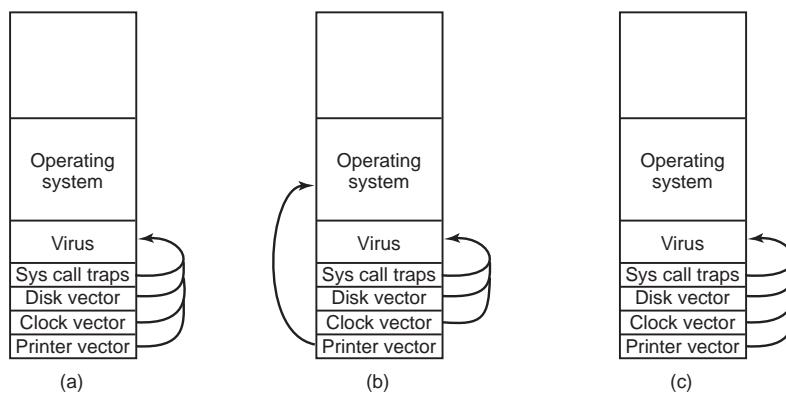
**Device driver virus** device drivere er jo vanligvis bare program de også, dermed kan virus hekte seg på de og lastes gjerne i kernel mode.

**Macro virus** befinner seg typisk i office dokumenter knyttet til forskjellige funksjoner (typisk ikke binær kode men interpretert kode).

**Source code virus** hekter seg inn som en typisk to linjer (en header og et funksjonskall) i kildekoden til programmer før de kompileres.

Husk sammenheng med aksess kontroll på objekter som vi pratet om sist: kjører du som regel med admin/root rettigheter til enhver tid har du store problemer om du får virus...

### Viruses



Et virus som legger seg i minne (og gjerne da i minne sammen med OS'et og interrupt vektorene i resident minne, dvs som ikke er gjenstand for paging) klarer å beholde kontroll som regel på følgende vis:

(a) ved oppstart skriver viruset over alle interrupt vektorer slik at de peker på viruskoden istedet for interrupthandler

(b) og (c) når device drivere lastes skrives interrupt vektorene over igjen (slik som for printer interrupt vektoren i (b)), MEN nøkkelen er at de gjør de ikke alle samtidig, dvs det vil alltid forekomme bruk av en eller flere interrupt vektorer (spesielt klokkeinterrupt) som viruset kontrollerer før alle er overskrevet, dermed vil viruset kunne overskrive igjen alle de som er blitt overskrevet av OS'et og gjenta full kontroll som i (c).

Virus ønsker å få kjøre for hvert eneste systemkall fordi da kan det dytte seg selv inn i minne på alle prosesser som gjør exec.

### 15.3.3 Worms

#### Worms

1. A *worm* replicates itself over the network.

Prat litt om internetormen, og vis

<http://www.ee.ryerson.ca/~elf/hack/iworm.html>

som fører til

<http://pdos.csail.mit.edu/~rtm/>



#### 15.3.4 Spyware

##### Spyware

1. Spyware is software that without the users consent leaks private information.

Brukes oftest for å rapportere data nyttig for markedsføringsformål.

Installeres ofte via toolbars, activeX kontroller eller pop-ups (hvor brukeren klikker 'No' istedet for å bare lukke vinduet).

Spyware bruker ofte DLL injection metodikk:

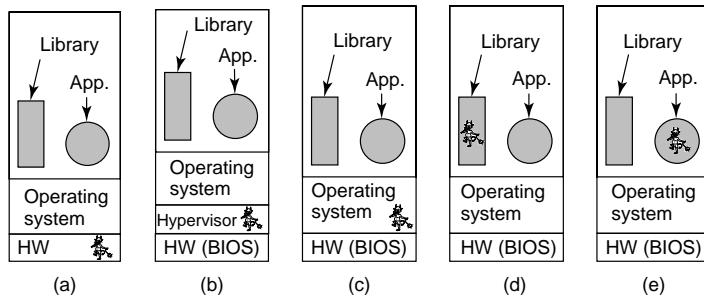
[http://en.wikipedia.org/wiki/DLL\\_injection](http://en.wikipedia.org/wiki/DLL_injection)

Nyttig innovativ norsk antispyware teknologi:

<http://www.promon.no>

### 15.3.5 Rootkits

#### Rootkits



TAVLE:

- Firmware rootkits
- Hypervisor rootkits
- Kernel rootkits
- Library rootkits
- Application rootkits

Generelt skiller rootkits seg fra trojanske hester med to hovedpoeng:

1. Rootkits er noe som installeres etter et angrep er kjørt mens en trojaner kan selv være et angrep.
2. Rootkits består som regel av mange filer og systemendringer og er langt mer komplisert enn en enkelt trojaner.

DEMO:

`sudo chkrootkit` som viser sjekk av typisk manipulerte filer og sjekk etter signaturer på kjente rootkits.

Som med all malware: den tryggeste måten er å boote fra et sikkert medium og scanne disk/partisjon.

When the news broke, Sony's initial reaction was that it had every right to protect its intellectual property. In an interview on National Public Radio, Thomas Hesse, the president of Sony BMG's global digital business, said: "Most people, I think, don't even know what a rootkit is, so why should they care about it?" When this response itself provoked a firestorm, Sony backtracked and released a patch that removed the cloaking of \$sys\$ files but kept the rootkit in place. Under increasing pressure, Sony eventually released an uninstaller on its Website, but to get it, users had to provide an e-mail address, and agree that Sony could send them promotional material in the future (what most people call spam).

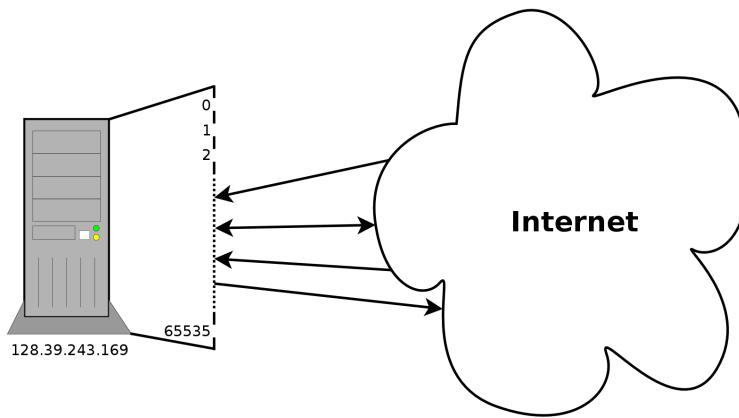
As the story continued to play out, it emerged that Sony's uninstaller contained technical flaws that made the infected computer highly vulnerable to attacks over the Internet. It was also revealed that the rootkit contained code from open source projects in violation of their copyrights (which permitted free use of the software *provided that the source code is released*).

In addition to an unparalleled public relations disaster, Sony also faced legal jeopardy. The state of Texas sued Sony for violating its antispyware law as well as for violating its deceptive trade practices law (because the rootkit was installed even if the license was declined). Class-action suits were later filed in 39 states. In December 2006, these suits were settled when Sony agreed to pay \$4.25 million, to stop including the rootkit on future CDs, and to give each victim the right to download three albums from a limited music catalog. On January 2007, Sony admitted that its software also secretly monitored users' listening habits and reported them back to Sony, in violation of U.S. law. In a settlement with the FTC, Sony agreed to pay people whose computers were damaged by its software compensation of \$150.

## 15.4 Defenses

### 15.4.1 Firewalls

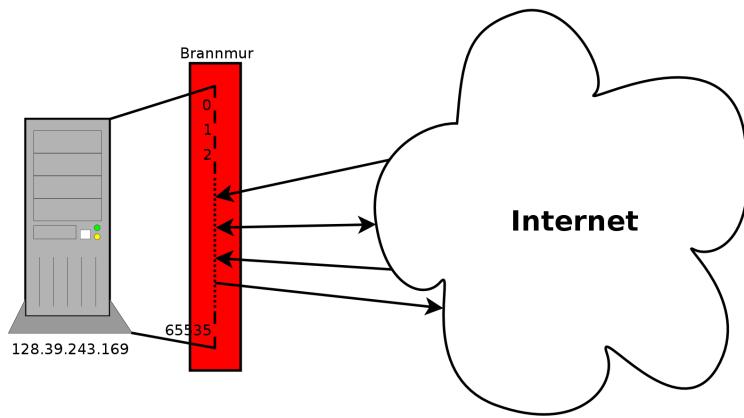
#### Without Firewall



Trafikk fra internett ankommer på porter som identifiseres med et 16-bits portnr (i tillegg til IP adresse selvfølgelig, og skille mellom protokoll TCP, UDP, og evn andre).

Generelt om beskyttelse: *Defense in depth*: Løkmodell - flest mulig lag med sikkerhet.

### With Firewall

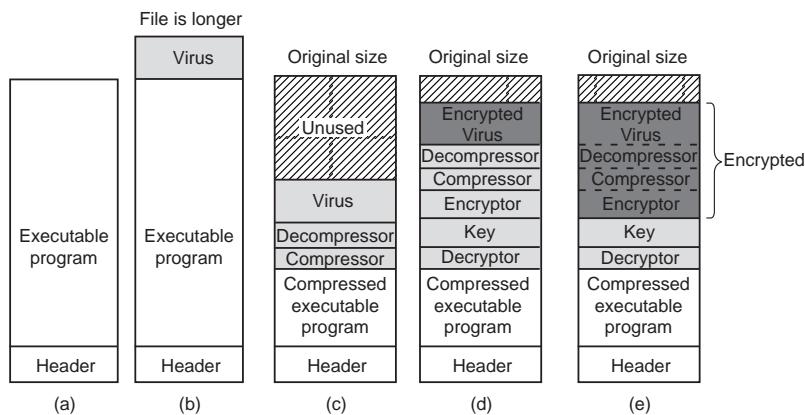


En brannmur er altså et sett med ACLer (aksess kontroll lister) som legger seg over disse portnummerne. Tenk på hver port som en fil som har hver sin aksess kontroll liste og subjektene som skal ha tilgang er identifisert ved IP, portnr, protokoll, o.l.

Vi tar med brannmur her fordi det er en av operativsystemets viktigste beskyttelsesmekanismer, selv om det er en mekanisme som ikke tilhører operativsystemet.

#### 15.4.2 Antivirus

##### How Viruses Hide



(b) og (c) et virus gjør jo filen større så host-programmet må komprimeres for å skjule at viruset er blitt en del av det, og komprimerings og dekomprimeringskoden må lagres også.

(d) for å skjule viruset må viruset krypteres med en ny nøkkel hver gang, random nøkkel kan f.eks. genereres ved XOR mellom tiden, og minneordene på addressene 34622 og 129836 (dvs to tilfeldige minnelokasjoner).

Et virus kan gjøre mye for å gjemme seg, men til slutt i (e) vil i denne figuren vil dekrypteringsrutinen fortsatt være synlig og antivirus programvaren kan lete etter denne.

### Polymorphic Virus

MOV A,R1 ADD B,R1 ADD C,R1 SUB #4,R1 MOV R1,X	MOV A,R1 NOP ADD B,R1 NOP ADD C,R1 NOP SUB #4,R1 NOP MOV R1,X	MOV A,R1 ADD #0,R1 ADD B,R1 OR R1,R1 ADD C,R1 SHL #0,R1 SUB #4,R1 JMP .+1 MOV R1,X	MOV A,R1 OR R1,R1 ADD B,R1 MOV R1,R5 ADD C,R1 SHL R1,0 SUB #4,R1 ADD R5,R5 MOV R1,X MOV R5,Y	MOV A,R1 TST R1 ADD C,R1 MOV R1,R5 ADD B,R1 CMP R2,R5 SUB #4,R1 JMP .+1 MOV R1,X MOV R5,Y
(a)	(b)	(c)	(d)	(e)

Fem varianter av samme kode, i (e) er R5 og Y bare avledningsmanøvre.

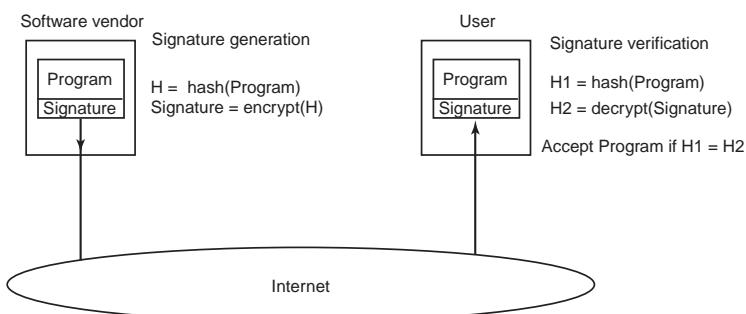
Et polymorfisk virus endrer seg selv (f.eks. sin dekrypteringsroutine) hver gang en ny fil infisieres. Typisk ved å sette inn NOP eller kode som gjør tilsvarende ingenting (dvs har ingen annen effekt enn å forvirre).

Måten disse endringene gjøres på bestemmes av en mutasjonsroutine (*mutation engine*) og denne kan gjemmes i den krypterte delen i (e) i forrige figur.

*Og dermed kan ingen fast signatur på noe av viruskoden finnes ...*

### 15.4.3 Code Signing

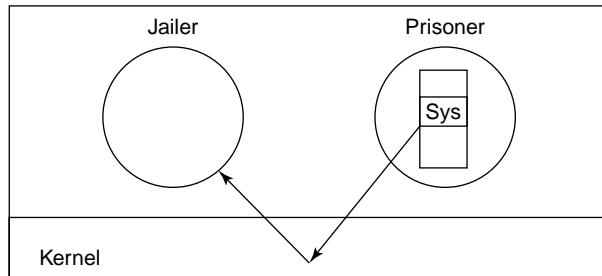
#### Code Signing



Rett frem ...

### 15.4.4 Jailing

#### Jailing



All oppførsel ved et program kan overvåkes med jailing, dvs hvert systemkall overføres til 'jailer' som bestemmer om de skal tillates, f.eks. åpne nettforbindelse til en ukjent IP.

Systemkallene overføres altså til 'jailer' istedet for å utføres av kernel direkte.

Det er også vanlig å "jaile" prosesser i sin egen del av filsystemet slik som godt forklart i "Create a simple chroot jail": <http://www.adminarticles.com/create-a-simple-chroot-jail/> Dette gjøres i stor grad på mobile enheter som installeres "app'er".

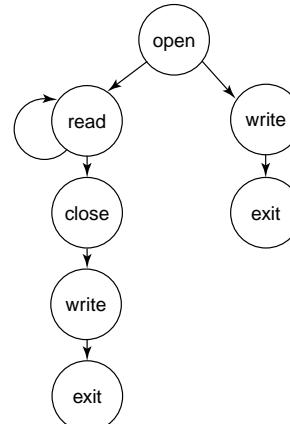
### 15.4.5 Host-based IDS

#### Model-based IDS

```
int main(int argc *char argv[])
{
    int fd, n = 0;
    char buf[1];

    fd = open("data", 0);
    if (fd < 0) {
        printf("Bad data file\n");
        exit(1);
    } else {
        while (1) {
            read(fd, buf, 1);
            if (buf[0] == 0) {
                close(fd);
                printf("n = %d\n", n);
                exit(0);
            }
            n = n + 1;
        }
    }
}
```

(a)



(b)

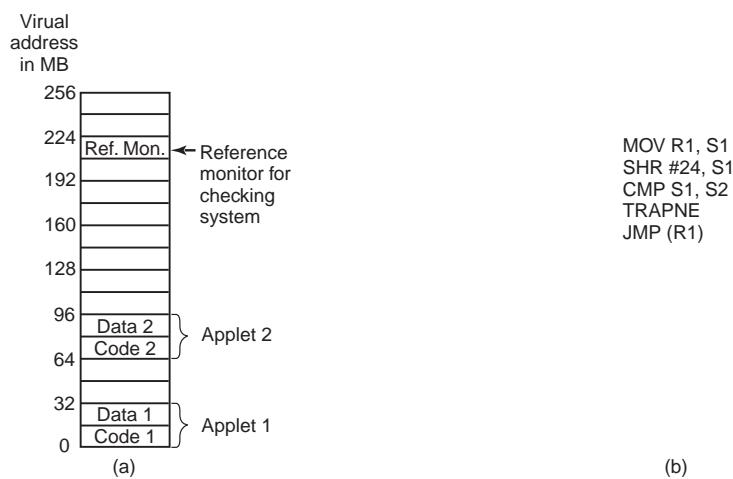
Finnes mange typer Host-based IDS, konseptuelt veldig likt antivirus (tenk på det som en generalisering av antivirus).

Model-based IDS er et eksempel hvor det sjekkes for avvik fra *systemkallgrafen* i (b). Systemkallgrafen er da på en måte et alternativ til en signatur.

Kan implementeres via jailing (fra forrige figur).

#### 15.4.6 Encapsulating Mobile Code

##### Sandboxing



Virtual memory kan deles opp i sandboxer hvor hver applet får en kode og en data sandbox som den må holde seg innenfor.

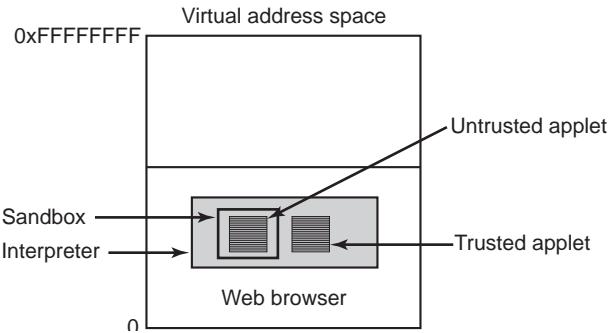
Ved å ha en egen kodesandbox kan denne merkes read-only slik at man unngår faren for selvmodifiserende kode.

I det en applet lastet relokaliseres alle minnereferanser til å være innenfor appletens sandboxer.

(b) viser hvordan 4 instruksjoner og 2 registre benyttes (settes inn) før hver dynamisk JMP instruksjon, for å sjekke om addresseringen er innenfor sandboxene (SHR er shift-right for å isolere prefix-bitene som er de som de skal sjekkes på, dvs alle minnereferanser innenfor samme sandbox vil ha lik higher-order bits prefix).

Alle forsøk på systemkall omdirigeres til en egen Reference monitor.

##### Interpretation



Alternativt kan kode også tolkes, dvs den mobile koden er ikke kompilert til direkte kjørbare kode men skal tolkes slik som f.eks. java applets gjerne funker, da er det enda letteres å sjekke koden før kjøring.

I figuren vises en trusted applet (f.eks. som er verifisert via kodesignatur) som har lov til å gjøre systemkall, og en untrusted som må kjøre i en sandbox.

## 15.5 Human in the Loop

### Levels of Human Intervention

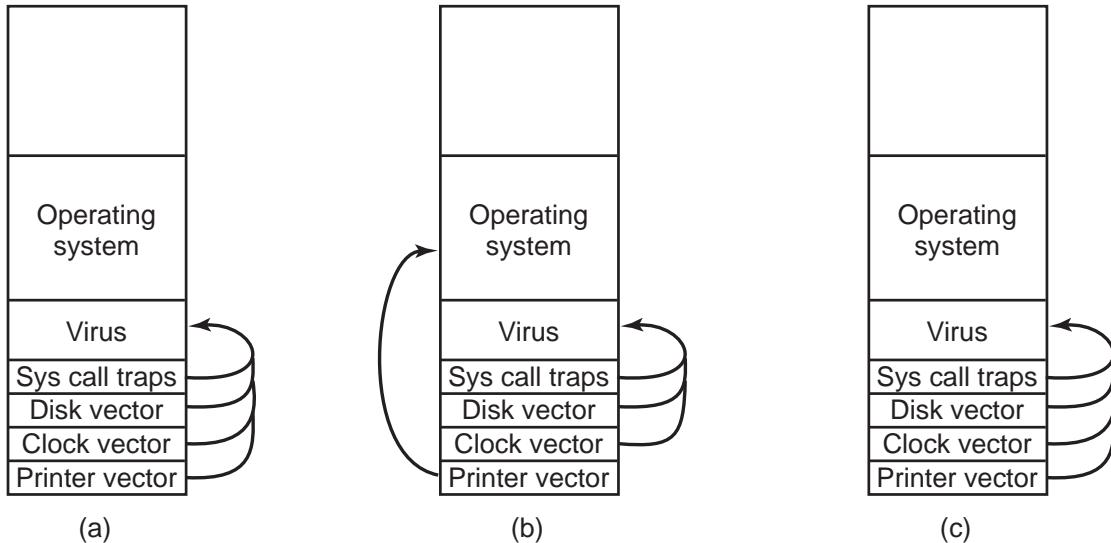
1. Fully automated attack script
2. Trick users into visiting a website with their buggy outdated browser
3. Trick users into double-clicking on attachments

Det er forskjellige nivåer med brukerinteraksjon påkrevet i angrepene. Husk at vi kan sikkert klare å sikre oss godt mot automatiserte angrep, men vi må også hjelpe brukerne beskytte seg mot seg selv så godt de kan (f.eks. Windows Protected Mode Internet Explorer som bruker low integrity level). INGEN har noengang verifisert på forhånd absolutt alt de har klikka på...

Husk at prosesskonseptet ble innført for å skille mellom prosesser som tilhører forskjellige brukere, men det gir liten beskyttelse mellom to prosesser fra samme bruker siden de tillates og sende meldinger til hverandre. Men code signing, jailing, sandboxing sammen med integrity levels hjelper altså brukeren å beskytte seg mot seg selv.

## 15.6 Theory questions

1. Forklar kort og presist klassisk buffer overflow angrep og return-to-libc angrep.
2. Tanenbaum oppgave 9.27  
Name a C compiler feature that could eliminate a large number of security holes.  
Why is it not more widely implemented?
3. Tanenbaum oppgave 9.38  
What is the difference between a virus and a worm? How do they each reproduce?
4. Forklar kort hva som menes med et *integer overflow attack*.
5. Forklar hvordan et virus kan beholde kontrollen over alle interruptvektorene med utgangspunkt i denne figuren:



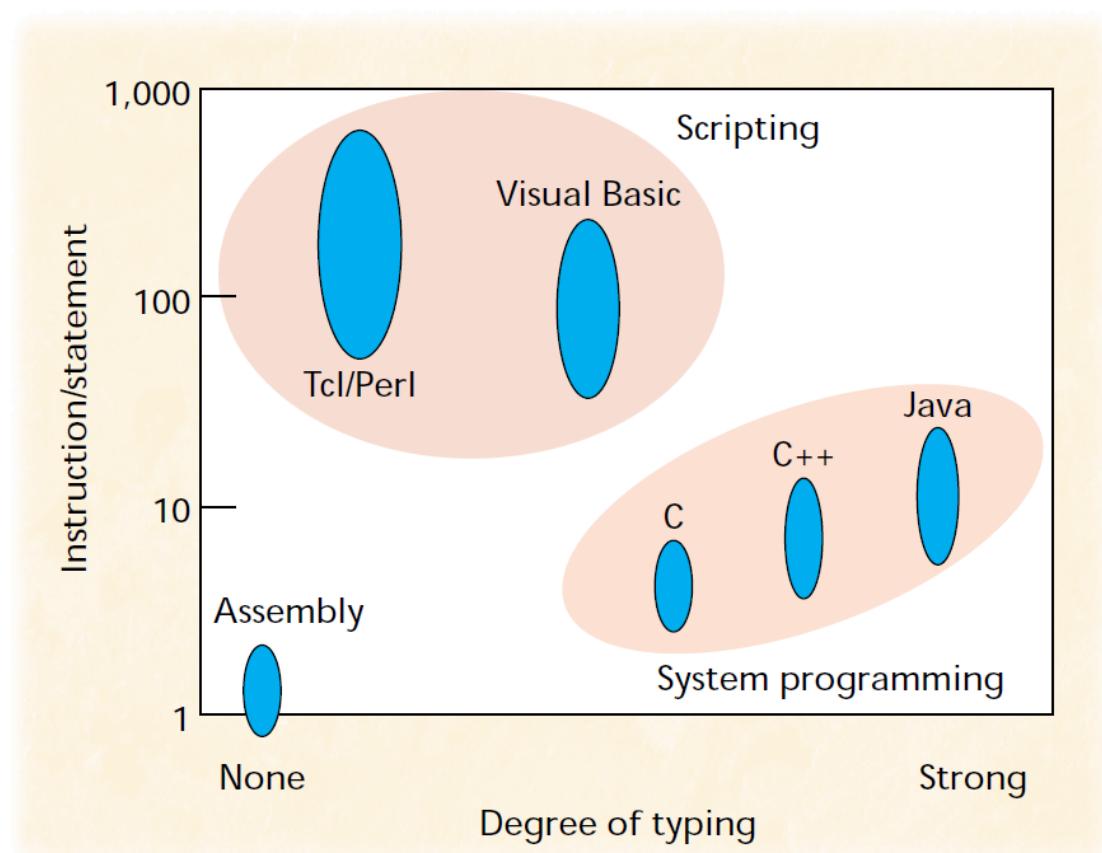
6. Hva er et *rootkit*? Hvilke fem steder kan et rootkit typisk gjemme seg?
7. Hva menes med *jailing*? Forklar kort hvordan jailing typisk fungerer.

## **15.7 Lab exercises**

1. Bare fortsett på forrige ukes oppgaver...

## Appendix A

### Bash



(OUSTERHOUT, J., "Scripting: Higher-Level Programming for the 21st Century", IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30.)

From Ousterhout, 1998:

While programming languages like C/C++ are designed for low-level construction of

data structures and algorithms, scripting languages are designed for high-level “gluing” of existing components. Components are created with low-level languages and glued together with scripting languages.

**WARNING!**

The following presentation is NOT meant to be a comprehensive/complete tour of the Bash language.

*The purpose is to get you started with some basic program constructions which you will recognize based on some-sort-of-programming-background.*

At the end of the presentation you will find pointers to more comprehensive material.

**Practice**

*You need a GNU/Linux distribution (e.g. Ubuntu) running on a physical or virtual machine with working access to the internet, and with wget installed.*

Log in and open a terminal window, download the examples as we go along with

```
wget http://www.hig.no/~erikh/tutorial-bash(FILENAME)
```

(or download all at once with filename bash-examples.tar)

You will find the FILENAME on the second line of each example. For each example do

1. Download

```
wget http://www.hig.no/~erikh/tutorial-bash(FILENAME)
```

2. View the code

```
cat FILENAME or less FILENAME
```

3. Execute the code

```
bash FILENAME
```

or (make it executable with chmod +x FILENAME)

```
./FILENAME
```

*It is easy to write bash scripts, but sometimes your scripts will behave strangely. This is due to the fact that there are many pitfalls in Bash. It is very easy to write statements that appear logical to you in your way of thinking programming, but due to the nature of a shell environment such as Bash, they will not produce the expected results. I strongly recommend that you quickly browse (and remember as a reference) the following excellent document: <http://mywiki.wooleedge.org/BashPitfalls>*

## Hello World

```
#!/bin/bash
# hello.bash

echo "Hello world!"
```

make executable and execute:

```
chmod +x hello.bash
./hello.bash
```

The SheBang/HashBang `#!` is treated as a comment by the interpreter, but it has a special meaning to the operating system's program loader (the code that is run when one of the `exec` system calls are executed) on Unix/Linux systems. The program loader will make sure this script is interpreted by the program listed after the `#!`

Since Unix/Linux does not use file endings for identifying the file, there is no reason for us to do so in scripts either. The OS knows from the SheBang/HashBang what interpreter to use. However during the development of a script it can be nice to know something about the file content based on the file ending, so it is common to use `.bash` or `.sh` endings. Since this tutorial is about the Bash language, including Bash-specific features that are not POSIX-compliant, we stick with `.bash` as file endings.

## A.1 Variables

### Single Variables

```
#!/bin/bash
# single-var.bash

firstname=Mysil
lastname=Bergsprekken
fullname="$firstname $lastname"
echo "Hello $fullname, may I call you $firstname?"
```

A single variable is not typed, it can be a number or a string.

Do not put spaces before or after `=` when assigning values to variables.

Scope: variables are global unless specified inside a block and starting with the keyword `local`.

(in general, use lower case variable names, upper case implies it's a SHELL/ENVIRONMENT variable)

## Single and Double Quotes

```
#!/bin/bash
# quotes.bash

name=Mysil
echo Hello $name
echo "Hello $name"
echo 'Hello $name'
```

Variables are expanded/interpolated inside double quotes, but not inside single quotes.  
We use double quotes when we have a string.

### A.1.1 Arrays

#### Arrays

Bash supports simple one-dimensional arrays

```
#!/bin/bash
# array.bash

os=('linux' 'windows')
os[2]='mac'
echo "${os[1]}" # print windows
echo "${os[@]}" # print array values
echo "${!os[@]}" # print array indices
echo "${#os[@]}" # length of array
```

Automatic expansion of arrays (automatic declaration and garbage collection). `os[2]='mac'` can also be written as `os+=('mac')`

#### Associative Arrays

```
#!/bin/bash
# assoc-array.bash

declare -A user          # must be declared
user=( \
    [frodeh]="Frode Haug" \
    [ivarm]="Ivar Moe" \
)
user[lailas]="Laila Skiaker"
echo "${user[ivarm]}" # print Ivar Moe
echo "${user[@]}"      # print array values
echo "${!user[@]}"     # print array indices (keys)
echo "${#user[@]}"     # length of array
```

Associative arrays were introduced with Bash version 4 in 2009. If we don't declare the variable as an associative array with `declare -A` before we use it, it will be an ordinary indexed array.

```
user[lailas]="Laila Skiaker" can also be written as  
user+=([lailas]="Laila Skiaker")
```

### A.1.2 Structures/Classes

#### Structures/Classes

Sorry, no structs or classes in Bash ...

### A.1.3 Command-line args

#### Command-Line Arguments

Scriptname in \$0, arguments in \$1, \$2, ...

```
#!/bin/bash  
# cli-args.bash  
  
echo "I am $0, and have $# arguments \  
      first is $1"
```

Bash accepts the first nine arguments as \$1...\$9, for further arguments use \${10}, \${11}, ...

## A.2 Input

### A.2.1 Input

#### Input From User

```
#!/bin/bash  
# input-user.bash  
  
echo -n "Say something here:"  
read something  
echo "you said $something"
```

## Input From STDIN

Same way, commonly without an echo first

```
#!/bin/bash
# input-stdin.bash

read something
echo "you said $something"
```

can be executed as

```
echo "hey hey!" | ./input-stdin.bash
```

Of course, input from user is from STDIN.

## A.2.2 System commands

### Input from System Commands

You can use `$(cmd)` (supports nesting) or `cmd` (deprecated)

```
#!/bin/bash
# input-commands.bash

kernel=$(uname -sr)
echo "I am running on $kernel in $(pwd)"
```

This is also called *command substitution*. `...` (backticks) is deprecated because it's difficult to read, and can create some problems, see <http://mywiki.wooleedge.org/BashFAQ/082>

## A.3 Conditions

### A.3.1 if/else

#### if/else

```
#!/bin/bash
# if.bash

if [[ "$#" -ne 1 ]]; then
    echo "usage: $0 <argument>"
fi
```

Note: there must be spaces around [[ and ]].

There is also an older (slower) and more portable (meaning POSIX defined) operator, [ which is actually an alias for the operator test, meaning

```
[ "$#" -ne 2 ]
# is the same as
test "$#" -ne 2
```

### A.3.2 Operators

#### Arithmetic Comparison

Operator	Meaning
-lt	Less than
-gt	Greater than
-le	Less than or equal to
-ge	Greater than or equal to
-eq	Equal to
-ne	Not equal to

#### String Comparison

Operator	Meaning
<	Less than, in ASCII alphabetical order
>	Greater than, in ASCII alphabetical order
=	Equal to
==	Equal to
!=	Not equal to

#### File Tests

Operator	Meaning
-e	Exists
-s	Not zero size
-f	Regular file
-d	Directory
-l	Symbolic link
-u	Set-user-id (SetUID) flag set

There are many more file test operators of course.

## Boolean

Operator	Meaning
!	Not
&&	And
	Or

## Numerical or String Compare

```
#!/bin/bash
# if-num-string.bash

if [[ "$#" -ne 2 ]]; then
    echo "usage: $0 <argument> <argument>"
    exit 0
elif [[ $1 -eq $2 ]]; then
    echo "$1 is arithmetic equal to $2"
else
    echo "$1 and $2 arithmetic differs"
fi
if [[ $1 == $2 ]]; then
    echo "$1 is string equal to $2"
else
    echo "$1 and $2 string differs"
fi
if [[ -f $1 ]]; then
    echo "$1 is also a file!"
fi
```

This shows the if-elif-else construction, the difference between string and numerical comparison, and a file test operator.

Note the difference between `-eq` and `==`

```
$ ./if-num-string.bash 1 01
1 is arithmetic equal to 01
1 and 01 string differs
```

## Boolean example

```
#!/bin/bash
# if-bool.bash

if [[ 1 -eq 2 && 1 -eq 1 || 1 -eq 1 ]]; then
    echo "And has precedence"
else
    echo "Or has precedence"
fi

# force OR precedence:

if [[ 1 -eq 2 && (1 -eq 1 || 1 -eq 1) ]]; then
    echo "And has precedence"
else
    echo "Or has precedence"
fi
```

AND is always (as known from mathematics courses) evaluated before OR (binds more tightly). Write it down in logic (truth table) if you are unsure.

### A.3.3 Switch/case

#### Case

```
#!/bin/bash
# switch.bash

read ans
case $ans in
yes)
    echo "yes!"
    ;;&                      # keep testing
no)
    echo "no?"
    ;;                      # do not keep testing
*)
    echo "$ans???"
```

;;
esac

See also select and whiptail.

## A.4 Iteration

### A.4.1 For

#### For loop

```
#!/bin/bash
# for.bash

for i in {1..10}; do
    echo -n "$i "
done
echo

# something more useful:

for i in /*; do
    if [[ -f $i ]]; then
        echo "$i is a regular file"
    else
        echo "$i is not a regular file"
    fi
done
```

We can also use `for i in $(ls -1 ~/)` or `for i in $(stat -c "%n" /*)` but this creates problems if we have filenames with spaces, so it's much better to use Bash' builtin expansion operator \* as we do in this example.

### A.4.2 While

#### While

We want to read from STDIN and do stuff line by line

```
#!/bin/bash
# while.bash

i=0
while read line; do
    foo[i]="$line"
    ((i++))
done
echo "i is $i, size of foo ${#foo[@]}"
```

```
$ ls -1 | ./while.bash
i is 20, size of foo is 20
```

**A problem ...**

What if we want to pipe into a while inside our script:

```
#!/bin/bash
# while-pipe-err.bash

i=0
ls -1 | while read line; do
    foo[i]="$line"
    ((i++))
done
echo "i is $i, size of foo ${#foo[@]}"

$ ./while-pipe-err.bash
i is 0, size of foo is 0
```

*In other words, this does not work due to a subshell being used (because of the pipe) inside while!*

Meaning that the variables outside the while loop are not accessible inside the while loop since it is run as a new process.

**Solution with here string**

```
#!/bin/bash
# while-pipe.bash

i=0
while read line; do
    foo[i]="$line"
    ((i++))
done <<< "$(ls -1)" # here string
echo "i is $i, size of foo ${#foo[@]}"

$ ./while-pipe.bash
i is 20, size of foo is 20
```

`command <<< "$var"` is called a here string (a special version of a here document), it expands `$var` and feeds the output to stdin of command.

We can also solve this problem without here string by rewriting it as a for loop instead:

### Solution with for instead of while

```
#!/bin/bash
# for-commands.bash

i=0
for line in $(ls -1); do
    foo[i]="$line"
    ((i++))
done
echo "i is $i, size of foo ${#foo[@]}"
```

## A.5 Math

### Operators

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus

*Only on integers!*

```
#!/bin/bash
# math.bash

i=0
((i++))
echo "i is $((i-2))" # print -1
```

### A trick for floats

```
#!/bin/bash
# math-float.bash

echo "3.1+5.6 is $(echo '3.1+5.6' | bc)"
```

## A.6 Functions

### Functions

```
#!/bin/bash
# func.bash

# declare:
function addfloat {
    echo "$1+$2" | bc
}
# use:
addfloat 5.12 2.56
```

## A.7 RegExp

### Regular expressions intro 1/5

Special/Meta-characters:

\ | ( ) [ ] { } ^ \$ \* + ? .

*These have to be protected with \, e.g.*

<http://www\.\.hig\.\.no>

To match c:\temp, you need to use the regex c:\\temp. As a string in C++ source code, this regex becomes "c:\\\\temp". Four backslashes to match a single one indeed.

(from <http://www.regular-expressions.info/characters.html>):

Regular expressions are the generic way to describe a string/syntax/word that you use for either syntax/compliance checking or searching. It is commonly used in configuration files. Think of it as a generic way of doing advanced search. Google would probably prefer user to only enter regular expression as search terms, but that would be too hard for the general population so Google offers “advanced search” instead:

[http://www.google.com/advanced\\_search](http://www.google.com/advanced_search)

There are many different regular expression engines, which differ mostly in features and speed. In this tutorial we will try to stick with simple examples which will work the same in most engines (perl, pcre, extended posix, .NET, ...).

### Regular expressions intro 2/5

Describing characters:

Operator	Meaning
.	Any single character
[abcd]	One of these characters
[^abcd]	Any one but these characters
[A-Za-z0-9]	A character in these ranges
:word:	A word (A-Za-z0-9_)
:digit:	A digit

\w is the same as [a-zA-Z0-9] and \d is the same as [0-9]. Many more of course ...

### Regular expressions intro 3/5

Grouping:

Operator	Meaning
( )	Group
	OR

Anchoring:

Operator	Meaning
^	Beginning of line
\$	End of line

### Regular expressions intro 4/5

Repetition operators/Modifiers/Quantifiers:

Operator	Meaning
?	0 or 1 time
*	0 or more times
+	1 or more times
{N}	N times
{N,}	At least N times
{N,M}	At least N but not more than M

Demo: example with cat a.html | egrep REGEXP (four steps).

### Regular expressions intro 5/5

Finding URLs in HTML:

(mailto|http) : [^"]\*

Each line should be an email address:

^ [A-Za-z0-9.\_-]+@[A-Za-z0-9.\_-]+\$

Remember that regexp engines are most often greedy, they try to match as much as possible, so using e.g. .\* might match more than you were planning for.

### A.7.1 Bash example

#### Bash example

```
#!/bin/bash
# regexp.bash

while read line; do
    if [[ $line =~ \
        ^[A-Za-z0-9._-]+@[A-Za-z0-9._-]+\$ ]]
    then
        echo "Valid email ${BASH_REMATCH[0]}"
        echo "Domain is ${BASH_REMATCH[1]}"
    else
        echo "Invalid email address!"
    fi
done
```

When we use regular expressions inside scripts, it is very useful to be able to extract parts of the match. We can do this by specifying the part with (part) and refer to it later in the \$BASH\_REMATCH array (if we specify two parts, the second one will be in \$BASH\_REMATCH[2] etc).

Of course you can use regexp in many different components which you can include in your bash script (sed, grep, perl, ...).

## A.8 Bash only

### Advanced stuff

See Advanced Bash-Scripting Guide at

<http://tldp.org/LDP/abs/html/>

for everything you can do with Bash

## A.9 Credits

**Credits** J. Ousterhout, "Scripting: Higher-Level

Programming for the 21st Century," IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30.

<http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html> [http://www.linuxconfig.org/Bash\\_scripting\\_Tutorial](http://www.linuxconfig.org/Bash_scripting_Tutorial)

<http://www.thegeekstuff.com/2010/06/bash-array-tutorial/>

<http://www.panix.com/~elflord/unix/bash-tute.html>

<http://www.codecoffee.com/tipsforlinux/articles2/043.html>

<http://tldp.org/LDP/abs/html/>

[http://linuxsig.org/files/bash\\_scripting.html](http://linuxsig.org/files/bash_scripting.html)

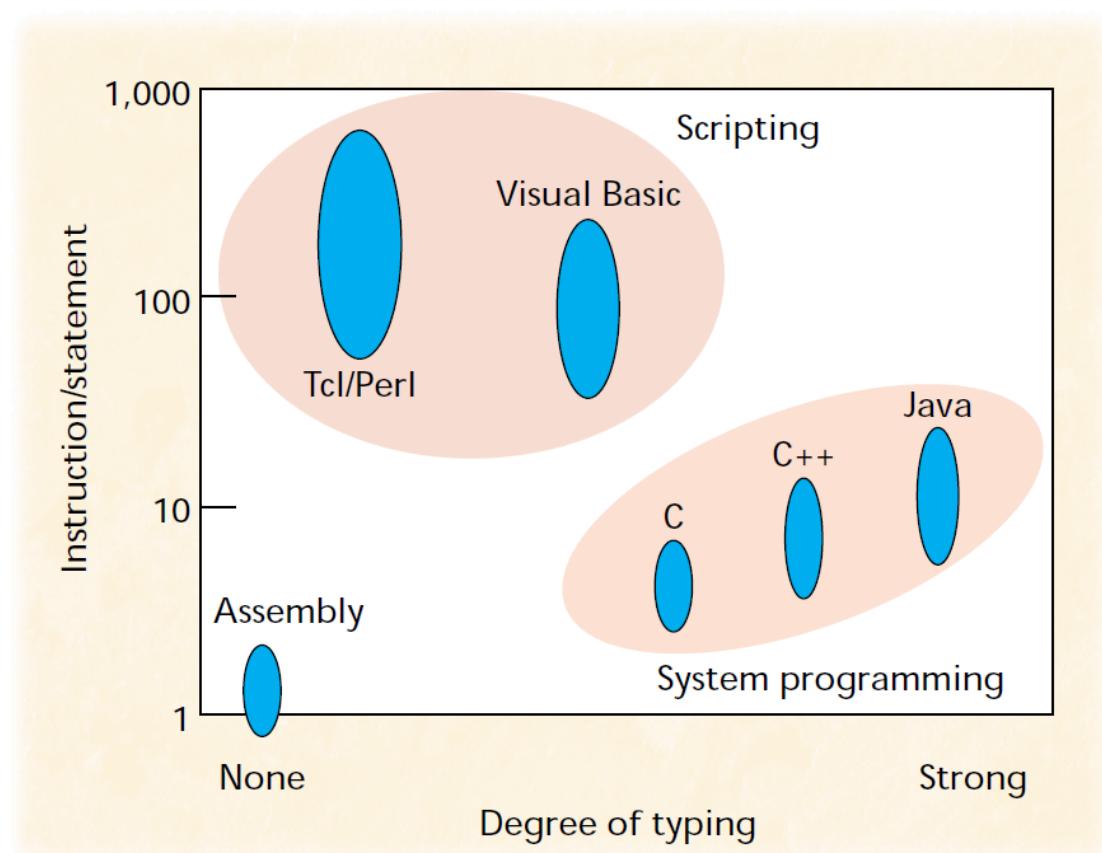
<http://mywiki.wooledge.org/BashGuide>

<http://www.regular-expressions.info/>

Alexander Berntsen

## Appendix B

### PowerShell



(OUSTERHOUT, J., "Scripting: Higher-Level Programming for the 21st Century", IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30.)

From Ousterhout, 1998:

While programming languages like C/C++ are designed for low-level construction of

data structures and algorithms, scripting languages are designed for high-level “gluing” of existing components. Components are created with low-level languages and glued together with scripting languages.

### **WARNING!**

The following presentation is NOT meant to be a comprehensive/complete tour of the PowerShell language.

*The purpose is to get you started with some basic program constructions which you will recognize based on some-sort-of-programming-background.*

At the end of the presentation (Credits section) you will find pointers to more comprehensive material (reference material).

### **Practice**

*You need a Windows host running on a physical or virtual machine with working access to the internet, and with PowerShell v2.0 installed.*

Log in and open a terminal window, download the examples as we go along from

`http://www.hig.no/~erikh/tutorial-powershell/FILENAME`

(or download all at once with filename `powershell-examples.zip` but remember to unblock before unzip)

You will find the `FILENAME` on the first line of each example. For each example do

1. Download

`wget http://www.hig.no/~erikh/tutorial-powershell/FILENAME`

2. View the code

`Get-Content FILENAME`

3. Execute the code

`.\\FILENAME`

We assume that you are using PowerShell 2.0 (as shipped with Windows 7 and Windows Server 2008R2) and have installed the PowerShell Community Extensions from <http://pscx.codeplex.com/> and the GnuWin32 utilities <http://sourceforge.net/projects/getgnuwin32/files/> (where you will find `wget` etc).

To allow for execution of scripts in powershell you need to set the correct execution policy:

```
# check what is current policy
Get-ExecutionPolicy
# change to only require signature on remote scripts
Set-ExecutionPolicy RemoteSigned
# you probably need to "run as administrator" to do this
```

---

To install PowerShell Community Extensions

```
# download Pscx-2.x.x.x.zip using a webbrowser
# windows explorer and browse to where it is
# right click on Pscx-2.x.x.x.zip, choose properties
# click unblock, ok
# right click, extract all to $PSHOME\Modules dir
# $PSHOME is probably
# C:\Windows\System32\Windows\PowerShell\v1.0
Import-Module Pscx
# place this command in $profile so it is run every time
# you start PowerShell, or do it globally with
# "run as administrator" and
New-Item $pshome\profile.ps1 -type file
notepad $pshome\profile.ps1
```

To install GnuWin32

```
# Run setup program from
# http://sourceforge.net/projects/getgnuwin32/files/
# cd to the directory where it was downloaded
download.bat # answer yes to a couple of questions
# run powershell as administrator
install.bat 'C:\Program files\GnuWin32'
notepad $pshome\profile.ps1
# add the following to include the gnuwin32 tools in PATH
# $env:path += ";C:/Program Files/GnuWin32/bin"
```

## Hello World

```
# hello.ps1
Write-Host "hello world!"
```

execute as long as filename ends with .ps1:

```
.\hello.ps1
```

or direct from command line cmd (DOSPROMPT)

```
powershell -command "Write-Host \"hello world!\""
```

or direct from command line powershell

```
Write-Host "hello world!"
```

## B.1 Variables

### Single Variables

```
# single-var.ps1

$firstname="Mysil"
$lastname="Bergsprekken"
$fullname="$firstname $lastname"
Write-Host "Hello $fullname, may I call you `"
$firstname `?"
```

All variables are prefixed with \$

We need to use ` between \$firstname and ? to avoid ? being “part of” the variable name.

A single variable (sometimes called a *scalar*) is typed, but PowerShell chooses the type automatically for us by “guessing”. Typing can be forced by prefixing the variable with e.g. [int]. *What is important to know is that variables are instances of .NET objects, and these objects are also what is being passed through the pipe of piped commands (as opposed to just piping byte streams in other shells).*

PowerShell uses namespaces, e.g. you can write \$fullname or \$variable:fullname. You can list all current variables with Get-Variable \$variable:\*

Scope for a variable can be defined with Set-Variable -Scope. PowerShell can also *dot-source* script files to make a script’s variables accessible from the command line.

PowerShell in itself, like much of Windows, is case-insensitive, however it preserves case when used.

Btw, ` is the protection character (and line continuation character) in PowerShell (same as \ in bash). PowerShell does this differently from Unix/Linux scripts since \ (in addition to /) is used as a directory separator on Windows, see also

```
Get-Help about_escape_characters
```

### Single and Double Quotes

```
# quotes.ps1

$name="Mysil"
Write-Host Hello $name
Write-Host "Hello $name"
Write-Host 'Hello $name'
```

Variables are expanded/interpolated inside double quotes, but not inside single quotes.

### B.1.1 Arrays

#### Arrays

One-dimensional arrays:

```
# array.ps1

$os=@("linux", "windows")
$os+=@("mac")
Write-Host $os[1]      # print windows
Write-Host $os          # print array values
Write-Host $os.Count    # length of array
```

*Arrays are created with @(...)*

Note how we display the length of the array by viewing a property (Count) of the object.  
Btw, Count is just a reference to the Length property

```
./array.ps1
$os.PSExtended | Get-Member
```

If you want to access an array element within an interpolated string, you have to place the array element in parentheses like this:

```
Write-Host "My operating system is $($os[1])"
```

#### Associative Arrays

```
# assoc-array.ps1

$user=@{
    "frodeh" = "Frode Haug";
    "ivarm"  = "Ivar Moe"
}
$user+=@{"lailas"="Laila Skiaker"}
Write-Host $user["ivarm"]  # print Ivar Moe
Write-Host @user          # print array values
Write-Host $user.Keys      # print array keys
Write-Host $user.Count     # print length of array
```

*Associative arrays are created with @{...} and are called Hashtables in PowerShell.*

### B.1.2 Structures/Classes

#### Structures/Classes

A simple object used as a struct:

```
# struct.ps1

$myhost=New-Object PSObject -Property `

@{os="";
  sw=@();
  user=@{}}

$myhost.os="linux"
$myhost.sw+=@("gcc","flex","vim")
$myhost.user+=@{
    "frodeh"="Frode Haug";
    "monicas"="Monica Strand"
}
Write-Host $myhost.os
Write-Host $myhost.sw[2]
Write-Host $myhost.user["monicas"]
```

Of course, since PowerShell is based on the object-oriented framework .NET, creating and manipulating objects is a world by it self, there are a plethora of ways of doing these things.

See what kind of object this is by running the commands on the command line and doing

```
$myhost
$myhost.GetType()
$myhost | Get-Member
```

Note also that we don't need the line continuation character ` when inside a block ({...}).

### B.1.3 Command-line args

#### Command-Line Arguments

All command-line arguments in the array \$args

Scriptname retrieved from the object \$MyInvocation

```
# cli-args.ps1

Write-Host "I am" $MyInvocation.InvocationName `

"and have" $args.Count "arguments" `

"first is" $args[0]
```

\$MyInvocation is one of PowerShell's builtin variables. Again, check what kind of object this is with

```
$MyInvocation.GetType()  
$MyInvocation | Get-Member  
# or check what a typical PowerShell command returns  
Get-Process | Get-Member  
(Get-Process).GetType()  
# contrast this with a traditional cmd command  
ipconfig | Get-Member  
(ipconfig).GetType()
```

## B.2 Input

### B.2.1 Input

#### Input From User

```
# input-user.ps1  
  
$something=Read-Host "Say something here"  
Write-Host "you said" $something
```

#### Input From the Pipeline

```
# input-pipe.ps1  
  
$something="$input"  
Write-Host "you said" $something
```

can be executed as

```
Write-Output "hey hey!" | .\input-pipe.ps1
```

\$input (another one of PowerShell's builtin variables) is a special variable which enumerates the incoming objects in the pipeline.

#### Input From Files

```
# input-file.ps1  
  
$file=Get-Content hello.ps1  
Write-Host @file -Separator "`n"
```

You can assign the entire output of a command directly to a variable.

## B.2.2 System commands

### Input from System Commands

```
# input-commands.ps1

$name=(Get-WmiObject Win32_OperatingSystem).Name
$kernel=(Get-WmiObject `-
    Win32_OperatingSystem).Version
Write-Host "I am running on $name, version" `-
    "$kernel in $(Get-Location)"
```

Using `$(expr)` inside a string will treat it as an *ad-hoc variable* evaluating the expression `expr` and inserting the output into the string.

## B.3 Conditions

### B.3.1 if/else

#### if/else

```
# if.ps1

if ($args.Length -ne 1) {
    Write-Host "usage: " `-
        "$MyInvocation.InvocationName `-
        <argument>"
}
```

### B.3.2 Operators

#### Comparison

Operator	Meaning
-lt	Less than
-gt	Greater than
-le	Less than or equal to
-ge	Greater than or equal to
-eq	Equal to
-ne	Not equal to

Note that many other test operators (e.g. file tests) are used as methods in the objects instead of separate operators.

## Boolean

Operator	Meaning
-not	Not
!	Not
-and	And
-or	Or

```
# if-num-string.ps1

if ($args.Count -ne 2) {
    Write-Host "usage: `n
                $MyInvocation.InvocationName `n
                <argument> <argument>`n
    exit 0
} elseif ($args[0] -gt $args[1]) {
    Write-Host $args[0] "larger than" $args[1]
} else {
    Write-Host $args[0] "smaller than or" `n
                    "equal to" $args[1]
}
if (Test-Path $args[0]) {
    if (!(Get-Item $args[0]).PSIsContainer) {
        Write-Host $args[0] "is a file"
    }
}
```

There are not separate comparison operators for numbers and strings. Be careful when comparing objects with different types. Behaviour might be a bit strange (see page 209 of "Mastering PowerShell" by Weltner):

```
$ 123 -lt "123.4"
False
$ 123 -lt "123.5"
True
```

A set of *file test operators* is not available since this functionality is covered through cmdlets (e.g. `Test-Path`) and methods (e.g. `PSIsContainer`).

## Boolean example

```
# if-bool.ps1

if ((1 -eq 2) -and (1 -eq 1) -or (1 -eq 1)) {
    Write-Host "And has precedence"
} else {
    Write-Host "Or has precedence"
}

# force OR precedence:

if ((1 -eq 2) -and ((1 -eq 1) -or (1 -eq 1))) {
    Write-Host "And has precedence"
} else {
    Write-Host "Or has precedence"
}
```

AND is always (as known from mathematics courses) evaluated before OR (binds more tightly). Write it down in logic (truth table) if you are unsure.

### B.3.3 Switch/case

#### Switch/Case

```
# switch.ps1

$short = @{ yes="y"; nope="n" }
$ans = Read-Host
switch ($ans) {
    yes { Write-Host "yes" }
    nope { Write-Host "nope"; break }
    {$short.ContainsKey("$ans")}`n
        { Write-Host $short[$ans] }
    default {Write-Host "$ans `???"}
}
```

Run example and see the difference between inputting yes, nope and nei.

In the example above `{$short.ContainsKey("$ans")}`n` checks if the content of `$ans` has an entry (matches a key) in the associative array `$short`. Switch in PowerShell continues testing each case unless it reads a `break`.

### B.3.4 Where

#### Where/Where-Object

```
# where.ps1

Get-ChildItem | Where-Object {$_['.Length -gt 1KB]}
```

In a pipeline we use `Where-Object` and `ForEach-Object`, but when processing a collection/array in a script we would use `Where` and `ForEach` (in other words: without the `-object`).

We can use KB, MB and GB and PowerShell understands what we mean.

## B.4 Iteration

### B.4.1 For

#### For loop

```
# for.ps1

for ($i=1;$i-le3;$i++) {
    Write-Host "$i"
}

# something more useful:

$file=Get-ChildItem
for ($i=0;$i-lt$file.Count;$i++) {
    if (!(Get-Item $file[$i]).PSIsContainer) {
        Write-Host $file[$i].Name "is a file"
    } else {
        Write-Host $file[$i].Name "is a directory"
    }
}
```

Normally you would use `ForEach` instead of `for` since you can simplify the first loop above like this:

```
ForEach ($i in 1..3) {
    Write-Host "$i"
}
```

### B.4.2 While

#### While

```
# while.ps1

while ($i -le 3) {
    Write-Host $i
    $i++
}

# something more useful:

$file=Get-ChildItem
$i=0
while ($i -lt $file.Count) {
    if (!(Get-Item $file[$i]).PSIsContainer) {
        Write-Host $file[$i].Name "is a file"
    } else {
        Write-Host $file[$i].Name "is a directory"
    }
    $i++
}
```

The for example converted to while.

### B.4.3 Foreach

#### Foreach loop

```
# foreach.ps1

foreach ($i in Get-ChildItem) {
    Write-Host $i.Name
}

# with associative arrays

$user=@{
    "frodeh" = "Frode Haug";
    "monicas" = "Monica Strand";
    "ivarm" = "Ivar Moe"
}
foreach ($key in $user.Keys) {
    Write-Host $user[$key]
}
```

In a pipeline we would use `ForEach-Object`.

#### ForEach

If we want to read from the pipeline and do stuff object by object:

```
# foreach-pipe.ps1

foreach ($i in $input) {
    $foo += @($i)
}
Write-Host "size of foo is" $foo.Count
```

or

```
# foreach-object-pipe.ps1

$input | ForEach-Object {
    $foo += @_}
}
Write-Host "size of foo is" $foo.Count
```

```
$ Get-ChildItem | ./foreach-object-pipe.ps1
size of foo is 20
```

\$input represents the pipeline and \$\_ the current object in the pipeline.

## B.5 Math

### Operators

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus

```
# math.ps1

Write-Host "3+5 is" (3+5)
```

```
Write-Host "3+5 is" 3+5
Write-Host "3+5 is" (3+5)
Write-Host "3+5 is" $(3+5)
Write-Host "3+5 is (3+5)"
Write-Host "3+5 is $(3+5)"
```

## B.6 Functions

### Functions

```
# func.ps1

# declare:
function add($a, $b) {
    Write-Host "$a+$b is" ($a+$b)
}
# use:
add 5.12 2.56
```

## B.7 RegExp

### Regular expressions intro 1/5

Special/Meta-characters:

\ | ( ) [ ] { } ^ \$ \* + ? .

*These have to be protected with \, e.g.*

<http://www\.\.hig\.no>

To match c:\temp, you need to use the regex c:\\temp. As a string in C++ source code, this regex becomes "c:\\\\temp". Four backslashes to match a single one indeed.

(from <http://www.regular-expressions.info/characters.html>):

There are many different regular expression engines, which differs mostly in features and speed. In this tutorial we will try to stick with simple examples which will work the same in most engines (perl, pcre, extended posix, .NET, ...).

### Regular expressions intro 2/5

Describing characters:

Operator	Meaning
.	Any single character
[abcd]	One of these characters
[^abcd]	Any one but these characters
[a-zA-Z0-9]	A character in these ranges

### Regular expressions intro 3/5

Grouping:

Operator	Meaning
( )	Group
	OR

Anchoring:

Operator	Meaning
^	Beginning of line
\$	End of line

### Regular expressions intro 4/5

Repetition operators/Modifiers/Quantifiers:

Operator	Meaning
?	0 or 1 time
*	0 or more times
+	1 or more times
{N}	N times
{N,}	At least N times
{N,M}	At least N but not more than M

Demo: four step example with

```
cat a.html | ForEach-Object {if($_ -match REGEXP)`{Write-Host $matches[0]}}
```

### Regular expressions intro 5/5

Finding URLs in HTML:

```
(mailto|http)://[^"]*
```

Each line should be an email address:

```
^ [A-Za-z0-9._-]+@[A-Za-z0-9._-]+\$
```

Remember that regexp engines are most often greedy, they try to match as much as possible, so using e.g. .\* might match more than you were planning for.

#### B.7.1 PowerShell example

##### PowerShell example

```
# regexp.ps1

$input | ForEach-Object {
    if ($_. -match
        "^[A-Za-z0-9._-]+@[A-Za-z0-9.-]+\$") {
        Write-Host "Valid email", $matches[0]
        Write-Host "Domain is", $matches[1]
    } else {
        Write-Host "Invalid email address!"
    }
}
```

When we use regular expressions inside scripts, it is very useful to be able to extract parts of the match. We can do this by specifying the part with (part) and refer to it later using `$matches[1]`, `$matches[2]`, etc. `$matches[0]` matches the entire expression.

<http://www.regular-expressions.info/powershell.html>

## B.8 PowerShell only

### Advanced stuff

See the complete Mastering PowerShell book at

<http://powershell.com/cs/blogs/ebook/>

for much more of what you can do with PowerShell

## B.9 Credits

### Credits

<http://refcardz.dzone.com/refcardz/windows-powershell>

<http://powershell.com/cs/blogs/ebook/>

<http://technet.microsoft.com/en-us/library/ee692948.aspx>

[http://www.techtropia.com/index.php/Windows\\_PowerShell\\_1.0\\_String\\_Quoting\\_and\\_Escape\\_Sequences](http://www.techtropia.com/index.php/Windows_PowerShell_1.0_String_Quoting_and_Escape_Sequences)

<http://dmitrysotnikov.wordpress.com/2008/11/26/input-gotchas/>

<http://stackoverflow.com/questions/59819/how-do-i-create-a-custom-type-in-powershell-for-my-scripts-to-use>

<http://www.powershellpro.com/powershell-tutorial-introduction/>

[http://en.wikipedia.org/wiki/Windows\\_PowerShell](http://en.wikipedia.org/wiki/Windows_PowerShell)

## *B.9. CREDITS*

---

<http://www.johndcook.com/powershell.html>

<http://www.regular-expressions.info/>

OUSTERHOUT, J., "Scripting: Higher-Level Programming for the 21st Century", IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30.)