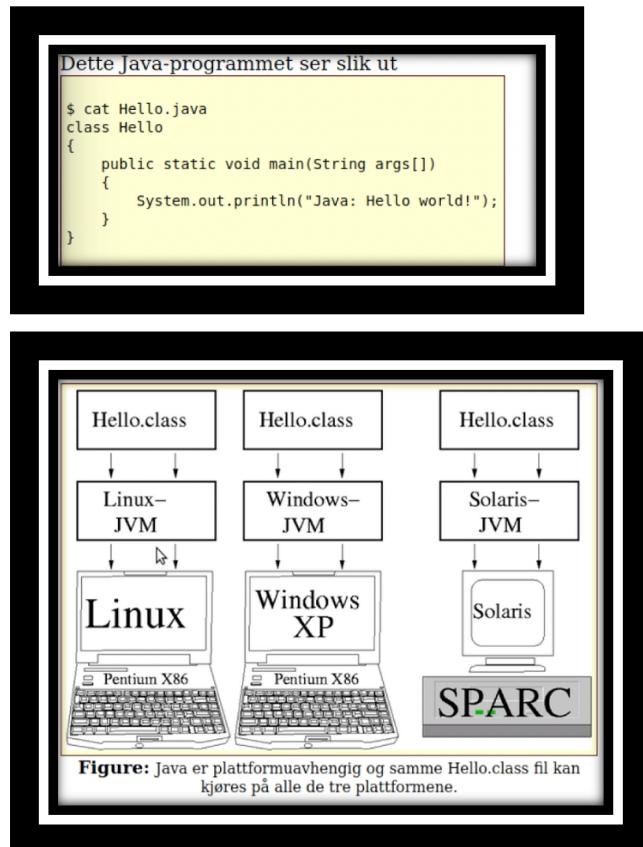

NOTATER UKE 13 –
Plattformavhengighet og Threads

Å KJØRE JAVA, C OG BASH-PROGRAMMER UNDER FORSKJELLIGE OS

x86 og Intel er begge relatert til datamaskiner og prosessorer. x86 er en type CPU-arkitektur som har vært brukt i mange personlige datamaskiner og servere, mens Intel er en av de største produsentene av CPU-er, inkludert x86-baserte prosessorer.

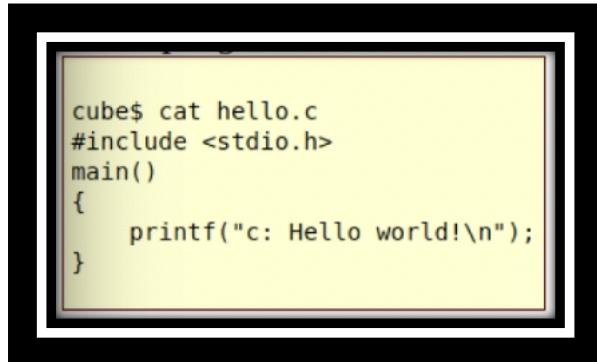
Solaris er et operativsystem utviklet av Sun Microsystems, nå en del av Oracle Corporation. Solaris kjører på mange forskjellige maskinarkitekturen, inkludert x86-prosessorer, og er kjent for sin skalerbarhet, sikkerhet og pålitelighet.

En a.out fil inneholder x86 instruksjoner og kjørr direkte på pc. Java-fil kjører på en JVM.



Plattformuavhengighet vil si at man kan ta en fil og kjøre den på andre plattformer. I bildet over ser vi at det er ulike Java virtuelle maskiner som kan tolke fila og kjøre den. Når man kompilerer Java fil med javac så dannes det en class fil. Det er derfor det står Hello.class. Enheten OS + Hardware er ofte omtalt som en plattform, og det er pga dette vi kan si Java er plattformuavhengig.

Så kommer vi til C-program som i høyeste grad er plattformavhengig.



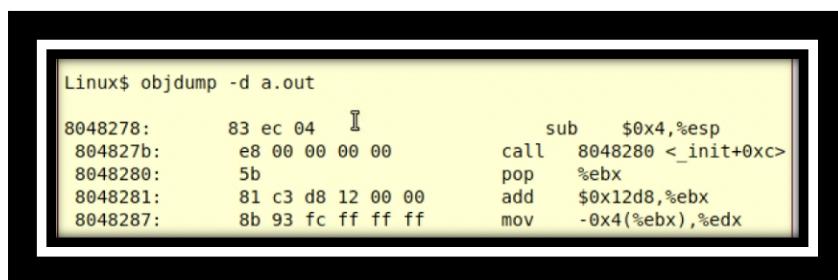
```
cube$ cat hello.c
#include <stdio.h>
main()
{
    printf("c: Hello world!\n");
}
```

Når vi kompilerer denne hello.c får vi en a.out fil. Den vil holde x86-instruksjoner som på en assembler som kan se ut som:

```
mov %eax, %ebx
```

og disse vil så klart kun forstås av en x86 CPU.

Kommandoen "objdump -d a.out" demonterer den binære eksekverbare filen "a.out" og viser assembler-koden for hver funksjon i filen. "-d" -alternativet forteller objdump å demontere den binære filen, som betyr at den vil konvertere maskinkodeinstruksjonene i filen til menneskellesbar assemblerkode. Resultatet vil vise assemblerkoden for hver funksjon i den binære filen, sammen med de tilsvarende minneadressene for hver instruksjon.

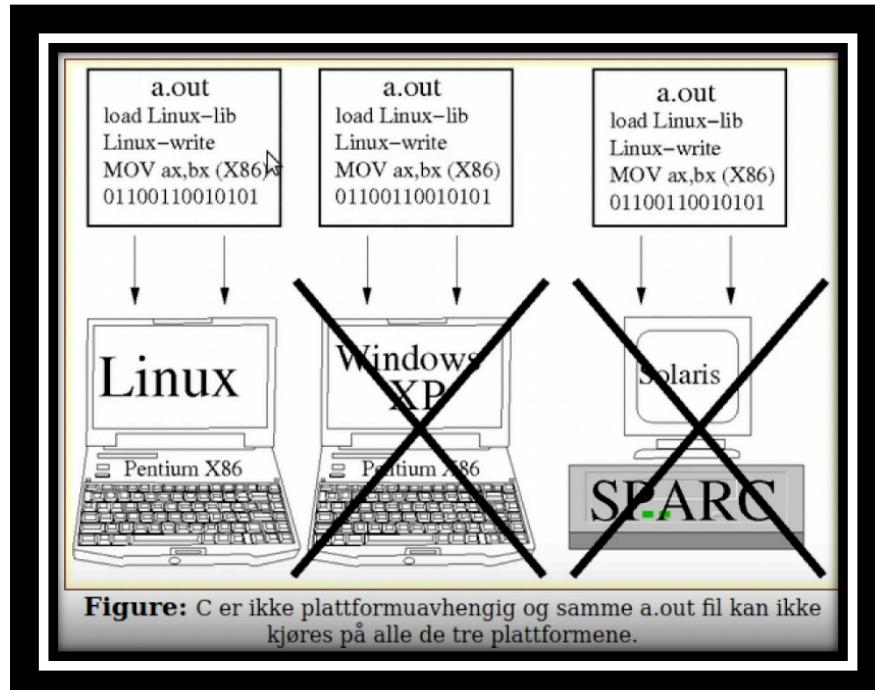


```
Linux$ objdump -d a.out
8048278: 83 ec 04  I          sub    $0x4,%esp
804827b: e8 00 00 00 00      call   8048280 <_init+0xc>
8048280: 5b                 pop    %ebx
8048281: 81 c3 d8 12 00 00      add    $0x12d8,%ebx
8048287: 8b 93 fc ff ff ff      mov    -0x4(%ebx),%edx
```

Under er det samme kommando på solaris:

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

a.out kjøring vil gi feilmelding på en Windows PC, til tross for x86-arkitekturen, og SPARC.



Hvordan kan man da kjøre a.out på Window-server og Solaris. Da må man ha en kompilator for den plattformen. Da kan vi komprimere, få en a.exe også kjøre på Windows eller en s.out.

TEST AV C, JAVA, PYTHON OG BASH UNDER ULIKE OS

Fem plattformer:

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

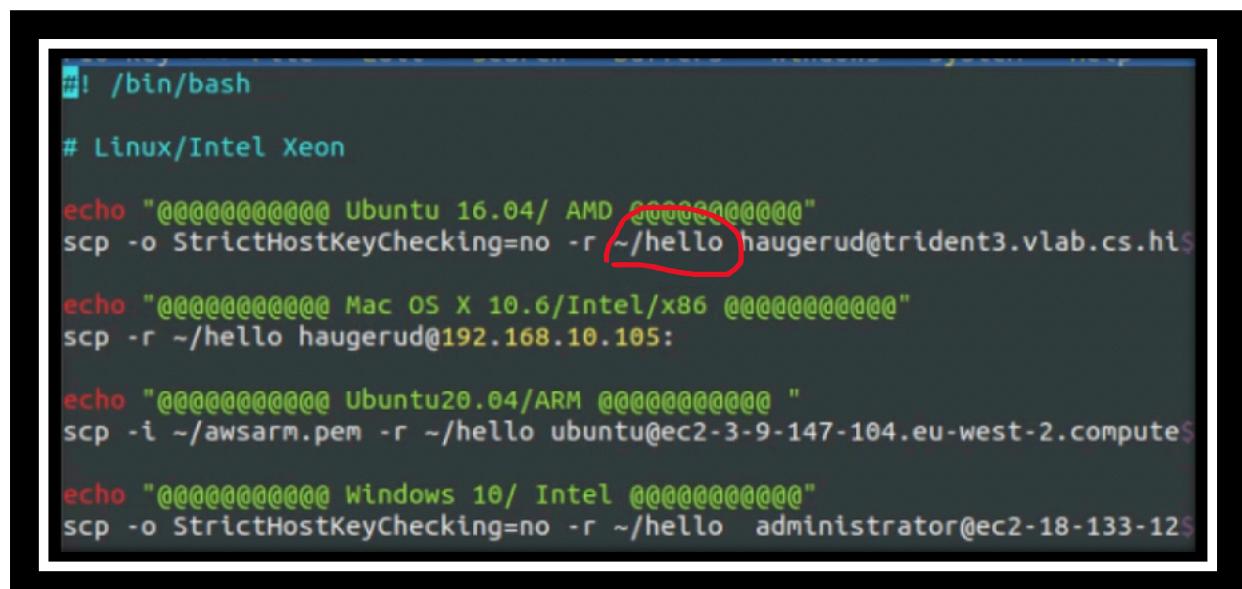
Intel og AMD har begge x86 instruksjoner. De tre første siste er Intel, men den 4 er ARM.

Under er det et shell som kopierer hele mappen kan står i (rød) til de andre plattformene.



```
haugerud@lap:~/hello$ jed dist.sh
```

1 er AMD serveren, 2 er MAC OS, 3 ARM serveren og 4 er Windows. De to siste kjører på Amazon cloud. Men før dette er meningen å kompilere programmene.



```
#!/bin/bash

# Linux/Intel Xeon
echo "@@@@@@@@@@@@ Ubuntu 16.04/ AMD @@@@@@@@"
scp -o StrictHostKeyChecking=no -r ~/hello haugerud@trident3.vlab.cs.hi$
```

The line above is circled in red.

```
echo "@@@@@@@@ Mac OS X 10.6/Intel/x86 @@@@@@@@"
scp -r ~/hello haugerud@192.168.10.105:
```

```
echo "@@@@@@@@ Ubuntu20.04/ARM @@@@@@@@"
scp -i ~/awsarm.pem -r ~/hello ubuntu@ec2-3-9-147-104.eu-west-2.compute$
```

```
echo "@@@@@@@@ Windows 10/ Intel @@@@@@@@"
scp -o StrictHostKeyChecking=no -r ~/hello administrator@ec2-18-133-125
```

Under er det et skript som kompilere GCC og Java:



```
haugerud@lap:~/hello$ jed compile
```



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

AMD-SERVEREN

Vi ser at a.out fungerer.



```
haugerud@trident3:~/hello$ ./a.out
c: Hello world!
```

Når vi kjører Java Hello så får vi feilmeldinger her.



```
haugerud@trident3:~/hello$ java Hello
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.UnsupportedClassVersionError: Hello
has been compiled by a more recent version of the Java Runtime (class f
ile version 55.0), this version of the Java Runtime only recognizes clas
s file versions up to 52.0
```

Hvis vi leser nøye under så står det at det er et versjonsproblem. Så problemet her er at Hårek har kompilert en Java 11, og prøver å kjøre det på Java 8. dermed kan vi si at Java er ikke helt plattformuavhengig.



```
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.UnsupportedClassVersionError: Hello
has been compiled by a more recent version of the Java Runtime (class f
ile version 55.0), this version of the Java Runtime only recognizes clas
s file versions up to 52.0
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:339)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:120)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:455)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:367)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:359)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:349)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:352)
```

Python og Bash går helt fint her:



```
haugerud@trident3:~/hello$ python hello.py
Python: Hello World!
haugerud@trident3:~/hello$ bash hello.bash
bash: Hello world!
```

MAC OS

Vi ser a.out vil ikke fungere. Selvom dette er en Intel CPU med x86-instruksjoner så er dette et annet operativsystem.



```
haugerud@trident3:~/hello$ ./a.out
-bash: ./a.out: cannot execute binary file
```

Vi får det samme problemet med Java fordi dette er Java 6, en eldre versjon, som ikke kjenner igjen en nyere Java 11 class fil.

```
MacOS X Darwin Kernel Intel Core Duo Java 6 (1.6) Python 2.6
at java.net.URLClassLoader.access$000(URLClassLoader.java:58)
at java.net.URLClassLoader$1.run(URLClassLoader.java:197)
```

Python og bash går helt fint.

```
localhost:hello haugerud$ python hello.py
Python: Hello World!
localhost:hello haugerud$ bash hello.bash
bash: Hello world!
```

ARM

Hovedpoenget er at de underliggende maskininstruksjonene her har helt annet instruksjonssett. Dermed ser vi at a.out vil ikke kjøre. CPU-en skjønner ikke instruksjonene som ligger i a.out.

```
ubuntu@ip-172-31-11-60:~/hello$ ./a.out
-bash: ./a.out: cannot execute binary file: Exec format error
```

Java fungerer ettersom vi kjører på en Java 14.

```
ubuntu@ip-172-31-11-60:~/hello$ java Hello
Java: Hello world!
```

Python fungerer helt fint.

```
ubuntu@ip-172-31-11-60:~/hello$ python hello.py
Python: Hello World!
```

Bash fungerer også

```
ubuntu@ip-172-31-11-60:~/hello$ bash hello.bash
bash: Hello world!
```

WINDOWS

Vi kan se at det ikke går å kjøre uten å konverte den til en filtype som Windows forstår, som .exe eller .dll. a.out snakker med Linux-operativsystemet. Det er til tross for at det er akkurat samme CPU, det er Intel Xeon CPU. Så a.out vil prøve å snakke men Linux også møter den Windows, og da fungerer ingenting.

```
PS C:\Users\Administrator\hello> .\a.out
PS C:\Users\Administrator\hello> cp .\a.out a.exe
PS C:\Users\Administrator\hello> .\a.exe
Program 'a.exe' failed to run: The specified executable is not a valid
application for this OS platform. At line:1 char:1
+ .\a.exe
+ ~~~~~.
At line:1 char:1
+ .\a.exe
+ ~~~~~
+ CategoryInfo          : ResourceUnavailable: (:) 
+ FullyQualifiedErrorId : NativeCommandFailed

PS C:\Users\Administrator\hello>
```



På Java får vi samme runtime problemet, her fra versjon 11 til 8.

```
again
Exception in thread "main" java.lang.UnsupportedClassVersionError: Hello
    has been compiled by a more recent version of the Java Runtime (class f
ile version 55.0), this version of the Java Runtime only recognizes clas
s file versions up to 52.0
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(Unknown Source)
    at java.security.SecureClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.access$100(Unknown Source)
    at java.net.URLClassLoader$1.run(Unknown Source)
```

Python fungerer fint.

```
PS C:\Users\Administrator\hello> python.exe hello.py
Python: Hello World!
```

Fungerer med bash:

```
PS C:\Users\Administrator\hello> .\hello.bash
PS C:\Users\Administrator\hello> bash hello.bash
bash: Hello world!
```

Fungerer fordi han har installert Bash eller den Linux-modulen som trengs for å kunne starte et Bash-shell her inne og utføre kommandoer.

```
PS C:\Users\Administrator\hello> bash
haugerud@EC2AMAZ-8FI1V48:/mnt/c/Users/Administrator/hello$ 
haugerud@EC2AMAZ-8FI1V48:/mnt/c/Users/Administrator/hello$ pwd
/mnt/c/Users/Administrator/hello
haugerud@EC2AMAZ-8FI1V48:/mnt/c/Users/Administrator/hello$ ls -l
total 26
-rwxrwxrwx 1 haugerud haugerud 422 Mar 16 08:07 Hello.class
-rwxrwxrwx 1 haugerud haugerud 116 Mar 16 08:07 Hello.java
-rwxrwxrwx 1 haugerud haugerud 8304 Mar 16 08:07 a.exe
-rwxrwxrwx 1 haugerud haugerud 8304 Mar 16 08:07 a.out
-rwxrwxrwx 1 haugerud haugerud 72 Mar 16 08:07 compile
-rwxrwxrwx 1 haugerud haugerud 564 Mar 16 08:07 dist.sh
-rwxrwxrwx 1 haugerud haugerud 563 Mar 16 08:07 dist.sh~
```

Men det går ann å kjøre a.out inne i shellet.

```
haugerud@EC2AMAZ-8FI1V48:/mnt/c/Users/Administrator/hello$ ./a.out
c: Hello world!
```

ARM assembly instruksjoner for sum.c

kompilerer

```
COLLECT2: error: no returned in exit status
ubuntu@ip-172-31-11-60:~$ gcc -S sum.c
```

Ser

```
ubuntu@ip-172-31-11-60:~$ ls -l
total 16
drwx----- 2 ubuntu ubuntu 4096 Mar 16 08:41 hello
-rwx----- 1 ubuntu ubuntu   68 Mar 15 12:34 hello.c
-rw-rw-r-- 1 ubuntu ubuntu   89 Mar 15 22:43 sum.c
-rw-rw-r-- 1 ubuntu ubuntu  553 Mar 16 08:58 sum.s
```

Den inneholder da ARM instruksjoner som vi vet er veldig annerledes enn x86. noen av de heter det samme, slik som add, men her er det add med tre argumenter. Disse instruksjonene vil det heller ikke gå ann å kjøre på en datamaskin med x86 struktur.

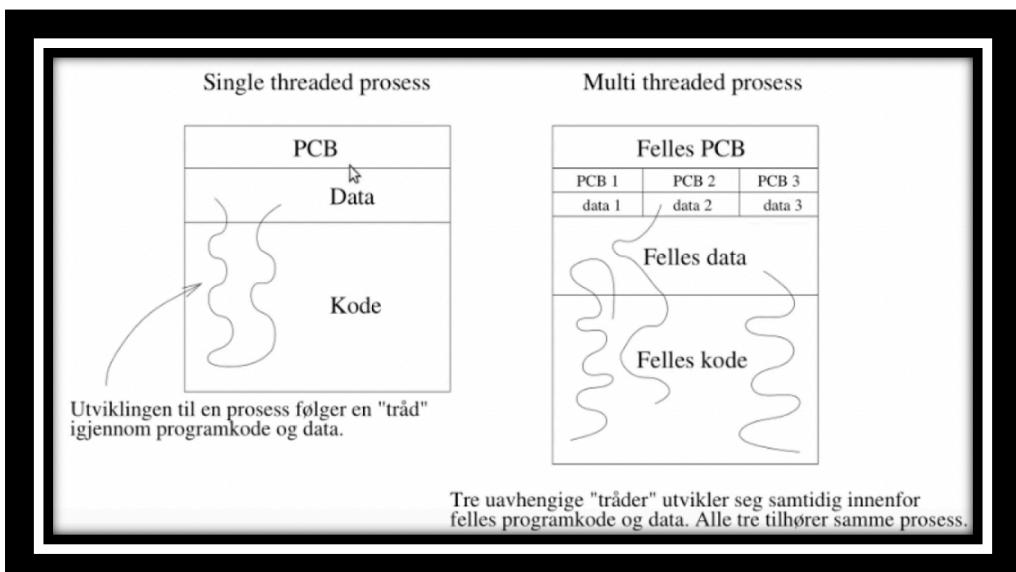
```

sum:
.LFB0:
    .cfi_startproc
    sub    sp, sp, #16
    .cfi_offset w0, 16
    str   w0, [sp, 8]
    str   w0, [sp, 12]
    b     .L2
.L3:
    ldr   w1, [sp, 8]
    ldr   w0, [sp, 12]
    add   w0, w1, w0
    str   w0, [sp, 8]
    ldr   w0, [sp, 12]
    add   w0, w0, 1
    str   w0, [sp, 12]
.L2:
    ldr   w0, [sp, 12]
    cmp   w0, 3
    ble   .L3

```

Threads (tråder)

Threads er en form for lettvektsprosesser eller små deler av en større prosess. En tråd deler ressurser (for eksempel minne og CPU-tid) med andre tråder i samme prosess og kan utføre oppgaver parallelt med andre tråder. Tråder brukes ofte for å økte ytelsen til et program ved å utnytte flertråding. Dette betyr at programmet kan utføre flere oppgaver samtidig i stedet for å utføre oppgaver i rekkefølge. Tråder kan også brukes til å øke programvarens responsivitet ved å la bakgrunnsoppgaver og brukerinteraksjon foregå parallelt.



THREADS VS PROSESSER

To alternative:

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

Tradisjonell prosess: kun en kokk jobber i et kjøkken og lager kun en porsjon av gangen.

Ønsker man å lage flere porsjoner samtidig, så må man lage flere kjøkken. En kokk som går fra kjøkken til kjøkken (multitasking) eller en kokk i hvert kjøkken (SMP, symmetric multi processing). **Med tråder:** flere kokker kan jobbe i samme kjøkken samtidig og lage flere porsjoner på en gang.

Definisjoner av tråder: den sammenhengende rekken av hendelser/instruksjoner som utføres når et program kjøres. «tråden» som følges når et program utføres og som går fra instruksjon til instruksjon.

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

FORDELER MED THREADS

- **Ressursdeling:** flere tråder eksisterer innenfor samme prosess. Deler på kode, data og delvis PCB
- **Respons:** interaktive applikasjoner kan ha en tråde med høy prioritet som kommuniserer med brukere og lavprioritettråder som gjør grovarbeid.
- **Effektivitet:** tar mindre tid å lage nye threads og mindre tid å context-switche mellom threads. Kan ta 30x så lang tid å lage en ny prosess som å lage en ny thread. Context-switch kan ta 5x så lang tid.
- **Multiprosessor:** hver tråd kan tildeles en egen CPU.

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

JAVA-THREADS OG SCHEDULING, VARIABLER OG TRÅDER

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

- start() → allokerer minne, stack etc. Og kaller run()
- run() → her utføres jobben tråden skal gjøres
- yield() → tråden gir fra seg CPU-en
- setPriority() → setter thread-prioritet. Min = 1, max = 10, default = 5.
- sleep() → tråden sover i ms millisekunder

SCHEDULING

Den opprinnelige måten eller JDK 1.1 så har man såkalt Green Threads hvor Java betraktes av OS som en prosess og JVM schedulerer trådene selv. Det var et ganske rart opplegg fordi da måtte man bruke yield for å gi fra seg CPU. Men det som er standarden er Native Threads: Java-tråder scheduleres av OS, slik at de kjører uavhengig av hverandre og samtidig.

Og da kan du sette i gang ti tråder også forventer du da at operativsystemet schedulerer mellom de, sånn at hvis du har ti CPU-er, får de en CPU hver. Hvis ikke multitasker operativsystemet.

Det går ikke klart frem av spesifikasjonene for JVM (Java Virtual Machine) hvordan prioritet skal implementeres og her er det forskjeller mellom Linux og Windows. Dette er et annet eksempel på at Java ikke er plattformuavhengig.

JAVA PÅ LINUX

På Linux VM-ene skal vi kunne installere jdk, kompilere og kjøre:

```
$ sudo apt-get install default-jdk
$ emacs Calc.java&

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

root@34beb5c52029:/# java -version
openjdk version "11.0.6" 2020-01-14

ubuntu16.04:~/$ java -version
openjdk version "1.8.0_121"
```

VARIABLER

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

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

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

Her kan vi se for en Java prosess som inneholder to tråder. Og id er en egen variabel for hver tråd som stårer, 1 i første, 2 i andre. Felles variablene, static count, her kan brukes for å telle tråder, og man kan f eks bruke en løkke.

Program	Felles data count
s-data id = 1	s2-data id = 2
s	s2

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

LINUX

10.2 DOCKER COMPOSE OG DOCKER-COMPOSE.YAML

Vi har sett at Dockerfile er en ryddigere og mer systematisk måte å bygge containere på. Der kan man definere alt man ønsker skal være med når man starter containeren, som hvilke programvare som skal være installert, hvilke filer som skal kopieres inn og så videre. Dette er et bedre alternativ enn å starte en container, installere det som trengs av programvare og innhold og så lagre denne containeren som et image og senere bruke denne. Da er det vanskeligere å gjøre endringer, vanskeligere å huske hva containeren egentlig inneholder og generelt vanskeligere å gjenskape det samme imaget med noen endringer til å bruke i andre sammenhenger.

Docker compose med den tilhørende docker-compose.yaml filen er en metode som gjør noe av den samme forenklingen for å kjøre containere som Dockerfile gjør for å bygge containere. Ofte trenger man å legge til mange flagg og oppsjetter når man starter en container og dette kan gi lange og uoversiktlige docker container run-kommandoer. Man kan gjøre dette på en mye mer ryddig og systematisk måte ved å definere alt som skal skje når man starter en container i en docker-compose.yaml fil.

Man kan i en slik fil også starte å velge å starte flere samtidige containere som skal samarbeide om å gi den tjenesten man ønsker. For eksempel kan man med Docker compose samtidig starte både en webserver og en database-server som webserveren henter dataene sine fra. Generelt kan man bruke dette til å sette opp mange forskjellig typer oppsett av samtidige containere på en ryddig og oversiktig måte. Dermed er det også enkelt å endre på konfigurasjonen og stoppe og starte hele clusteret av containere for å få alt til å virke som man ønsker.

YAML sto opprinnelig for "Yet Another Markup Language" og er som XML et maskinlesbart format som også er inspirert av Python i den forstand at riktig innrykk i teksten er viktig og det fører til feilmeldinger om dette ikke er riktig definert. Derfor må man være svært nøyne med innrykk/antall mellomrom og også med å ha mellomrom på riktige steder. Dette gir noe av de samme syntaks-

problemene som ved bash-scripting, derfor er det også her en god strategi å sakte bygge opp en docker-compose.yaml fil og teste hver gang man gjør endringer.

10.2 DOCKER COMPOSE hello-world

En docker-compose.yaml som starter en hello-world container kan se slik ut:

```
version: '3.0' # Yaml-versjon
services:
  hello:
    image: hello-world:latest
```

Første linje forteller hvilken Yaml-versjon som skal brukes og deretter listes alle services som skal være med. I dette er det kun en som vi gir navnet 'hello'. Deretter følger hvilket image som skal brukes. Dermed er man klar til å kjøre hello-world containeren med:

```
# docker-compose up
Starting hello_hello_1 ... done
Attaching to hello_hello_1
hello_1 | 
hello_1 | Hello from Docker!
```

Hvis man her ikke marker at 'hello' er en av tjeneste og skriver filen slik:

```
version: '3.0' # Yaml-versjon
services:
  hello:
    image: hello-world:latest
```

får man med en gang en feilmelding:

```
# docker-compose up
ERROR: In file './docker-compose.yaml', service 'image' must be a mapping not a string.
```

10.3 DOCKER COMPOSE NGINX

Hvis man ønsker å starte en nginx-container kan man bruke følgende yaml-fil:

```
version: '3.1'
services:
  nginx:
    image: nginx:latest
```

Og starte tjenesten med:

```
# docker-compose up -d
Creating network "test_default" with the default driver
Creating test_nginx_1 ... done
# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                 NAMES
df21da9d3c67        nginx:latest       "/docker-entrypoint."   5 seconds ago      Up 4 seconds          80/tcp                test_nginx_1
```

Hvis dette er første gang man laster ned nginx-imaget, vil det først lastes ned på samme måte som når man kjører `docker run nginx`. Deretter kan man stoppe det hele med:

```
# docker-compose down
Stopping test_nginx_1 ... done
Removing test_nginx_1 ... done
Removing network test_default
```

Docker-compose rydder opp etter seg ved å fjerne containeren som ble kjørt.

Hvis man ønsker at port 80 som nginx default bruker som source port skal vises som port 8080 på host'en som kjører containeren, kan man gjøre det ved å definere følgende i yaml-filen:

```
version: '3.1'
services:
  nginx:
    image: nginx:latest
    ports:
      - 8080:80
```

Generelt kan alle parametre og opsjoner man kan gi til `docker container run` defineres i yaml-filen. Deretter kan man starte nginx:

```
# docker-compose up -d
Starting test_nginx_1 ... done
root@os800:~/test# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9b37c7a48e8 nginx:latest "/docker-entrypoint." 59 seconds ago Up 10 seconds 0.0.0.0:8080->80/tcp test_nginx_1
```

og man vil kunne få en hilsen fra nginx:

```
# curl localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

Videre kan man på en enkel måte definere en mappe på host'en som nginx skal hente sine web-sider fra ved å legge til følgende rett under ports: i yaml-filen:

```
volumes:
- ./innhold:/usr/share/nginx/html:ro
```

Dette gjør at man nå vil få opp innhold/index.html filen på host'en når man starter nginx.

10.4 TJENESTER MED FLERE SAMTIDIGE CONTAINERE

Docker Compose er et kraftig verktøy som lar deg sette opp flere containere som jobber sammen for å gi en tjeneste. Hvis du definerer flere tjenester i samme Docker Compose-fil, vil det også opprette et privat nettverk der containerne kan kommunisere med hverandre. Dette lar deg enkelt sette opp en database og en webserver som kan kommunisere med hverandre på en sikker måte.

Følgende yaml-fil definerer to containere som leverer forskjellig innhold på henholdsvis port 8080 og 8081:

```

version: '3'
services:
  nginx:
    image: nginx:latest
    ports:
      - 8080:80
    volumes:
      - ./innhold:/usr/share/nginx/html:ro
  nginx2:
    image: nginx:latest
    ports:
      - 8081:80
    volumes:
      - ./innhold2:/usr/share/nginx/html:ro

```

Dermed kan man starte begge containerne samtidig og se at de virker som de skal. Og stoppe begge etterpå.

```

# docker-compose up -d
Starting test_nginx_1 ... done
Starting test_nginx2_1 ... done
# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f507d9c369c5 nginx:latest "/docker-entrypoint." About a minute ago Up 4 seconds 0.0.0.0:8081->80/tcp test_nginx2_1
fe8b6ca0922d nginx:latest "/docker-entrypoint." About a minute ago Up 4 seconds 0.0.0.0:8080->80/tcp test_nginx_1
fb87e1b854ba ubuntu "tail -f /dev/null" 4 weeks ago Up 4 weeks
# curl localhost:8080
Container innhold!
# curl localhost:8081
Container innhold 2!
# docker-compose down
Stopping test_nginx2_1 ... done
Stopping test_nginx_1 ... done
Removing test_nginx2_1 ...
Removing test_nginx_1 ...

```

Om man går inn i den ene nginx og installerer ping, vil man kunne se at man kan kommunisere over det lokale private nettverket med den andre containeren ved å bruke navnet som er definert i yaml-filen:

```

# docker exec -it f2 bash
root@f2d45f6e9bf5:/# ping nginx2
PING nginx2 (192.168.32.3) 56(84) bytes of data.
64 bytes from test_nginx2_1.test_default (192.168.32.3): icmp_seq=1 ttl=64 time=0.137 ms

```

De to containerene har IPer 192.168.32.3 og 192.168.32.2 og kan kommunisere med hverandre som på andre nettverk. Dette gjør det mulig å sette opp relistiske systemer, ikke minst for å teste kode som man utvikler for dette service-scenariet.

10.5 DOCKER-COMPOSE BUILD

Man kan kombinere docker-compose med en eller flere Dockerfiles ved å spesifisere en mappe der den tilhørende Dockerfile ligger ved å angi 'build' istedet for image i yaml-filen:

```
version: '3'  
services:  
  nginx:  
    build: ./nginx  
    ports:  
      - 8080:80
```

Dermed vil docker-compose prøve å bygge et image fra Dockerfile i mappen ./nginx og starte en container med det image't som er resultatet av denne byggingen. Og man kan sette opp en oversiktlig mappestruktur som definerer alle containerene som er med i et compose-prosjekt.

