

NOTATER UKE 12 –

Prosesser og operativsystem-arkitektur

Funksjonen (**systemkall**) `getppid()` i C-programmeringsspråket brukes til å returnere prosess-IDen til forelderprosessen til den gjeldende prosessen. «`ppid`» står for «parent process ID». Funksjonen tar ikke noen argumenter og returnerer en heltallsverdi som representerer prosess-IDen til foreldreprosessen.

For å utføre systemkallet med funksjonen må `#include` være med riktig på topp:

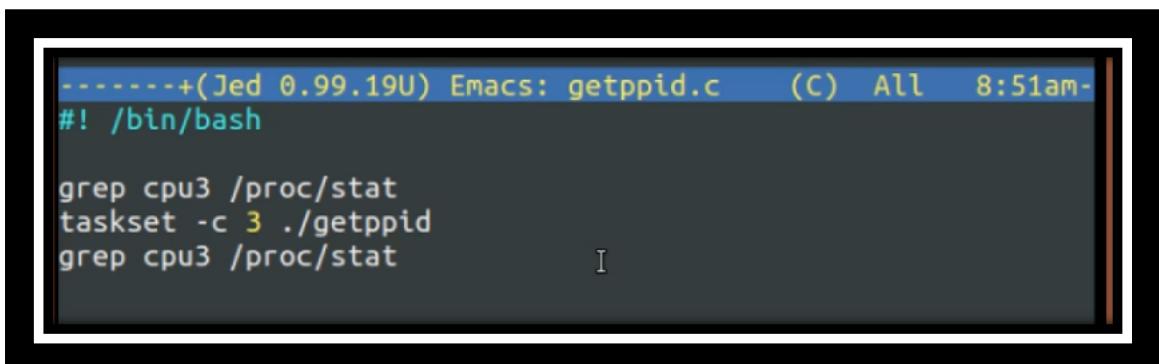


```
haugerud@lap: ~/os9 61x34
F10 key ==> File Edit Mode Search Buffers Windows
#include<unistd.h>

int main(void) {
    int i;
    for (i=0; i<100000000; i++) {
        getppid();
    }
    return(0);
}
```

MEN! Hvor mye av dette foregår i user mode og hvor mye i kernel mode?

Hårek åpner opp et annet script som heter `sys.sh`. Skriptet prøver å telle opp antall ticks i user mode og kernel mode. I dette tilfellet spør Hårek om CPU 3 i `/proc/stat` fila hvor man får informasjon om antall ticks (tidsenheter) brukt av brukerprosesser og systemprosesser, antall kontekstbytter, antall avbrudd behandlet osv.



```
-----+(Jed 0.99.19U) Emacs: getppid.c (C) All 8:51am-
#! /bin/bash

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

Husk at ticks/jiffies er den minste tidsenheten, oftest hundredelssekund. En vanlig regnejobb som kjører bruker vanligvis 100 ticks per sekund. Den vil kjøre i usermode hele tiden.

```
haugerud@lap:~/os9$ ./regn
./regn : regner....
./regn, resultat: 140450265000
Real:2,407 User:2,407 System:0,000 100,00%
```

Under kjører Hårek sys.sh skriptet og ser at to felt øker. Hvis man ikke vet hva det betyr kan man søke opp manualsiden for proc med kommandoen «man proc».

```
haugerud@lap:~/os9$ gcc -o getppid getppid.c
haugerud@lap:~/os9$ ./getppid
haugerud@lap:~/os9$ ./sys.sh
cpu3 2603913 927 1143992 34368696 15110 0 15921 0 0 0
cpu3 2604255 927 1144126 34368696 15110 0 15921 0 0 0
```

Under er det gjort cat /proc/stat og på høyresiden er det åpnet opp manualsiden:

State	Description
user	(1) Time spent in user mode.
nice	(2) Time spent in user mode with low priority (nice).
system	(3) Time spent in system mode.
idle	(4) Time spent in the idle task. This value should be USER_HZ times the second entry in the /proc/uptime pseudo-file.
wait	(since Linux 2.5.41) (5) Time waiting for I/O to complete. This value is not

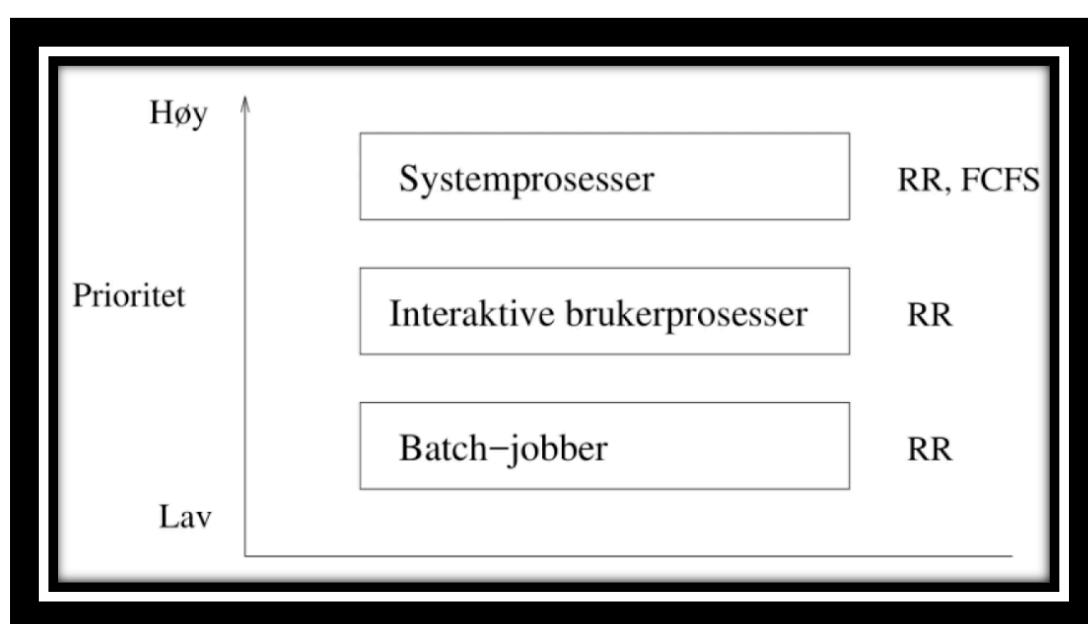
PRIORITET OG SCHEDULING ALGORITMER

Prioritet mellom prosesser refererer til en mekanisme i operativsystemet som bestemmer rekkefølgen for hvordan prosesser og oppgaver blir utført. Prioritetsnivået til en prosess avgjør hvor mye CPU-tid den får tildelt i forhold til andre prosesser som kjører på systemet. Prosesser med høyere prioritet vil få tildelt mer CPU-tid enn de med lavere prioritet.

Disse nivåene er viktige for å sørge for at systemet fungerer jevnt og effektivt. Hvis en prosess får for mye CPU-tid kan det føre til at andre prosesser blir tregere eller krasjer. På en annen side kan en prosess som får for lite CPU-tid føre til at systemet blir tregt eller uresponsivt.

Prioritetsmekanismen gjør det også mulig for systemadministrator å styre systemressursene på en mer effektiv måte. For eksempel kan man prioritere en viktig applikasjon eller tjeneste høyere enn mindre viktige bakgrunns-oppgaver.

I Linux er det 140 prioriteringsnivåer, der 0 er høyest prioritet og 139 er laveste. Dette gir systemadministratorer en finjustering for å optimalisere ressursbruk og systemytelse. Interaktive brukerprosesser vil få mer tid, men de bruker ikke så mye CPU, så med engang de er ferdige går de ut igjen.



4 VANLIGE SCHEDULING-ALGORITMER

1. **Round-robin:** dette er en vanlig algoritme som gir like mye tid til hver prosess i en fastsatt rekkefølge. Hvis en prosess ikke er ferdig når tiden er ute, blir den satt bakerst i køen og vil få en annen sjanse senere.
2. **FCFS (First-Come, First-Served):** denne algoritmen prioritiserer prosessene basert på når de ankommer køen. Prosessen som først ankommer vil bli behandlet først, og de andre prosessene vil bli behandlet i samme rekkefølge som de ankom.

3. **FIFO (First-In, First-Out):** dette er en enkel algoritme der prosessene behandles i den rekkefølgen de ankom køen. Prosessene blir satt bakerst i køen etter hvert som de fullføres.
4. **SJF (Shortest Job First):** denne algoritmen prioriterer prosessene basert på hvor kort tid de vil ta å fullføre. Den prosessen som tar kortest tid å fullføre, vil bli behandlet først, og de andre prosessene vil bli behandlet i den rekkefølgen de trenger minst tid.

NICE

Nice er en Linux-kommando som brukes til å endre prioriteringsnivået til en prosess. Ved å endre prioriteringsnivået kan man justere hvor mye prosessorkraft en prosess skal ha tilgjengelig. Jo høyere prioriteringsnivå, jo mer ressurser vil prosessen ha til rådighet, og vice versa. Kommandoen fungerer ved å legge til et prioriteringsnivå mellom -20 (høyest prioritet) og 19 (lavest prioritet) til en allerede kjørende prosess eller til en ny prosess som startes.

Hvordan kan vi endre prioritering selv på egne prosesser? Hvis det ikke er vi som eksplisitt sier ifra vil operativsystemet sette dette selv:

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

For å endre nice verdi bruker man «renice» og da må man skrive prosessID, som man finner ved å taste top:

```
haugerud@Lap:~/os9$ sudo renice -5 25046  
25046 (process ID) old priority 5, new pri  
ority -5
```

REAL-TIME OG BRUKERPROSESSER

Vi kan merke at i bildet under, eller når vi taster top, så er det noen prosesser som ikke har prioritet skrevet til seg, men kun «rt». Real-time prosesser er en type program som krever svært rask responsid og kjøres ofte på dedikert maskinvare. Disse prosessene blir prioritert høyt og kjøres ofte i en egen kategori for å sikre at de får ressursene de trenger til å kjøre jevnt og raskt.

Brukerprosesser er vanlige programmer som kjøres på en datamaskin eller et annet system. Disse kan være alt fra tekstbehandlingsprogrammer og nettlesere til mer komplekse applikasjoner som databaser og servere. Brukerprosesser har vanligvis ikke de samme strenge kravene til responstid som real-time prosesser, men de kan likevel kreve at maskinen gir dem en viss mengde ressurser for å fungere skikkelig.

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND	P
20536	haugerud	20	0	47,344g	629888	331320	S	64,1	1,9	56:31.23	obs	4
20271	haugerud	20	0	6697356	797924	256540	S	63,1	2,4	47:30.68	zoom	2
25046	haugerud	15	-5	14436	3460	3288	R	62,8	0,0	2:01.95	regn	3
25087	haugerud	20	0	14436	3532	3288	R	20,6	0,0	1:40.47	regn	3
2856	haugerud	20	0	4754712	854728	104676	S	16,6	2,6	671:46.80	gnome-shell	4
25066	haugerud	21	1	14436	3536	3288	R	16,6	0,0	2:05.74	regn	3
3554	haugerud	20	0	2597504	20624	14976	S	10,0	0,1	151:15.84	pulseaudio	1
1847	root	20	0	1465576	780916	125936	S	6,6	2,4	907:25.07	Xorg	1
1925	root	-51	0	0	0	0	S	3,3	0,0	46:27.65	irq/142-nv+	0
14587	haugerud	20	0	719508	64096	36372	S	1,0	0,2	0:15.91	terminator	0
23743	haugerud	20	0	43464	4012	3364	R	0,3	0,0	0:07.61	top	6
1	root	20	0	226032	9712	6668	S	0,0	0,0	0:07.25	systemd	6
2	root	20	0	0	0	0	S	0,0	0,0	0:00.31	kthreadd	4
4	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/0:+	0
6	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	mm_percpu_+ 1	
7	root	20	0	0	0	0	S	0,0	0,0	0:17.01	ksoftirqd/0	0
8	root	20	0	0	0	0	I	0,0	0,0	4:01.88	rcu_sched	0
9	root	20	0	0	0	0	I	0,0	0,0	0:00.00	rcu_bh	0
10	root	rt	0	0	0	0	S	0,0	0,0	0:00.90	migration/0	0
11	root	rt	0	0	0	0	S	0,0	0,0	0:00.71	steal/0	0

PRIORITETER I WINDOWS

I Windows brukes en prioritatingsalgoritme som kalles Priority Boost. Denne bestemmer prioriteringen til prosesser basert på en kombinasjon av flere faktorer, som prosessens bruk av CPU og I/O-enheter, hvor lenge prosessen har vært inaktiv, og om prosessen er en systemprosess eller en brukerprosess.

Windows har en rekke forskjellige prioritetsnivåer som

Realtime

High

Above Normal

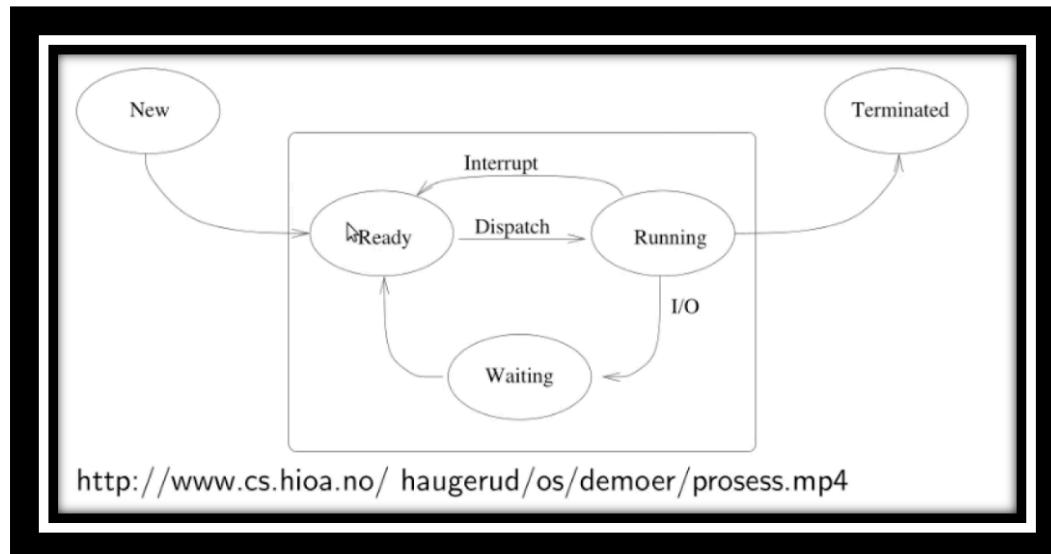
Normal

Below Normal

Low

Nivået påvirker hvor mye CPU-tid en prosess vil få tildelt sammenliknet med andre prosesser. For eksempel vil en prosess med høy prioritet (High) få tildelt mer CPU-tid enn en med Normal.

PROSESSFORLØP



COMPLETELY FAIR 2.6 LINUX-KJERNE

Den 2.6 Linux-kjernen introduserte Completely Fair Scheduler (CFS) i 2007 som en ny måte å planlegge prosessene på. CFS erstattet den eldre O(1) scheduleren som hadde vært standard i tidligere versjoner av kjernen. CFS har som mål å tildele like mye CPU-tid til hver prosess, slik at det blir en jevn fordeling av ressurser mellom prosessene. Dette er en forbedring fra den eldre scheduleren som ikke alltid var like rettferdig i tildelingen av CPU-tid, spesielt under høyt belastede situasjoner. CFS er fortsatt standard i moderne Linux-kjerner.

SCHEDULINGBEGREPER

En enqueueer er ansvarlig for å legge til oppgaver i en kø eller liste, mens en dispatcher er ansvarlig for å plukke oppgaver fra køen eller listen og utføre dem. På den måten sørger enqueueuer og dispatcher for at programmet utfører oppgavene i en ordnet og effektiv måte.

Enqueueer	<ul style="list-style-type: none"> • Legger i kø • Beregner prioritet
Dispatcher	<ul style="list-style-type: none"> • Velger prosess fra READY LIST; liste med prosesser som er klare til å kjøre

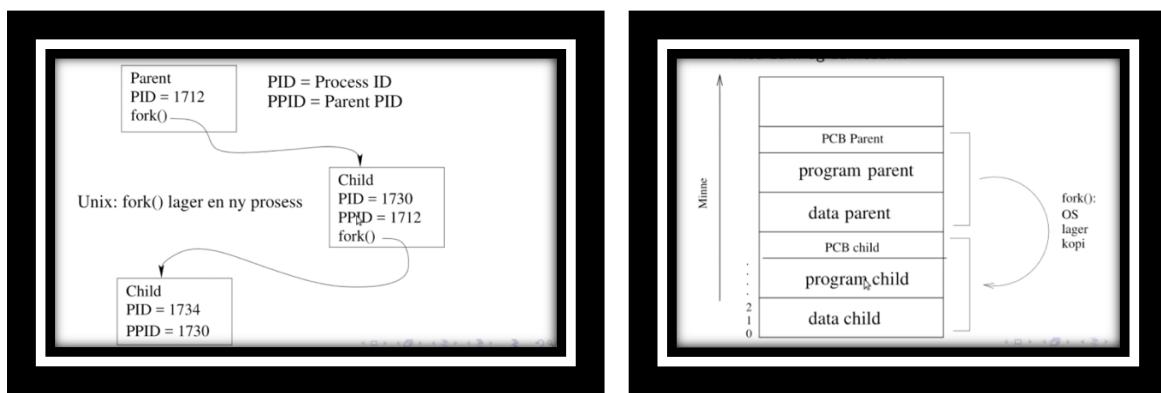
LAGE EN NY PROSESS – FORK()

For å lage en prosess i Unix/Linux-basert operativsystem bruker man systemkallet fork().

Dette vil lage en kopi av den nåværende prosessen og kjøre en separat instans av programmet. Man kan også bruke funksjonen exec() for å erstatte den nye prosessen med et annet program.

Linux-prosesser lager på denne måten et hierarki av prosesser med barn og barnebarn. I Linux-verden er det alltid en prosess som lager en annen. Den prosessen kalles en forelder-prosess, og parent-prosess lager en child-prosess. I prosessverdenen er det bare en forelder til ett eller flere barn.

Hvis parent blir drept uten at barna blir drept først får man såkalte zombie-prosesser.



Det som egentlig skjer i bildet til høyre er at det er faktisk ikke ALT som blir kopiert over. Det er mer som copy-on-demand slik at hvis child trenger noe av data av parent, så kopieres de over. Men det meste av strukturen kopieres, men effektivitetsmessig tar ikke clone med alt. I child kan man ha en if-test: if (jeg er et barn, start en annen prosess).

WINDOWS CREATEPROCESS

Windows CreateProcess er en funksjon i Windows-operativsystem som lar deg opprette en ny prosess (program) fra en eksisterende prosess. Den nye prosessen kjører uavhengig av den eksisterende prosessen, og kan utføre forskjellige oppgaver eller utføre forskjellige programmeringsoppgaver. CreateProcess funksjonen tar inn flere parameter, inkludert navnet på programmet du ønsker å kjøre og argumenter til programmet. Når du kaller CreateProcess funksjonen, vil operativsystemet opprette en ny prosess for programmet og starte utførelsen av programmet i denne prosessen.

AVSLUTTE PROSESSER

I Linux kan man avslutte en prosess på forskjellige måter, avhengig av hva slags verktøy og tilgangsnivå du har. Noen av måtene å avslutte en prosess på:

1. «kill»:
 - a. Finn PID for prosessen som skal avsluttes med kommando «ps aux»
 - b. Skriv «kill <PID>» i terminalen
 - c. Hvis man vil tvinge avslutningen av en prosess, kan man bruke «kill -9 <PID>» i stedet. Dette sender SIGKILL-signal til prosessen som avslutter den med engang
2. «pkill»:
 - a. Skrive «pkill <prosessnavn>»
 - b. Avslutter alle prosesser med dette navnet
3. System Monitor:
 - a. Åpne system monitor
 - b. Finn prosessen du vil avslutte, høyreklikk på den og velg «end process»

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

SIGNALER

Et signal kan referere til en form for beskjed eller instruksjon som sendes til en prosess eller et program, vanligvis fra OS eller en annen prosess. Signaler kan brukes til å varsle en prosess om en hendelse, for eksempel når brukeren vil avslutte programmet eller når en feil oppstår, eller til å gi instruksjoner til en prosess, for eksempel å avslutte en prosess eller starte en.

I Linux og andre Unix-baserte OS kan signaler sendes til en prosess ved hjelp av «kill»-kommandoen eller gjennom systemkall som kill() eller raise(). Det finnes forskjellige typer signaler for eksempel:

- **SIGTERM:** Dette er det vanligste signalet som brukes til å avslutte en prosess. Det gir en prosess muligheten til å rydde opp etter seg selv og avslutte på en kontrollert måte.
- **SIGKILL:** Dette signalet tvinger en prosess til å avslutte umiddelbart, uten mulighet for å rydde opp etter seg selv. Det brukes vanligvis som en siste utvei når en prosess ikke kan avsluttes på en annen måte.
- **SIGINT:** dette signalet sendes når brukeren trykker på Ctrl + C i en terminal. Det varsler en prosess om at den skal avslutte på en kontrollert måte.
- **SIGSTOP og SIGCONT:** disse signalene brukes til å midlertidig stoppe og gjenoppta en prosess, henholdsvis.

- Prosesser kan kommunisere med hverandre ved hjelp av signaler
- En bruker kan sende et signal til en prosess, for eksempel CTRL-C
- En prosess kan med noen unntak velge hvordan et signal skal behandles

SIGNALER OG TRAP

Hvis man kjører en prosess og taster Control + C, så dreper man prosessen. Vi har også sett at man kan bruke kommandoen kill for å drepe prosessen. Vi skal se på hvordan prosesser kan behandle denne type signaler som sendes til dem, og det er det vi ser i shell-skriptet her. Under er det vist noen definisjoner fra en Linux-headerfile som definerer forskjellige type killsignaler. Dette er da signaler som kan sendes til prosesser.

Det er disse trap instruksjonene under som tar imot signalet

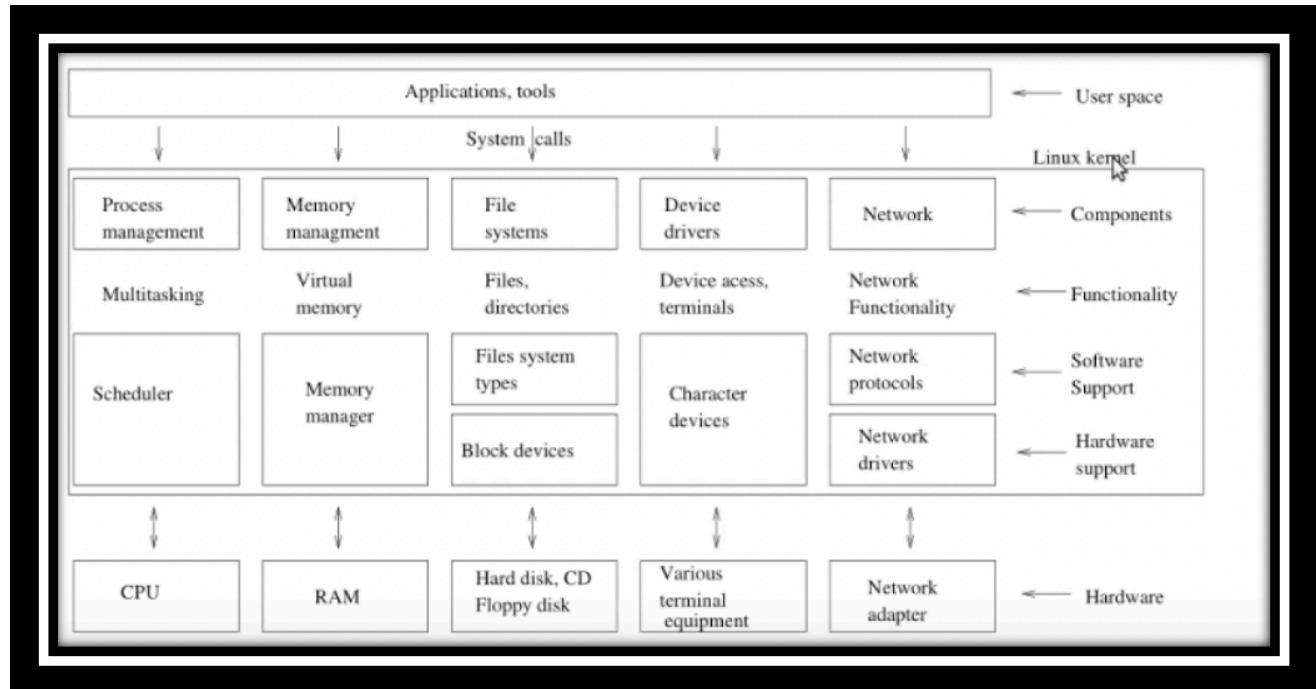
```
F10 key ==> File Edit Search Buffers Windows System
#! /bin/bash

# Definisjoner fra
# /usr/src/linux-headers-3.13.0-106/arch/x86/include/uapi/asm/S

#define SIGHUP      1      /* Hangup (POSIX). */
#define SIGINT      2      /* Interrupt (ANSI). */
#define SIGKILL     9      /* Kill, unblockable (POSIX). */
#define SIGTERM    15      /* Termination (ANSI). */
#define SIGSTP     20      /* Keyboard stop (POSIX). */

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

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

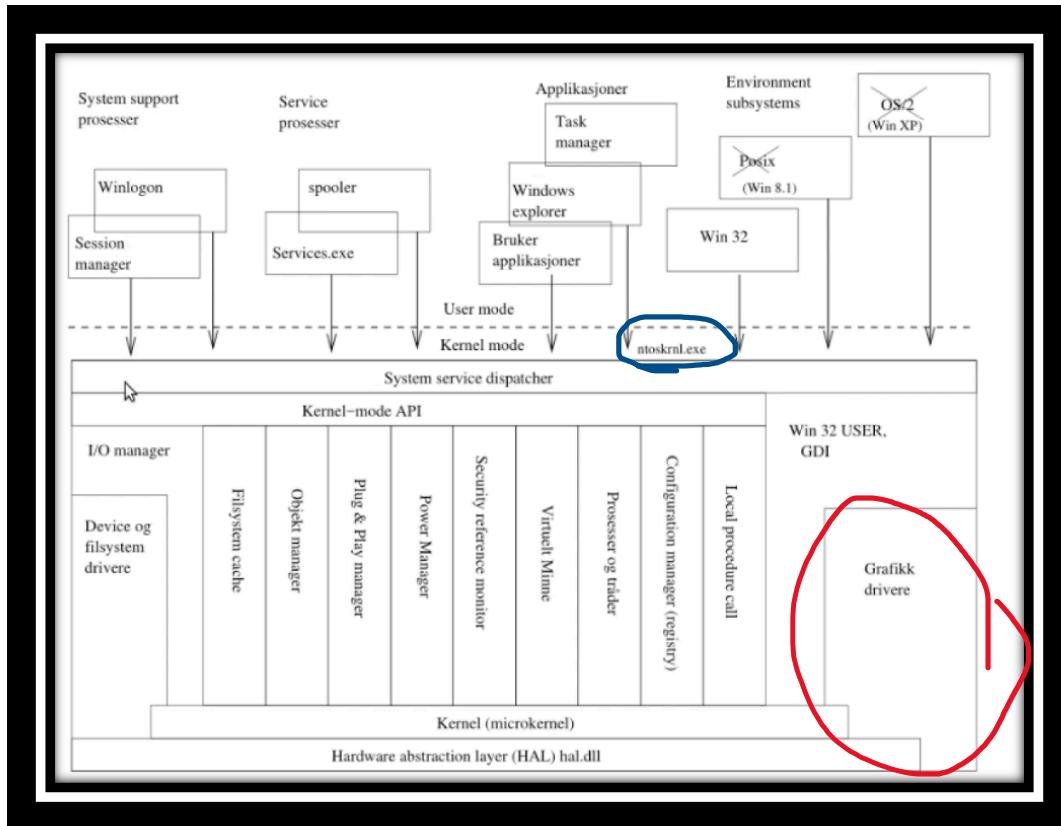


Det som er inni boksen, er Linux-kjernen. Alle applikasjoner og verktøy er i user space og de gjør systemkall til Linux kjernen. Vi kan se det vagt er delt opp i fem hovedmoduler. Det første som vi har sett på er alt med prosess-management og multitasking, som er relatert til hardware CPU. Neste vi skal gå gjennom er RAM og cache, hvor cache ikke styres direkte av operativsystemet, det er det hardware som gjør.

WINDOWS ARKITEKTUR

Masse som likner, men det er noen forskjeller. Man kan se at alt over kjører i user mode og ber da OS-kjerne om hjelp gjennom API, gjennom systemkall. Med liten skrift (se tegning), er ntoskernel.exe som man finner i en Windows maskin, og er hele programmet (boksen nede). Windows er en såkalt monolittisk kjerne som refererer til en type programvarearkitektur der hele programmet er bygd som en enkelt enhet. Dette betyr at alle deler av programmet er kombinert til en enkelt fil eller modul. I monolittiske kjerner kan alle deler av kjerner snakke med alle deler av kjernen.

En viktigst forskjell på Linux og Windows er at vi kan se nede at grafikken er en del av Windows kjernen (rød). I Linux kjører grafikk i user mode



LINUX – Docker og Docker Hub, shell script ytelse

Vi skal gjøre oppgave 12 sammen som går ut på å sammenlikne hastigheten til ulike programmeringsspråk. Vi skal sette opp en Docker-container som installerer alle programmeringsmiljøene som er nødvendige for å kompilere og kjøre disse programmene.

I denne første omgangen skal vi bare lage Dockercontaineren for å kjøre dette iterativt.

Man trenger først disse verktøyene:

```
apt-get install build-essential  
apt-get install default-jdk  
apt-get install python3
```

- Build-essential er for gcc, for å kompilere C
- Jdk er for Java
- Python for Python

Programmet som skal brukes ligger på GitHub hvor et repository heter sum.git:

```
git clone https://github.com/hauierud/sum.git
```

Det er veldig nyttig å kunne bruke Git fra kommandolinjen fordi da kan man automatisere alt som har med å pushe og pulle kode.

Husk at man en gang i blant må kjøre kode for å slette alle docker og image slik at man har clean slates. Kan sjekke lagring med (her 1,5 GB):

```
root@osG70:~# df -h  
Filesystem      Size  Used Avail Use% Mounted on  
udev            476M    0  476M   0% /dev  
tmpfs           98M   11M  87M  11% /run  
/dev/xvda1       8,2G  6,4G  1,5G  82% /  
tmpfs           486M    0  486M   0% /dev/shm  
tmpfs           5,0M    0  5,0M   0% /run/lock  
tmpfs           486M    0  486M   0% /sys/fs/cgroup  
tmpfs           98M    0   98M   0% /run/user/1001
```

Under klones det fra Git. Og da lages det en mappe sum som man kan gå inn og se med.

```
[root@osG70:~# git clone https://github.com/haugerud/sum.git
Cloning into 'sum'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 15 (delta 4), reused 12 (delta 4), pack-reused 0
Unpacking objects: 100% (15/15), done.
```

Sum.bash skriptet regner ut en sum 50k ganger og de andre programmene regner ut denne arbeidsoppgaven x ganger, og vi skal finne ut hvem som klarer flest sånne oppgaver på like lang tid. Dette må vi få til å gjøre INNE i dockercontaineren.

Under legges det inn fra det oppgaveteksten sa trengs:

```
FROM ubuntu
RUN apt-get -y update
RUN apt-get -y install build-essential
RUN apt-get -y install default-jdk
RUN apt-get -y install python3
```

I tillegg må vi installere Git og vi må gjøre en Git-clone, med linjen til repository:

```
FROM ubuntu
RUN apt-get -y update
RUN apt-get -y install build-essential
RUN apt-get -y install default-jdk
RUN apt-get -y install python3
RUN apt-get -y install git
RUN git clone https://github.com/haugerud/sum.git
```

Søker rekursivt med Control + R, og dette tar 8 min, fordi det er masse kode å kopiere inn:

```
root@osG70:~# cd sum
root@osG70:~/sum# time docker build -t sum .
```

```
FROM ubuntu
RUN apt-get -y update
RUN apt-get -y install build-essential
RUN apt-get -y install default-jdk
RUN apt-get -y install python3
RUN apt-get -y install git
RUN git clone https://github.com/haugerud/sum.git
COPY sum.sh /sum
RUN chmod 755 /sum/sum.sh
```

IMENS GJØRE ITERATIVT DET HAN ETTERPÅ SKAL GJØRE I DOCKER

Først kloner han filene og går inn i mappen:

```
rex:~$ git clone https://github.com/haugerud/sum.git
Cloning into 'sum'...
remote: Enumerating objects: 15, done.
```

Går inn i mappen og sjekker:

```
rex:~/sum$ ls -l
totalt 32
-rw----- 1 haugerud haugerud 35 mars 27 10:54 README.md
-rwx----- 1 haugerud haugerud 210 mars 27 10:54 sum.bash
-rwx----- 1 haugerud haugerud 355 mars 27 10:54 sum.c 1
-rwx----- 1 haugerud haugerud 416 mars 27 10:54 Sum.java 2
-rwx----- 1 haugerud haugerud 420 mars 27 10:54 sum0.c 3
-rwx----- 1 haugerud haugerud 276 mars 27 10:54 sum.perl 4
-rwx----- 1 haugerud haugerud 279 mars 27 10:54 sum.php 5
-rwx----- 1 haugerud haugerud 206 mars 27 10:54 sum.py 6
```

Så tar han time på **sum.bash** skriptet for å se hvor lang tid det bruker. Ca. 5.5 sekunder

```
rex:~/sum$ time ./sum.bash
Ferdig, sum: 250000000000
Real:5.534 User:5.528 System:0.004 99.96%
```

Også starter han med den første **sum.c**, kompilerer med gcc, kjører time Ca. 5.6 sekunder

```
rex:~/sum$ gcc sum.c
rex:~/sum$ time ./a.out
Ferdig 250000000000
Real:5.650 User:5.644 System:0.000 99.89%
```

sum.java må kompileres med javac først, kjører med time Ca. 5.7 sekunder

```
rex:~/sum$ javac Sum.java
rex:~/sum$ java Sum
^C
rex:~/sum$ time java Sum
Ferdig 250000000000
Real:5.731 User:6.316 System:1.072 128.91%
```

Perl er som regel installert default på ubuntu. **Sum.perl** Ca. 5.9 sekunder.

Sum.py Ca. 5.2 sekunder

```
rex:~/sum$ time ./sum.py
Ferdig!
Real:5.497 User:5.492 System:0.000 99.90%
rex:~/sum$ time ./sum.py
Ferdig! Sum: 250000000000
Real:5.520 User:5.492 System:0.012 99.71%
```

Sum.php Ca. 5.2 sekunder

```
rex:~/sum$ time php ./sum.php
Ferdig. Sum 250000000000
Real:5.303 User:5.288 System:0.004 99.79%
```

Hvilket av disse språkene tror jeg er raskest?

KODE SOM AUTOMATISERER KOMPILERING OG KJØRING MED TIME

```
TIMEFORMAT="Real:%R User:%U System:%S %P%"

gcc sum.c
javac Sum.java
echo -n "a.out "
time ./a.out
echo -n "java "
time Java Sum
echo -n "python "
time python sum.py
echo -n "bash "
time ./sum.bash
```

Også må vi endre Dockerfile slik at koden legges inn i imaget med COPY nederst:

```
FROM ubuntu
RUN apt-get -y update
RUN apt-get -y install build-essential
RUN apt-get -y install default-jdk
RUN apt-get -y install python3
RUN apt-get -y install git
RUN git clone https://github.com/haugerud/sum.git
COPY sum.sh /sum
RUN chmod 755 /sum/sum.sh
```

FORK DEMO

Under er det et C-program, med importerte biblioteker som trengs for å gjøre et fork() systemkall. Programmet skriver først ut en linje med fork demo. Også skjer systemkallet som

starter opp en child prosess som starter med den neste print-linjen. Forelderprosessen vil fortsette som skriptet etter kallet.

```

F10 key ==> File Edit Mode Search Buffers Windows Sy
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    printf("Fork demo!\n");
    fork();
    printf("Fork demo!\n");

```

Man kan enkelt til høyre se at foreldrepresessen kjører vanlig og child prosessen også skriver ut fra videre etter systemkallet.

```

haugerud@lap:~/os9/fork 64x35
F10 key ==> File Edit Mode Search Buffers Windows Sy
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    printf("Fork demo! 1\n");
    fork();
    printf("Fork demo! 2\n");
}

haugerud@lap:~/os9/fork$ cd fork/
haugerud@lap:~/os9/fork$ gcc f.c
haugerud@lap:~/os9/fork$ ./a.out
Fork demo!
Fork demo!
Fork demo!
Fork demo!
haugerud@lap:~/os9/fork$ gcc f.c
haugerud@lap:~/os9/fork$ ./a.out
Fork demo! 1
Fork demo! 2
Fork demo! 2
haugerud@lap:~/os9/fork$ 

```

FORK MED PID

```

haugerud@lap:~/os9/fork 64x35
F10 key ==> File Edit Mode Search Buffers Windows Sy
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    printf("Fork demo! 1\n");
    int pid=fork();
    printf("Fork pid=%d\n",pid);
    fork();
    printf("Fork demo! 3\n");
}

haugerud@lap:~/os9/fork$ ./a.out
Fork demo! 1
Fork pid=26726
Fork pid=0
Fork demo! 3
Fork demo! 3
Fork demo! 3
Fork demo! 3
haugerud@lap:~/os9/fork$ 

```

FORK MED PID OG IF-TEST

```

haugerud@lap:~/os9/fork 64x35
F10 key ==> File Edit Mode Search Buffers Windows Sy
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    printf("Fork demo! 1\n");
    int pid=fork();
    if(pid == 0)
    {
        printf("Jeg er child, Fork ga pid=%d \n",pid);
    }
    else
    {
        printf("Jeg er parent, Fork ga pid=%d \n",pid);
    }
    printf("Fork demo avsluttet!\n");
}

haugerud@lap:~/os9/fork$ ./a.out
Fork demo! 1
Fork pid=26726
Fork pid=0
Fork demo! 3
Fork demo! 3
Fork demo! 3
Fork demo! 3
haugerud@lap:~/os9/fork$ gcc f.c
haugerud@lap:~/os9/fork$ ./a.out
Fork demo! 1
Jeg er parent, Fork ga pid=26923
Fork demo avsluttet!
Jeg er child, Fork ga pid=0
Fork demo avsluttet!
haugerud@lap:~/os9/fork$ 

```

```

int pid = fork();      //opprettes en ny prosess med
if(pid == 0) {         //sjekker om vi er i den nye prosessen
    x                  //kode som skal kjøres i den nye prosessen
}

```

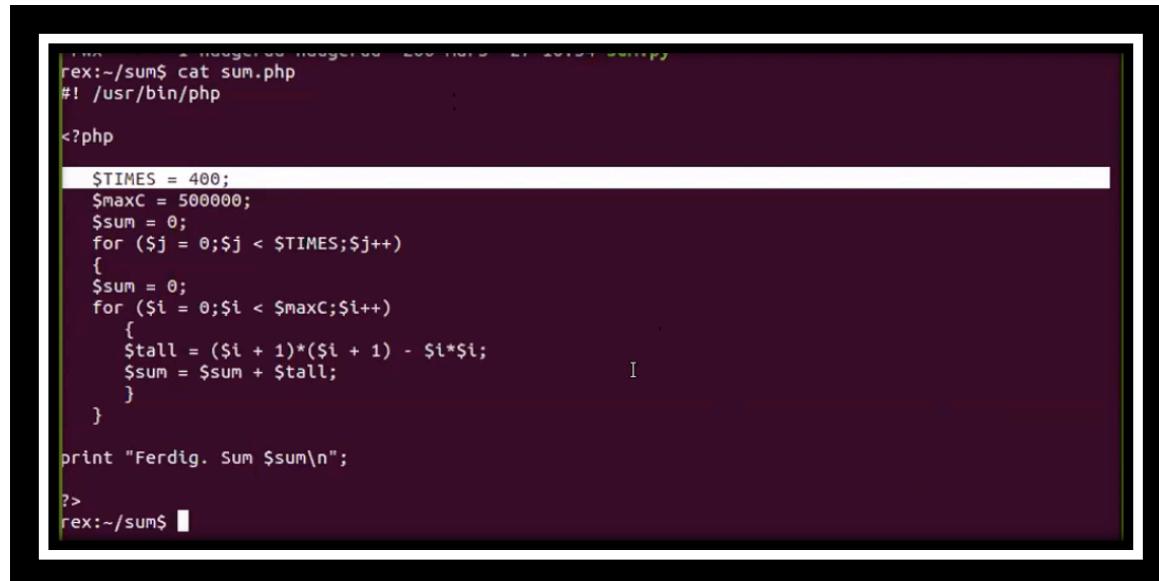
Når FORK-funksjonen kalles, opprettes en kopi av prosessen i et helt nytt adresseområde, men med samme programkode og miljøvariabler som den opprinnelige prosessen. Den nye prosessen har en unik prosess-ID og et eget adresseområde, men deler filåpninger, signalbehandling og annen tilstand med den opprinnelige prosessen.

PHP

Er en forkortelse for «Hypertext Preprocessor», og det er et populært server-side programmeringsspråk som brukes til å utvikle dynamiske websider og applikasjoner. Det er et av de mest kjente sidene for webutvikling.

PHP er spesielt godt egnet til å arbeide med webapplikasjoner som krever interaksjon med databasesystemer, og kan integreres med en rekke forskjellige databasesystemer som MySQL, PostgreSQL og Oracle. Det støtter også en rekke webprotokoller og formater, inkludert HTTP, HTML, XML og JSON.

Under viser han skriptet til php og vi ser at den har klart å kjøre koden 400 ganger.



```
rex:~/sum$ cat sum.php
#!/usr/bin/php

<?php

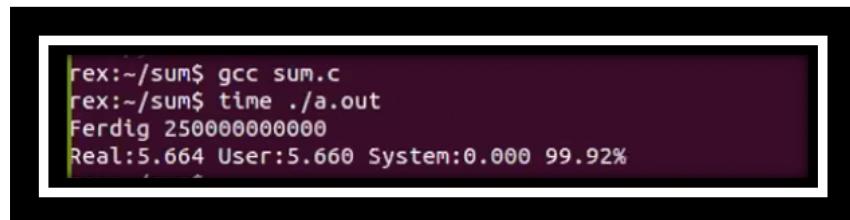
$TIMES = 400;
$maxC = 500000;
$sum = 0;
for ($j = 0;$j < $TIMES;$j++)
{
    $sum = 0;
    for ($i = 0;$i < $maxC;$i++)
    {
        $stall = ($i + 1)*($i + 1) - $i*$i;
        $sum = $sum + $stall;
    }
}
print "Ferdig. Sum $sum\n";
?>
rex:~/sum$
```

Under vises antall ganger fra alle kodene:



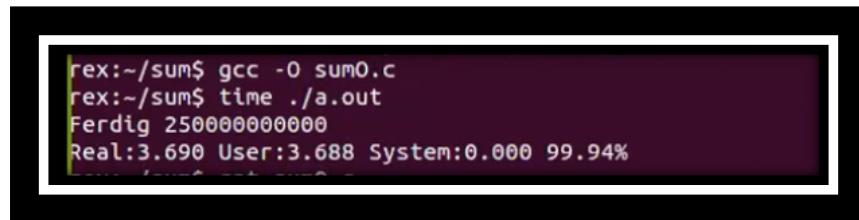
```
rex:~/sum$ grep "TIMES =" *
sum.c:    int TIMES = 4100;
Sum.java:    int TIMES = 20000;
sum0.c:    int TIMES = 28000;
sum.perl: $TIMES = 36;
sum.php: $TIMES = 400;
sum.py:TIMES = 44
```

Når vi kompilerer gcc slik, da er gcc optimalisert for å kompile raskt, ikke optimalisert for å gi rask kode.



```
rex:~/sum$ gcc sum.c
rex:~/sum$ time ./a.out
Ferdig 2500000000000
Real:5.664 User:5.660 System:0.000 99.92%
```

Derfor har han laget en sum0 hvor 0 står for optimalisert, og det er samme koden, men hvis man kompilerer den, og kjører, så får man en optimalisert ferdig kode som går raskest mulig.



```
rex:~/sum$ gcc -O sum0.c
rex:~/sum$ time ./a.out
Ferdig 2500000000000
Real:3.690 User:3.688 System:0.000 99.94%
```

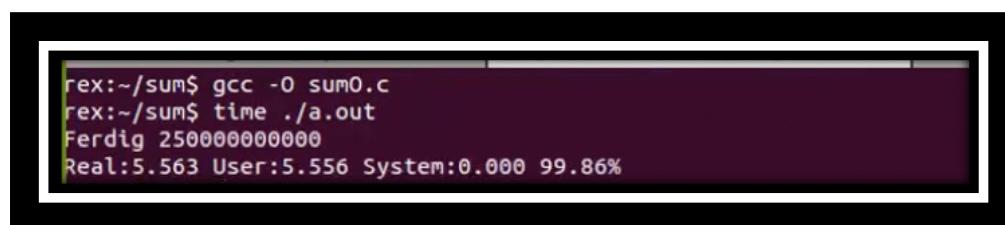
Konklusjon:

Så c er faktisk det aller raskeste språket, og den optimaliserte koden er nesten 7 ganger raskere enn om -o (optimalisering) ikke er med

Tanken er at hvis tiden ikke er helt det samme som real på alle tre, må vi tilpasse kallene slik at vi får et antall ganger i docker på Linux-VM.

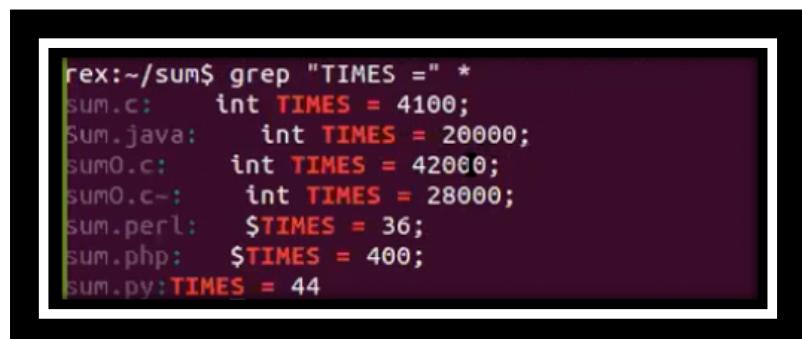
Man kan ta tiden det tok på shell-skriptet også dele på 3.69 (tiden c programmet bruker).

Svaret blir 1.5 og det betyr at programmet kan kjøre 1.5 ganger forttere. Da tar vi $1.5 * 28000 = 42000$. så tanken er da at vi går inn i sumO.c og endrer TIMES til 42000. kompilerer med opsjonen -O og kjører.



```
rex:~/sum$ gcc -O sumO.c
rex:~/sum$ time ./a.out
Ferdig 250000000000
Real:5.563 User:5.556 System:0.000 99.86%
```

Riktig fasit: optimalisert c kode er 42k ganger raskere enn et shell-skript.



```
rex:~/sum$ grep "TIMES =" *
sum.c:    int TIMES = 4100;
Sum.java:    int TIMES = 20000;
sumO.c:    int TIMES = 42000;
sumO.c~:   int TIMES = 28000;
sum.perl: $TIMES = 36;
sum.php: $TIMES = 400;
sum.py:TIMES = 44
```

Når man undrer på hvilket språk man bør velge, er det nyttig med c eller c++, c++ er omtrent like raskt som c. Alternativt eller java som også er veldig raskt pga just-in-time kompilator.

Da er byggingen av denne ferdig «docker container build....»



```
F10 key ==> File Edit Search Buffers Windows System Help
FROM ubuntu
RUN apt-get -y update
RUN apt-get -y install build-essential
RUN apt-get -y install default-jdk
RUN apt-get -y install python3
RUN apt-get -y install git
RUN git clone https://github.com/haugerud/sum.git
```

Og han runner den interaktivt slik

```
root@osG70:~/sum# docker container run -it sum bash  
root@cba99aa57f6f:/#
```

«cd sum» for å sjekke at alt er blitt lastet ned:

```
root@cba99aa57f6f:/# cd sum  
root@cba99aa57f6f:/sum# ls -l  
total 32  
-rwxr--r-- 1 root root 35 Mar 27 09:38 README.md  
-rwxr-xr-x 1 root root 416 Mar 27 09:38 Sum.java  
-rwxr-xr-x 1 root root 210 Mar 27 09:38 sum.bash  
-rwxr-xr-x 1 root root 355 Mar 27 09:38 sum.c  
-rwxr-xr-x 1 root root 276 Mar 27 09:38 sum.perl  
-rwxr-xr-x 1 root root 279 Mar 27 09:38 sum.php  
-rwxr-xr-x 1 root root 206 Mar 27 09:38 sum.py  
-rwxr-xr-x 1 root root 420 Mar 27 09:38 sum0.c
```

Under er det kompilering av programmet C og kjøring:

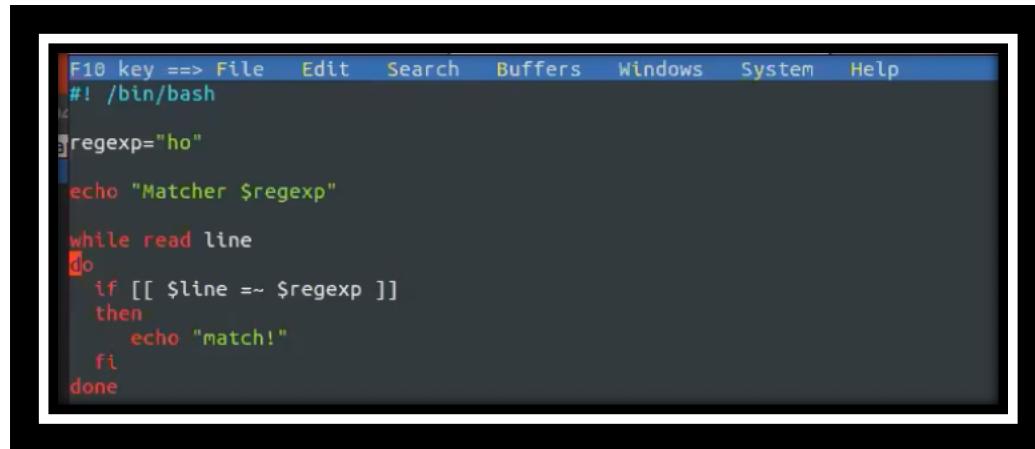
```
root@cba99aa57f6f:/sum# gcc sum.c  
root@cba99aa57f6f:/sum# time ./a.out  
Ferdig 2500000000000  
  
real 0m9.539s  
user 0m9.512s  
sys 0m0.023s
```

Her er det kompilering av java filen med kjøring

```
root@cba99aa57f6f:/sum# javac Sum.java  
root@cba99aa57f6f:/sum# time java Sum  
Ferdig 2500000000000  
  
real 0m11.992s  
user 0m11.727s  
sys 0m0.242s
```

REGULÆRE UTTRYKK

Det skriptet her lager ett regulært shell og er det en while løkke som leser inn. Hvis vi ser på testen inne med if: `=~` er operatoren som bash bruker for å teste regulære uttrykk (også det samme som perl bruker), og det vi gjør er at vi tar en og en linje i programmet og tester mot det regulære utrykket.



```
F10 key ==> File Edit Search Buffers Windows System Help
#!/bin/bash

$regexp="ho"

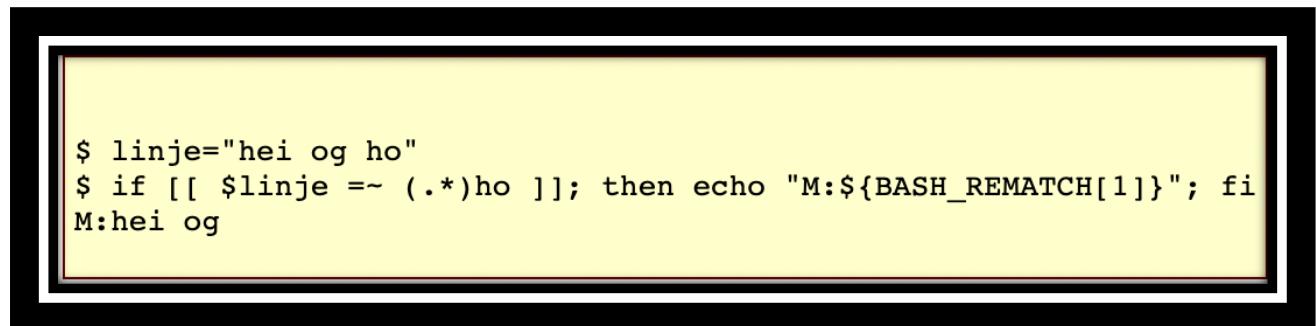
echo "Matcher $regexp"

while read line
do
  if [[ $line =~ $regexp ]]
  then
    echo "match!"
  fi
done
```

Dette er et eksempel på en Bash-skript som bruker regulære uttrykk for å matche et mønster i en streng og hente ut en del av strengen basert på mønsteret.

Først opprettes en variabel \$linje med verdien "hei og ho". Deretter brukes en if-setning med et vilkår som sjekker om \$linje inneholder et mønster som slutter med "ho". Hvis vilkåret er sant, utføres en kommando som bruker \$BASH_REMATCH-variabelen til å hente ut delen av \$linje som matcher det første uttrykket i parentesene i det regulære uttrykket.

I dette tilfellet vil if-setningen være sann, fordi \$linje inneholder "ho" på slutten. Dermed vil kommandoen inne i if-setningen skrive ut "M:hei og", fordi det første uttrykket i parentesene i det regulære uttrykket matcher "hei og".



```
$ linje="hei og ho"
$ if [[ $linje =~ (.*)ho ]]; then echo "M:${BASH_REMATCH[1]}"; fi
M:hei og
```

ASSOSIATIV ARRAY

En assosiativ array, også kjent som en mappe eller et dictionary, er en type datastruktur som brukes i programmering for å lagre data i form av nøkkel-verdi-par. I et slikt array tilordnes en nøkkel til en verdi, og denne nøkkelen brukes deretter for å hente ut verdien fra arrayet.

```
$person = array(  
    "navn" => "Ola",  
    "alder" => 30,  
    "adresse" => "Oslo"  
)
```

Over vil vi skrive

```
$person[«navn»]
```

For å få tilgang til navnet Ola.