Oblig 3

Uke 10

10.1. Se på CPU-usage på en Linux-VM. Hvor mye av CPU'en brukes vanligvis? Her ser det ut til å være mye ressurser til overs. Hvordan kan de utnyttes bedre, eller gjør de allerede det?

```
Ved å se på TOP med et delay på 3 sekunder på intel3-serveren, kan vi se CPU bruk
ligger rundt ~0.5% i løpet av de tre sekundene.

Det utnyttes så langt vi kan se veldig effektivt.
```

10.4 Linux-kjernen konfigurerer vanligvis hardware-timeren til å sende et interrupt hvert hundredels sekund. Forklar kort hvorfor dette timer-interruptet er nødvendig for at Linux-kjernen skal kunne fordele CPU-tid mellom alle prosessene uten at en av disse prosessene kan ta over styringen.

```
Et interrupt er noe som gjøres omtrent hver tiende millisekund og lar systemet
kjøre kode i en veldig kort periode. Dette brukes til å stoppe programmer fra å ta
styringen ved at de ved oppstart får lov til å kjøre over et sett antall
interrupts, før programmet må be om å få tildelt en ny periode. Ticksene styres av
hardware-timeren, siden det ville blitt for mye systemoverhead om OS skulle styrt
dette selv.
```

10.6.Problem 1.15 i Tanenbaum: Hva er den viktigste forskjellen mellom en trap og et interrupt? (I videoen som demonstrerer multitasking av forelesning og vaffelrørelaging : forskjellen på når vaffelprosessen ønsket å knuse et egg og på avbruddene fra timeren og fra melkemannen når melken leveres)

```
Den viktigste forskjellen mellom en trap og et interrupt er at en trap utføres av
systemet, mens et interrupt gjøres av hardware.

En trap er et signal fra brukerprogram som ber OS om å utføre en viss
funksjonalitet umiddelbart
```

```
Et interrupt er et signal til CPU`en som sendes ut av hardware som indikerer en
hendelse som krever umiddelbar oppmerksomhet

En trap er også kjent som et software interrupt
Trap utfører den spesifikke oppgaven i OS og gir kontroll til Trap handler
Brukerprogram instruksjonerer generer en Trap

Et interrupt er også kjent som et hardware interrupt
Hardware genererer et interrupt
Interrupt tvinger CPU`en til å utløse en spesifikk interrupt rutine.d
```

10.8(Oblig) Tre 100% CPU-avhengige prosesser A, B og C kjører på en Linux-PC med kun én CPU og med en Linux 2.6 kjerne. Ved starten av en epoke er alle tre klare til å kjøre og de er de eneste prosessene i ready-list. Prosess A har en tildelt timeslice på 30 ticks/jiffies, B har 20 og C har 10. Forklar kort hvordan disse prosessene kjøres og hvordan fordelingen av CPU-tid vil skje for disse tre prosessene i løpet av epoken. Anta at det ikke forekommer andre interrupts enn fra timeren og at prosessene ikke gjør noen systemkall.

```
1 prosess 30 ticks - bestemt av prioritet
2 prosess 20 ticks - bestemt av prioritet
3 prosess 10 ticks - bestemt av prioritet

OS kjører Round robin  -
Prosess 1 går i 30 ticks - hvis counter = 0 kalles neste prosess(Scheduleren)
Prosess 2 går i 20 ticks - hvis counter = 0 kalles neste prosess(Scheduleren)
prosess 3 går i 10 ticks - hvis counter = 0 kalles neste prosess(Scheduleren)

Siden det ikke kommer noen andre forstyrrelser vil de kjøre en etter en. A vil
først kjøre over 30 ticks, før B kjører over 20 og så C over 10 ticks.

Epoken er over når alle prosesser har brukt opp sin tid.
```

10.9(Oblig) Forklar kort hvor mange timer-interrupts og context switches som vil skje i denne epoken.

```
2 context switches fordi det tar ganske lang tid å utføre en context switch så
hvis det ikke finnes interrputs så vil hver prosess kjøre ferdig sine ticks før ny
prosess får lov til å kjøre.
```

```
Det vil skje 60 interrupts, og 3 context-switches
```

10.10(Oblig) Hva skjer når epoken er over? Vil prosessene nødvendigvis få tildelt samme antall ticks som før? Forklar kort.

```
Om det er noen endringer i en av prosessene sin prioritet, eller andre prosesser
også starter kan de får en annen mengde med ticks.
```

10.14(Oblig) Lag et script som først stopper alle kjørende docker containere og deretter fjerner dem og alle image. Det er viktig at dere jevnlig fjerne gamle containere og image som ikke er i bruk siden serveren har lite diskplass og risikerer å bli helt full om man ikke rydder jevnlig.

```
#!/bin/bash

echo 'stopping containers...'
while [[ -n "$( docker ps -a -q )" ]]
do
docker stop $(docker ps -a -q)
done

echo 'deleting containers...'
if [[ -n "$( docker ps -a -q )" ]] ; then
docker rm $(docker ps -a -q)
fi

echo 'deleting images...'
if [[ -n "$( docker images -q )" ]] ; then
    docker rmi $(docker images -q)
fi
```

```
Alternativ 2:
```

```
#! /bin/bash
docker kill $(docker ps -q)
```

```
docker system prune -a
```

10.15(Oblig) I denne oppgaven skal du på samme måte som i oppgaven fra forrige uke lage to docker containere som kjører en apache2 webserver på port 8081 og 8082 på Linux VM. Men denne gangen skal du først lage en Dockerfile som sørger for at apache2 installeres på et ubuntu-image og bygge image'et med denne. Prøv å få til at apache2 automatisk starter når containeren startes opp, slik at du ikke trenger å gå inn og eksplisitt starter den. Lag en index.html-fil som inneholder 'hei fra dockerfile' eller lignende. Når image bygges skal denne filen legges i /var/www/html i docker-imaget. Legges denne filen inn når image'et bygges eller når containeren starter?

Hint: Apache2-installasjonen spør interaktivt om tidssone og byggingen vil henge. Legg derfor inn riktig tidsone som andre og tredje linje i Dockerfile:

```
ENV TZ=Europe/Oslo
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime  && echo $TZ > /etc/timezone
```

Hint nr 2: Kommandoen som er spesifisert med CMD er den kommandoen man ønsker å kjøre når containeren starter og om denne stoppes, vil containeren avsluttes. Derfor kan ikke apache startes med 'service apache2 start' for denne kommandoen avsluttes og containeren stoppes. Bruk derfor 'apachectl -D FOREGROUND' istedet som fortsetter å kjøre i forgrunnen og dermed fortsetter også containeren å kjøre. (Start containeren med -d og ikke med -it og /bin/bash. Hvis du gjør det site tar /bin/bash over som container-kommandoen, apachectl utføres ikke og apache startes ikke.)

```
FROM ubuntu
ENV TZ=Europe/Oslo
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone
RUN apt-get --fix-missing -y update
Run apt-get -y update
RUN apt-get -y install apache2
```

10.17

```
Oppgave levert på https://nexus.cs.hioa.no/~haugerud/os10.php

Har ikke lagret koden, men kan finne den på websiden nevnt over.

Containere og images er slettet, får derfor ikke hentet den på nytt
```

Uke11

11.2(Oblig) Man kan installere kildekoden for linux med

```
sudo apt-get install linux-source
```

Dette er gjort på data2500 og kildekoden ligger der i

```
/usr/src/linux-source-5.10
```

Finn ut fra filen

```
/usr/src/linux-source-5.10/arch/x86/entry/syscalls/syscall_64.tbl
```

hvor mange systemkall det er i denne versjonen av Linux-kjernen. I mappen over er det en fil som heter entry_64.S . Hvilken funksjon har koden i denne filen og hvilket språk er den skrevet i?

```
Det er 385 systemkall i den filen. Filen entry_64.S inneholder andre systemkall,
og er skrevet i Assembly.
```

11.4(Oblig) Start to regne-jobber på Linux VM. Bruk scriptet regn fra tidligere oppgaver, men legg til minst en null i løkken så de kjører lenger og kommenter bort litt av utskriften så det blir mindre utskrift i terminalen. Sjekk at de kjører med top. Hvor stor andel CPU-tid får de hver? Gi så den ene av prosessene så stor nice verdi som mulig, 19. Hvor stor andel CPU-tid får de nå hver? Hvis du ikke ser noen endring, hva kan det komme av? Hva om du starter tre regnejobber, ser du noen effekt av at en av dem har nice = 19? Hint: Sjekk 'Last Used Cpu' i top.

```
Hvis det ikke er andre jobber kjørende vil de bruke 100% hver av cpu. Hvis jeg
kjører 3 regnejobber på en vm med to cpu vil den med høyest nice verdi bruke
mindre cpu tid og la de to andre prosssesene få mer cpu tid.
```

11.5(Oblig) Start to regne-jobber på din Linux-VM med følgende kommandoer:

```
group70@os70:~$ taskset -c 0 ./regn&
group70@os70:~$ taskset -c 0 ./regn&
```

Forklar hva taskset gjør med prosessene. Sjekk gjerne at det stemmer ved å kjøre top, taste f, gå med piltaster ned til P = Last Used Cpu, taste space og så Esc. Hvor stor andel CPU-tid får de nå hver?

Stopp så regn-jobbene og start to nye med

```
group70@os70:~$ killall regn
group70@os70:~$ taskset -c 0 ./regn&
group70@os70:~$ taskset -c 0 nice -n 19 ./regn&
```

Hvor stor andel CPU-tid får de nå hver?

```
Tasketset setter regne jobben eksplisitt på cpu 0.
Kjører jeg 2 regnejobber vil de kjøre på samme cpu(0) og bruke 50% av cpu hver

Den med høyest nice verdi bruker 1,3%, lavest nice verdi bruker nesten 100%
prosent hele tiden når den med lavest nice verdi er ferdig begynner den med høyest
nice verdi å få mer og mer andel av CPU
```

11.8(Oblig) Studer og kjør følgende kode:

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
```

```c
#include <sys/types.h>
#include <sys/wait.h>

int main(){

int i,id,pid,mypid,ppid;
id = getpid();
printf ("Fork demo!\n Jeg er parent-prosess med pid = %d \n",id);
sleep(2);
pid = fork();     /* Prosessen lager nå en ny uavhengig prosess
                     I den nye barne-prosessen er $pid = 0
                     I foreldre prosessen er $pid = barnets PID */
if (pid == 0)
  {
    mypid = getpid();
    ppid  = getppid();
    printf("       Jeg er child (mypid = %d) av parent med pid =
%d.\n",mypid,ppid);
    printf("       Here er variabelen pid = %d.\n",pid);
    for (i = 1;i < 10;i++)
      {
        printf("       %d \n",i);
        sleep(1);
      }
    printf("       Child prosess avslutter.\n");
    exit(0);
  }
else
  {
    sleep(2);
    printf("Parent %d venter på child %d her...\n",id,pid);
    wait(&pid);
    printf("Child har avsluttet, Parent avslutter!\n");
  }

}
```

Vær oppmerksom på at etter fork()-kallet er det to uavhengige prosesser som kjører den samme koden! Eneste forskjell er at variabelen pid er forskjellig.

Modifiser denne koden og lag et C-program som ved hjelp av fork() lager en child-prosess. Denne prosessen skal skrive ut sin egen og parent-prosessen sin PID, sove i 5 sekunder (sleep(5);) og deretter ved hjelp av fork() lage en ny child-prosess. Denne prosessen blir "grandchild" av den opprinnelige prosessen. Grandchild-prosessen skal skrive ut sin egen og parent-prosessen sin PID, sove i 5 sekunder og deretter avslutte. Alle prosesser skal vente på sine children og si ifra når de avslutter.

Kanskje noe sånt?

```c
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(){
    printf ("Fork demo!\n Jeg er parent-prosess med pid = %d \n",getpid());
    sleep(2);
    forking(1);
}

void forking(int done) {
int i,id,pid,mypid,ppid;
id = getpid();
pid = fork();

if(pid == 0)
  {
    mypid = getpid();
    ppid  = getppid();
    printf("       Jeg er child (mypid = %d) av parent med pid =
%d.\n",mypid,ppid);
    printf("       Here er variabelen pid = %d.\n",pid);
    sleep(5);
    printf("       Child prosess avslutter.\n");
    if(done == 1)
    forking(0);
    exit(0);
  }
else
  {
    sleep(2);
    printf("Parent %d venter på child %d her...\n",id,pid);
    wait(&pid);
    printf("Child har avsluttet, Parent avslutter!\n");
  }
}
```

11.10(Oblig) Målet med denne oppgaven er å forberede en sammenligning av hastigheten til programeringsspråk for CPU-intensive programmer. I GitHub repositoriet https://github.com/haugerud/sum.git ligger det noen programmer skrevet i bash, C, Python, perl og java. Og en PHP-fil, men den kan du se bort ifra. Test at du kan klone dette til Linux VM med kommandoen git clone https://github.com/haugerud/sum.git slik at det lages en mappe med navn sum som inneholder kildekoden for programmer som i alle disse språkene utfører de samme

CPU-intensive regneoperasjonene. Istedet for å installere kompilatorer og runtime miljøer for alle disse språkene i Linux-VM, skal du lage en Dockerfile som laster ned filene med git og installerer gcc, java og python i en Ubuntu-container. Du trenger å installere følgende:

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

Hvis du får en feilmelding av typen

```
E: Unable to fetch some archives, maybe run apt-get update or try with --fix-
missing?
```

legg til --fix-missing når du kjører apt-get update i starten av Dockerfile. Det vil si, bruk:

```
RUN apt-get --fix-missing -y update
```

Hint: Hvis installasjonen spør interaktivt om tidssone og byggingen henger; Legg inn riktig tidsone som andre og tredje linje i Dockerfile:

```
ENV TZ=Europe/Oslo
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime  && echo $TZ > /etc/timezone
```

Etter at du har laget en Dockerfile som gjør dette og laster ned git-repositoriet sum, start containeren, gå inn i sum-mappen og sjekk at du kan kompilere og kjøre java-programmet med:

```
javac Sum.java
java Sum
```

Og at du kan kompilere med gcc og kjøre C-programmet på vanlig måte. Vider skal du kunne kjøre

```
./sum.perl
python3 sum.py
```

Hvis du står fast på denne oppgaven, kan du se de tre videoene i ukens linux-forelesning som viser et forsøk på å løse denne og litt av de følgende oppgavene med samme tema. (de hadde den gang andre oppgavenummer).

```
Git repo klonet med kommando: git clone https://github.com/haugerud/sum.git

Dockerfile

FROM ubuntu
RUN apt-get --fix-missing -y update
ENV TZ=Europe/Oslo
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /e$
RUN apt-get -y update
RUN apt-get -y install build-essential
RUN apt-get -y install python3
RUN apt-get -y install default-jdk
RUN apt-get -y install git
RUN git clone https://github.com/haugerud/sum.git
```

```
javac Sum.java
java Sum
Ferdig 250000000000
```

```
time ./sum.perl
Ferdig.
real    0m10.423s
user    0m10.419s
sys 0m0.004s
```

```
python3 sum.py
Ferdig! Sum: 250000000000
```

11.11 (Oblig) Nå skal du ta tiden på disse programmene inne i containeren du lagde i forrige oppgave. Du vil se at sum.bash er litt anderledes enn de andre. I dette programmet (eller scriptet) regnes det ut en sum 500.000 ganger. Alle de andre

programmene regner ut denne summen TIMES ganger. Du måler tiden sum.bash bruker ved kommandoen*

```
root@33256e20c0ee:/sum# time ./sum.bash
Ferdig, sum: 250000000000

real    0m5.564s
user    0m5.518s
sys 0m0.037s
```

Resultatet du bør se på er user + sys som vanligvis er det samme som real (for de andre programmene er stort sett sys = 0) og det brukte altså 5.5 sekunder (ett desimal holder).Hvis tiden real er mye større enn user + sys betyr det at andre programmer bruker mye CPU. Det kan innvirke på resultatet hvis den underliggende VM-serveren er tungt belastet, men den har 4 uavhengige CPUer, så det vil trolig stort sett ikke være et problem. Sett gjerne variabelen

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

for å få et mer kompakt tidsresultat.

Ved å bruke kommandoen time på de andre programmene skal du nå lage en tabell over hvor mange ganger raskere de andre programmene er enn sum.bash. Om de andre programmene ikke bruker 5.5 sekunder(eller den tiden du måler for sum.bash så kan du prøve å endre størrelsen på TIMES slik at programmet bruker ca like lang tid som sum.bash. Hvis f. eks. time sum.perl bruker 11 sekunder, så kan du halvere verdien på TIMES og prøve igjen, da bruker det trolig ca 5.5 sekunder. Etter å ha tilpasset TIMES slik at alle programmene bruker omtrent like lang tid som bash-scriptet, vil verdien på TIMES for et program si hvor mange ganger raskere det er enn bash-scriptet. Lag en tabell over hvor mange ganger raskere C, PHP, Python og java-programmene er enn bash-programmet. Sammenlign med resultatene fra en forelesning for noen år siden, listet nedenfor og kommenter.

```
2010

| Språk | CPU-tid i sekunder på cube | TIMES |
|-------|----------------------------|-------|
| Bash  | 5.7                        | 1     |
| php   | 5.6                        | 21    |
| perl  | 5.6                        | 75    |
| Java  | 5.6                        | 13500 |
| C/C++ | 5.6                        | 63000 |
```

```
2019

| Språk     | Times |
|-----------|-------|
| Bash      | 1     |
| sum.perl  | 40    |
| sum.py    | 46    |
| sum.pho   | 155   |
| sum.c     | 4200  |
| Sum.java  | 15000 |
| sumO.c    | 25000 |
```

Bash-scriptet brukte ca 5.1 sek

Språk	CPU-tid i sekunder på cube	TIMES
bash	5.1	1
perl	5.1	28
php	5.1	215
Java	5.1	18200
C	5.1	36000

11.15(Oblig) Lag et bash-script som teller hvor mange blanke(tomme) linjer det er i filen scriptet tar som argument. Linjer som inneholder blanke (" "), tab og lignende tegn som ikke synes, regnes også som blanke. Test om det virker på noen filer du lager!

```
#! /bin/bash
fil=minFil.txt
egrep '^\s*$' $fil | wc -l
```

** 11.18**

```
Oppgave levert på https://nexus.cs.hioa.no/~haugerud/os10.php

Har ikke lagret koden, men kan finne den på websiden nevnt over.

Containere og images er slettet, får derfor ikke hentet den på nytt
```

Uke12

12.1(Oblig) Ta utgangspunkt i følgende C-program:

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

Skriv dette inn i en fil hello.c ved å starte f. eks. editoren jed:

```
$ jed hello.c
```

lagre filen og kompiler den med

```
$ gcc hello.c
```

gcc er C-kompilatoren. For å kunne kompilere og kjøre C-programmet, må gcc installeres: apt-get install gcc (hvis du da får en feilmelding om at stdio.h ikke finnes, installer også følgende: apt-get install build-essential. Den lager nå maskinkode som lagres i den kjørbare filen a.out. For å få lov til å kjøre den, må filen gis kjørerettigheter og det gjør du med kommandoen

```
$ chmod 700 a.out
```

Kjør programmet med

```
$ a.out
```

og du bør nå få ut Hello world i terminalvinduet. Klarer du å se hvilke maskininstruksjoner den kjørbare koden består av? Logg så inn på en Windows maskin. Kopier a.out med Winscp eller på annen måte, og kjør det fra der du har lagt det med

```
C:\>a.out
```

Forklar hva som skjer. Hvordan kan du få til å kjøre hello.c på Windows?

```
maskinkode/Assembly:
```

```
root@os608:/home/s108# cat hello.s
    .file   "hello.c"
    .text
    .section    .rodata
.LC0:
    .string "c: Hello world!"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long    1f - 0f
    .long    4f - 1f
    .long    5
0:
    .string  "GNU"
1:
    .align 8
    .long    0xc0000002
    .long    3f - 2f
2:
    .long    0x3
```

```
3:
    .align 8
4:
```

```
Fra forelesningen - ingen ting ut. a.out snakker med linux operativsystemet så
denne vil ikke kjøre på windows. Samme cpu(x86) men a.out prøver å snakke med
linux men treffer på windows server og derfor vil dette ikke fungere.

Hvis linux modul er installert(Bash-shell)
hvis a.out kjøres inne i linux modulen vil dette fungere.
```

12.2(Oblig) Ta utgangspunkt i følgende Java-program

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

og lagre det i filen Hello.java på Linux-VM. Man kan installere java og java-kompilatoren javac med

```
apt-get install default-jdk
```

På Linux kompileres et Java-program med

```
$ javac Hello.java
```

det lages da byte-kode som lagres i en fil Hello.class som kjøres med

```
$ java Hello
```

og sjekk at den kjører som den skal. Kopier over Hello.class til en Windows-PC. Får du den til å kjøre her? Vil det i prinsippet være mulig?

```
Run time problem - bakoverkompabilitet fungerer ikke. Hvis man har riktig jdk
versjon installert vil det fungere fint. Derfor også kalt et plattforuavhengig
språk(java)
```

12.4(Oblig) Kopier Java-programmet Calc.java fra avsnitt 10.9 i forelesningen til Linux-VM og lagre det med samme navn. Om du ikke har gjort det før, kan du installere java og java-kompilatoren javac med

```
apt-get install default-jdk
```

Kompiler så med

```
javac Calc.java
```

og sjekk at det lages en Calc.class fil. Kjør programmet med

```
java Calc
```

Om det går for fort, gang verdien på telleren i programmet med 10 eller mer, kompiler og kjør igjen. Tving java til å kjøre på bare en CPU med taskset. Hvor mye CPU får hver tråd? Endre koden slik at det starter tre tråder istedet for to og tving igjen alle til å kjøre på samme CPU med taskset. Tast H i top hvis tråder ikke vises. Hvor mye CPU-tid får hver tråd i snitt når det kjører? Hvor mange java-tråder kjøres totalt når du starter dette Java-programmet? Hint: Se på nTH = Number of Threads i top eller bruk ps -L. Hvis man taster 'u' i top og så skriver inn sitt brukernavn (f.eks. group100) vil man kun se denne brukerens prosesser/tråder. En annen mulighet er å installere htop som viser tråder.

```
nTH sier at de får 15 tråder. Prosessene får i snitt 33,3% hver hvis java
programmet kjøres med taskset
```

12.5(Oblig) Studer og kjør følgende kode på din Linux-VM:

```
import java.lang.Thread;
import java.io.PrintStream;

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

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

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

   private long work()
        {
       long i,j;
       if(id == 1)
         {
        System.out.println("Thread nr." + id + " calculating");
        for(i = 1;i <= 4000000000L;i++)
          {
             res += i;
          }
         }
       return(res);
    }
}

class Calc
{
   public static void main(String args[])
   {
      System.out.println("Starts two threads!");
      CalcThread s1 = new CalcThread();
      System.out.println("Thread s1 has id " + s1.id);
      s1.start(); // Allokerer minne og kaller s.run()

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

```
    {
       s1.join();
    }
      catch (InterruptedException e)
    {
    }

      try
    {
       s2.join();
    }
      catch (InterruptedException e)
    {
    }
      System.out.printf("Thread nr. 1 calculated %d\n",s1.res);
      System.out.printf("Thread nr. 2 calculated %d\n",s2.res);
   }
}
```

Resultatet av denne kjøringen er et veldig stort heltall, derfor heter variablene long istedet for int, long er 64 bit lange, mens int er 32 bit. Det er også årsaken til at tallet 4 milliarder i work-løkken slutter med en L, den må med når så lange siffer skal skrives inn i koden. Den ene tråden i koden regner ut summen av alle tall mellom 1 og 4 milliarder, mens den andre tråden ikke gjør noe. Kallene s1.join og s2.join inne i main gjør at main venter til begge trådene er ferdige med sine beregninger før den avslutter.Ta tiden på hvor lang tid beregningen tar. Klarer du å få programmet til å regne ut samme sluttresultat dobbelt så fort ved å fordele beregningene på begge trådene og utnytte to CPUer samtidig?

```
jorber@JBMain:~/uke12/oppgave5/1$ time java Calc
Starts two threads!
Thread s1 has id 1
Thread s2 has id 2
s2 started !

Thread nr.1 is starting
Thread nr.2 is starting
Thread nr.1 calculating
Thread nr. 1 calculated 8000000002000000000
Thread nr. 2 calculated 0

real    0m6.074s
user    0m6.068s
sys     0m0.071s
jorber@JBMain:~/uke12/oppgave5/2$ time java Calc
Starts two threads!
Thread s1 has id 1
Thread s2 has id 2
s2 started !
```

```
Thread nr.2 is starting
Thread nr.1 is starting
Thread nr.2 calculating
Thread nr.1 calculating
Thread nr. 1 calculated 2000000001000000000
Thread nr. 2 calculated 6000000001000000000
Total: 8000000002000000000

real    0m3.093s
user    0m6.141s
sys     0m0.030s
```

Endret kode

```
import java.lang.Thread;

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

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

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

    private long work()
    {
        long i,j;
        if(id == 1)
        {
            System.out.println("Thread nr." + id + " calculating");
            for(i = 1;i <= 2000000000L;i++)
            {
                res += i;
            }
        }
        else if(id == 2) {
            System.out.println("Thread nr." + id + " calculating");
            for(i = 2000000001L;i <= 4000000000L;i++)
            {
                res += i;
            }
```

```
            }
            return(res);
        }
    }

    class Calc
    {
        public static void main(String args[])
        {
            System.out.println("Starts two threads!");
            CalcThread s1 = new CalcThread();
            System.out.println("Thread s1 has id " + s1.id);
            s1.start(); // Allokerer minne og kaller s.run()

            CalcThread s2 = new CalcThread();
            System.out.println("Thread s2 has id " + s2.id);
            s2.start();
            System.out.println("s2 started !\n");
            try
            {
                s1.join();
            }
            catch (InterruptedException e)
            {
            }

            try
            {
                s2.join();
            }
            catch (InterruptedException e)
            {
            }
            System.out.printf("Thread nr. 1 calculated %d\n",s1.res);
            System.out.printf("Thread nr. 2 calculated %d\n",s2.res);
            System.out.println("Total: " + (s1.res + s2.res));
        }
    }
```

12.12(Oblig) Inntil 2005 var det slik at det fantes noen x86 instruksjoner (som for eksempel POPF) som bare ble oversett om de ble utført i user mode. Det disse instruksjonene utførte skulle bare være lov å gjøre i kernel mode og ingen ting skjedde når de ble utført i user mode. Forklar kort hva som gjorde dette problematisk i forbindelse med virtualisering og hvordan disse hardware-instruksjonene ble endret for å støtte x86-virtualisering.

```
Det ble problematisk for virtualisering fordi en virtuell maskin ble kjørt i user
mode, og derfor ikke kunne nå den overordnede OS'en sin hypervisor.Det ble løst
ved at alle sensistive instruksjoner som utføres i user mode av gjeste-os må
trap`e til kernel mode og fanges opp av hypervisor
```

12.13(Oblig) Last ned følgende fil på gruppens virtuelle os Linux-VM med

wget http://www.cs.hioa.no/~haugerud/virt.tgz

Pakk så denne mappen ut med tar xfz, gå inn i mappen, kompiler og kjør med scriptene som vist på forelesning. Før du kjører scriptet, må du installere /usr/bin/time og gcc med

```
sudo apt install time gcc
```

for å kunne kjøre gcc og få riktig tids-output fra scriptet. Sammenlign med resultatene som er vist nedenfor og som er det man får ved å kompilere og kjøre de samme programmen på bare metal. Kommenter forskjellene, er de som forventet? Å kjøre på bare metal vil si å kjøre direkte på serveren uten noen form for virtualisering, direkte på hardware. Resultatet fra denne kjøringen så slik ut:

```
root@amdock:~/hh/virt/virt# ./runiter.bash
sum getppid gettimeofday forkwait fork
1.52;0.06;0.02;0.06;3.05;
1.52;0.06;0.02;0.06;3.05;
1.52;0.06;0.02;0.06;3.04;
1.52;0.06;0.02;0.06;3.05;
1.52;0.06;0.02;0.06;3.05;
```

Når samme program kjøres på en vanlig Docker container som kjøres direkte på amdock blir resultatet som følger:

```
root@66aef203414c:~/virt# ./runiter.bash
sum getppid gettimeofday forkwait fork
1.52;0.11;0.02;0.07;3.05;
1.52;0.11;0.02;0.06;3.05;
1.52;0.11;0.02;0.06;3.05;
1.52;0.11;0.02;0.07;3.05;
1.52;0.11;0.02;0.07;3.05;
```

Sammenlign disse to resultatene med dine resultater på Linux-VM. Husk at din VM egentlig ikke er en virtuell maskin men en sysbox-container som startes med noe slikt som

```
docker container run  --cpus="2"  --runtime=sysbox-runc -d os65
```

Sammenlign også forskjellene du finner over med forskjellene mellom bare metal og virtuelle maskiner (VMer) i forelesningsnotatene i avsnitt 10.27. Disse VM'ene ble kjørt med klassisk virtualisering, qemu-KVM på en Linux-server.

Bare Metal

```
group20@os20:~/uke12/oppgave13/virt$ ./runiter.bash
sum getppid gettimeofday forkwait fork
1.52;0.12;0.02;0.07;6.11;
1.52;0.12;0.02;0.07;6.06;
1.53;0.12;0.02;0.07;6.12;
1.52;0.12;0.02;0.07;6.13;
1.52;0.12;0.02;0.07;6.05;
```

Docker container

```
root@d1fc89281464:/uke13/virt# ./runiter.bash
sum getppid gettimeofday forkwait fork
1.52;0.15;0.02;0.07;6.09;
1.52;0.15;0.02;0.06;6.11;
1.55;0.15;0.02;0.06;6.09;
1.52;0.15;0.02;0.06;6.11;
1.52;0.15;0.02;0.07;6.10;
```

Tidene er ganske identiske med tidene i oppgaveteksten, men fork bruker dobbelt så lang tid.

** 12.14**

```
Oppgave levert på https://nexus.cs.hioa.no/~haugerud/os12.php

Har ikke lagret koden, men kan finne den på websiden nevnt over.
```

```
Containere og images er slettet, får derfor ikke hentet den på nytt
```

Uke 13

13.2(Oblig) Lagre filen Prior.java fra forelesningen på Linux-VM og kompiler og kjør den. Hvordan behandler Linux Java Virtual Machine (JVM = /usr/lib/bin/java) prioriteringen? Endre programmet slik at tråd to endrer prioritet til 4 istedet for 1. Ser dette ut til å gi noen endring i hvordan trådene blir prioritert? Eksperimenter med forskjellige prioriteter. Ser det ut til at noen av trådene får mer prioritet? Prøv å kjøre java med opsjonen -XX:ThreadPriorityPolicy=1 som vanlig bruker og som root med sudo java. Gir dette noen endring? Kommenter kort det du ser. (Les http://tech.stolsvik.com/2010/01/linux-java-thread-priorities-workaround.html for mer info om dette). Bruk taskset -c 0 hvis du kjører på en maskin med flere CPUer, ellers vil ikke trådene konkurrere om CPU-tid. NB! Linux-VM er egentlig en docker-container og man har noe begrensede rettigheter i forhold til om man kjører på en Linux server eller en Linux viruell maskin. Følgende

```
root@os100:/home/group100# nice -n -5 ./regn
nice: cannot set niceness: Permission denied
```

viser at man ikke får lov til å sette negativ niceness som root som man normalt kunne gjort. Dette vil ha innvirkning på resultatet du får.

```
Ved å kjøre med prioritet 4 eller 1 ga det ikke noe forskjellig resultat, det ser
ut til at
linux ikke bryr seg om java sin thread priorioty uten å ha -
XX:ThreadPriorityPolicy=1
opsjonen. Ved å skru på denne opsjonen, så forandret prioriteten seg.

Når thread 1 hadde høyere prioritet enn 2, så avsluttet den først, og omvendt.
```

13.4(Oblig) I denne oppgaven skal vi ta utgangspunkt i programmet listet nedenfor. Hvis threads eller prosesser skal jobbe mot felles data, må man generelt synkronisere dem slik at de ikke modifiserer felles data samtidig; noe som har uforutsigbare konsekvenser. Programmet har en felles variabel static int saldo som de to trådene skal

jobbe mot. Som vi har sett har Java en egen klasse Thread og for å lage egne threads kan du utvide denne klassen med extends. I main() lages nye tråder med

```
SaldoThread s1 = new SaldoThread();
```

En tråd settes i gang med s1.start(). Da startes run() som er en metode alle threads må ha. Variabler som defineres som static, er felles data for alle tråder.

Studer først programmet "NosynchThread.java", som blir listet nedenfor, ved å lese programteksten (legg merke til at main også er en thread):

```
// Kompileres med  javac NosynchThread.java
// Run: java NosynchThread

import java.lang.Thread;

class SaldoThread extends Thread
{
    static int MAX = 50000000;
    static int count = 0;
    public static int saldo; // Felles variable, gir race condition
    int id;

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

    public void run()
    {
    System.out.println("Thread nr. "+ id +", med prioritet " + getPriority() + "
starter");
    updateSaldo();
    }

    public void updateSaldo()
    {
    int i;
    if(id == 1)
    {
        for(i = 1;i < MAX;i++)
        {
        saldo++;
        }
    }
    else
    {
        for(i = 1;i < MAX;i++)
```

```
            {
                saldo--;
            }
        }
        System.out.println("Thread nr. " + id + " ferdig. Saldo: " + saldo);
        }
}

class NosynchThread extends Thread
{
   public static void main (String args[])
   {
       int i;
       System.out.println("Starter to threads!");

       SaldoThread s1 = new SaldoThread();
       SaldoThread s2 = new SaldoThread();
       s1.start();
       s2.start();

       try{s1.join();} catch (InterruptedException e){}
       try{s2.join();} catch (InterruptedException e){}

       System.out.println("Endelig total saldo: " +SaldoThread.saldo);
   }
}
```

Kompiler og kjør NosynchThread.java. Kan du forklare hva som skjer og det merkelige sluttresultatet for total saldo? Hvorfor endrer sluttresultatet seg fra gang til gang når det samme programmet kjøres omigjen? Gjør det noen forskjell på resultatet om du kjører programmet med

```
$ taskset -c 0 java NosynchThread

Når det kjøres to prosesser så kan context switch oppstå når som helst. Så når
tråd 1 skal til å oppdatere saldoen slik
saldo = saldo + i
så kan tråden bli avbrutt rett før denne instruksen. Da kan tråd 2 i mellomtiden
forandre verdien i saldo. Da vil dette gi feil verdi når tråd 1 starter igjen, og
gjennomfører instruksen. Dette blir en race condition.
```

Hva endres med hvordan trådene kjøres når du kjører java med taskset på denne måten? Det kan være du må gjenta taskset-kjøringen mange ganger (eventuelt øke max-verdien i loopen) for at du skal få 'gale' resultater. Hva er forskjellen på antall mulige "kollisjoner" når trådene kjører på samme CPU sammenlignet med når de kjører på hver sin CPU?

```
Det er fortsatt mulig å få feil resultater om man kjører med taskset, men som
oftest får man korrekt resultat (saldo = 0).
```

13.5(Oblig) Ta utgangspunkt i programmene thread.c, minimal.s og en.c fra forelesningen og lagre dem på data2500. Editer først thread.c og gang antall runder i løkken med en faktor 100, til en milliard, ellers vil du ikke se den ønskede effekten på data2500. Kompiler og kjør dem først med

```
haugerud@data2500:~/lock$ gcc -pthread thread.c minimal.s
haugerud@data2500:~/lock$ taskset -c 0 ./a.out
```

Hvorfor får du nå alltid samme resultat selvom det er to uavhengige tråder som kjører?

```
Om vi hadde f.eks hadde kodet følgende i C:

for (int i = 0; i < 1000000000; i++)
{
    svar++;
}

ville det laget flere instruksjoner i assembly, hvor tråden kunne blitt satt på
pause hvor som helst. Da kunne tråd nr 2 endret på svar variabelen før tråd 1 er
ferdig.
Siden assembly instruksjon som kalles fra metoden bare er en instruks, kan ikke
tråden
stoppes mitt i instruksjon som det ville gjort om vi hadde kodet det i C.
```

Gjenta kjøringen ved å istedet bruke en.c på følgende måte:

```
haugerud@data2500:~/lock$ gcc -pthread thread.c en.c
haugerud@data2500:~/lock$ taskset -c 0 ./a.out
```

Kanskje må du kjøre den flere ganger for å se effekten, men hvorfor får du nå noen ganger forskjellig resultat? Kan det ha noe å gjøre med at kompilatoren kan lage flere linjer maskinkode av en linje høynivå-kode? Kompiler en.c på følgende måte:

```
haugerud@data2500:~/lock$ gcc -c -S en.c
```

Se så på den resulterende filen en.s og avgjør om flere linjer maskinkode kan være årsaken.

```
        movl    svar(%rip), %eax
        addl    $1, %eax
        movl    %eax, svar(%rip)
```

```
Denne biten fra en.s viser at det lages 3 instruksjon for å inkrementere svar
(svar++).
```

Kompiler og kjør til slutt på følgende måte:

```
haugerud@data2500:~/lock$ gcc -O -pthread thread.c en.c
haugerud@data2500:~/lock$ taskset -c 0 ./a.out
```

**Hvorfor blir resultatet nå alltid det samme?

```
Det er fordi med -O opsjonen optimaliserer gcc koden, det vil si den prøver å lage
færre instruksjoner. Om vi ser på assembly koden for en.c nå, er det bare denne
instruksjonen som brukes for å inkrementere svar

addl    $1, svar(%rip)
```

13.8(Oblig) Lag en docker-compose.yaml fil som starter opp to nginx-containere som begge er bygget fra den samme Dockerfile som ligger i en mappe du kan kalle nginx. Når de starter opp skal de to containerne som kjører nginx-webserveren på henholdsvis port 8080 og 8081 gi data fra to forskjellige index.html filer fra mapper på host'en som containerene kjører på. Omtrent slik som i eksempelet fra forelesningsnotatene. Dockerfile skal sørge for at både iputils-ping og net-tools er installert når de to containerene starter opp. Hvis så at du kan gå inn på en av containerene og pinge den andre. Vis også ved hjelp av kommandoen ifconfig inne i containerene hvilke IP-adresser de har fått på det lokale nettverket. Er selve host'en også på dette nettverket?

```
Dockerfile
FROM nginx:latest
```

```
RUN apt-get update -y
RUN apt install iputils-ping -y
RUN apt install net-tools -y
RUN apt install jed -y
```

```
Docker compose
version: '3'
services:
    nginx:
        build: ./nginx
        ports:
            - 8080:80
        volumes:
            - ./innhold:/usr/share/nginx/html:ro
    nginx2:
        build: ./nginx
        ports:
            - 8081:80
        volumes:
            - ./innhold2:/usr/share/nginx/html:ro
```

** Ping resultat fra nginx2 til nginx**

```
PING nginx (172.18.0.2) 56(84) bytes of data.
64 bytes from dockercompose_nginx_1.dockercompose_default (172.18.0.2): icmp_seq=1
ttl=64 time=0.090 ms
64 bytes from dockercompose_nginx_1.dockercompose_default (172.18.0.2): icmp_seq=2
ttl=64 time=0.020 ms
64 bytes from dockercompose_nginx_1.dockercompose_default (172.18.0.2): icmp_seq=3
ttl=64 time=0.044 ms
64 bytes from dockercompose_nginx_1.dockercompose_default (172.18.0.2): icmp_seq=4
ttl=64 time=0.019 ms
```

ifconfig på begge containere

```
nginx: inet 172.18.0.2  netmask 255.255.0.0  broadcast 172.18.255.255
nginx2: inet 172.18.0.3  netmask 255.255.0.0  broadcast 172.18.255.255
Host has adresse 172.18.0.1
```