

Ukeoppgaver 16 – Threads, serialisering, race condition og virtualisering

RØD - Obligoppgaver

GUL – Ikke obligoppgaver

TURKIS – Ukens nøtt og utfordringer

Teorioppgaver besvares ved hjelp av hjelpebidrifter.

1. (Ikke oblig)

En av de største **fordelene** med å implementere tråder i brukerområdet (user space) er at det gir en mer effektiv bruk av ressurser og mindre overhead enn å bruke tråder som administreres av os. Når tråder administreres av os, krever det en overgang fra brukermodus til kjernemodus, noe som kan være en dyr operasjon. Ved å administrere trådene i brukerområdet, kan denne overgangen unngås og trådene kan skape og ødelegges raskere.

En annen fordel er at det gir større grad av kontroll over trådene og hvordan de kjører. Fordi trådene administreres av applikasjonen selv, kan applikasjonen selv ta beslutninger om hvordan tråden skal prioritertes og hvordan de skal kommunisere og dele ressurser. Dette gir en mulighet for mer finjustering og optimalisering av trådhåndtering for å møte spesifikke applikasjonskrav.

En av de største ulempene med å implementere tråder i brukerområdet er at det kan være vanskeligere å implementere riktig og sikkert. Når trådadministrasjonen blir overlatt til applikasjonen, må applikasjonen selv ta ansvar for å unngå problemer som race conditions og deadlocks. Dette krever ofte mye innsats og kan føre til at applikasjonen blir mer kompleks og mindre pålitelig.

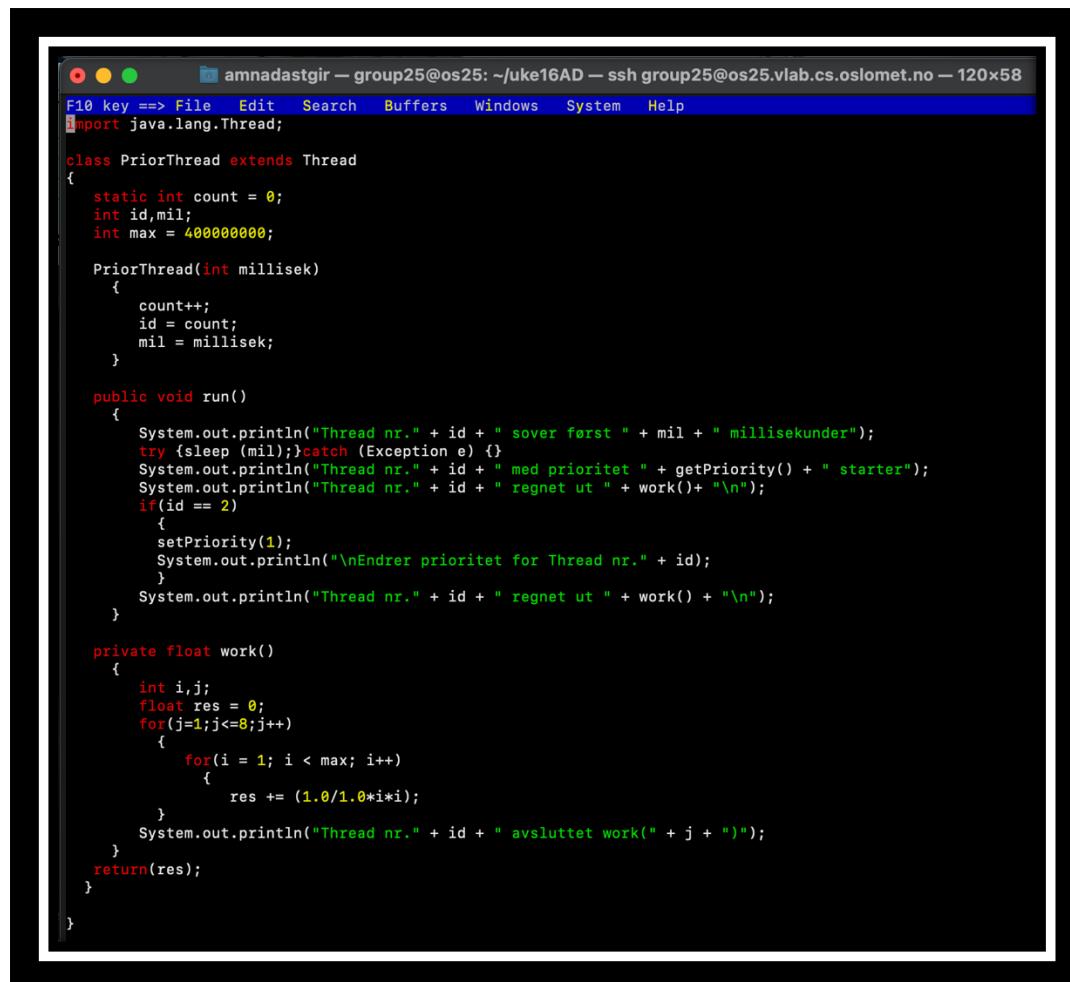
En annen ulempe er begrenset skalerbarhet. Når tråder administreres av os, kan flere tråder enkelt kjøres på flere prosessorer. Med tråder som administreres i brukerområdet, kan applikasjonen kun dra nytte av enkelt prosessorer, noe som kan føre til at ytelsen begrenses på høyt belastede systemer.

2. (Oblig)

Svar med bruk av kilder: Linux operativsystemet behandler Java Virtual Machine (JVM) prioriteten ved hjelp av prosessprioriteter. Hver prosess i Linux har en prioritetsverdi som bestemmer hvor mye CPU-tid den vil motta sammenlignet med andre prosesser. Når JVM kjører, vil den få tildelt en standard prioritet basert på systemets konfigurasjon, men dette kan endres ved å bruke kommandoer som "nice" og "renice".

"nice" kommandoen kan brukes til å øke eller redusere prioriteringen til JVM, mens "renice" kan brukes til å endre prioriteringen til en JVM-prosess som allerede kjører. Ved å endre prioriteringen til JVM kan du gi den mer eller mindre CPU-tid, avhengig av behovene til systemet. Dette kan hjelpe deg med å optimalisere ytelsen til Java-applikasjoner som kjører på Linux-systemer.

Første del av Prior.java:



The screenshot shows a terminal window titled "amnadarstgir" with the command "group25@os25: ~/uke16AD" and "ssh group25@os25.vlab.cs.oslomet.no". The window displays Java code for a thread class named "PriorThread". The code includes a constructor "PriorThread(int millisek)", a "run()" method that prints thread information and sleeps for "mil" milliseconds, and a "work()" private method that performs a floating-point calculation. The code uses static variables "count", "id", and "mil", and a constant "max = 400000000". It also includes exception handling and priority setting logic for threads 1 and 2.

```
import java.lang.Thread;

class PriorThread extends Thread
{
    static int count = 0;
    int id,mil;
    int max = 400000000;

    PriorThread(int millisek)
    {
        count++;
        id = count;
        mil = millisek;
    }

    public void run()
    {
        System.out.println("Thread nr." + id + " sover først " + mil + " millsekunder");
        try {sleep (mil);}catch (Exception e) {}
        System.out.println("Thread nr." + id + " med prioritet " + getPriority() + " starter");
        System.out.println("Thread nr." + id + " regnet ut " + work()+"\n");
        if(id == 2)
        {
            setPriority(1);
            System.out.println("\nEndrer prioritet for Thread nr." + id);
        }
        System.out.println("Thread nr." + id + " regnet ut " + work() + "\n");
    }

    private float work()
    {
        int i,j;
        float res = 0;
        for(j=1;j<8;j++)
        {
            for(i = 1; i < max; i++)
            {
                res += (1.0/1.0*i*i);
            }
            System.out.println("Thread nr." + id + " avsluttet work(" + j + ")");
        }
        return(res);
    }
}
```

Andre del av koden:

```
class Prior
{
    public static void main(String args[])
    {
        System.out.println("\nStarter to threads!\n");
        PriorThread s1 = new PriorThread(0);
        s1.start();
        s1.setPriority(5);
        System.out.println("Default prioritet er " + s1.NORM_PRIORITY + " for en thread");
        System.out.println("Max er " + s1.MAX_PRIORITY + " og min er " + s1.MIN_PRIORITY + "\n");

        PriorThread s2 = new PriorThread(0);
        s2.setPriority(10);
        s2.start();
    }
}
```

Under kjører jeg koden:

```
[group25@os25:~/uke16AD$ jed Prior.java
[group25@os25:~/uke16AD$ javac Prior.java
[group25@os25:~/uke16AD$ java Prior

Starter to threads!

Default prioritet er 5 for en thread
Max er 10 og min er 1

Thread nr.1 sover først 0 millisekunder
Thread nr.2 sover først 0 millisekunder
Thread nr.1 med prioritet 5 starter
Thread nr.2 med prioritet 10 starter
Thread nr.2 avsluttet work(1)
Thread nr.1 avsluttet work(1)
Thread nr.2 avsluttet work(2)
Thread nr.1 avsluttet work(2)
Thread nr.2 avsluttet work(3)
Thread nr.1 avsluttet work(3)
Thread nr.2 avsluttet work(4)
Thread nr.1 avsluttet work(4)
Thread nr.2 avsluttet work(5)
Thread nr.1 avsluttet work(5)
Thread nr.2 avsluttet work(6)
Thread nr.1 avsluttet work(6)
Thread nr.2 avsluttet work(7)
Thread nr.1 avsluttet work(7)
Thread nr.2 avsluttet work(8)
Thread nr.2 regnet ut 4.8357033E24

Endrer prioritet for Thread nr.2
Thread nr.1 avsluttet work(8)
Thread nr.1 regnet ut 4.8357033E24

Thread nr.2 avsluttet work(1)
Thread nr.1 avsluttet work(1)
Thread nr.2 avsluttet work(2)
Thread nr.1 avsluttet work(2)
Thread nr.2 avsluttet work(3)
Thread nr.1 avsluttet work(3)
Thread nr.2 avsluttet work(4)
Thread nr.1 avsluttet work(4)
Thread nr.2 avsluttet work(5)
Thread nr.1 avsluttet work(5)
Thread nr.2 avsluttet work(6)
Thread nr.1 avsluttet work(6)
Thread nr.2 avsluttet work(7)
Thread nr.1 avsluttet work(7)
Thread nr.2 avsluttet work(8)
Thread nr.2 regnet ut 4.8357033E24

Thread nr.1 avsluttet work(8)
Thread nr.1 regnet ut 4.8357033E24
```

Koden kjører to tråder som utfører beregninger og sover i noen millisekunder før de starter å jobbe igjen. Også endrer jeg prioritet på en av trådene fra 10 til 1 og jeg lar trådene fortsette å jobbe.

Etter å ha eksperimentert med ulike prioriteter i tråder ser vi at tråden med lavest prioritet får tildelt mindre tid enn den andre tråden. Det er i midlertidig verdt å merke seg at endringer i trådprioriteter ikke alltid har en stor effekt på ytelsen til et program, og det kan være andre faktorer som påvirker hvordan trådene blir prioritert og tildelt CPU-tid på ett gitt tidspunkt.

Opsjonen **-XX:ThreadPriorityPolicy=1** indikerer at JVM skal bruke en «round-robin» strategi for å velge hvilken tråd som skal få tildelt en CPU-tid, uavhengig av trådens prioritet. Dette kan føre til at trådene blir tildelt mer jevn CPU-tid, selv om de har forskjellige prioriteter. Når man kjører Java som vanlig bruker med denne opsjonen, vil effekten avhenge av hvordan systemressursene er fordelt.

Hvis det er mange andre prosesser som konkurrerer om CPU-tid, kan denne opsjonen føre til at Java-trådene får tildelt mindre CPU-tid enn de ville ha fått ellers.

Under kjører jeg koden på en CPU. På siden kan vi se at det er 14 threads som jobber.

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND | nTH | P |
|--------|------|----|----|-------|-------|-------|---|------|------|---------|---------|-----|---|
| 486544 | root | 20 | 0 | 32.3g | 41272 | 25392 | S | 80.1 | 0.0 | 0:07.61 | java | 14 | 0 |

3. (Ikke oblig). Gjør ikke denne oppgaven her fordi jeg i øyeblikket ikke har tilgang til en Windows PC og jeg gir ikke lage en vm i virtualbox.

4. (Oblig)

Dette programmet oppretter to tråder som øker og reduserer en felles saldo-variabel uten noen form for synkronisering. Det som skjer når de to trådene kjører parallelt, er at de konkurrerer

om å endre saldo-variabelen. Dette kan føre til en race condition, der resultatet avhenger av hvilken tråd som får tilgang til variabelen først.

I løpet av kjøringen kan man få ulike resultater hver gang, da rekkefølgen og timingen av trådene som kjører, kan variere. Resultatet kan også påvirkes av maskinvaren, og om trådene kjører på samme CPU eller forskjellige CPU-er.

Når programmet kjøres med "taskset -c 0 java NosynchThread", tvinges programmet til å kjøre begge trådene på samme CPU-kjerne (i dette tilfellet kjerne 0). Dette vil øke sannsynligheten for race condition, da de to trådene konkurrerer om å få tilgang til samme CPU-ressurs. Dette kan føre til enda flere kollisjoner mellom trådene og dermed øke sjansene for gale resultater.

Når trådene kjører på samme CPU-kjerne, vil antall mulige kollisjoner øke, sammenlignet med når de kjører på forskjellige CPU-er. Dette skyldes at de deler samme CPU-ressurs og konkurrerer om å få tilgang til denne ressursen samtidig. Når trådene kjører på forskjellige CPU-er, vil de ha separate ressurser og vil dermed ikke kolidere så ofte.

```
/ 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);
    }
}
```

63.1

5. (Ikke oblig)

Race condition (konkurransebetingelse) oppstår når to eller flere tråder i et program forsøker å samhandle med en delt ressurs (som en variabel, fil eller nettverksressurs) samtidig, og resultatet avhenger av rekkefølgen og timingen av trådene som utfører operasjonene. Dette kan føre til uventede eller uønskede resultater, som feil i programmet, datakorrasjon eller programkrasj.

Race condition oppstår når trådene ikke er synkronisert eller koordinert på riktig måte, og konkurrerer om å få tilgang til den delte ressursen uten noen form for beskyttelse eller koordinering. Dette kan føre til at en tråd overskriver en annen tråds data, eller at dataene blir inkonsistente på grunn av inkonsistente eller uforutsigbare operasjoner.

For å unngå race condition, må trådene koordineres og synkroniseres på riktig måte, ved hjelp av teknikker som låsing, venting, semaforer eller monitorer, slik at trådene kan få tilgang til de delte ressursene i en kontrollert og koordinert måte, og unngå uventede og uønskede resultater.

6. (Oblig)

Thread.c:

```
group25@os26:~/uke16AD/oppg5$ cat thread.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int svar = 0;

extern void enlinje();

void *inc()
{
    printf("Starter; svar verdi: %d\n", svar);
    for (int i = 0; i < 10000000; i++)
    {
        enlinje();
    }
    printf("Avslutter; svar verdi: %d\n", svar);
}

int main()
{
    pthread_t thread1, thread2;

    /* Lager uavhengige threads som utfører inc-funksjonen */
    pthread_create( &thread1, NULL, inc,NULL);
    pthread_create( &thread2, NULL, inc, NULL);

    /* Venter med join til begge tråder er ferdige */
    pthread_join( thread1, NULL);
    pthread_join( thread0, NULL);

    printf("Main avslutter; svar verdi: %d\n", svar);
    exit(0);
}
```

Minimal.s:

```
[group25@os25:~/uke16AD/oppg5$ cat minimal.s
.globl enlinje
enlinje:
    incl svar(%rip)
    ret
```

En.c:

```
[group25@os25:~/uke16AD/oppg5$ cat en.c
void enlinje()
{
    extern int svar;
    svar++;
}
```

Under endrer jeg koden i thread.c slik som oppgaven ber om. Kompilerer og kjører, og da ser jeg at jeg alltid får samme resultat.

```
[group25@os25:~/uke16AD/oppg5$ vim thread.c
[group25@os25:~/uke16AD/oppg5$ gcc -pthread thread.c minimal.s
[group25@os25:~/uke16AD/oppg5$ taskset -c 0 ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 4917353
Avslutter; svar verdi: 18659883
Avslutter; svar verdi: 20000000
Main avslutter; svar verdi: 20000000
[group25@os25:~/uke16AD/oppg5$ taskset -c 0 ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 5866988
Avslutter; svar verdi: 17518373
Avslutter; svar verdi: 20000000
Main avslutter; svar verdi: 20000000
[group25@os25:~/uke16AD/oppg5$ taskset -c 0 ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 6946557
Avslutter; svar verdi: 17503503
Avslutter; svar verdi: 20000000
Main avslutter; svar verdi: 20000000
```

Når begge trådene kjører parallelt, kan vi fortsatt se at verdien av "svar" øker likt og riktig for hver iterasjon av kritisk seksjon, fordi trådene blir tvunget til å vente på hverandre før de kan fortsette. Dette sikrer at programmet alltid produserer det forventede resultatet, uavhengig av timingen mellom trådene. Dermed vil vi alltid få det samme resultatet, selv om to uavhengige tråder kjører.

Fra hjelpemdir fordi jeg ikke girde å skrive svaret selv:

Når vi bruker en.c på den første måten, der funksjonen enlinje() er definert som en enkel inkrementering av variabelen svar, så kan det føre til forskjellige resultater avhengig av når trådene kjører og når de konkurrerer om å øke verdien av svar. Dette skyldes at trådene kan utføre instruksjonene sine på forskjellige tidspunkter, og det kan føre til at de leser og skriver

til svar-variabelen på forskjellige tidspunkter. Derfor kan resultatet avhenge av den spesifikke timingen av trådene.

Når vi kompilerer en.c til en.s, kan vi se at funksjonen enlinje() faktisk blir oversatt til tre instruksjoner i maskinkode: load, increment og store. Dette betyr at selv om høynivå-koden ser ut som en enkel inkrementering av variabelen, så er det faktisk flere instruksjoner i maskinkoden som utføres. Dette kan føre til ytterligere konkurranse mellom trådene om å få tilgang til svar-variabelen, siden det kan ta litt tid å utføre alle tre instruksjonene.

Når vi kompilerer med flagget -O, som betyr å aktivere kompilatorens optimalisering, kan kompilatoren gjøre noen optimaliseringer som reduserer antall instruksjoner i maskinkoden. Dette kan føre til mindre konkurranse mellom trådene og dermed mer forutsigbare resultater. Derfor vil resultatet avhenge mindre av timingen til trådene, og vi vil se en mer forutsigbar oppførsel uavhengig av kjøring.

7. (UKENS NØTT)

Under er C-programmet. Jeg måtte gjøre endringer grunnet feilmeldinger fra terminalen som blant annet at jeg må ha en returtype og definere 'int' foran main().

```
[group25@os25:~/uke16AD/oppg7$ cat program.c
int main() {
    int i,S = 0;
    for(i=1;i < 5;i++){
        S = S + i;
    }
    return 0;
}
```

Kompilerer og kjører:

```
[group25@os25:~/uke16AD/oppg7$ vim program.c
[group25@os25:~/uke16AD/oppg7$ gcc program.c
```

Koden:

```
[group25@os25:~/uke16AD/oppg7$ objdump -d a.out
```

Under bruker jeg hjelpemidler.

```
0000000000001129 <main>:
1129: f3 0f 1e fa        endbr64
112d: 55                 push    %rbp
112e: 48 89 e5          mov     %rsp,%rbp
1131: c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
1138: c7 45 f8 01 00 00 00  movl   $0x1,-0x8(%rbp)
113f: eb 0a              jmp    114b <main+0x22>
1141: 8b 45 f8          mov     -0x8(%rbp),%eax
1144: 01 45 fc          add    %eax,-0x4(%rbp)
1147: 83 45 f8 01        addl   $0x1,-0x8(%rbp)
114b: 83 7d f8 04        cmpl   $0x4,-0x8(%rbp)
114f: 7e f0              jle    1141 <main+0x18>
1151: b8 00 00 00 00 00  mov    $0x0,%eax
1156: 5d                 pop    %rbp
1157: c3                 retq
1158: 0f 1f 84 00 00 00 00  nopl   0x0(%rax,%rax,1)
115f: 00
```

- **f3 0f 1e fa** er **endbr64** instruksjonen, som brukes for å indikere at en grenseoverskridelse (branch overshoot) har skjedd.
- **55** instruksjonen pusher verdien av registeret **%rbp** på stabelen. Dette brukes for å lagre verdien av stabelpekeren, slik at vi kan gjenopprett den senere.
- **48 89 e5** instruksjonen flytter verdien av stabelpekeren (%rsp) inn i registeret %rbp. Dette setter rammeverket for funksjonen ved å sette stabelpekeren til bunnen av funksjonskallet.
- **c7 45 fc 00 00 00 00** instruksjonen flytter verdien **0x0** inn i minnet som ligger **4** bytes under %rbp. Dette setter variabelen som ligger nederst på stabelen til 0.
- **c7 45 f8 01 00 00 00** instruksjonen flytter verdien **0x1** inn i minnet som ligger **8** bytes under %rbp. Dette setter variabelen som ligger nest nederst på stabelen til 1.
- **eb 0a** instruksjonen er en kort form for **jmp**. Den hopper til instruksjonen som ligger **10** bytes lengre frem i koden.
- **8b 45 f8** instruksjonen flytter verdien som ligger **8** bytes under %rbp inn i registeret %eax. Dette er verdien av variabelen som ble satt til 1 tidligere.
- **01 45 fc** instruksjonen legger verdien i %eax til minnet som ligger **4** bytes under %rbp. Dette tilsvarer å legge verdien til variablen som ligger nederst på stabelen.
- **83 45 f8 01** instruksjonen legger til verdien **1** til minnet som ligger **8** bytes under %rbp. Dette tilsvarer å øke verdien av variablen som ligger nest nederst på stabelen med **1**.
- **83 7d f8 04** instruksjonen sammenligner verdien som ligger **8** bytes under %rbp med verdien **0x4**. Dette tilsvarer å sammenligne verdien av variablen som ligger nest nederst på stabelen med **4**.

- **7e f0** instruksjonen hopper tilbake til instruksjonen som ligger **16 bytes** bak i koden hvis sammenligningen tidligere var mindre enn eller lik 4.
- **b8 00 00 00 00** instruksjonen flytter verdien **0x0** inn i registeret %eax. Dette brukes til å indikere at programmet har avsluttet kjøringen uten noen

8. (UKENS NØTT)

Under er det koden. Jeg har en linje kode med system for å kunne se om koden kjører.

```
[group25@os25:~/uke16AD/oppg8$ cat løkke.java
class For{
    public static void main(String args[]){
        int i,S = 0;
        for(i=1;i < 5;i++){
            S = S + i;
        }
        System.out.println("The sum of numbers from 1 to 4 is: " + S);
    }
}
```

Jeg brukte hjelpe midler for å skjonne koden:

```
[group25@os25:~/uke16AD/oppg8$ javap -c For
Compiled from "løkke.java"
class For {
    For();
    Code:
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V
    }
    public static void main(java.lang.String[]);
    Code:
        0: iconst_0
        1: istore_2
        2: iconst_1
        3: istore_1
        4: iload_1
        5: iconst_5
        6: if_icmpge   19
        9: iload_2
       10: iload_1
       11: iadd
       12: istore_2
       13: iinc      1, 1
       16: goto     4
       19: getstatic #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       22: iload_2
       23: invokedynamic #3,  0          // InvokeDynamic #0:makeConcatWithConstants:(I)Ljava/lang/String;
       28: invokevirtual #4              // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    }
}
```

I bytekoden initialiseres "i" og "S" på følgende måte:

"iconst_0" legger verdien 0 på stacken.

"istore_2" tar verdien 0 fra stacken og lagrer den i variabelen "S".

"iconst_1" legger verdien 1 på stacken.

"istore_1" tar verdien 1 fra stacken og lagrer den i variabelen "i".

Dermed er "i" initialisert til 1 og "S" er initialisert til 0 i koden.

Videre brukes "iload_1" og "iload_2" for å hente verdiene av "i" og "S" fra variablene, og "iinc" brukes for å øke verdien av "i" med 1 i hver iterasjon av løkken. "iadd" brukes til å

legge verdien av "i" til verdien av "S" i hver iterasjon, som representerer summen av tallene fra 1 til 4. Til slutt skrives summen ut ved hjelp av "getstatic", "invokedynamic" og "invokevirtual" instruksjoner.

Så i bytecode-koden representerer "i" og "S" to heltallsvariabler som brukes til å telle gjennom løkken og beregne summen av tallene i løkken.

9. (Oblig)

Mitt notat fra forelesningen som svarer på spørsmålet.

HARDWARE STØTTET VISUALISERING

Grunnen til at det er et problem med privilegerte instruksjoner er fordi det finnen sensitive instruksjoner som ikke er privilegerte. Når vi har en VM så kjører den oppå en hypervisor og hypervisor er på en måte det vi har tenkt på som os-kjernen tidligere. Den kjører i kernel mode. Også kjører det en eller flere VM oppå hypervisor, og de vil være prosesser som kjører i usermode. Og da får man et problem for hvis gjeste os tror at det er et ekte os som kjører på ordentlig hardware, det vil jo forvente når gjeste-os gjør en privilegert instruksjon, eller for å være presis, når en gjeste-os gjør en sensitiv instruksjon (en instruksjon som bare kan gjøres i kernel mode), så forventer gjeste-os at det faktisk skjer noe. For la oss si gjeste-os utfører et POPF for å skru av på et interrupt, os kjermen må jo ha lov til å gjøre det. Men hvis da POPF ikke er en privilegert instruksjon, altså hvis POPF utføres og ingenting skjer, så vil jo ikke det gjeste-os i bildet under å fungere. Derfor er det viktig at en sensitiv instruksjon må trappe til hypervisor, slik at hypervisor kan behandle dette riktig. Den må skjonne at 'siden det er gjeste-os som ønsker å fjerne et interrupt, da må jeg fjerne interrupt'.

Hvis vi da har en brukerprosess som gjør det, så vil den også trappe, men da vil hypervisor avgjøre at dette er en brukerprosess, den får ikke lov til det. Det er dette som gjorde at hardware begynte å støtte visualisering etter 2005.

10. (Oblig)

Pakker ut fila.

```
[group25@os25:~/uke16AD/oppg10$ tar -xzf virt.tgz
```

Bytter til mappa og tar ls:

```
[group25@os25:~/uke16AD/oppg10$ cd virt
[group25@os25:~/uke16AD/oppg10/virt$ ls -l
total 28
-rwxr-xr-x 1 group25 group25 81 Apr 22 2017 comp.sh
-rw----- 1 group25 group25 437 Apr  3 2019 fork.c
-rw----- 1 group25 group25 241 Apr  3 2019 forkwait.c
-rw----- 1 group25 group25 104 Apr  3 2019 getppid.c
-rw----- 1 group25 group25 196 Apr  3 2019 gettimeofdayday.c
-rwx----- 1 group25 group25 187 Apr 22 2018 runiter.bash
-rw----- 1 group25 group25 202 Apr 24 2017 sum.c
```

11. (Oblig)**EFFEKTIVITET FOR 4 C-PROGRAMMER NÅR DE KJØRER BARE
METAL OG VIRTUELT**

Vi skal se hvordan 4 programmer kjøres virtuelt og bare metal (som er på en fysisk server).

Den første koden her er en sum slik som de vi har hatt tidligere, den kjører stort sett bare i user mode, og bør ha omtrent samme ytelse virtuelt.

```
#include <stdio.h>
#define uint64_t unsigned long int

int main(void) {
    uint64_t i, s = 0;
    for (i=0; i<50000000000; i++) {
        s = s + i;
    }
    return(0);
}
```

Figur 1 SUM.C

Neste program er et veldig enkelt systemkall, kan være tyngre virtuelt, men bør gå ganske greit. Består av å gjøre systemkallet getppid om og om igjen.

```
#include<unistd.h>

int main(void) {
    int i;
    for (i=0; i<20000000; i++) {
        getppid();
    }
    return(0);
}
```

Figur 2 GETPPID.C

Er et litt mer kompleks systemkall som også kaller andre systemkall. Kan tenkes å være virtuelt tungt fordi det da må trappes mange ganger til kjernen. Fordi når en vanlig applikasjon gjør et systemkall innenfor et gjeste-os, så må gjeste-os gjøre instruksjoner som er sensitive.

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

int main(void)
{
    struct timeval c;           ↴
    int i;
    for(i=0; i<40000000; i++) {
        gettimeofday(&c,NULL);
    }
    return(0);
}
```

Figur 3 GETTIMEOFDAY.C

Fork program som setter opp omrent 10k forks. Det siste eksempelet gir også mange systemkall og en god del minne-bruk. Dette vil for en eller annen grunn kjøre hurtigere på VM.

```
#include <unistd.h>
#include <sys/wait.h>

#include<unistd.h>

int main(void) {
    int i,pid;
    for (i=0; i<10000; i++) {
        pid = fork();
        if (pid < 0) return -1;
        if (pid == 0) return 0;
        waitpid(pid,NULL,0);
    }
    return(0);
}
```

Figur 4 FORKWAIT.C

Dette er bare metal. Vi har kompilert de fire programmene og kjørt de 5 ganger for å se litt hvordan tidene varierer på en fysisk server (intel server).

```
Total tid programmene bruker i sekunder.
```

```
root@intel:~/virt# ./runiter.bash
sum getppid gettimeofday forkwait
1.47;0.89;0.70;2.49;
1.46;0.89;0.71;2.54;
1.46;0.85;0.70;2.48;
1.47;0.89;0.71;2.49;
1.46;0.85;0.71;2.45;
```

- De to første programmene går som forventet omtrent like raskt.
- Noe overraskende går det litt raskere på den virtuelle maskinen.
- Kall til gettimeofday går vesentlig saktere på VM-en , ikke unaturlig, trap til hypervisor tar mye tid
- Overraskende at fork-kall går raskere på den virtuelle maskinen
- QEMU gjør binær-versjonskifte av koden, kan spille en rolle her

```
root@server2:~/virt# ./runiter.bash
sum getppid gettimeofday forkwait
1.46;0.75;1.69;0.44;
1.39;0.75;1.69;0.43;
1.39;0.75;1.69;0.44;
1.47;0.75;1.68;0.51;
1.39;0.75;1.69;0.46;
```

Virtuelle maskinen uten -enable-kvm. Vi ser at det går vesentlig saktere med en ren qemu-emulering uten bruk av hardware-aksellerasjon. Programmet sum som kjører i user mode bruker omtrent åtte ganger så lang tid og forkwait bruker 20 ganger så lang tid.

```
root@server2:~/virt# ./runiter.bash
sum getppid gettimeofday forkwait
10.91;4.38;3.67;10.47;
11.06;4.28;3.42;10.51;
10.51;4.38;3.35;10.73;
10.80;4.29;3.50;10.24;
10.04;4.34;3.54;10.58;
```

En unikernel er en minimalistisk operativsystemkjerner. Tanken rundt en unikernel er at når man kjører virtuelle maskiner, slik som ubuntu. Hvis vi går tilbake og ser så startet vi opp en stor ubuntu kjernen på 2 GB som vist under. Når denne er startet så bruker den masse minne og

vi satte av en GB med minne, så den klarer å kjøre på 1GB men det er relativt mye minne.

Tanken med unikernel er at veldig mye av det som finnes i imaget under er egentlig overflødig. La oss si vi skal kjøre en webserver, da er det mye i koden under som aldri kommer til å brukes. Tanken med unikernel er at den nøyaktig skal gjøre det som trengs av den. Os delen skal være nøyaktig det man trenger av den.

12. (EFFICODE)

```
amnadarstgir --@a726b89032c:~ ssh -p 5264 s264@intel1.vlab.cs.oslomet.no - 200x56
<tr><td valign="top"></td><td align="right">2023-04-16 23:43 </td><td align="right"> 17K</td><td align="right"> 17K</td><td align="right"> 17K</td>
<tr><td valign="top"></td><td align="right">2023-04-16 23:43 </td><td align="right"> 17K</td><td align="right"> 17K</td><td align="right"> 17K</td>
<tr><td colspan="5"><br></td></tr>
</table>
</body></html>
[root@4a726b89032c ~]# ls -l
total 56
dr-xr-xr-x  2 root  root  4096 Aug  9  2022 afs
lrwxrwxrwx  1 root  root   7 Aug  9  2022 bin -> /usr/bin
dr-xr-xr-x  2 root  root  4096 Aug  9  2022 boot
drwxr-xr-x  5 root  root  848 Apr  9  2022 dev
drwxr-xr-x  1 root  root  4096 Apr  9  2022 devpts
drwxr-xr-x  2 root  root  4096 Aug  9  2022 home
lrwxrwxrwx  1 root  root   7 Aug  9  2022 lib -> /usr/lib
lrwxrwxrwx  1 root  root   9 Aug  9  2022 lib64 -> /usr/lib64
drwxr----- 2 root  root  4096 May 11 05:49 lost+found
drwxr-xr-x  2 root  root  4096 Aug  9  2022 media
drwxr-xr-x  2 root  root  4096 Aug  9  2022 mnt
drwxr-xr-x  2 root  root  4096 Aug  9  2022 opt
dr-xr-xr-x  5199 root  root   9 Apr 16 22:16 proc
dr-xr-xr-x  2 root  root  4096 Mar 11 05:49 root
drwxr-xr-x  2 root  root  4096 Mar 11 05:49 run
drwxrwxrwx  1 root  root   8 Aug  9  2022 sbin -> /usr/sbin
drwxr-xr-x  2 root  root  4096 Aug  9  2022 srv
dr-xr-xr-x  1 nobody nobody  0 Apr 16 22:16 sys
drwxrwxr-x  2 root  root  4096 Mar 11 05:49 tmp
drwxr-xr-x  12 root  root  4096 Mar 11 05:49 usr
drwxr-xr-x  18 root  root  4096 Mar 11 05:49 var
[root@4a726b89032c ~]# curl https://os.cs.oslomet.no/os/hvaerdette/p264
Warning: Binary output can mess up your terminal. Use "--output -" to tell
Warning: curl to output it to your terminal anyway, or consider "--output
Warning: <FILE>" to save to a file.
[root@4a726b89032c ~]# curl -O https://os.cs.oslomet.no/os/hvaerdette/p264
  % Total    % Received   % Xferd  Average Speed   Time   Time     Current
                                 Dload  Upload Total   Spent   Left  Speed
100 17032  100 17932   0   282k      0  --:--:--:--:--:--:--:--:--:--:--:286k
[root@4a726b89032c ~]# ./p264
bash: ./p264: No such file or directory
[root@4a726b89032c ~]# ./p282
bash: ./p282: No such file or directory
[root@4a726b89032c ~]# ./p100
bash: ./p100: No such file or directory
[root@4a726b89032c ~]# ./p264
bash: ./p264: Permission denied
[root@4a726b89032c ~]# ./p264
bash: ./p264: Permission denied
[root@4a726b89032c ~]# ./p264
bash: ./p264: command not found
[root@4a726b89032c ~]# ./p264
bash: ./p264: Permission denied
[root@4a726b89032c ~]# chmod 700 p264
[root@4a726b89032c ~]# ./p264
ob74kA81fz
[root@4a726b89032c ~]#
```