

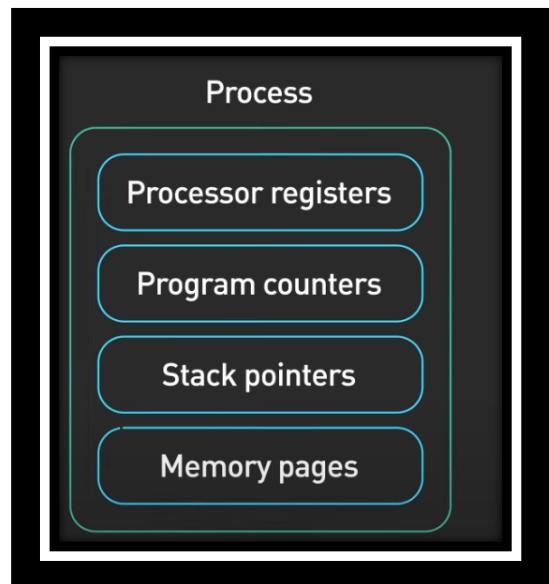
Notater 16 – Java Threads og synkronisering

SIST:

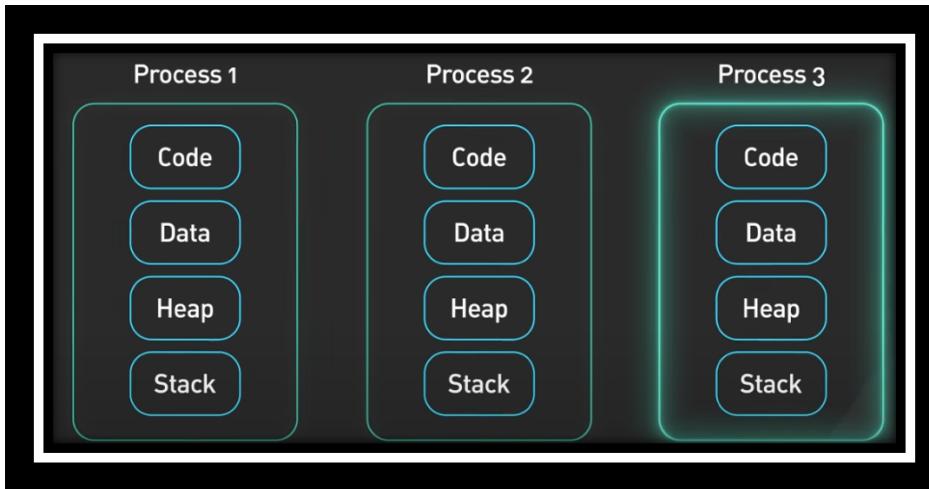
Forrige gang så vi på plattformavhengighet, hva som skjer når man prøver å flytte programmer på plattformer. Hovedkonklusjonen er at programmer som er kompilert, som C og C++, som kompileres til maskinkode er plattformavhengige. Det betyr at de kun kan kjøre der de er kompilert, altså på den plattformen de kompileres på. De fleste moderne språk som Java og Python, C-sharp også, kjører VM, som betyr man kompilerer til en type byte kode, og det er en type VM på hver plattform, og derfor kan man teoretisk flytte kode fra en plattform til en annen.

PROSESS VS THREAD:

Først må vi se på hva et program er. Et program er en kjørbar fil og inneholder kode, eller et sett med instruksjoner, som er lagret som en fil på disk. Når koden i et program lastes inn til minnet og kjøres med en prosessor → dannes det til en prosess. En aktiv prosess inkluderer også ressursene programmet trenger til å kjøres. Disse ressursene håndteres av operativsystemet. Noen eksempler er prosess registere, program counters, stack pointers, minne sider designert til prosessen for dens heap og stack, etc.



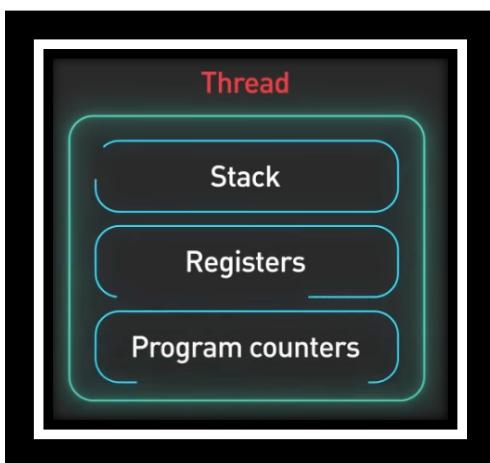
En viktig egenskap ved en prosess som er viktig å nevne: en prosess kan ikke ødelegge minneområdet til en annen prosess. Dette betyr at når en prosess feiler, fortsetter andre prosesser å kjøre. Chrome er kjente for å utnytte dette ved å kjøre hver fane i sin egen prosess. Når en fane oppfører seg rart av en bug eller et angrep, vil alle andre faner være upåvirket.



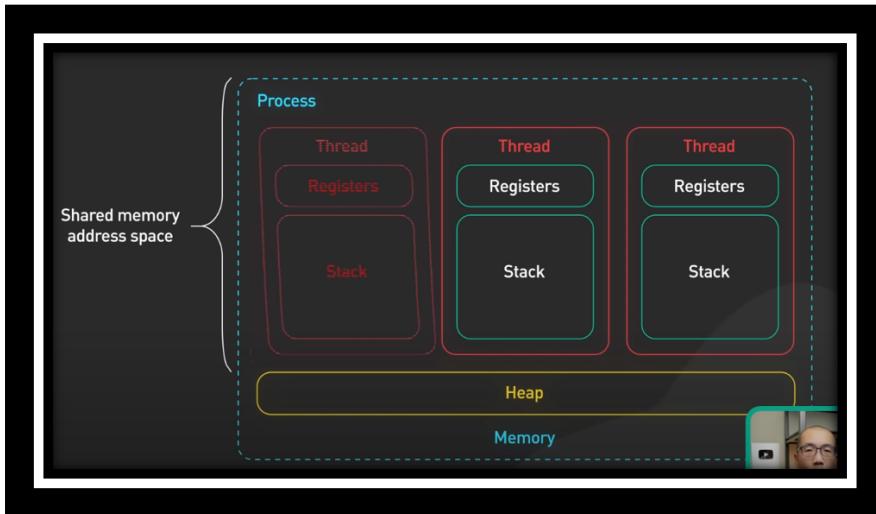
Så hva er en thread? En thread er en enhet for utførelse innenfor en prosess. Hver prosess har minst en tråd som er hoved thread. Det er vanlig at prosesser har mange threads.



Hver thread har sin egen stack. Tidligere nevnte vi at registre, program counters, og stack pointers er en del av prosesser. Det er mer nøyaktig å si at de delene tilhører en thread.

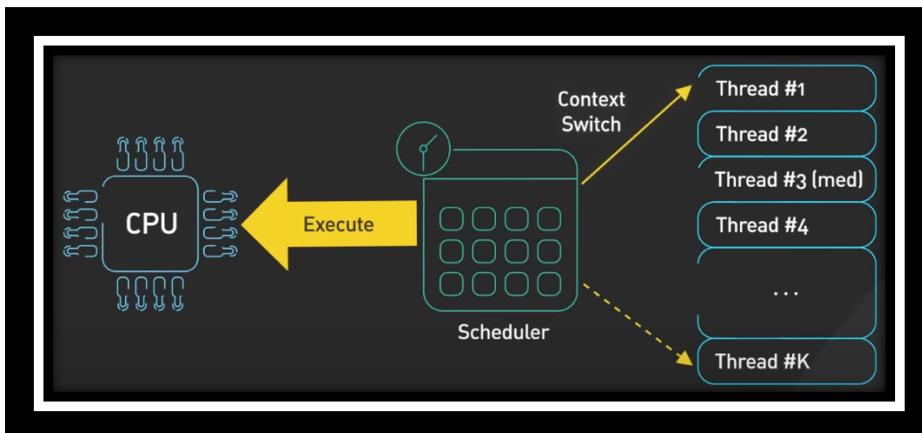


Threads innenfor en prosess deler minne-adresse område. Det er mulig å kommunisere mellom trådene ved å bruke det minne-adresse området. En thread sin dårlige oppførelse kan påvirke hele prosessen.

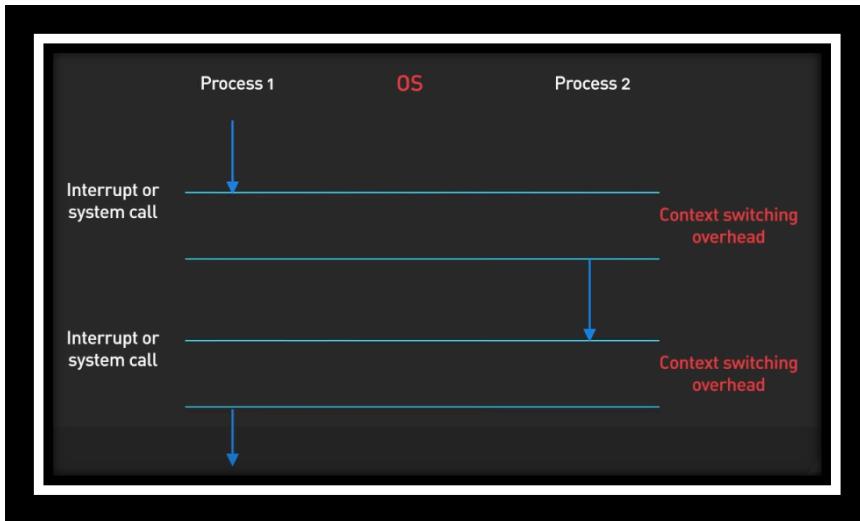


HVORDAN KAN OS RUNNE THREAD ELLER PROSESS PÅ EN CPU?

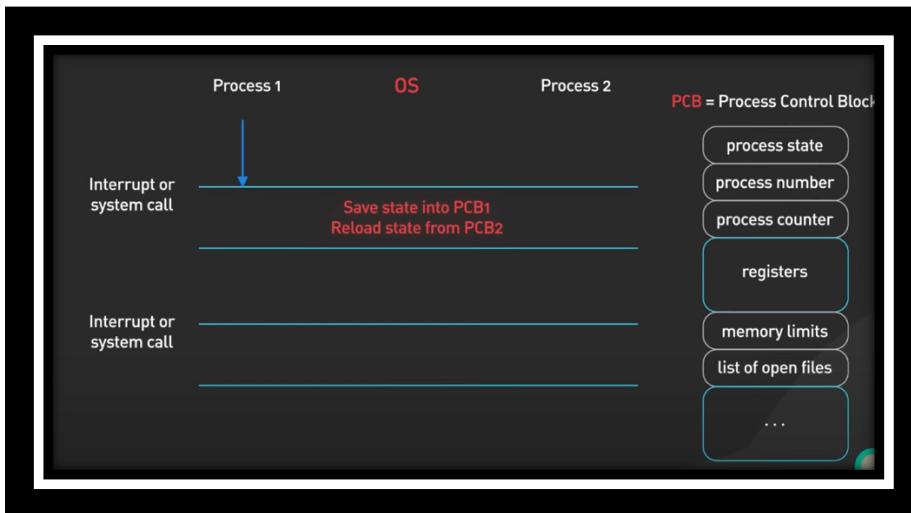
Dette håndteres av context switching.



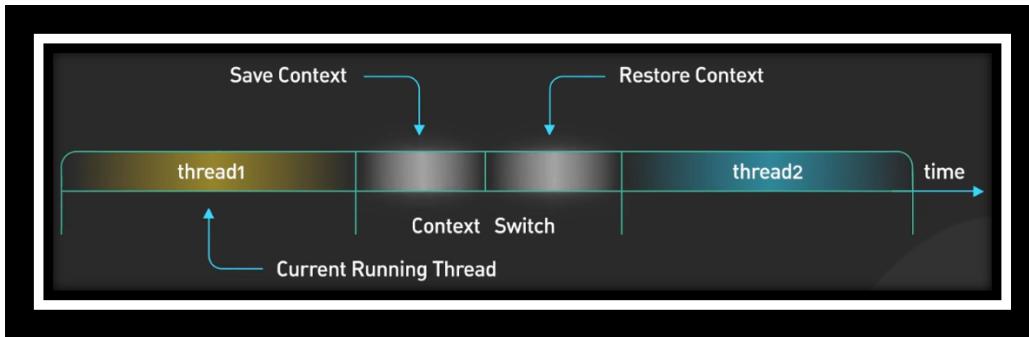
Under et context switch byttes en prosess ut av CPU så en annen prosess kan kjøre.



Operativsystemet lagrer tilstandene til den nåværende kjørende prosessen så prosessen kan gjenopprettes og fortsettes på et senere tidspunkt. Også gjenoppretter den tidligere lagrede tilstanden av en annerledes prosess og fortsetter kjøringen av den prosessen.



Context switching er dyrt. Det innebærer lagring og lasting av registre, bytting ut av minne områder, og oppdatering av varierte kernel datastrukturer. Switching execution mellom threads innebærer også context switching. Det er generelt raskere å bytte context mellom threads enn mellom tråder. Det er færre steder å tracke, mer viktig, siden threads deler samme minne-adresse område er det ingen grunn til å nytte ut virtuelle minne sider, som er en av de mer dyre handlingene under context switch.



Context switching er så dyrt at det er andre mekanismer som prøver å minimalisere det. Noen eksempler er fiber og co-routines. Disse mekanismene bytter kompleksitet for enda en lavere context-switching kostnad. Generelt er de scheduled samarbeidende, og de må yield control (gi fra seg CPU) for at andre skal kjøre. Med andre ord kan vi si at applikasjonen selv håndterer task scheduling. Det er ansvaret til en applikasjon å sørge for at en lenge kjørende oppgave er delt opp med yield periodisk.

CalcMany. Java – PROGRAM SOM STARTER 20 TRÅDER

Det koster veldig lite for operativsystemet å ha mange tråder stående klare for å kjøre. Vi kunne i forrige ukes oppgaver se omtrent 20 tråder – hjelpe-threads.

I klassen under main definerer vi en ny thread som er 20. så det vi skal gjøre er å starte 20 nye uavhengige tråder. Også opprettes det et nytt trådobjekt med new, også skriver vi ut at det startes 20 threads. Også går vi inn i en for-løkke og da startes da først thread nummer 0, også oppover, også skriver vi ut trådene som starter. Også kaller vi på start()-funksjonen.

```

class CalcMany
{
    public static void main(String args[])
    {
        int k;
        int threads = 20;
        CalcThread tr[] = new CalcThread[threads];
        System.out.println("Starts " + threads + " threads !\n");

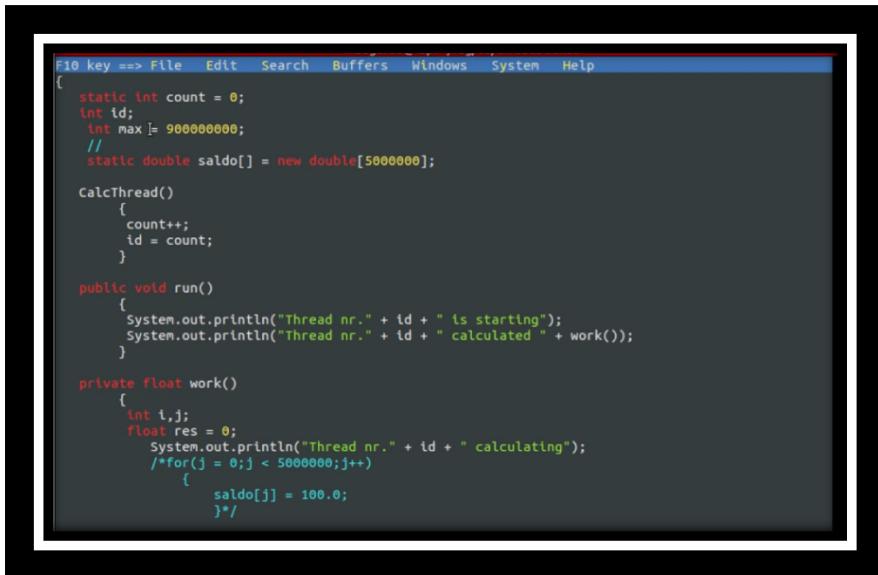
        for(k = 0;k < threads;k++)
        {
            tr[k] = new CalcThread();
            System.out.println("Thread has id " + tr[k].id + "\n");
            tr[k].start();
        }
    }
}

```

Når `CalcThread()` startes så kjøres denne init-metoden. Count++, da er det viktig å huske at count er definert som static, som betyr at det finnes bare en av den for alle trådene. Vi har 20 tråder men bare en enkelt count. En variabel som deklarereres ikke static, slik som int id, den

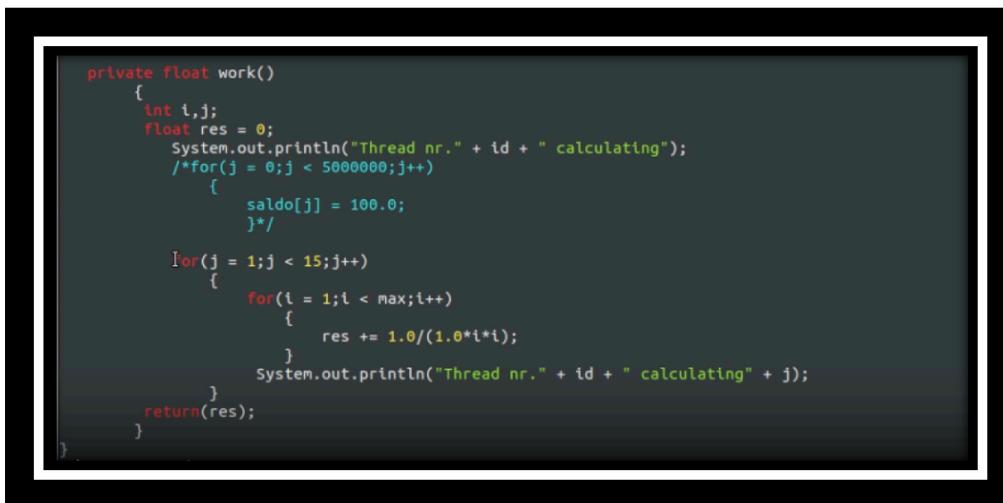
har vi da 20 stykker av. Siden max også er static skal den også lages 20 av. Men det viktigste her er count og id, så hver gang vi øker count, da telles den felles variabelen og da telles det opp fra 0 til 20. også setter vi en id lik count, så da blir det en id for hver.

Også printer vi thread nr x starter og at thread nr x kalkuleres, også kalles work()-funksjonen.



```
F10 key ==> File Edit Search Buffers Windows System Help
{
    static int count = 0;
    int id;
    int max = 90000000;
    //
    //static double saldo[] = new double[5000000];
    CalcThread()
    {
        count++;
        id = count;
    }
    public void run()
    {
        System.out.println("Thread nr." + id + " is starting");
        System.out.println("Thread nr." + id + " calculated " + work());
    }
    private float work()
    {
        int i,j;
        float res = 0;
        System.out.println("Thread nr." + id + " calculating");
        /*for(j = 0;j < 5000000;j++)
        {
            saldo[j] = 100.0;
       }*/
    }
}
```

Det som skjer inni work() er at 15 ganger så regner den ut en kjempe lang løkke. Java er så rask at selv om man står og regner ut en kjempe lang løkke så går det relativt fort. Hovedpoenget her med tråder er at når vi setter dem opp her så kjører de samtidig.



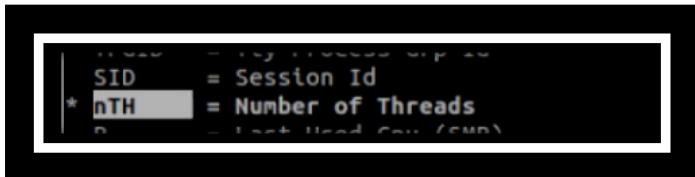
```
private float work()
{
    int i,j;
    float res = 0;
    System.out.println("Thread nr." + id + " calculating");
    /*for(j = 0;j < 5000000;j++)
    {
        saldo[j] = 100.0;
    }*/
    for(j = 1;j < 15;j++)
    {
        for(i = 1;i < max;i++)
        {
            res += 1.0/(1.0*i*i);
        }
        System.out.println("Thread nr." + id + " calculating" + j);
    }
    return(res);
}
```

Kompilerer og kjører

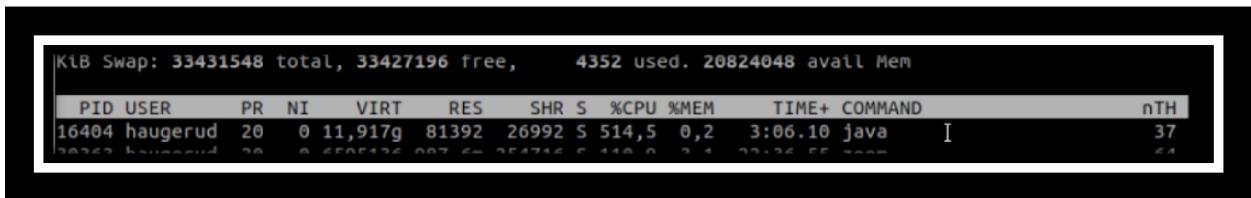


```
haugerud@lap:~/fag/os/threads$ jed CalcMany.java
haugerud@lap:~/fag/os/threads$ javac CalcMany.java
```

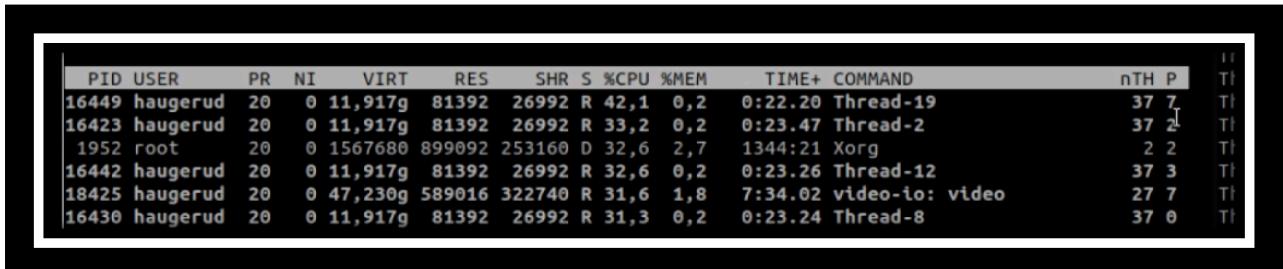
Når man ser på det i top, kan man i utgangspunktet se at det er en prosess som kjører. Vi velger fra top sin ekstra meny nTH (number of threads), så ser vi at den ene prosessen som kjører har 37 tråder:



Vi ser også at de andre prosessene som kjører har en rekke tråder, slik som zoom har 64. det som er spesielt med 20 av 37 som vi har på Java er at de er regnetråder.



Hvis vi taster stor H så kan vi se alle trådene. Hvis vi legger på last used CPU med top sitt ekstra menyvalg så ser vi at trådene er schedulert på forskjellige CPU. Java setter i gang mange tråder, også blir de schedulert av OS. Og akkurat likt er det med de andre programmene.



PRIORITET AV JAVA THREADS PÅ LINUX

Det vi skal se er at prioritet er litt merkelig implementert i java. Det er annerledes implementert i Windows og Linux. Java burde være helt plattformuavhengig, og her skal vi se at det ikke er det.



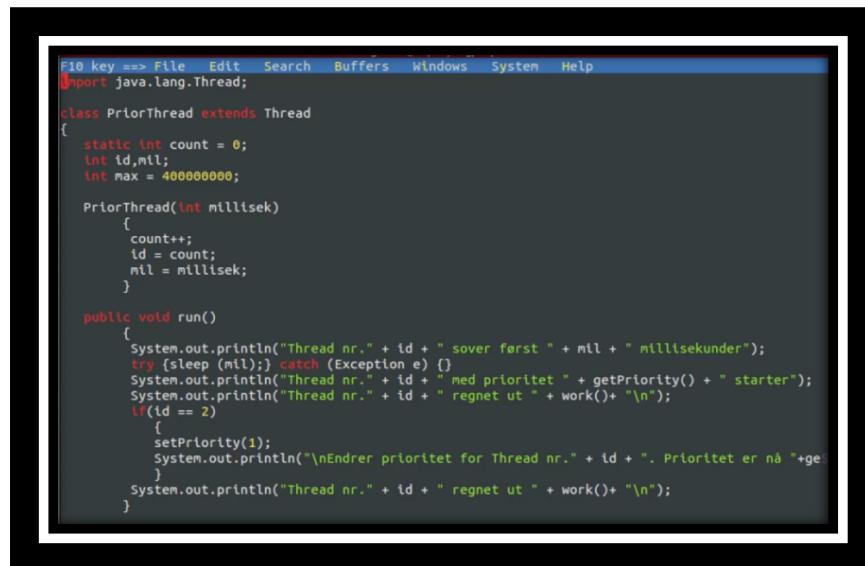
Dette programmet implementerer prioritet, og likner andre program vi har sett. Det sendes hvor mange millsekunder tråden skal sove før den starter. Begge har 0 så de startes med engang.

Her startes det to tråder, s1 og s2. s1 starter og vi setter prioritet til 5 som også er default, så hvis det ikke hadde blitt implementert så hadde prioriteten fortsatt vært 5. også er det noen variabler som skrives ut etter det. Også skrives det ut max-prioritet og min-prioritet. MAX-prioritet er 10 og MIN-prioritet er 1. dette er motsatt av nice (hvor snill man er med andre).

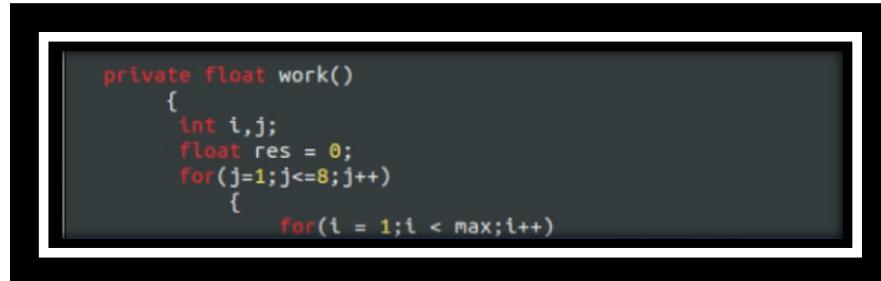
```
class Prior
{
    public static void main(String args[])
    {
        System.out.println("\nStarter to threads!\n");
        PriorThread s1 = new PriorThread(0);
        s1.start();
        s1.setPriority(5);
        System.out.println("Default prioritet er " + s1.NORM_PRIORITY + " for en thread");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

Koden er under



```
File key ==> File Edit Search Buffers Windows System Help
Import java.lang.Thread;
class PriorThread extends Thread
{
    static int count = 0;
    int id,mil;
    int max = 40000000;
    PriorThread(int millsek)
    {
        count++;
        id = count;
        mil = millsek;
    }
    public void run()
    {
        System.out.println("Thread nr." + id + " sover først " + mil + " millisekunder");
        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 + ". Prioritet er nå " +getPriority());
        }
        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++)
    }
```

```

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

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();
    }
}

```

Vi har en teller count og id. Også har vi millisekunder for senere da vi skal kjøre på Windows, da skal vi gi tråden noen millisekunder før den starter. Vi gir den noen millisekunder her også for å se hva som skjer, slik at det er enklere for oss.

Vi kan også se hvordan vi behandler to forskjellige tråder, fordi de kjører i utgangspunktet samme kode. Men så ser vi run metoden som først sier at tråd når 1 sov først i noen millisekunder, også starter den. I if-testen, hvis id er lik 2, så settes det prioritet til 1. så på den måten kan man behandle tråder og gjøre ulike ting selv om trådene i utgangspunktet er like.

```

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 + ". Prioritet er nå "+getPriority());
        }
        System.out.println("Thread nr." + id + " regnet ut " + work()+"\n");
    }
}

```

Vi kompilerer og kjører. Vi starter tråd 1 med prioritet 5 og 2 med prioritet 10. men de kjører annenhver gang og det går like fort. Når vi ser på top kan vi se at begge får like mye CPU.

Hvis vi starter koden igjen med takset slik at de kjører på samme CPU så ser vi at prioritet er ulik så gjør ikke dette noe utslag på CPU. Vi kan konkludere med at Linux ikke bryr seg om prioritetene, om du setter 1 eller 10, så spiller det ingen rolle.

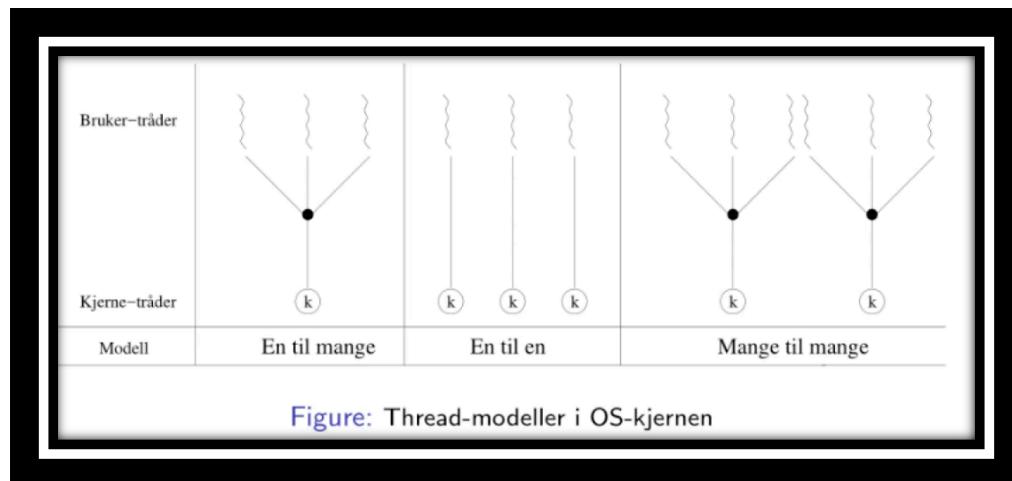
I oppgavene blir vi bedt om å prøve å få Linux til å ta hensyn til Java-prioritetene, det er mulig hvis man kjører java på en spesiell måte, med et spesielt flagg/opsjon. Da må man kjøre som root, og i tillegg med det flagget og da implementerer JVM at de har prioritet i forhold til hverandre. Vi skal nå se at hvis vi kjører samme kode eller klasse fil på Windows vil den ta kraftig hensyn til prioritet.

BLOKKERENDE SYSTEMKALL OG TRÅD-MODELLER

Blokkerende systemkall er den viktigste grunnen til at vi i det hele tatt har tråder. Blokkerende systemkall er systemkall som blokkerer prosessen fordi den venter på I/O forespørsler. Det kan være at man ønsker å lese noe fra en disk, og da er det naturlig at programmet ikke går videre før den har lest det som er på disken. Det kan være problematisk fordi vi ønsker at en applikasjon skal være responsiv selv om den leser noe fra disk, eller gjør noe annet som krever I/O. Det er en fordel med tråder, at man ikke trenger å styre med blokkerende systemkall og programmet/programmeren kan drive med de andre oppgavene samtidig. Noen eksempler på blokkerende systemkall er ‘read’, ‘write’, ‘wait’ og ‘sleep’. Eksempler på ikke-blokkerende systemkall er ‘getPID’ og ‘gettimeofday’ fordi de eksekverer veldig raskt og blokkerer ikke prosessen.

THREAD-MODELLER

Det finnes mange implementasjoner av tråder, og dette er de tre mest vanligste og brukte modellene. Den aller vanligste er ‘en til en’, hvor for hver tråd brukeren starter som for eksempel at vi starter et java program, JVM starter 20 tråder. Da vil det tilsvarende være 20 tråder, som vil si at du har 20 tråder som scheduleres som 20 uavhengige enheter.



En til mange: vil si at alle tråder scheduleres som en prosess, en enhet. Java: green-threads, JVM sørger selv for scheduling, ingen multitasking. Default på gamle versjoner av Linux(Debian) og Solaris. Dette er litt upraktisk fordi programmereren selv må schedulerere, vi har yield() som sier at den selv nå vil gi fra seg CPU-en.

En til en: den mest vanlige. Hver tråd scheduleres uavhengig av de andre. Windows Java-threads, Linux native Java-threads, Linux Posix-threads (pthreads).

Mange til mange: tråder scheduleres uavhengig om de ikke er for mange. Kjernen kan begrense antall tråder i RR-køen, Solaris, Digital Unix, IRIX pthreads.

SYNKRONISERING OG SERIALISERING

Samtidige prosesser som deler felles ressurser/data, må synkroniseres. Da følger vi noen prinsipper og de er:

1. prosesser må ikke endre felles data samtidig
2. en prosess bør ikke lese felles data mens en annen endrer dem
3. en prosess må kunne vente på (f eks resultater fra) en annen prosess

Prosesser/tråder som aksesserer felles data må serialiseres. De må jobbe en av gangen på felles data. Problemstillingen kalles Race Condition (konkurranse om felles ressurser).

Programmereren må selv serialisere sine prosesser. OS legger mulighetene til rette.

HVA ER MENINGEN MED TRÅDER NÅR PRIORITETEN IKKE TAS

HENSYN TIL?

Tråder er en grunnleggende enhet for utførelse av programvare. I denne konteksten kan «prioriteter» referere til viktigheten av trådene i et gitt program, og hvordan disse prioritetene

påvirker utførelsen av programmet. Hvis prioritetene ikke tas hensyn til trådene i et program, kan det føre til ineffektiv bruk av systemressurser og redusert ytelse. For eksempel kan en tråd med lav prioritet hindre en tråd med høy prioritet i å få tilgang til nødvendige ressurser eller hindre at viktige oppgaver blir fullført i tide. Derfor er det viktig å ta hensyn til prioritetene når man utvikler og administrerer tråder i et program for å sikre at de fungerer effektivt og på en måte som oppfyller programkravene.

Prioritet av Java-tråder på Windows

Implementasjonen av prioritet på Java er plattformavhengig. Moderne OS er flinke på å gi respons på de som trenger det interaktivt og gi litt saktere respons til de prosessene som bruker mye CPU. Det er dette som gjør OS-ene dynamiske og det er da som regel ikke så stort behov for programmereren å selv gi prioritet dynamisk.

PROBLEM MED TO WEB-PROSESSER SOM SKRIVER UT

BILLETTER. MÅ SERIALISERES.

Hvis man jobber med felles data må man serialisere. Hvis ikke kan man plutselig sloss om å bruke en felles variabel, hvor resultatet kan bli helt feil.

Et enkelt eksempel på dette er at vi tenker oss en webside som skriver ut billetter. Da er det ofte en database som holder på antall billetter. Da må vi ha en trådsikker database, som vil si at den er serialisert, hvor dette er tatt hensyn til. Men vi kan også bare tenke oss at vi har en web-server hvor det er to prosesser som står og kjører. Og begge må ha tilgang til antall billetter, ellers kan de ikke dele ut billetter i det hele tatt. I dette tilfellet tenker vi oss at vi har en felles variabel `LedigeBilletter` som er antall ledige billetter.

```
if(LedigeBilletter > 0){  
    LedigeBilletter--;  
    SkrivUtBillet();  
}
```

Så kan vi tenke oss at koden for webserveren er noe som vist over. Hvis `LedigeBilletter` er større enn 0, da trekker vi fra en ledig billett og skriver den ut. Dette ser enkelt og greit ut men vi kan få et mulig problem.

Hva om vi er litt uheldige, og prosess 1 står og kjører og sjekker om det er ledige billetter, fellesvariabelen er 1. en If-test utføres ikke i en instruksjon, det er først en if-test, altså en sammenlikning av ledige billetter med 0 for å se om den er større enn 0. i det første tilfellet vil det slå til, men før prosess 1 har rukket å minske variablen med 1, så skjer det en context switch. Da hopper vi over til prosess 2 som sjekker om ledige billetter er større enn 0, ja det er den, så trekker den fra, og så skriver ut en billett.

P1-kode	P2-kode	LedigeBilletter
if(LedigeBilletter > 0){		1
Context Switch==>	if(LedigeBilletter > 0){	1
	LedigeBilletter - -;	0
	SkrivUtBillett();	0
	}	0
LedigeBilletter - -;	====Context Switch	-1
SkrivUtBillett();		-1
}		-1

En Context Switch kan forekomme når som helst. Må serialiseres!

Også etter at vi tar context switchen tilbake, så trekkes det fra ledige billetter, og da er vi på -1, så skriver den ut enda en billett. Og her går det veldig galt, derfor må man serialisere, for å sørge for at slikt ikke skjer. Det er enda mer problematisk om prosesser kjører på ulike CPU. Da kan ikke de sjekke i mellom om den andre har trukket fra en billett eller ikke, så dermed er det et vanskeligere tilfelle å serialisere i.

RACE CONDITION MED EN KODELINJE!

En linje høynivåkode kan oversettes ofte til mange linjer maskinkode av kompilatoren. En context switch kan oppstå mellom to hvilke som helst maskininstruksjer. Vi har ofte sett at os aner egentlig ikke noe om hva de ulike instruksjonene betyr eller gjør. Så os sier bare 'kjør instruksjon, også neste, også neste'... og når som helst kan det komme en context switch. Hvis prosessen opererer på to ulike CPU så har os enda mindre kontroll.

Her skal vi se på to prosesser som oppdaterer en felles variabel. Og der er det saldo som skal oppdateres. P1 gjør masse kode og eventuelt vil den oppdatere saldo til saldo - mill. P2 vil også gjøre en del kode og tilslutt oppdatere saldo til saldo + mill.

P1-kode	P2-kode
static int saldo;	static int saldo;
.	.
.	.
saldo = saldo - mill;	saldo = saldo + mill;

Variabelen saldo er da en felles variabel begge kan endre.

Problem: hva skjer om OS switcher fra P1 til P2 mens P1 utfører saldo = saldo – mill?

Da må vi tenke på maskinarkitektur, assembly og at prosesser faktisk bare utfører maskinkode. I dette tilfellet et det maskininstruksjoner med x86. vi kan senere se at det er noe liknende når vi utfører Java med JVM og bytekode. Vi vet at x86 ikke tillater to referanser til minne samtidig og derfor kan ikke en enkelt kodelinje med addisjon utføres som en linje maskinkode.

P1	P2
saldo = saldo - mill;	saldo = saldo + mill;
mov saldo, %ax	mov saldo, %ax
mov mill, %bx	mov mill, %bx
sub %bx, %ax	add %bx, %ax
mov %ax, saldo	mov %ax, saldo

Og da har vi et problem. Det kan hende at 1 million forsvinner. Da tenker vi at saldo er 5 først også begynner vi med P1 som trekker ifra. Den tar også flytter saldo inn i en av registrene sine %ax, også skjer det en context switch. Og da er det viktig å huske at alt lagres. Og da kommer P2 inn og den vil gjøre det samme og flytte saldo og mill inn i %ax og %bx. Da går det galt fordi P2 vil ta også legge til 1 mill inn i registrene hvor det allerede er 5 mill også flytter den verdien ut til at saldo da er 6 mill. Senere skjer den en context switch tilbake til P1. også tar P1 subtraksjon %bx, %ax, og den bruker jo de gamle verdiene. Den trekker fra saldo 5 så da blir det 4, også flytter den det inn i saldo variabelen.

Prosess som kjører	Instruksjon (IR)	%ax	%bx	saldo
P1	mov saldo, %ax	5	0	5
P1	mov mill, %bx	5	1	5
OS	Context switch	0	0	5
P2	mov saldo, %ax	5	0	5
P2	mov mill, %bx	5	1	5
P2	add %bx, %ax	6	1	5
P2	mov %ax, saldo	6	1	6
OS	Context switch	5	1	6
P1	sub %bx, %ax	4	1	6
P1	mov %ax, saldo	4	1	4

Det burde ha blitt saldo = 5 og en mill er borte!! Konklusjon: må serialisere aksess til felles data!

Konklusjon: det burde ha blitt saldo = 5 og dermed ser vi at 1 million er borte. Til konklusjon kan vi si at alt må serialiseres, det som skal ha tilgang til felles data. Det vil i dette tilfellet si at P1 vil gjøre sitt ferdig med saldo også kan P2 komme inn, eller motsatt, men de må ikke få lov til å gjøre det samtidig.

Utregningen av saldo er et kritisk avsnitt i koden til P1 og P2. kritisk avsnitt må fullføres av prosessene som utfører det uten at andre prosesser slipper til. Det medfører at prosessene må serialiseres.

DEMO:

Det gjør litt av det samme som saldo programmet vi så på gjorde. I vårt tilfelle er det to saldo-threads. Det er s1 og s2. Vi har en public static int saldo, som vil si den er felles. Også har vi to tråder som oppdaterer denne saldoen. Det som vi ser skjer er at hvis id er lik 1, så øker saldo med 1 antall MAX ganger og hvis saldo ikke er 1 så minkes saldo en antall MAX ganger.

```
[~] haugerud@lap:~/fag/os/threads$ jed Saldo.java
```

Når vi kjører den ser vi at to tråder starter med samme prioritet. Men det skjer noe veldig galt fordi den endelige saldoen er helt feil.

```
haugerud@lap:~/fag/os/threads$ java Saldo
Starts two threads !
Thread nr. 2, priority 5 starts
Thread nr. 1, priority 5 starts
Thread nr. 2 finished. Saldo: 38048
Thread nr. 1 finished. Saldo: 850391
Final total saldo: 38048
```

Hvis vi prøver å kjøre på nytt ser vi at det blir annerledes igjen:

```
haugerud@lap:~/fag/os/threads$ java Saldo
Starts two threads !
Thread nr. 1, priority 5 starts
Thread nr. 2, priority 5 starts
Thread nr. 1 finished. Saldo: 92063
Thread nr. 2 finished. Saldo: -670084
Final total saldo: 92063
```

Vi ser at hver gang det kjøres er det en ulik verdi, men det er sånt som skjer når det ikke er serialisering mellom tråder. Her ser vi hvor feil det kan gå når de jobber ujevnt.

JAVAP -PRIVATE

Javap er en egen applikasjon som kan vise java byte kode. Assemblykode er for x86 fysiske maskiner mens denne java bytekoden er for en virtuell maskin.

```
haugerud@lap:~/fag/os/threads$ javap -private -c SaldoThread
```

Også er det en metode 'private void updateSaldo()' som oppdaterer saldo. Og under ser vi linje 21 en iadd og linje 44 en isub.

```
private void updateSaldo();
  Code:
    0: aload_0
    1: getfield      #3           // Field id:I
    4: iconst_1
    5: if_icmpne    31
    8: iconst_1
    9: istore_1
   10: iload_1
   11: getstatic     #12          // Field MAX:I
   14: if_icmpge    54
   17: getstatic     #13          // Field saldo:I
   20: iconst_1
   21: iadd
   22: putstatic     #13          // Field saldo:I
   25: iinc         1, 1
   28: goto        10
   31: iconst_1
   32: istore_1
   33: iload_1
   34: getstatic     #12          // Field MAX:I
   37: if_icmpge    54
   40: getstatic     #13          // Field saldo:I
   43: iconst_1
   44: isub
   45: putstatic     #13          // Field saldo:I
   48: iinc         1, 1
   51: goto        33
   54: getstatic     #7           // Field java/lang/System.out:Ljava/io/PrintStream;
   57: aload_0
   58: getfield      #3           // Field id:I
   61: getstatic     #13          // Field saldo:I
   64: invokedynamic #14,  0     // InvokeDynamic #1:makeConcatWithConstants:(II)Ljava/
```

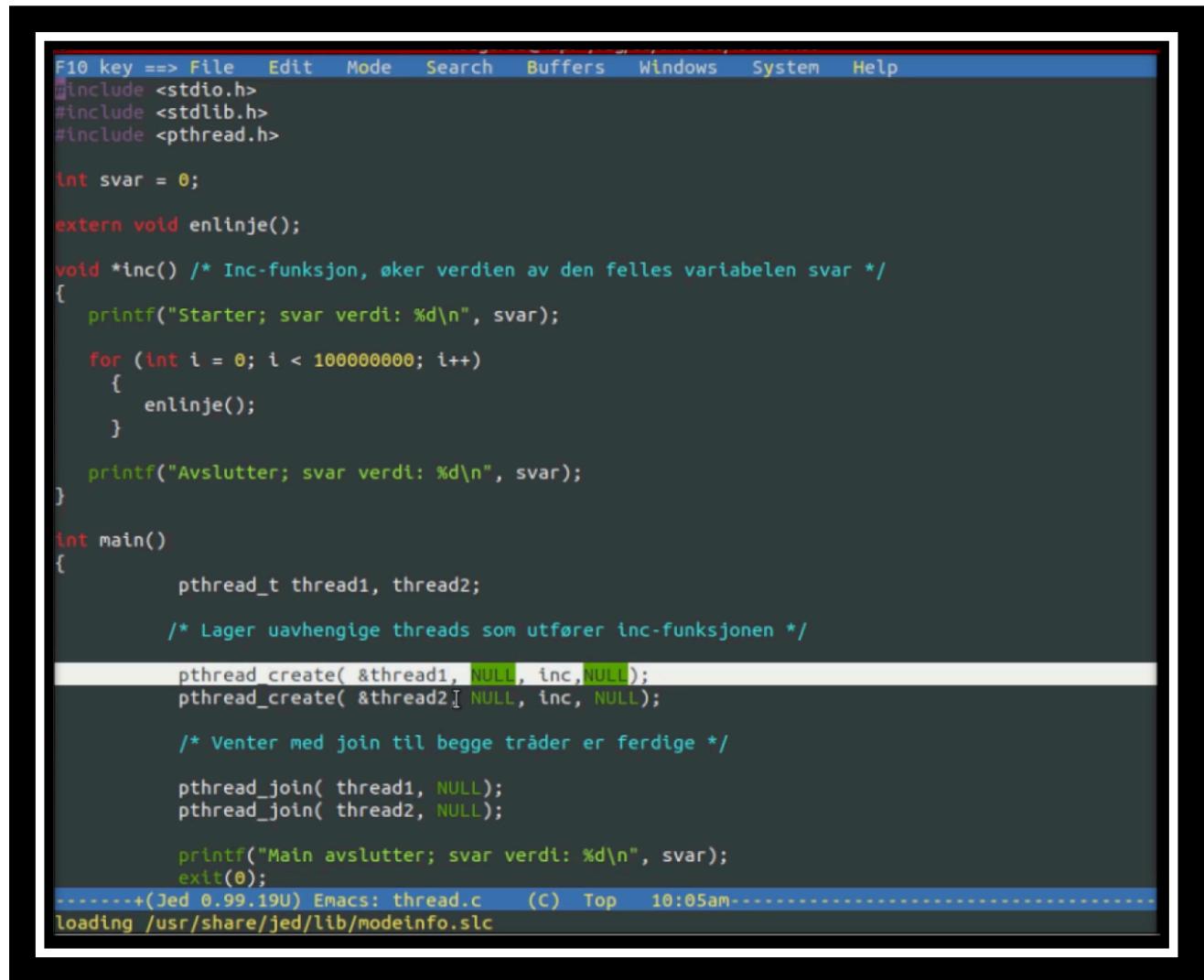
Fra ca. linje 11 til 22 er der tråd 1 tar saldo ++. JVM er en stack maskin, og den legger variabler på stacken også opererer den på stacken. Det som faktisk foregår her er at linje 17: getstatic, betyr hent inn variabelen saldo. Så hvis tallet er 50, så legges saldo 50 på stacken. Også iconst 1, legger tallet 1 på stacken. Og iadd legger sammen de to tallene. Så da får den 51 (50 + 1), også gjør den putstatic, og legger verdien ut. Og helt tilsvarende skjer med isub nede fra linjene ca 40-45. Og da er det klart, og vi ser at det tar litt tid med å hente verdien inn og legges på stacken og dette implementeres jo av JVM. Som så kjører maskinkode som gjør det her. Det som er tydelig er at saldo blir lagt inn, også legges det i registeret som bare denne prosessen eier, og den aner ikke noe om hva som skjer nede fra linje 34 og ned. Så når dette kjøres er det helt kaos. Saldo hentes inn og det er ikke noe kontroll over hvor mange ganger det gjøres eller ingen kontroll på at det kan komme inn noe avbrudd fra linje 21. Det er enda mindre kontroll når prosesser jobber på to ulike CPU.

DEMO RACE CONDITION MED C

Vi skal se på enda et eksempel, men her er det maskinkode.

```
haugerud@lap:~/fag/os/threads/lock$ jed thread.c
```

Dette er en implementasjon av pthreads i C. Det er et bibliotek som gjør det mulig å kjøre tråder i et C-program. I den markerte linja under startes det en tråd, tråd 1, og tråd 2 rett under. De skapes også sendes det en metode, inc, og siden det sendes med så kjører trådene den metoden. Også ser vi join nede, som betyr trådene venter på hverandre, sånn at begge er ferdige før main avslutter og skriver ut svaret. Hundre millioner ganger kallen koden på en metode ‘enlinje()’.



The screenshot shows a terminal window with a dark background and light-colored text. At the top, there's a menu bar with options: F10 key ==> File Edit Mode Search Buffers Windows System Help. Below the menu, the code is displayed in a monospaced font. The code defines a global variable 'svar' and two functions: 'inc()' and 'main()'. 'inc()' increments 'svar' 100,000,000 times and prints its value. 'main()' creates two threads, 'thread1' and 'thread2', which both call 'inc()'. After the threads have finished, 'main()' joins them and prints the final value of 'svar' before exiting. The terminal window also shows some status information at the bottom, including the Emacs version and file name.

```
F10 key ==> File Edit Mode Search Buffers Windows System Help
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int svar = 0;

extern void enlinje();

void *inc() /* Inc-funksjon, øker verdien av den felles variabelen svar */
{
    printf("Starter; svar verdi: %d\n", svar);

    for (int i = 0; i < 100000000; 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( thread2, NULL);

    printf("Main avslutter; svar verdi: %d\n", svar);
    exit(0);
-----+(Jed 0.99.19U) Emacs: thread.c (C) Top 10:05am-----
loading /usr/share/jed/lib/modeinfo.slc
```

Men hva er det enlinje() gjør. Koden tar et extern int ‘svar’, også øker den det med en. Øverst i bildet over ser vi at ‘svar’ variablen er deklarert globalt. Når vi kompilerer koden over vil variablen være global som da enlinje() får tak i.

```
haugerud@lap:~/fag/os/threads/lock$ cat en.c
void enlinje()
{
    extern int svar;
    svar++;
}
```

Det som da vil skje er at, nå er det ikke en som trekker fra og en som legger til, men begge disse legger til, så hundre millioner ganger så kaller vi svar ++ for den ene tråden og hundre millioner ganger for den andre, så svaret bør bli 2 hundre millioner.

Hvis vi kompilerer som under får vi feilmeldinger om pthread og det er fordi vi må ha med en opsjon.

```
haugerud@lap:~/fag/os/threads/lock$ gcc thread.c en.c
/tmp/ccINTHwI.o: In function `main':
thread.c:(.text+0x8d): undefined reference to `pthread_create'
thread.c:(.text+0xaa): undefined reference to `pthread_create'
thread.c:(.text+0xbb): undefined reference to `pthread_join'
thread.c:(.text+0xcc): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
```

Under vises den riktige måten å kompilere på:

```
haugerud@lap:~/fag/os/threads/lock$ gcc -pthread thread.c en.c
```

Vi kjører flere ganger og ser at akkurat som i java trådene, så blir svaret ulikt hver gang.

```
haugerud@lap:~/fag/os/threads/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 0
Avslutter; svar verdi: 113238853
Avslutter; svar verdi: 115967681
Main avslutter; svar verdi: 115967681
```

Kan det være at når kompilatoren kompilerer en.c, at kompilatoren lager flere linjer når den kompileres. For å sjekke ut det, og være sikker på at den ikke gjør det kan vi lage en veldig liten assembly fil.

```
haugerud@lap:~/fag/os/threads/lock$ cat en.c
void enlinje()
{
    extern int svar;
    svar++;
}
```

Det lille assembly programmet implementerer enlinje. Og vi ser under at den implementerer med en veldig enkel, en linje instruksjon. %rip er et veldig spesielt register som brukes til å overføre variabler, så det som skjer her er at den variabelen øker med 1.

```
haugerud@lap:~/fag/os/threads/lock$ cat minimal.s
.globl enlinje
enlinje:
    incl svar(%rip)
    ret
```

Vi kompilerer på nytt så istedet for å ta med en.c tar vi med minimal.s.

```
haugerud@lap:~/fag/os/threads/lock$ gcc -pthread thread.c minimal.s
```

Også kjører vi igjen og ser at resultatet fremdeles ikke er 2 hundre millioner. Men det er forskjellig svar hver gang man kjører. Og det virker jo veldig rart fordi i minimal.s er det jo bare en instruksjon. Så det ikke være at det er noe context switch i mellom fordi her er det bare en instruksjon, så det blir ikke noe mellomlagring slik som vi hadde i java.

```
haugerud@lap:~/fag/os/threads/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 15873
Avslutter; svar verdi: 90881828
Avslutter; svar verdi: 121506423
Main avslutter; svar verdi: 121506423
```

Hva om vi prøver med taskset her. Fordi disse trådene vil jo kunne scheduleres på hver sin CPU. Og da har vi plutselig ikke noe kontroll på det som skjer. Da ser vi at når vi tvinger den begge trådene om å kjøre på samme CPU så vil ikke svarer kunne være korrupt av noe context switch fordi det bare er en instruksjon som skal skje. Med engang vi kjører de med to ulike CPU-er så får man ulike svar.

```
haugerud@lap:~/fag/os/threads/lock$ taskset -c 0 ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 7036640
Avslutter; svar verdi: 199287181
Avslutter; svar verdi: 200000000
Main avslutter; svar verdi: 200000000
```

Under er det en assemblykode:

```
haugerud@lap:~/fag/os/threads/lock$ jed lockMinimal.s
```

Der er fortsatt den ene instruksjonen. Men før det er det en instruksjon som heter lock. Det er den første metoden vi skal se på som kan løse dette problemet. Lock må videreformidle til alle CPU-ene at 'nå må ingen bruke databussen, nå må vi låse av databussen'. Etter at lock instruksjonen sendes kan ingen andre bruke databussen.

```
F10 key ==> File Edit Search Buffers Windows System
.
.globl enlinje
enlinje:
    lock
    incl    svar(%rip)
    ret
```

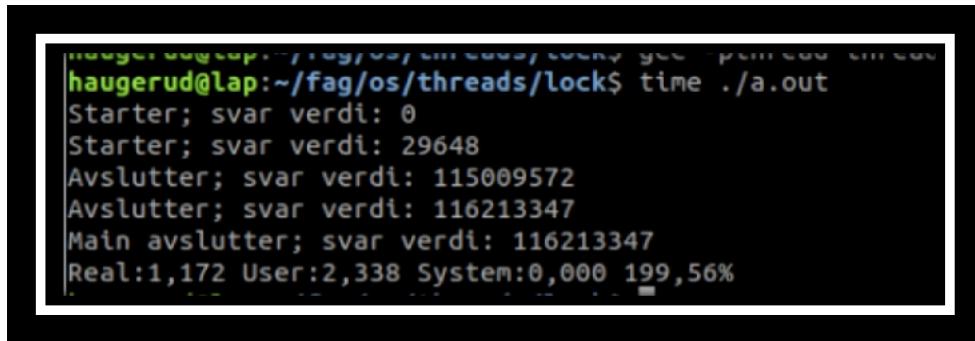
Under kompilerer vi igjen med lockmininal-koden som er vist rett over og da får vi riktig svar, og dette er på tross av at trådene ikke er på samme CPU.

```
haugerud@lap:~/fag/os/threads/lock$ gcc -pthread thread.c lockMinimal.s
haugerud@lap:~/fag/os/threads/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 7075
Avslutter; svar verdi: 198413568
Avslutter; svar verdi: 200000000
Main avslutter; svar verdi: 200000000
```

Dette tar også litt lengre tid fordi databussen må låses hele tiden og trådene må koordineres og vente på hverandre.

```
haugerud@lap:~/fag/os/threads/lock$ time ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 0
Avslutter; svar verdi: 186510288
Avslutter; svar verdi: 200000000
Main avslutter; svar verdi: 200000000
Real:4,385 User:8,517 System:0,068 195,78%
```

Uten lock-koden så tar det omtrent $\frac{1}{4}$ av tiden det tar med lock:



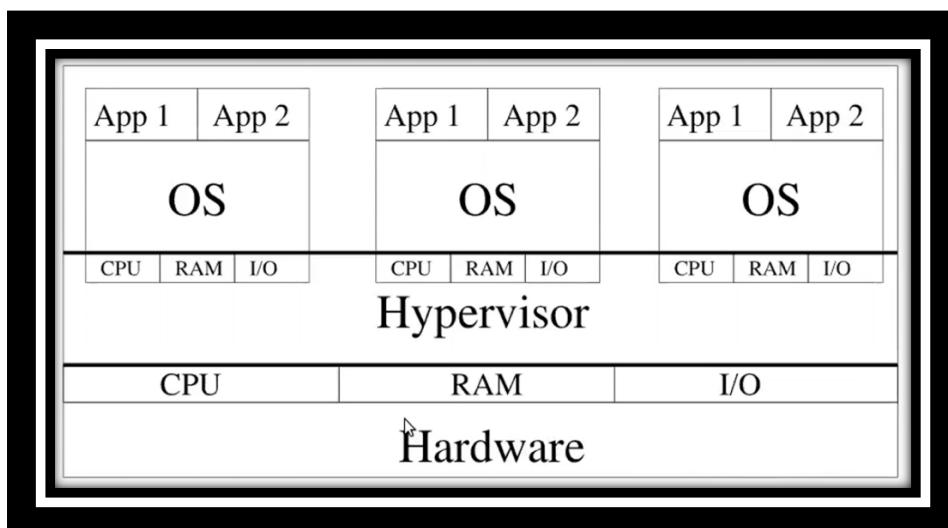
```
haugerud@lap:~/fag/os/threads/lock$ time ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 29648
Avslutter; svar verdi: 115009572
Avslutter; svar verdi: 116213347
Main avslutter; svar verdi: 116213347
Real:1,172 User:2,338 System:0,000 199,56%
```

En CPU er koblet med databussen til RAM, men når disse to instruksjonene kjører på samme CPU, med taskset, så når man da gjør svar++ eller instruksjonen i lock koden, og vi tenker oss at vi har to tråder, den ene tråden utfører den ++ instruksjonen, så vil hele instruksjonen utføres før den gjør en context switch. Instruksjonen jeg snakker om, igjen vist under, henter fra RAM la oss si variabelen svar er 50, så øker den med 1 til 51 og legger den tilbake. Og dette foregår på en atomisk operasjon, det er bare en operasjon i CPU-en. CPU-en blir aldri avbrutt midt i CPU-en. Det kan skje en context switch når denne operasjonen er ferdig, og da kan den andre tråden komme inn. Med ulike CPU sier vi derfor at koordinasjonen er feil.

LINUX

Grunnleggende om hva virtualisering er. Hardware, hypervisor og gjeste-os

Når vi snakker om virtualisering på server eller maskinnivå, så ser vi for det av en hel server eller desktop. Det som virtualiseres er all hardware til den serveren. Bildet under har vi sett masse i os og når vi først så det var det et operativsystem (som styrte alt) der det nå står hypervisor. Med virtualisering er det en hypervisor som kjører oppå her, og det kan være at det inneholder et operativsystem, eller så er det et os som inneholder en del kjernemoduler som hjelper til med virtualisering. Uansett vil det være et lag mellom hardware og operativsystemer og applikasjoner som kjører på topp. Hypervisoren i bildet tilbyr tre grensesnitt, 3 CPU, 3 RAM og 3 I/O til os-er. Her i bildet kjøres tre virtuelle maskiner, og de tre ser det ut som om kjører på en fysisk hardware, det kan ikke sees om de kjøres på et virtuelt grensesnitt eller hardware. Det er ikke helt sant, det vil være mulig å finne ut med hypervisor 2 hvor os kan kjøre helt under endringer rett på hypervisor. Da kan du ta en helt vanlig ubuntu 18.04 på hypervisor og den kjører akkurat som om man hadde bootet den opp på hardware direkte. **Hypervisor simulerer hardware og gir samme grensesnitt.** Dermed vil os-ene som kjører på en virtuell maskin tro at de kjører på ekte hardware. Dette er ganske viktig sikkerhetsmessig også, det gjør at denne type virtualisering er en sikker måte å fordele applikasjoner på VM. De ulike VM-ene er veldig godt isolert fra de ved siden av. Selvom det kommer en hacker i VM i midten har den null sjans for å komme seg til de ved siden av. Det er litt annerledes i docker fordi da kjøres prosessene på samme operativsystem. Det likner på det her men da kan vi ha to ulike os på hver dockerinstans, men da kjører ikke de på noe hypervisor men de kjører inne i samme os, hvor skillelinjene er mindre.



HVORFOR VIRTUALISERING

Da må vi kanskje gå 20 år tilbake hvor det typiske var at man hadde fysiske servere og da hadde man ofte en stor fysisk server med mange CPU-er og mye minne. Man kunne ha en fysisk server for drift, en for database osv. Mange driftsproblemer dukket opp rundt dette, f.eks feil i hardware kunne ødelegge hele serveren. Da eksperimenterte man med VM. Noen fordeler er isolasjon, ressurssparing, fleksibilitet, programvare-utvikling og skytjenester.

Alt dette gjelder også for containere og docker som vi også har sett på, bortsett fra den første, isolasjon og sikkerhet. Men fleksibiliteten blir enda større med containere.

Isolasjon: en fordel med visualisering er at man setter opp tjenester som bare kjører på en VM, og kun der. Da unngår man at ulike tjenester ødelegger hverandre. Sikkerhet: hvis en tjeneste blir hacket, vil det ikke påvirke de andre tjenestene. Dette er fordi os og applikasjonene kun kommuniserer mot det virtuelle hardware-API'et som hypervisor gir dem tilgang til. De har ingen mulighet til å kommunisere med andre deler av en hypervisor eller andre VM-er.

Ressurssparing: man kan oppnå isolasjon ved å ha en fysisk server for hver tjeneste, men det gir store driftskostnader. Med virtualisering kan det samme oppnås på en enkelt server. Virtuelle maskiner som for eksempel bruker lite CPU kan settes på samme fysiske server. VM-er kan enkelt flyttes til og fra fysiske servere og man kan dermed spare hardware og strøm.

Fleksibilitet: kapasiteten kan enkelt økes ved å legge til flere VM-er, lastbalansering blir enklere. **Elastisitet:** man kan dynamisk tildele CPU-er og internminne til VM-er. Har en VM blitt ødelagt eller kompromittert kan man enkelt starte opp en ny kopi. Tradisjonelt er det arbeidskrevende å flytte en tjeneste eller et softwareprosjekt til en ny server på grunn av avhengighet av os og annen programvare: når noe er utviklet på en VM så kan hele VM flyttes eller kopieres. **Live migration:** Hele VM flyttes til en annen fysisk server uten nedetid på tjenestene.

PROGRAMVAREUTVIKLING

Har vi sett på mye med docker hvor man kan enkelt nøyaktig sette opp miljøet man ønsker og raskt teste ut programvare på forskjellige os Windows, Linux, Mac, etc. Ved å kjøre VM-er med en rekke forskjellige os. Ønsker man å teste ut nye ideer, kan man raskt sette opp miljøer for å teste dem ut.

Virtualisering er grunnlaget for fleksible skytjenester. Kunder kan gis egne VM-er med et antall CPU-er, disk og minne. Disse kundene kan dele fysiske servere, noe som gir store besparelser av hardware.

SENSITIVE OG PRIVILEGERTE INSTRUKSJONER

IBM startet med virtualisering av stormaskiner på 1960-tallet. Dette var en tidlig form for visualiseringsteknologi som gjorde det mulig å kjøre flere virtuelle maskiner på en enkelt fysisk maskin. IBM (International Business Machines Corporation) er et amerikansk teknologiselskap. Xen er den som kjører i våre VM.

- Første virtualiseringsløsning for x86: VMware i 1999
- Deretter fulgte Xen, VirtualBox, KVM og mange andre
- Hardware-støtte for x86 virtualisering kom først i 2005

Vi skal se hvordan hardware-støtte for x86 virtualisering kom i 2005. Det var arbeid fra Popek og Goldberg som så på store trekk som skulle til for virtualisering, i den betydningen om at man skal kunne kjøre et operativsystem og en VM. Det de sa er at en maskin kan bare virtualiseres hvis alle sensitive instruksjoner også er privilegerte instruksjoner. En sensitiv instruksjon kan bare utføres i kernel mode. I user mode er det ikke sånn at en hvilken som helst prosess kan skru av maskinen. Det er en sensitiv instruksjon. En privilegert instruksjoner forårsaker en trap til kernel mode hvis den gjøres i user mode.

Eksempel:

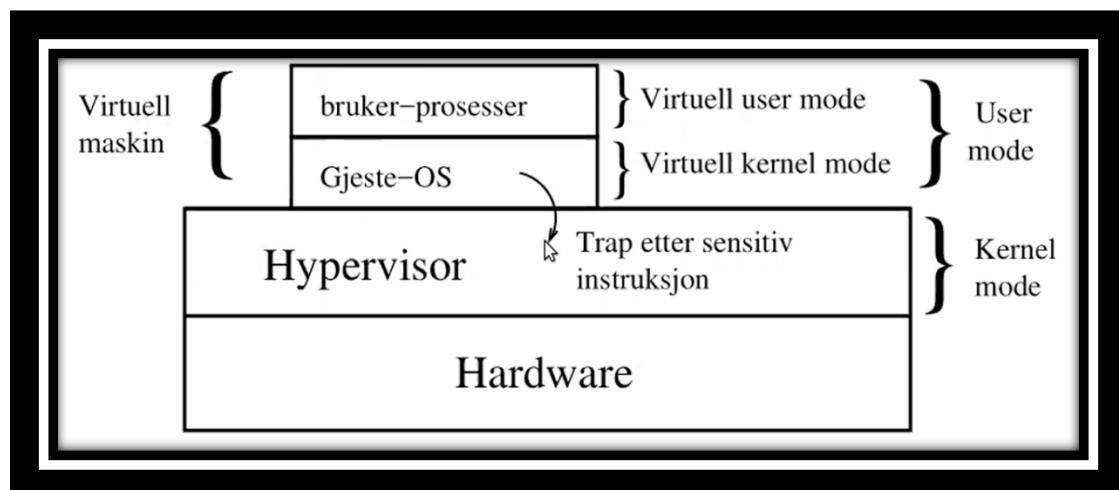
- x86 instruksjonen POPF (skrur av og på interrupts) er en sensitiv instruksjon. Hvis den utføres i user mode vil ingenting skje, som for NOP (no operation).

- Instruksjonen CLI (clear interrupt flag) er en sensitiv instruksjon, men den er også privilegert. Hvis den utføres i user mode, gjøres en trap til kernel mode. Dette er også en sensitiv instruksjon = privilegert.
- Vanlige instruksjoner som ADD, CMP og MOV er hverken sensitive eller privilegerte. Vi skal se på at et Virtuelt os kjører i user mode.

HARDWARE STØTTET VISUALISERING

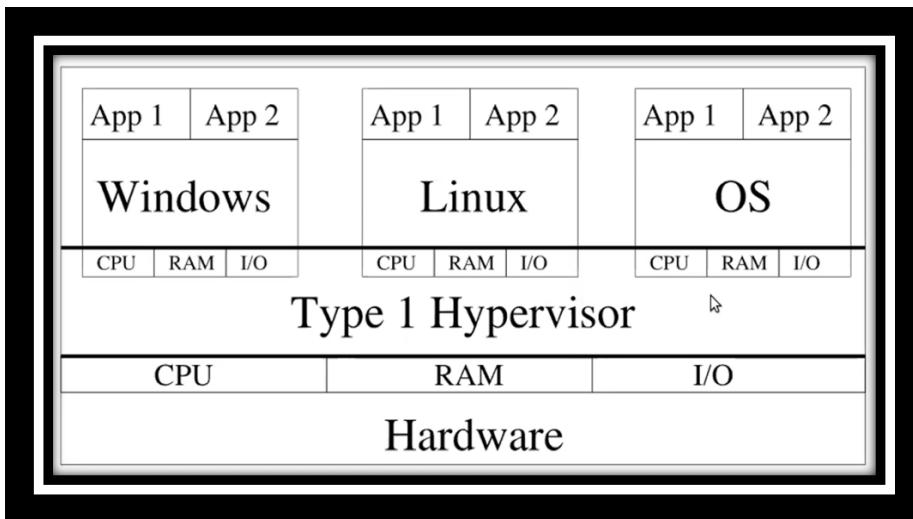
Grunnen til at det er et problem med privilegerte instruksjoner er fordi det finnen sensitive instruksjoner som ikke er privilegerte. Når vi har en VM så kjører den oppå en hypervisor og hypervisor er på en måte det vi har tenkt på som os-kjernen tidligere. Den kjører i kernel mode. Også kjører det en 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 kjernen 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.

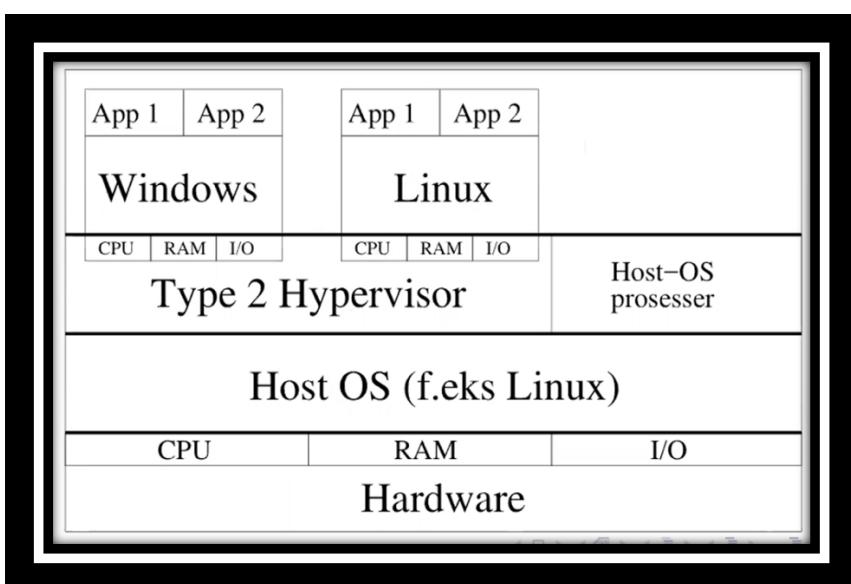


TYPE 1 OG 2 HYPERVISOR

Type 1 hypervisor er en helt selvstendig hypervisor som kjører oppå hardware direkte. Slik som med VM-er hvis du skal virtualisere med VM-er, så må du installere VM-er på en server, også etterpå installerer du virtuelle maskiner oppå det. Det samme gjelder Xen og Hyper-V(Microsoft sin virtualisering) da setter vi hypervisor i bunnen som på bildet, også oppå den kan vi kjøre forskjellige os, slik som Windows, Linux og andre os. Hypervisor tilbyr bare grensesnittet som hardware gir. Alle sensitive instruksjoner som utføres i user mode av gjeste-OS må trappe til kernel mode og fanges opp av hypervisor.



Type 2 vil kjøre oppå et eksisterende OS. Deler av hypervisor kan inngå i det underliggende OS i form av kjernemoduler. Hvis type 2 hypervisor ikke får hjelp fra kjernemoduler vil det gå veldig sakte. Det kan være litt flytende grenser mellom type 1 og type 2 hypervisor. I host os (f eks Linux) har vi kjernemoduler KVM som hjelper Linux gjøre dette mer effektivt.



BINÆR OVERSETTELSE

Før 2005 måtte alternative metoder brukes uten hardware støtte. Hvis man ikke gjorde det fikk man problemer med de instruksjoner som ikke trappet. Og da lagde VM-er en hypervisor som mens programmet kjørte scannet scannet den koden og så etter sensitive instruksjoner. Og da sensitive instruksjoner som ikke trapper til kjernen. Og da gjorde den det for hver kodeblokk som ender i jump, call, trap eller liknende hvor det ble ønsket om å snakke med kjernen. Og da fikk man til å kjøre de kodebitene veldig effektivt.

Et problem med hardware støttet virtualisering: genererer mange traps. Hver gang gjeste-os gjør et kall som trapper tar det litt ekstra tid. Slik kan det i enkelte tilfeller kan virtualisering gå raskere enn å kjøre på hardware.

PARAVIRTUALISERING

Er en type virtualisering som gjør at gjeste-os må endres, og derfor er ikke det helt optimalt fordi da må man gå inn og gjøre endringer i gjeste-os. Alle sensitive instruksjoner erstattes med kall til hypervisor. Gjeste-os kan optimaliseres for virtualisering. Ved å installere drivere laget for paravirtualisering, kan denne metoden bli meget effektiv.

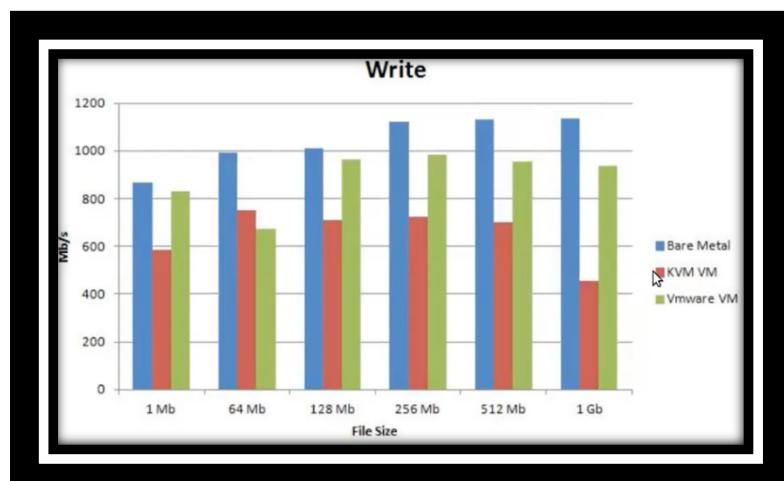
YTELSE OG VIRTUALISERING

Følgende fakta osv. har Hårek hentet herifra:

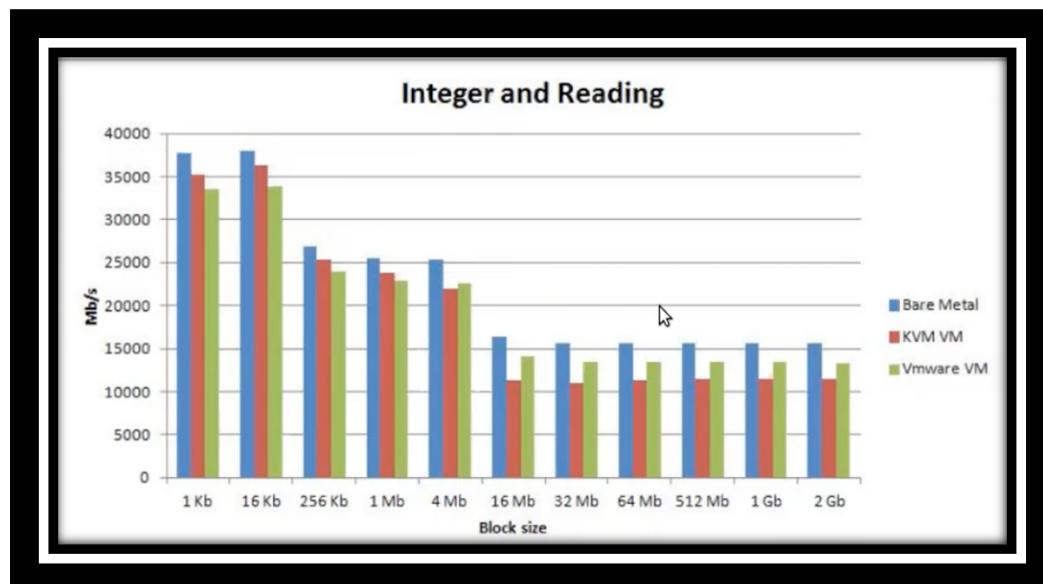
Bare metal vil si å kjøre rett på jernet altså rett på hardware.

Ved disk ytelse og I/O er det der hvor virtualisering gjør det dårligst.

- Yaqub, Naveed: Comparison of Virtualization Performance: VMWare and KVM, NSA masteroppgave (2012)



Det er mer jevnt med RAM ytelse:



Ikke så mye forskjell på CPU ytelse heller. Virtualisering lar kode kjøre direkte uten avbrytelser. Eneste avbrytelsen som skjer er når gjeste-os trapper til hypervisor, det skjer veldig sjeldent når man bare regner med CPU.

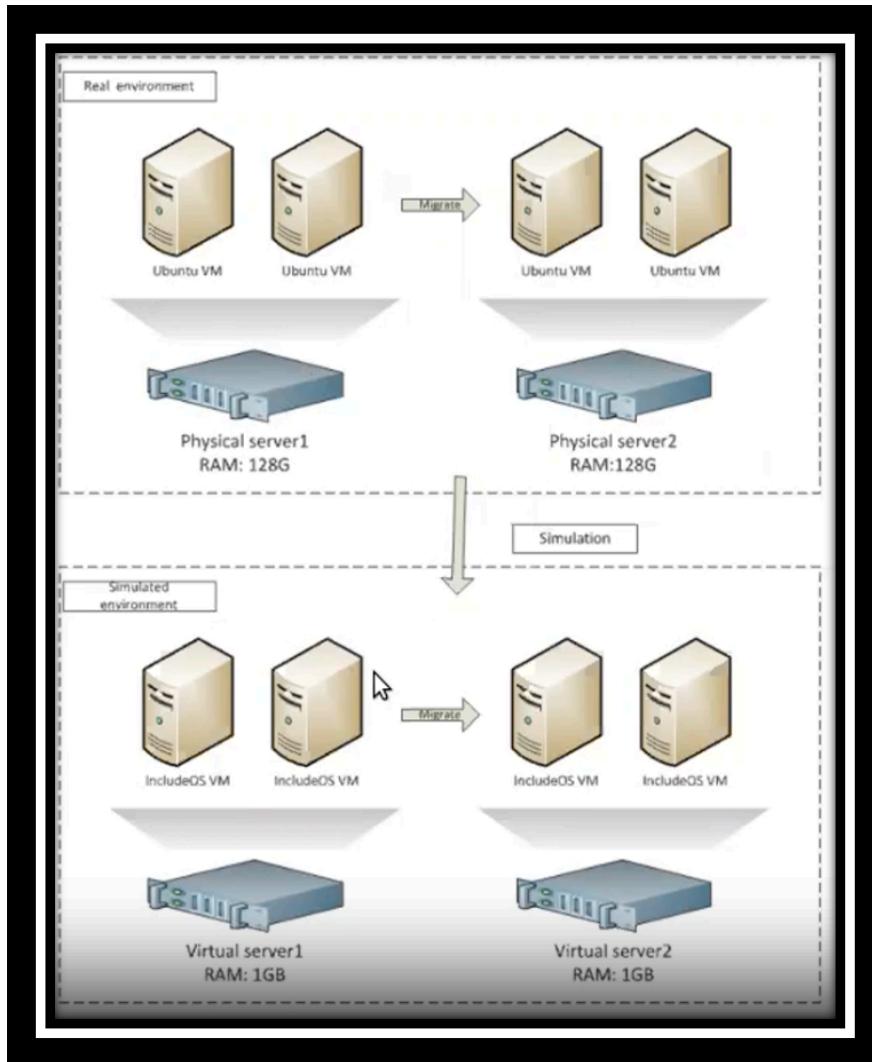
LIVE MIGRATION OG NESTED VIRTUALISERING

Ved live migration flyttes en VM fra en fysisk server til en annen samtidig som den kjører kontinuerlig. Først kopierer man over de statiske delene av minnet over mens VM'en kjører og tas imot på den andre servere. Når nesten alt er kopiert, stoppes VM-en, siste rest kopieres over og når den startes opp igjen på den andre serveren. Disken med VM-imaget og VM sitt filsystem er vanligvis et nettverksfilesystem.

Masteroppgave fra noen år tilbake:

- Wu, Hai: *Simulating Live Migration using Micro Virtual Machines* NSA masteroppgave (2015)

Han satte opp en simulering av en live migration. Vi kan tenke oss at i et virkelig miljø har du fysiske servere også kjører du ubuntu VM oppå her. Også kopierer man hele VM-en over når man kopierer over fra den ene fysiske serveren til den andre. Det var et av eksperimentene han gjorde, også gjorde han også en simulering



Han virtualiserte 2 servere. Han kjørte includeOS Vm-er på disse VM-ene. Dette er det som kalles nested virtualisering, da har man tre lag. Det er ganske effektivt, spesielt om du har små VM.

NESTED VIRTUALISERING

Ved nesten virtualisering kjører man en hypervisor inne i en annen VM, slik at det blir tre lag. Dette gir ytterlig ytelsesreduksjon. Intel har hatt hardwarestøtte for nested virtualisering siden 2013. kan brukes til å flytte VM-er mellom ulike Cloud-providere.

QEMU OG KVM, KJØRE QEMU I LINUX KOMMANDOLINJE

QEMU (Quick Emulator) er en open source programvare som kan emulere en komplett datamaskin. QEMU kan emulere mange instruksjonssett, inkludert x86, MIPS, 32-bit ARMv7, ARMv8, PowerPC. QEMU kan boote en rekke OS: Linux, Windows, Solaris, DOS og BSD.

KVM (Kernel virtual machine) er en Linux kjernemodul som gjør at user mode programmer kan utnytte hardware virtualisering for en rekke forskjellige prosessorer, inkludert Intel og AMDs x86-prosessorer. For å utnytte KVM må man kjøre QEMU-prosessen med opsjonen -enable-kvm.

Rekker ikke å se hvordan man kan kjøre sånne prosesser rett på linux server.

I praksis er det sånn man gjør det. Det første markert er programmet som kjører, prosessen. Dermed kan man bare skrive dette inn og da starter en hel server med alt innhold, og dette kunne vært, linux, ubuntu eller hva som helst. Et at argumentene er at det er et ubuntu 14 image, og det er et svært image som kanskje er 2 GB stort. Det er disk image og inneholder alt som kreves av ubuntu. Vi kan også se at det defineres et antall cores. Med enkelte tastetrykk kan man endre core til 10 og da får man 10 CPU.

-enable-kvm betyr at man skal bruke kjernemoduler som gjør virtualiseringen mye mer effektiv.

- `# qemu-system-x86_64 -enable-kvm -name server2 -m 1024 -realtime mlock=off -smp 1,sockets=1,cores=1,threads=1 -hda /root/cslab/vmimages/ubuntu14.04.amd64.4G2.img -netdev tap,script=./qemu-ifup,id=hostnet0 -device rtl8139,netdev=hostnet0,id=net0 -nographic`
- -enable-kvm betyr at man skal bruke kjernemoduler som gjør virtualiseringen mye mer effektiv.
- En hel virtuell maskin med alt innhold kjøres som en enkelt prosess i user-mode.
- Image'et ubuntu14.04.amd64.4G2.img er på 2.2 Gbyte.

```
root@intel:~/vm# lsmod | grep kvm
kvm_intel           172032  16
kvm                 540672  1 kvm_intel
```

Under er bilde av hvordan man kan liste kjernemoduler. Kvm-intel er her selve kjernemodulen

```
root@intel:~/vm# lsmod | grep kvm
kvm_intel           172032  6
kvm                 540672  1 kvm_intel
```

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 omrent 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);
}
```

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);
}
```

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);
}
```

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);
}
```

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 kjerner 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 1 GB 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.

- `# qemu-system-x86_64 -enable-kvm -name server2 -m 1024 -realtime mlock=off -smp 1,sockets=1,cores=1,threads=1 -hda /root/cslab/vmimages/ubuntu14.04.amd64.4G2.img -netdev tap,script=./qemu-ifup,id=hostnet0 -device rtl8139,netdev=hostnet0,id=net0 -nographic`