# Dagens tema: Tråder 1

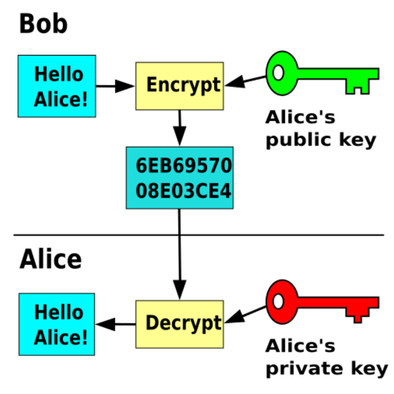
- Hva er tråder og hva kan vi bruke dem til?
- Tråder i Java

# Noen ting tar lang tid

## Kryptering og primtall

Det meste av kryptering i dag er basert på at det kreves svært mye datakraft for å avgjøre om et tall er et primtall eller hvilke primfaktorer det består av.

Er det så vanskelig da?



## En primtallssjekker

Det er enkelt å programmere en metode som sjekker om et tall er et primtall:

```
private static boolean erPrimtall (long x) {
    for (long i = 2;  i < x;  ++i)
        if (x%i == 0) return false;
    return true;
}
```

**NB!**

Dette er langt fra den smarteste måten å skrive en primtallssjekker på. Hvis du vil vite hvordan du bør gjøre det, kan du ta IN3030 – Effektiv parallellprogrammering.

# Hvor fort går dette?

La oss sjekke tallene fra 1.000.000.000 til 1.000.000.100:

```
class FinnPrimtall {
    public static void main (String[] arg) {
        long a1 = 1_000_000_000, a2 = a1+100;
        for (long i = a1;  i <= a2;  ++i)
            if (erPrimtall(i)) System.out.print(i + " ");
        System.out.println();
    }
}
```

Slik går kjøringen:

```
$ javac FinnPrimtall.java
$ time java FinnPrimtall
1000000007 1000000009 1000000021 1000000033 1000000087 1000000093 1000000097

real    0m21.973s
user    0m21.719s
sys     0m0.119s
```

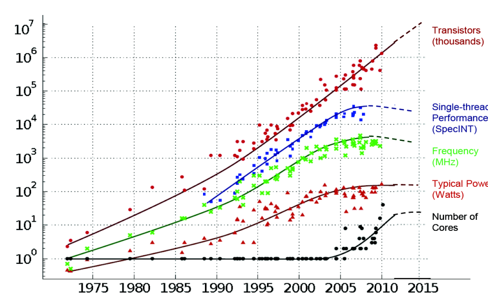
Kan det gjøres raskere?

# Et historisk blikk

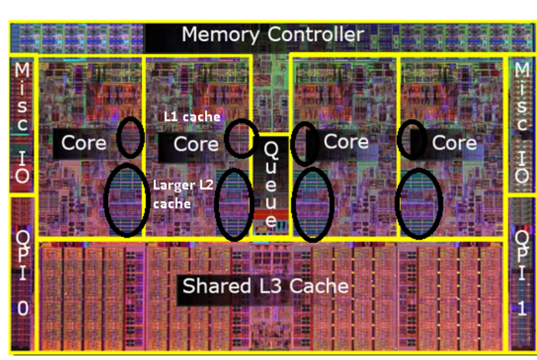
I mange år ble prosessorene raskere og raskere fordi man klarte å lage prosessorkretsene mer og mer kompakte.

Fra ca år 2008 gikk det ikke lenger.

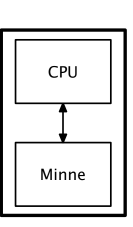
Hva kunne man da gjøre?

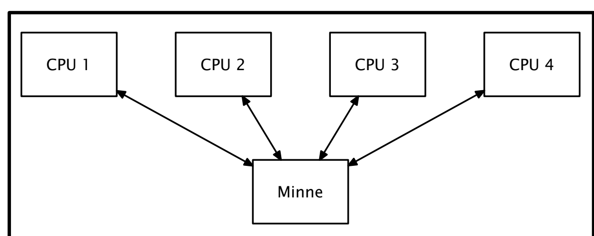


I stedet for å gjøre enkeltprosessoren raskere begynte man å legge flere prosessorer på hver brikke, i dag typisk 2–16.



Hver prosessor er på mange måter en egen datamaskin, men de har felles minne.



La oss i stedet kjøre primtallssjekkeren våre på 4 prosessorer samtidig. (Koden blir forklart senere.)

```
long a1 = 1_000_000_000;
Thread t1 = new Thread(new Leter(a1, a1+25));
Thread t2 = new Thread(new Leter(a1+26, a1+50));
Thread t3 = new Thread(new Leter(a1+51, a1+75));
Thread t4 = new Thread(new Leter(a1+76, a1+100));
t1.start();  t2.start();  t3.start();  t4.start();
```

## Kjøring

### Med 1 tråd

```
$ time java FinnPrimtall
1000000007 1000000009 1000000021 1000000033
1000000087 1000000093 1000000097

real    0m21.973s
user    0m21.719s
sys     0m0.119s
```

### Med 4 tråder

```
$ time java FinnPrimtall4
1000000007 1000000033 1000000087 1000000009
1000000093 1000000021 1000000097
real    0m9.565s
user    0m22.030s
sys     0m0.099s
```

# Når bruker vi tråder?

- Få programmer til å gå raskere
- Noen programmer blir enklere å programmere

## Prosesser og tråder (en oppklaring)

**Prosesser** er laget for hovedsaklig å jobbe alene; kommunikasjon med andre prosesser er vanskelig. Typisk vil hver kommando vi gir, starte en prosess.

**Tråder** (også kalt *lettvektsprosesser*) er laget for samarbeid; de har felles lager så kommunikasjon er enkelt. Hver prosess vil inneholde én eller flere tråder.

# Hvordan bruke tråder i Java?

For å starte en egen tråd må man gjøre tre ting:

1. Skrive koden som skal kjøres i tråden; dette gjøres i metoden run i en Runnable-klasse:

```
class Kode implements Runnable {
     @Override public void run() { ... } }
```

2. Lage en tråd for koden:

```
Thread traad = new Thread(new Kode());
```

3. Starte tråden:

```
traad.start();
```

## Når en tråd starter

... får tråden i utgangspunktet en egen prosessor å kjøre på. Hver tråd går derfor i parallell med og uavhengig av andre tråder.

## Hvis det ikke er nok prosessorer

... kommer operativsystemet til hjelp:

- Hver prosessor kan kjøre én eller flere tråder.
- OSet vil sjonglere med trådene og la prosessoren bytte på å kjøre dem.

# Eksempel

Her er et minimalt program som oppretter to nye tråder:

```
class PerKari {
    public static void main (String[] arg) {
        Thread per = new Thread(new Per());
        Thread kari = new Thread(new Kari());
        per.start();
        kari.start();
        System.out.println("Det var det.");
    }
}


class Per implements Runnable {
    @Override
    public void run () {
        System.out.println("Hei, jeg er Per.");
    }
}

class Kari implements Runnable {
    @Override
    public void run () {
        System.out.println("Hei, jeg er Kari.");
    }
}
```

# Kjøring

Kjøringen går slik:

```
$ java PerKari
Hei, jeg er Per.
Hei, jeg er Kari.
Det var det.
```

```
$ java PerKari
Det var det.
Hei, jeg er Kari.
Hei, jeg er Per.
```

(Siden trådene går i parallell, vil de kunne skrive ut i ulik rekkefølge.)

# Et eksempel til

La oss se på et eksempel på at noen programmer blir enklere med tråder:

## En stoppeklokke

Vi skal lage en stoppeklokke som vi kan starte og stoppe ved å trykke på Return-tasten. Når stoppeklokken går, skal den hvert sekund skrive ut en sekundteller.

Dette programmet vil vente på to ulike ting:

1. At det er gått et nytt sekund
2. At brukeren trykker på Return for å avslutte

Dette løses enklest ved å ha to tråder.

# Slumring

Klokketråden må kunne sove ett sekund av gangen; dette kan vi oppnå med metoden Thread.sleep:

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) { ... }
```

(Vi må ha med try-catch siden vi må angi hva som skal skje om søvnen blir avbrutt.)

# Stoppeklokken

```
class Stoppeklokke {
    public static void main (String[] arg) {
        Scanner tastatur = new Scanner(System.in);
        System.out.print("Trykk Return for aa starte... ");
        tastatur.nextLine();

        Thread klokke = new Thread(new Klokke());
        klokke.start();

        System.out.print("Trykk Return for aa stoppe...");
        tastatur.nextLine();
        System.out.println("Takk for naa");
    }
}

class Klokke implements Runnable {
    @Override
    public void run () {
        int tid = 0;
        while (true) {
            System.out.print(tid + " ");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) { return; }
            ++tid;
        }
    }
}
```

## Kjøring

```
$ java Stoppeklokke
Trykk Return for aa starte...  ↵
Trykk Return for aa stoppe...0 1 2 3 4 5  ↵
Takk for naa
6 7 8 9 10 11 12 13 14 15 16  Ctrl + C
```

## Forklaring

Dette gikk ikke helt som forventet. Hva skjer?

- Et Java-program starter med én tråd men kan lage flere.
- Selv om én tråd er ferdig, fortsetter de andre.
- Først når alle trådene er avsluttet, er programmet ferdig.

## Løsning

Vi kan heldigvis bryte av andre tråder med interrupt:

```
Thread klokke = new Thread(new Klokke());
klokke.start();

System.out.print("Trykk Return for aa stoppe...");
tastatur.nextLine();
klokke.interrupt();
System.out.println("Takk for naa");
```

Nå går det bedre:

```
$ java Stoppeklokke2
Trykk Return for aa starte... ↩
Trykk Return for aa stoppe...0 1 2 3 4 5 ↩
Takk for naa
$
```

# Eksempel: En bank

La oss lage et banksystem der kunder kan opprette konto og sette inn penger:

```
class Banksystem {
    public static void main (String[] arg) {
        Bankkonto k = new Bankkonto();
        Thread ola = new Thread(new Kunde(k, "Ola"));
        Thread anne = new Thread(new Kunde(k, "Anne"));
        ola.start();  anne.start();
        System.out.println("Kontoen er paa kr " + k.status());
    }
}
```

```
class Bankkonto {
    private int saldo = 0;

    public void settInn (int kr) {
        int nySaldo = saldo + kr;
        saldo = nySaldo;
    }

    public int status () {
        return saldo;
    }
}
```

```
class Kunde implements Runnable {
    Bankkonto konto;
    String navn;

    Kunde (Bankkonto k, String n) {
        konto = k;  navn = n;
    }

    @Override
    public void run () {
        System.out.println(navn +
            " setter inn 100 kr.");
        konto.settInn(100);
    }
}
```

## Kjøring

Dette går ikke helt som forventet:

```
$ java Banksystem
Kontoen er paa kr 0
Ola setter inn 100 kr.
Anne setter inn 100 kr.
```

## Hva gikk galt?

De tre trådene (hovedtråden, Ola-tråden og Anne-tråden) går i parallell og hovedtråden blir ferdig før Ola og Anne rekker å sette inn pengene sine.

# Vente på andre tråder

For å få riktig saldo må hovedtråden vente til Ola-tråden og Anne-tråden er ferdig. Dette kan vi gjøre slik med join:

```
public static void main (String[] arg) {
    Bankkonto k = new Bankkonto();
    Thread ola = new Thread(new Kunde(k, "Ola"));
    Thread anne = new Thread(new Kunde(k, "Anne"));
    ola.start();  anne.start();
    try {
        ola.join();  anne.join();
    } catch (InterruptedException e) {}
    System.out.println("Kontoen er paa kr " + k.status());
}
```

Dette går bedre:

```
$ java Banksystem2
Ola setter inn 100 kr.
Anne setter inn 100 kr.
Kontoen er paa kr 200
```

# Felles variabler under stress

Bankeksemplet vårt virker tilsynelatende helt fint, men hva skjer om vi lar de to kundene sette inn 1 million kroner i stort tempo:

```
class Kunde implements Runnable {
    @Override
    public void run () {
        for (int i = 1;  i <= 1_000_000;  ++i)
            konto.settInn(1);
    }
}
```

Det går dårlig:

```
$ java Banksystem3
Kontoen er paa kr 1022634
```

# Hva går galt?

```
class Bankkonto {
    private int saldo = 0;

    public void settInn (int kr) {
        int nySaldo = saldo + kr;
        saldo = nySaldo;
    }
```

Det har oppstått et *kappløp* («race condition», en situasjon der svaret kan avhenge av hvor fort trådene går og vil variere fra kjøring til kjøring). Det kan skje slik: La oss anta at saldo er 0.

| Ola | Anne |
|---|---|
| kaller settInn(1) | |
| beregner nySaldo=1 | |
| | kaller settInn(1) |
| | beregner nySaldo=1 |
| setter saldo=1 | |
| | setter saldo=1 |

# Viktige begreper

## Felles ressurs

Instansvariabelen `saldo` er en **felles ressurs** som flere tråder er interessert i å bruke.

## Kritisk region

Kode som *endrer* en felles ressurs (metoden `settInn` i vårt tilfelle), kalles en **kritisk region**. For å unngå feil må kun én tråd utføre den kritiske regionen av gangen.

## Kappløp

Ved manglende sikring av kritiske regioner kan det oppstå **kappløp** («race condition»).

# Synkronisering

For å unngå kappløp kan vi i Java beskytte kritiske regioner ved å **låse** tilgangen:

- Vi trenger en lås fra Java-biblioteket:
```
import java.util.concurrent.locks.*;
```
- Vi må opprette en lås:
```
    Lock laas = new ReentrantLock(true);
```
- Vi kan nå låse den kritiske aksessen:
```
    laas.lock();
```
- Etterpå må vi låse opp igjen:
```
    laas.unlock();
```

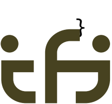
# Koden

```
import java.util.concurrent.locks.*;

class Bankkonto {
    private int saldo = 0;
    Lock laas = new ReentrantLock(true);

    public void settInn (int kr) {
        laas.lock();
        try {
            int nySaldo = saldo + kr;
            saldo = nySaldo;
        } finally {
            laas.unlock();
        }
    }

    public int status () {
        return saldo;
    }
}
```



## Bruk finally!

Legg merke til

```
laas.lock();
try {
  ...
} finally {
  laas.unlock();
}
```

Bruk av **try-finally** sikrer at låsen alltid (absolutt alltid!) blir låst opp.

## Kjøring

Nå går kjøringen bra:

```
$ java Banksystem4
Kontoen er paa kr 2000000
```

## Men det koster . . .

```
$ time java Banksystem3 # Uten laas
Kontoen er paa kr 1047793

real    0m0.112s
user    0m0.089s
sys     0m0.038s
```

```
$ time java Banksystem4 # Med laas
Kontoen er paa kr 2000000

real    0m6.860s
user    0m3.083s
sys     0m4.166s
```

# Oppsummering

I dag har dere lært:

- En **tråd** er en del av programmet som utføres i parallell med andre tråder.
- En tråd kan sove ved å kalle **Thread.sleep**.
- En trå kan vente på en annen ved å kalle **join**.
- En **kritisk region** er programkode (f eks en metode) som kun skal utføres av én tråd av gangen. Hvis ikke, kan det oppstå **kappløp** der resultatet er uvisst.
- En kritisk region kan beskyttes med en **lås**.

## Neste uke

Neste uke skal Stein fortelle om hvordan man får mange tråder til å samarbeide på en god og trygg måte.