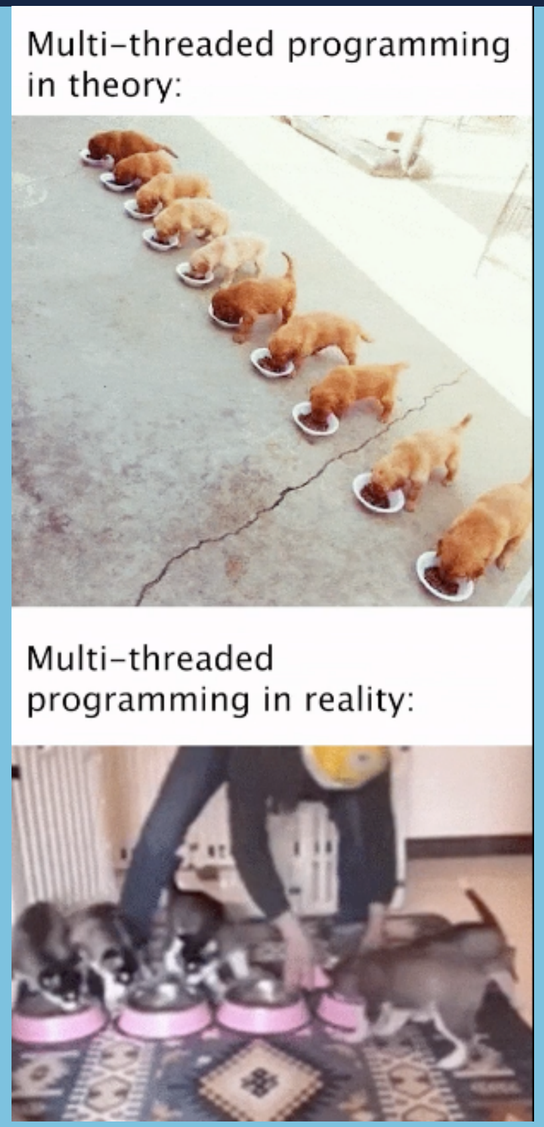
Uke 11 - Tråder

Gruppe 11 - jonasbny@ifi.uio.no

Agenda



Parallellprogrammering

Tråder

Låser og Monitorer

Conditions

Oppgaver

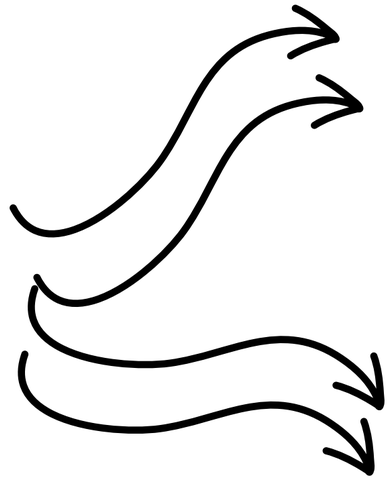
Parallellprogrammering

Moderne datamaskiner har mer enn en prosessorkjerne

Vi kan lage programmer som utnytter flere kjerner samtidig

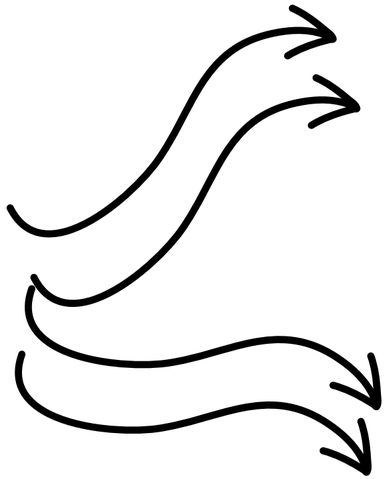
Dette gjør at vi kan effektivisere krevende oppgaver som sortering, finne primtall, hente informasjon fra nettet

Til dette bruker vi **tråder**



Fordeler

Hittil har vi skrevet sekvensielle, ikke-parallelle programmer. Programmer utført av en tråd, *main tråden*



Hvis vi vil at to eller flere oppgaver skal kjøre *samtidig* kan vi lage flere tråder

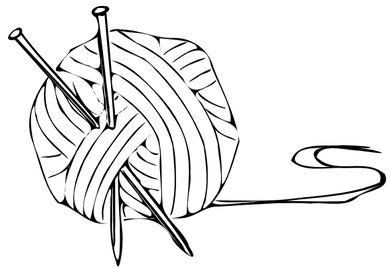
Vi snakker om *naturlig parallellitet* når det er naturlig for oss å stykke opp en oppgave i flere deloppgaver og løse de samtidig for å komme frem til ønsket resultat

Skal vi lage taco sammen, er det naturlig at en kutter grønnsaker og en annen steker kjøttdeigen

Tråder

Vi kan tenke på tråder i form av arbeidsfordeling på jobben. Man er flere ansatte som utfører ulike oppgaver samtidig (i parallell)

Innen programmering, er en tråd en sekvens av instruksjoner i et program (“thread of execution”)



Når flere tråder utfører forskjellige oppgaver (instruksjoner), får vi gjort mer på samme tid

Tråder kan vente, sove og avbrytes (merk InterruptedException)

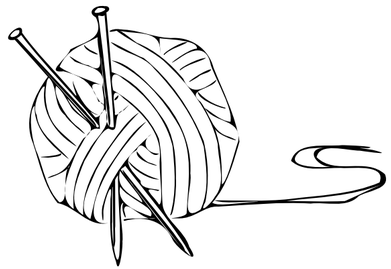
I IN1010 skal ikke trådene kommunisere direkte, kun bruke delte objekter (som kalles monitorer med kritiske regioner)

(workerClass må ha implementert Runnable)

Tråder

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

Thread er et innebygd objekt i Java som kan starte en ny **tråd**



Som parameter trenger en tråd et objekt som implementerer interfacet **Runnable**

Deretter kan man kalle på metoden start() på tråd objektet for å sette i gang en ny separat tråd

Det er koden i arbeider objektet som utføres i den nye tråden

Arbeiderklasse



Arbeider-klassen kan hete akkurat hva vi vil, men gjerne noe beskrivende

Klassen må implementere **Runnable**, som krever at vi skriver metoden **run()**

Det er i **run()** metoden vi skriver koden vi vil at arbeideren skal utføre

```
@Override
public void run(){
    // ...
}
```

Opprette og sette i gang tråder

Vi vil ofte opprette flere tråder

Det kan vi gjøre med for eksempel for loops

```
Thread[] threads = new Thread[5];
for(int i = 0; i < 5; i++){
    threads[i] = new Thread(new workerClass());
    threads[i].start();
}
```

Vente på tråder

Noen ganger må vi vente med å forsette main-tråden til alle trådene er ferdige

Det kan vi gjøre **join()**

```
try{
    for(int i = 0; i < 5; i++){
        threads[i].join();
    }
} catch(InterruptedException e){
    e.printStackTrace();
    // håndtere exception ...
}
```

Sovende tråder

En tråd kan “sove” et gitt antall millisekunder

Det kan være nyttig hvis vi ønsker å se ting skje i et litt langsommere tempo

```
static void                                    sleep(long millis)
                                               Causes the currently ex
                                               system timers and sche
```

```java
try {
    Thread.sleep(millis:1000);//"sover" et sekund
} catch (InterruptedException e) {
    e.printStackTrace();
    // håndtere exception
}
```

Synkroniseringsproblemer

Dersom to eller flere tråder deler data, kan vi få uheldige resultater om vi ikke lar bare en tråd utføre endringer på dataene av gangen og de prøver å oppdatere samme verdi *samtidig*

Dette kalles *race condition* og er kanskje den største kilden til feil i parallelle programmer

Løsningen får vi ved å bruke låser

(Noen ganger klarer de det, noen ganger ikke:)

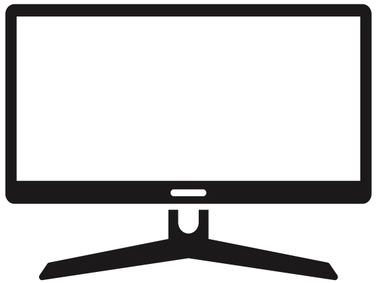
```
Expected result: 20000
starting thread, shared counter: 0
starting thread, shared counter: 0
done!, shared counter: 11070
done!, shared counter: 15668
```

```
Expected result: 20000
starting thread, shared counter: 0
starting thread, shared counter: 0
done!, shared counter: 10000
done!, shared counter: 20000
```

Låser og Monitorer

Når flere tråder jobber i samme minneområde kan vi få synkroniseringsproblemer

Flere tråder kan prøve å lese og skrive til samme minneområde samtidig, og da får vi ofte feil svar



Det finnes flere måter å løse dette på, men i IN1010 bruker vi hovedsakelig **låser**

Som oftes oppretter vi en egen klasse for hver data vi trenger å låse for synkronisering, og den klassen kalles da en **monitor**

Låser og Monitorer

For å løse synkroniseringsproblemer sørger vi for at kun en tråd kan aksessere et delt objekt til en gitt tid

Vi lager en kritisk region hvor kun en tråd kan gjøre endringer av gangen, andre tråder må stoppe og vente på sin tur

Man kan se for seg at man har dører på hver side av koden eller dataen man vil beskytte

Hvis en tråd låser døren når de går inn og skal gjøre endringer, må de andre trådene vente på at tråden som låste låser opp døren igjen



Try {. . .} Finally {. . .}

Når vi bruker låser er vi nødt til å forsikre oss om at låsen *alltid!* blir låst opp igjen etter bruk



Dette gjør at programmet vårt ikke nødvendigvis krasjer selv om en tråd krasjer



Vi “låser” locken(døren) ,deretter putter vi koden vår i en **try**-blokk, etterpå låser vi opp locken(døren) i en **finally**-blokk

Det som står i **finally**-blokken blir *alltid* utført, selv hvis tråden krasjer

Conditions

Av og til har vi flere *ulike* arbeidere som jobber i samme minneområde
For eksempel:

- En type arbeider som “produserer” data og legger dem i en beholder
- En type arbeider som henter ut data fra beholderen når den første arbeidertypen har lagt noe inn
-





Her kan to ting skje:

- Beholderen er full, og den første arbeideren må vente til noen har hentet data ut
- Beholderen er tom, og den andre arbeideren må vente til det kommer data inn

Hvis arbeiderene bare sjekker hele tiden bruker det veldig mye overflødig prosessortid

Conditions

Med conditions kan vi la trådene vente *passivt*, vi lar de "sove" til de får beskjed om å våkne

En condition opprettes via en lås slik:
Lock laas = new ReentrantLock();
Condition cond = laas.newCondition();



En tråd kan vente på at noe skjer ved å kalle på **await()**
Dette gjør at tråden midlertidig låser opp låsen den holder

En annen tråd kan gi beskjed til trådene som venter med **signal()** eller **signalAll()**

Tråder og exceptions

Når vi jobber med tråder kan det være litt tricky å huske akkurat hvor vi må håndtere exceptions



Her er en liten oversikt:

- **join()** og **sleep()** må skje i **en try / catch** blokk der vi fanger **InterruptedException**
- Når vi låser og låser opp en lås må vi alltid bruke **try / finally**
- **await()** kaster **InterruptedException**, men siden vi som oftest har dette inni i en **try / finally** er det vanlig at vi kaster denne videre og håndterer det et annet sted

Oppgaver

