IN1010

Repetisjon Uke 9
Tråder og parallellitet, del 1

(Basert på Stein Gjessings presentasjon)

21. mars, 2025

Tråder



© Aardman Animations

- Parallellitet (parallelt eller samtidig?)
- CPU og kjerner
- Prosesser og tråder
- Hvordan?
  - Låser, venting, signaler
  - Produsent/konsument

Parallellitet

En bonde og 50 sauer

Alene bruker han kanskje noen timer på å klippe sauene.

Men hvis bonden får hjelp fra en til,

så kan de gjøre det på halve tiden

Og om de er 50 til å klippe, så går det veldig fort

Men de må huske å ikke bare klippe den samme sauen

Parallellitet

Skogen

Hvis vi planter 2000 trær, så har vi en skog om 2000 dager.

Men vil vi ha bare *ett* tre, tar det også like lang tid.

2000 personer som planter ett tre, kan

ikke få det til å vokse på en dag

Parallellitet

Kortstokk

Hvis vi vil sortere en kortstokk med 52 kort:

er det raskere om *en* gjør det?

to?

eller 52 personer?

Parallellitet

Dyrehage

Du har en liten hund,

går det fortere å mate den om du får hjelp?

I en dyrehage med 25 dyr, så skal de mates 3-4 ganger om dagen.

Hvor mange trenger vi for å mate de?

Parallellitet

Kjøleskapet

Si at du bor i en stor familie eller kollektiv

En morgen er det tomt for melk, og du bestemmer deg for å kjøpe

Men det samme tenker og gjør alle de andre.

Parallellitet

Kjøleskapet II

Siden dere sist gang kjøpte for mye melk, så

har dere bestemt dere nå for å gi beskjed på forhånd

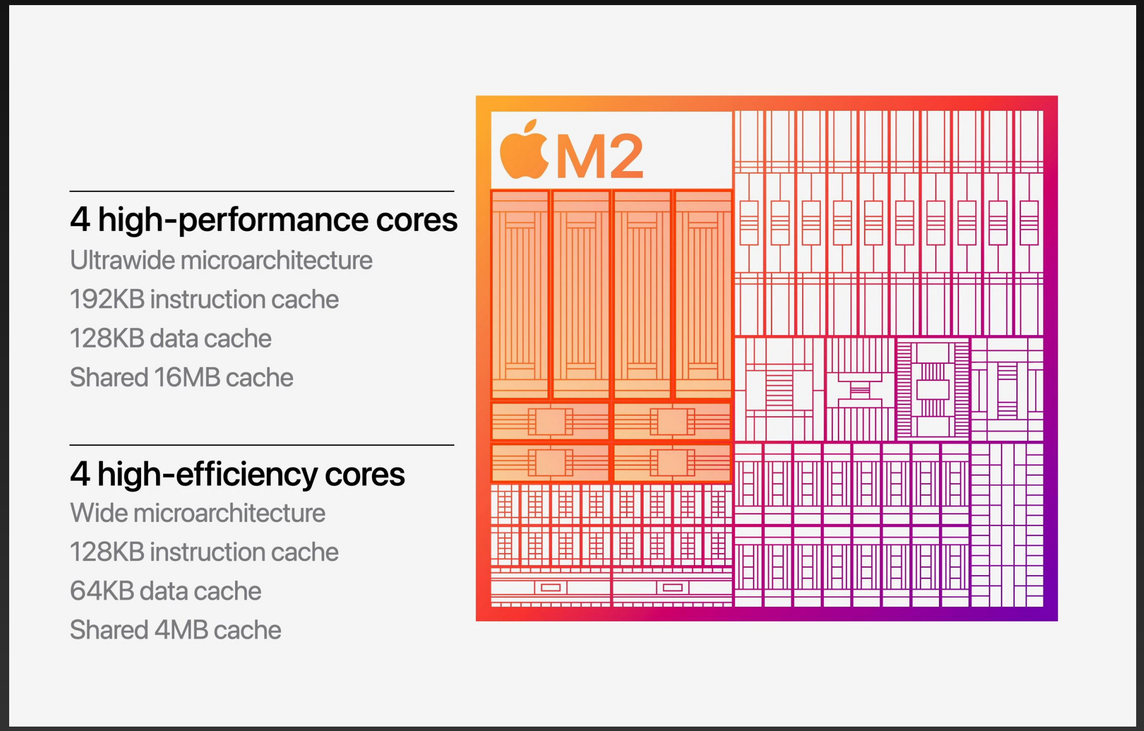
Så når det er tomt, så sender alle en melding om at han/hun skal kjøpe og at da trenger ikke de andre å gjøre det

CPU, prosessorer og kjerner

CPU er hovedprosessoren i en datamaskin

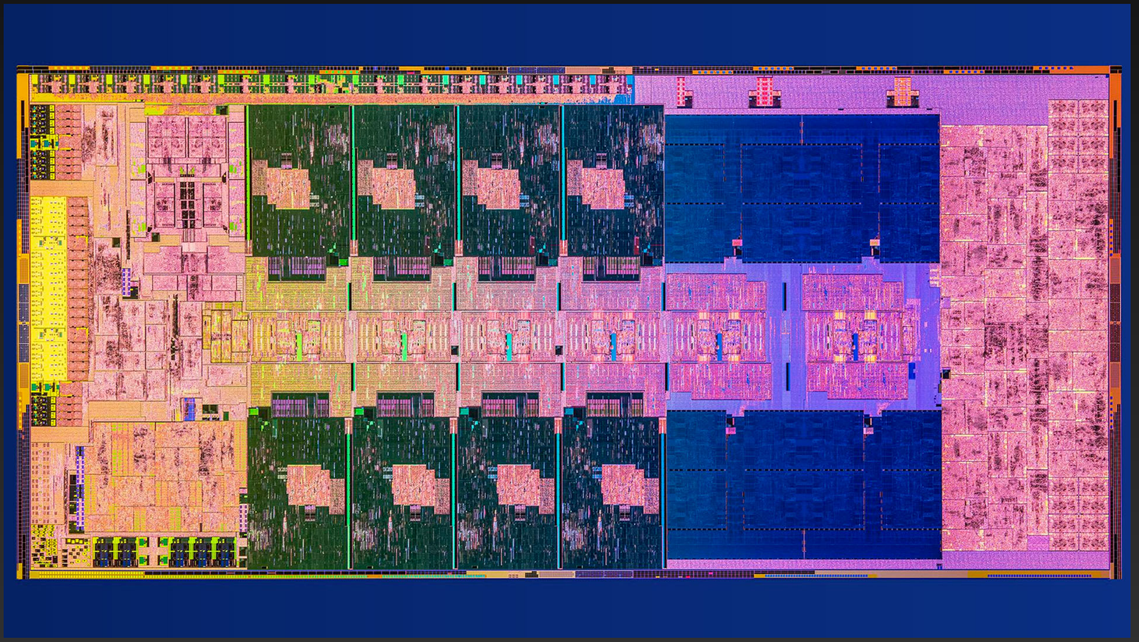
Men det kan også være flere

I nyere prosessorer så består hver CPU av flere kjerner (cores), og hver av de fungerer i praksis som en CPU



Apple sin M2 prosessor

CPU, prosessorer og kjerner



Intel 13. generasjon CPU

Vi har også minne, fra harddisk, SSD, RAM til cache og registere

Flere kjerner og CPU har ofte egen cache, men de deler RAM og disker

Prosesser og tråder

Prosesser er "programmer"

De har "eget" minneområde, og kan bare kommunisere med andre prossesser gjennom operativsystemet (IPC / system calls)

Oppstart av en prosess er "mye" jobb og kan ta "lang" tid

Prosesser og tråder

Tråder er egne enheter i en prosess

De deler det tildelte minneområdet, og kan derfor kommunisere mellom seg gjennom dette uten å gå gjennom OS'et

"Mye" raskere å opprette enn prosesser

Operativsystemet

Med operativsystem mener vi helst OS-kjernen

(kernel – ikke samme som CPU-kjerne (core))

Den bestemmer når prosesser og tråder* kan kjøre

Vi har en *kjøre-kø* og en *blokk-kø* for hver CPU-kjerne

Prosesser og tråder som er klar for å kjøre venter i kjøre-køen

Men hvis de sover eller venter på andre hendelser så settes de i blokk-køen

Falsk parallellitet

Hvordan kan en CPU-kjerne kjøre flere 100 eller 1000 prosesser og tråder?

Alle som venter i kjøre-køen får tildelt et tidsvindu som de får lov til å kjøre. Men de vet aldri når det skjer og hvor lenge.

Det kalles concurrency

Ekte parallellitet

Hvis en prosess eller tråd får tildelt en CPU-kjerne alene, så kan den kjøre hele tiden hvis den vil.

Og hvis flere tråder eller kjerner er alene om hver sin kjerne, så er det ekte parallellitet

Men i praksis er det ikke så merkbart for vanlig programmering

og prosessene og trådene vil både kjøre parallelt og i serie

Antall kjerner

```
class Kjerner {
    public static void main(String[] args) {
        System.out.println(Runtime.getRuntime().availableProcessors());
    }
}
```

Et program som skriver ut hvor mange CPU kjerner som Java har tilgjengelig

Opprette tråder i Java

Klassen *Thread* og interfacet *Runnable*

Når vi har en klasse som vi ønsker skal ha en tråd, så bruker vi grensesnittet *Runnable*

Selv om det er mulig å opprette en *Thread* direkte (bl.a. arv), så er *Runnable* mer fleksibelt siden det er et interface

Opprette tråder i Java

```
class MinFineRunnable implements Runnable {
    …
    @Override
    public void run() {
        … // Hva tråden skal gjøre
    }
}
```

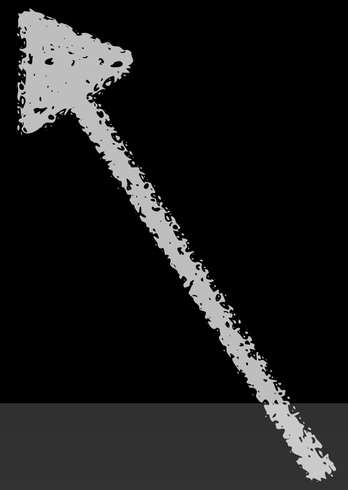
Vi må ha en metode run() som tråden kommer til å kjøre

Viktige metoder i Thread

- *start()* – Starter tråden (for at tråden skal kjøre *run()*-metoden vår)
- *join()* – Venter til tråden er ferdig
- *getName()* – Henter navnet til tråden
- *currentThread()* – Gir oss en referanse til tråden som kjøres
- *interrupt()* – Avbryter tråden, fører til et avbrudds-unntak
- *sleep(long millis)* – Forteller at tråden skal sove en viss tid

Opprette tråder i Java

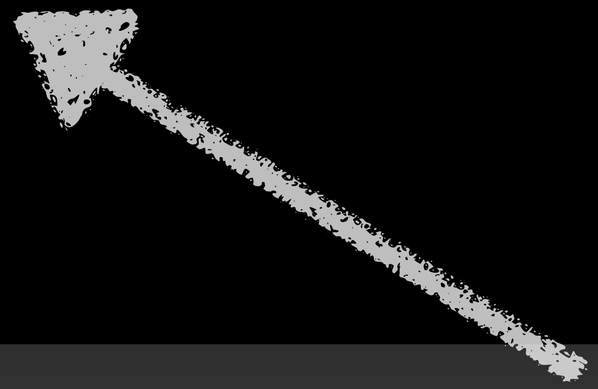
```
class MittFineProgram implements Runnable {
    public static void main(String[] args) {
        Runnable runnable = new MinFineRunnable();
        tråd = new Thread(runnable);
        tråd.start();
    }
}
```



Først må vi ha et *Runnable* objekt

Opprette tråder i Java

```
class MittFineProgram implements Runnable {
    public static void main(String[] args) {
        Runnable runnable = new MinFineRunnable();
        tråd = new Thread(runnable);
        tråd.start();
    }
}
```



Så oppretter vi et tråd objekt

Opprette tråder i Java

```
class MittFineProgram implements Runnable {
    public static void main(String[] args) {
        Runnable runnable = new MinFineRunnable();
        tråd = new Thread(runnable);
        tråd.start();
    }
}
```



Som vi til slutt starter med en *start()*-metode

Pass på at du bruker *start()* og ikke prøver å kjøre *run()* i *Runnable*-objektet

Opprette tråder i Java

```
class MittFineProgram implements Runnable {
    public static void main(String[] args) {
        tråd = new Thread(new MinFineRunnable());
        tråd.start();
        try {
            tråd.join();
        } catch (InterruptedException e) {
            System.out.println("Tråd-unntak i hovedtråd");
        }
    }
```



Hvis vi vil at hovedtråden skal vente med å kjøre videre til tråden har kjørt ferdig, så bruker vi en *join()*-metode. Den må ta hensyn til at ventingen kan bli avbrutt

Sove

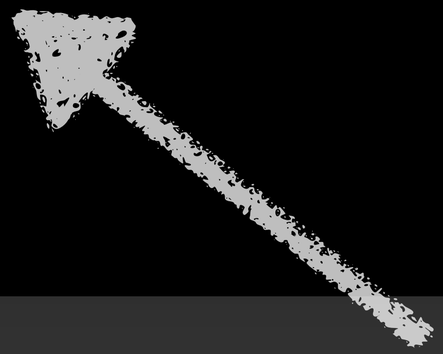
```
class MittFineProgram implements Runnable {
    public static void main(String[] args) {
        tråd = new Thread(new MinFineRunnable());
        tråd.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Tråd-unntak i hovedtråd");
        }
    }
```



Vi bruker sleep på samme måte når vi vil at tråden vi er i skal sove en viss tid, dette er en statisk metode så vi kaller ikke på et tråd-objekt

Avbrudd / Interrupt

```
class MittFineProgram implements Runnable {
    public static void main(String[] args) {
        tråd = new Thread(new MinFineRunnable());
        tråd.start();
        …
        tråd.interrupt();
    }
}
```



Vi kan avbryte tråden som vi har referanse til og hvis den sover/venter så utløser det et unntak, ellers så settes tråden som status avbrutt som koden i tråden kan sjekke

Avbrudd / Interrupt

Vi kan sjekke interrupt status ved hjelp av metodene

- *interrupted()* – Boolean som er sann hvis vi har blitt avbrutt, og fjerner avbrudd-statusen
- *isInterrupted()* – Samme som over men beholder avbrudd-statusen

Når oppdateres en variabel?

Når en tråd kjøres, bestemmes av operativsystemet. Det betyr at når vi gjør endringer i f.eks. en variabel, så vet vi ikke når det skjer.

Vi kan heller ikke gå ut fra hvilken tråd som kjører før eller etter tråden vi er i selv. Husk også at *main* kjøres i sin egen tråd.

Vi har heldigvis noen klasser med metoder som hjelper oss. Men først må vi identifisere det som vi kalle for kritiske regioner.

Når oppdateres en variabel?

Men hva gjør en seksjon eller region til en kritisk?

Hvis vi har en ressurs (som en variabel) som er felles mellom flere tråder, som de kan endre (altså ikke immutable) så vil de delene av koden hvor vi gjør endringer som oftest være en kritisk region.

Her må vi sørge for at kun en tråd av gangen får tilgang.

Låser

Når vi skal endre en variabel, så det som skjer i praksis som oftest

Lese inn verdi → beregne en ny verdi → skrive verdien tilbake

Så for å sørge for at vi kan fullføre hele prosessen sikkert, uten forstyrrelser, eller at andre tråder oppdaterer verdien i mens, så lager vi en "lås"

Låser

En lås sørger for at kun en tråd kan få tilgang, og låsen har en tilhørende kø som de neste trådene kan bruke til å vente i til det er klart

Låser

```
private static int resultatTeller = 0;
private static final ReentrantLock resultatLås = new ReentrantLock();
private static int hentResultat() {
    resultatLås.lock();
    try {
        return ++resultatTeller;
    } finally {
        resultatLås.unlock();
    }
}
```

Her har vi en teller som oppdateres når noe fullføres

Låser

```
private static int resultatTeller = 0;
private static final ReentrantLock resultatLås = new ReentrantLock();
private static int hentResultat() {
    resultatLås.lock();
    try {
        return ++resultatTeller;
    } finally {
        resultatLås.unlock();
    }
}
```



Her er resultatTeller en teller, som skal telle oppover når noe fullføres for å lage en resultatliste. Og den må telle riktig selv når noe fullføres på omtrent samme tid.

Låser

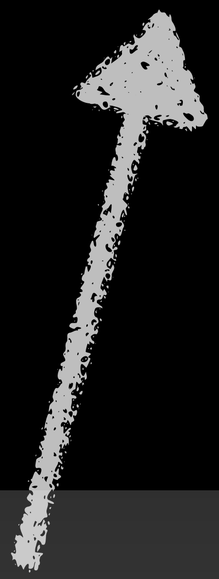
```
private static int resultatTeller = 0;
private static final ReentrantLock resultatLås = new ReentrantLock();
private static int hentResultat() {
    resultatLås.lock();
    try {
        return ++resultatTeller;
    } finally {
        resultatLås.unlock();
    }
}
```



Det er en felles ressurs for alle trådene, og der med en kritisk region

Låser

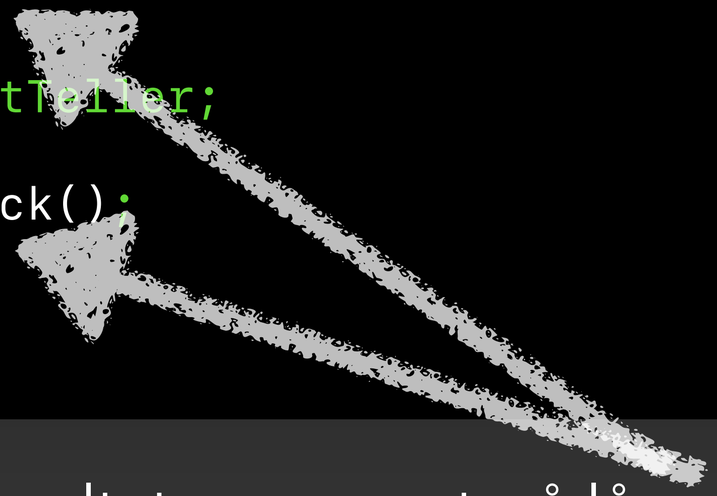
```
private static int resultatTeller = 0;
private static final ReentrantLock resultatLås = new ReentrantLock();
private static int hentResultat() {
    resultatLås.lock();
    try {
        return ++resultatTeller;
    } finally {
        resultatLås.unlock();
    }
}
```



Vi oppretter en lås

Låser

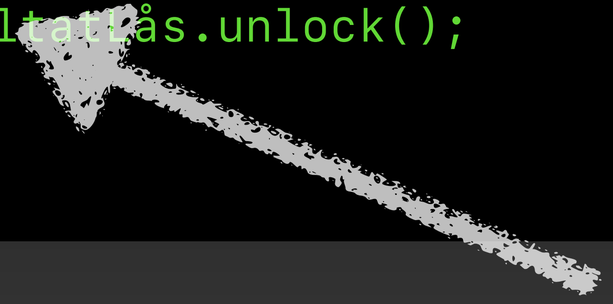
```
private static int resultatTeller = 0;
private static final ReentrantLock resultatLås = new ReentrantLock();
private static int hentResultat() {
    resultatLås.lock();
    try {
        return ++resultatTeller;
    } finally {
        resultatLås.unlock();
    }
}
```



Når vi skal endre og hente ut resultatnummeret, så låser vi slik at bare den første tråden i køen kan hente ut og samtidig øke telleren

Låser

```
private static int resultatTeller = 0;
private static final ReentrantLock resultatLås = new ReentrantLock();
private static int hentResultat() {
    resultatLås.lock();
    try {
        return ++resultatTeller;
    } finally {
        resultatLås.unlock();
    }
}
```



Vi bruker en finally-blokk, fordi det sørger for at den vil da alltid låses opp

Venting

Det er to måter som vi kan vente på, aktivt og passivt.

Aktivt er en løkke som hele tiden sjekker om en tilstand er sann. Bør unngås siden det betyr bruk av CPU og kan også hindre at annen kode kan kjøres.

Passiv venting bruker OS'et til å sette tråden i vente-køen, men er avhengig av å bli vekket opp.

Passiv venting

```
Lock lås = new ReentrantLock();
Condition ikkeTom = lås.newCondition();

public void settInn (Ting tingen) {
  lås.lock();
  try {
    behold.settInn(tingen);
    ikkeTom.signal();
  } finally {
    lås.unlock()
  }
}
```

```
public Ting taUt () throws
InterrupedException {
  lås.lock();
  try {
    if (behold.tom()) {
      ikkeTom.await();
    }
    return (behold.hentUt());
  } finally {
    lås.unlock()
  }
}
```

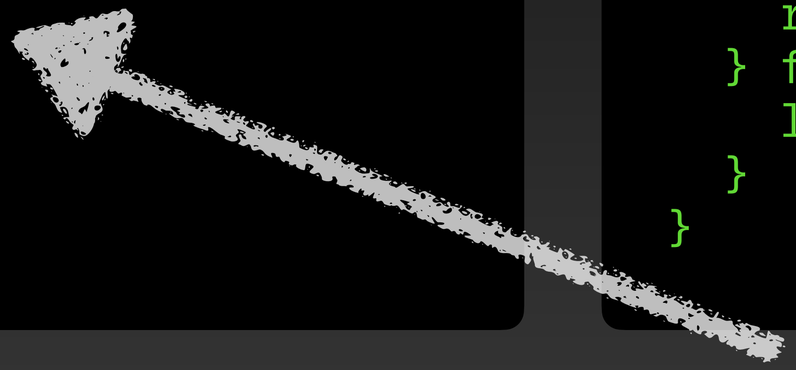
To metoder der den ene venter på noe som kan tas ut, og venter på signal fra den første metoden

Passiv venting

```
Lock lås = new ReentrantLock();
Condition ikkeTom = lås.newCondition();

public void settInn (Ting tingen) {
  lås.lock();
  try {
    behold.settInn(tingen);
    ikkeTom.signal();
  } finally {
    lås.unlock()
  }
}
```

```
public Ting taUt () throws
InterrupedException {
  lås.lock();
  try {
    if (behold.tom()) {
      ikkeTom.await();
    }
    return (behold.hentUt());
  } finally {
    lås.unlock()
  }
}
```



Men våkner opp når signalet kommer fra den andre tråden

Passiv venting

```
Lock lås = new ReentrantLock();
Condition ikkeTom = lås.newCondition();

public void settInn (Ting tingen) {
  lås.lock();
  try {
    behold.settInn(tingen);
    ikkeTom.signal();
  } finally {
    lås.unlock()
  }
}
```

```
public Ting taUt () throws
InterrupedException {
  lås.lock();
  try {
    if (behold.tom()) {
      ikkeTom.await();
    }
    return (behold.hentUt());
  } finally {
    lås.unlock()
  }
}
```



Gjør at tråden venter (sover)

Produsent/konsument

Et vanlig mønster for å strukturere parallell-programmering.

Vi har et viss antall tråder som "produserer"

Disse setter inn det de har produsert inn i det vi kaller en monitor

Så har vi et viss antall "konsument"-tråder

Disse venter på at produsenten har produsert noe

Produsent/konsument

Monitoren inneholder alle synkroniseringsprimitiver som trådene da ikke trenger å tenke på.

Java Object

Fra Java sin opprinnelse

Alle objekter har en innebygget lås. Her bruker vi `synchronized` som markerer hvor den kritiske regionen er. Mer begrenset enn nyere concurrency-biblioteket.

Bruker wait(), notify(), og notifyAll() i steden for await(), signal(), og signalAll(). Men kan føre til starvation og "livelock", altså at trådene aldri kommer til eller forbigått.

En del begreper

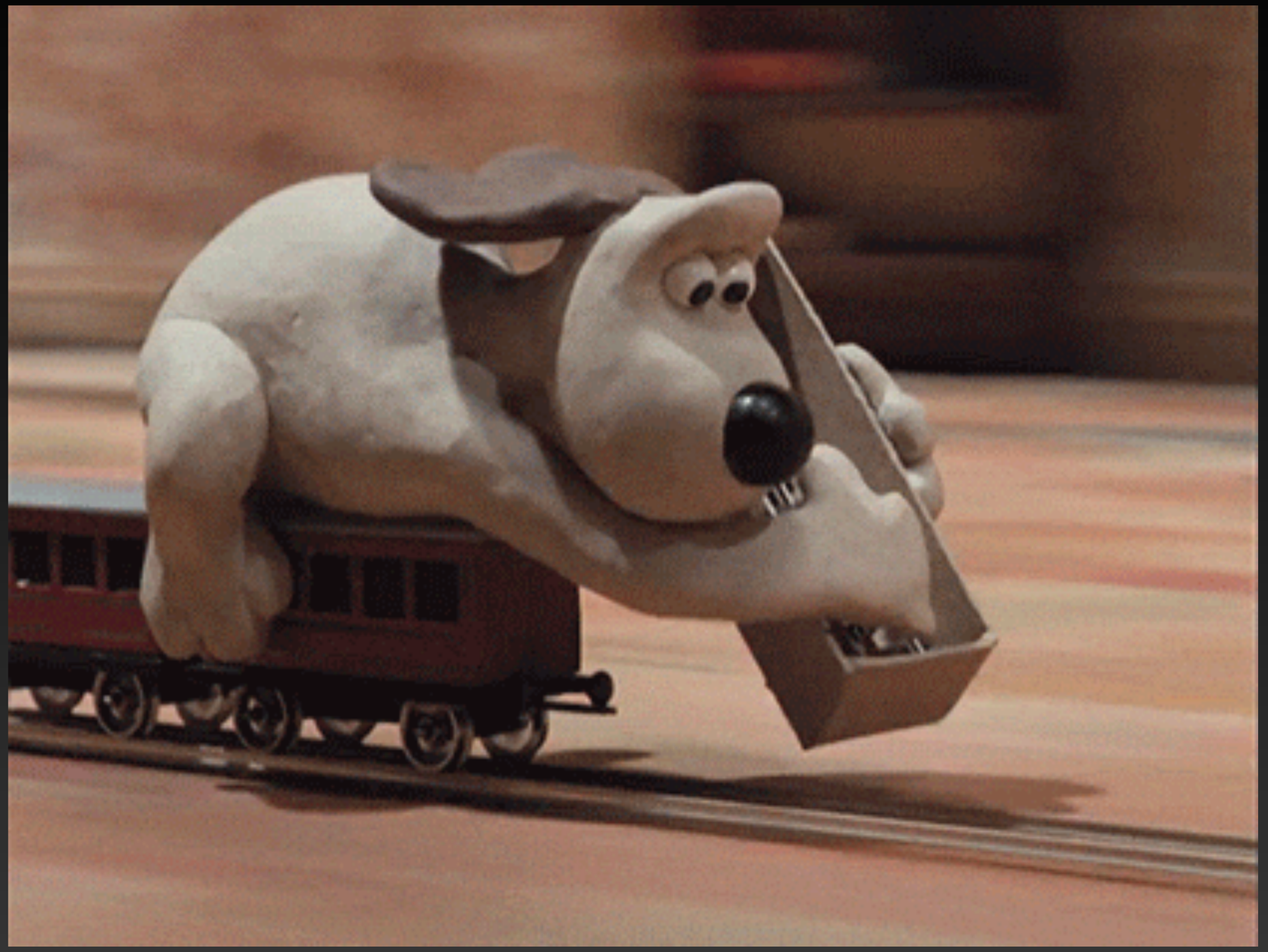
- *Synkronisering* – koordinere tråder slik at de bruker delte ressurser på en kontrollert og forutsigbar måte
- *Kritisk region* – en delt ressurs som ikke må kunne brukes samtidig av flere tråder
- *Race condition* – når to eller flere tråder bruker en delt ressurs samtidig, hvor resultatet både er uforutsigbart og avhenger av tidspunktet
- *Atomisk* – en operasjon som skjer i et øyeblikk hvis man ser fra en annens tråds perspektiv

En del begreper

- *Lås (mutex)* – et synkroniseringshjelpemiddel som gir tilgang til kun en tråd av gangen
- *Signal (condition)* – en variabel som lar tråder vente (sove) til en betingelse er sann
- *Monitor* – en struktur som kombinerer data og metoder for synkronisert tilgang
- *Spinlock* – aktiv venting, en løkke som hele tiden sjekker en betingelse, bør kun benyttes ved svært kort venting

En del begreper

- *Asynkront* – hendelse eller prosedyre som starter og kjører uavhengig
- *Read Write Lås* – lås som bare låser når vi vil skrive til variabelen men ikke lese
- *Volatile* – en variabel som skrives og hentes fra RAM og ikke CPU-kjernens cache, sikrer "synlighet" for alle tråder
- *Deadlock* – hvis to tråder hvor begge venter på at den andre skal bli ferdig før de kan fortsette



© Aardman Animations