

# IN1010 - Våren 2020

## Tråder

### del 1

11. mars 2019

Stein Gjessing,  
Institutt for informatikk,  
Universitetet i Oslo

Horstmann kap 20.1 – 20.3

Activity Monitor

All Processes Filter

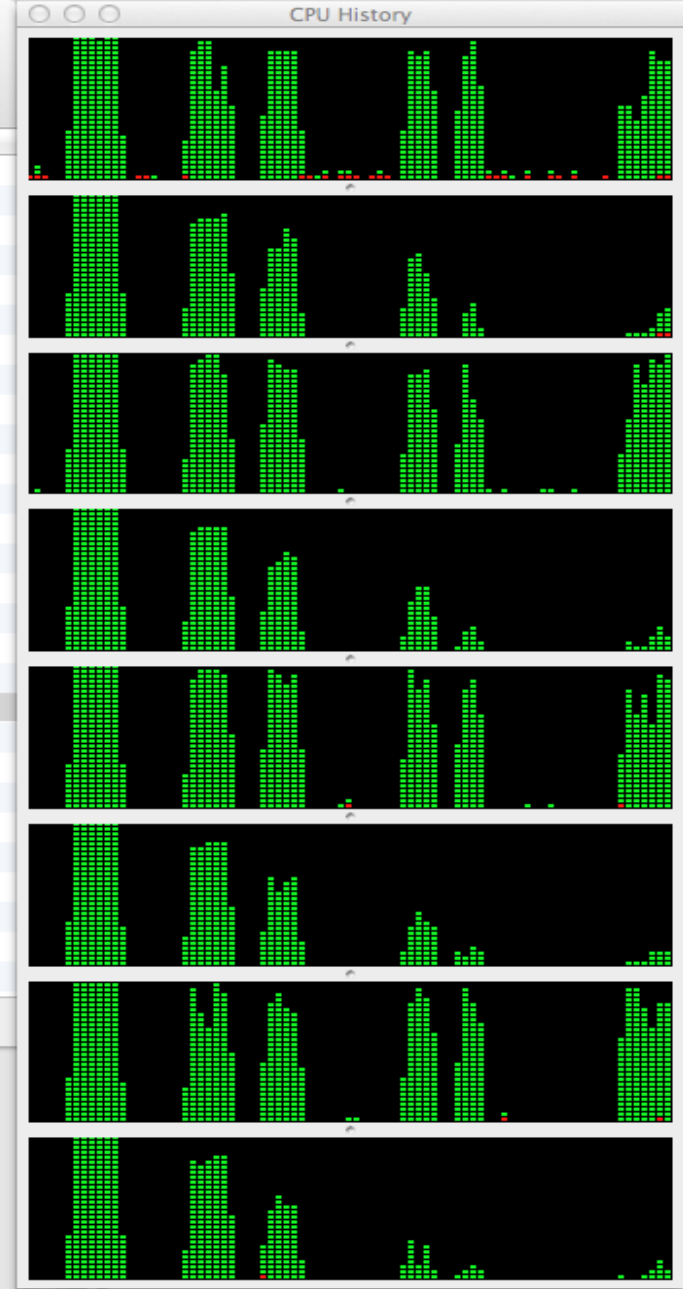
PID	Process Name	User	% CPU	Threads	Real Mem	Kind
29177	java	steing	300,3	25	20,5 MB	Intel (64 bit)
29178	cf-agent	root	77,1	1	5,2 MB	Intel (64 bit)
14	opendirectoryd	root	3,0	15	14,3 MB	Intel (64 bit)
19	distnoted	root	1,1	9	3,3 MB	Intel (64 bit)
27253	Activity Monitor	steing	1,1	2	27,2 MB	Intel (64 bit)
27265	activitymonitord	root	0,6	1	1,8 MB	Intel (64 bit)
0	kernel_task	root	0,6	79	677,3 MB	Intel (64 bit)
91	WindowServer	_windowserve	0,6	11	207,5 MB	Intel (64 bit)
1	launchd	root	0,6	3	4,0 MB	Intel (64 bit)
133	distnoted	steing	0,5	7	5,9 MB	Intel (64 bit)
12	notifd	root	0,5	5	1,5 MB	Intel (64 bit)
155	Grab	steing	0,3	5	136,0 MB	Intel (64 bit)
150	Acrobat	steing	0,3	21	144,8 MB	Intel
159	Microsoft PowerPoint	steing	0,2	12	183,9 MB	Intel
161	Microsoft Word	steing	0,2	5	125,6 MB	Intel
165	SystemUIServer	steing	0,2	2	24,8 MB	Intel (64 bit)
29204	writeconfig	root	0,1	3	3,7 MB	Intel (64 bit)
65	mDNS	root	0,1	10	110,0 MB	Intel (64 bit)
127	distnoted	_spotlight	0,0	5	1,6 MB	Intel (64 bit)
73	hidd	root	0,0	3	1,9 MB	Intel (64 bit)
13	mDNSResponder	_mdnsrespon	0,0	3	3,8 MB	Intel (64 bit)
29205	launchdadd	root	0,0	3	1,8 MB	Intel (64 bit)
22	securityd	root	0,0	4	8,1 MB	Intel (64 bit)
68	loginwindow	steing	0,0	3	55,9 MB	Intel (64 bit)
166	Finder	steing	0,0	15	151,7 MB	Intel (64 bit)
40469	mdworker	_spotlight	0,0	4	15,1 MB	Intel (64 bit)
17	syslogd	root	0,0	4	1,2 MB	Intel (64 bit)
20	fsevents	root	0,0	36	10,0 MB	Intel (64 bit)

CPU System Memory Disk Activity Disk Usage Network

% User: 50,12  
% System: 1,38  
% Idle: 48,51

Threads: 557  
Processes: 95

CPU Usage



# Oversikt

- Hva er parallelle programmer?
- Hvorfor parallelle programmer ?
- Hvordan kan dette skje på én kjerne/CPU/prosessor ?
- Hvordan kan dette skje med flere kjerner/CPU-er/prosessorer ?
- På engelsk: Parallel vs Concurrent computing
- Noen begreper:
  - Programmer, prosesser og **tråder**
- **Tråder** i Java
- Hvordan bruker vi **tråder**
- Oppdateringsproblemet (race conditions)
  - Samtidig oppdatering av data
    - Løsningen: "Monitorer" = Metodene er kritiske regioner



# Hva er parallelle programmer - I

- På en datamaskin kan flere programmer kjøre samtidig:
- **Virkelig samtidig:** To programmer får på samme tidspunkt utført hver sin instruksjon (må da ha flere kjerner/CPU-er/prosessorer)
  - Engelsk: Parallel computing
- **Tilsynelatende samtidig:** Maskinen (kjernen/CPU-en/prosessoren) skifter så raskt mellom programmene at du ikke merker det (flere ganger i sekundet). Maskinen utfører da noen få millioner instruksjoner for hvert program før den skynder seg til neste program.
  - Engelsk: Concurrent computing: Programmet/programmene kjører ekte eller bare tilsynelatende samtidig.
- På norsk har vi vanligvis ikke dette skille mellom ”parallel” og ”concurrent” – vi kaller alt parallell (parallelle programmer) eller samtidig programmer.  
Bruk: Virkelig/Ekte vs. Tilsynelatende parallellitet
- Uansett om programmene går virkelig eller tilsynelatende samtidig, må de behandles som om de går virkelig samtidig (fordi man ikke kan forutsi når maskinen skifter fra et program til et annet)



# Hva er parallelle programmer – I I

- Administrasjonen av hvilke program som kjører, og hvordan de deler tiden og kjernene mellom seg, gjøres av **operativsystemet** (Unix, Windows, Mac-OS). Operativsystemet er et meget stort program og er det første programmet som startes i maskinen. Det har alle tillatelser, men skrur mange av disse tillatelsene av for vanlige programmer (eks: skrive direkte til disken eller til alle steder i primærlageret/RAM).
- På alle (?) operativsystemer i dag kan *flere* brukere være pålogget og kjøre sine programmer samtidig og *flere* uavhengige programmer kan kjøre samtidig
  - Datamaskinen kan skifte rask mellom alle de ulike programmene som er i gang i datamaskinen
    - Eks: Du kjører både et Java-program, skriver et brev i Word og får en melding
    - De fleste mobiltelefoner og nettbrett har flere kjerner
  - Unntak:
    - en del svært enkle mobiltelefoner, nettbrett ol. ??
    - Mikrokontrollere – enkle embedded systems

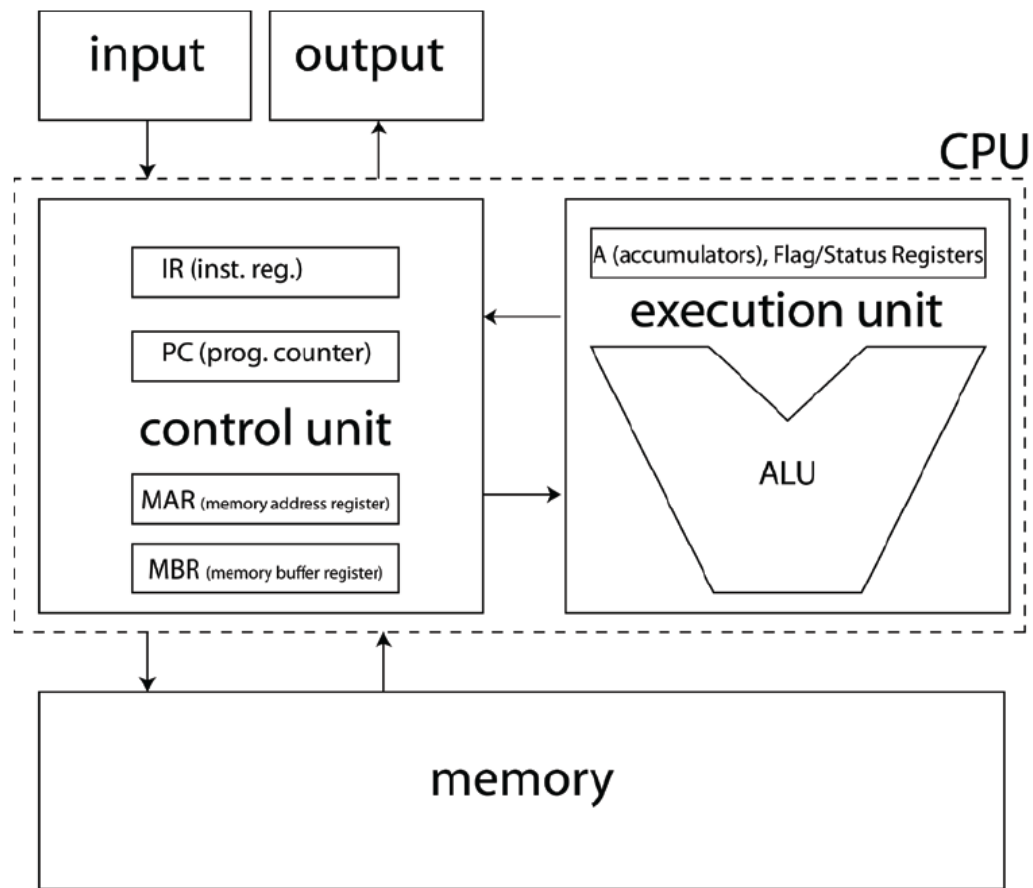


# Hvorfor parallelle programmer ?

- Maskinen har mye større kapasitet enn det vi vanligvis trenger; går for det meste på tomgang.
  - Min PC utfører vanligvis bare 0,1 % fornuftig arbeide
    - og mer enn 99% av tiden tomgangsprogrammet ' the System Idle Process'
  - Når jeg simulerer deler av Internet bruker jeg så mange kjerner som mulig
- En bruker trenger å kjøre flere programmer samtidig (se film/Skype samtidig som du jobber....)
- Ofte *må* mange ulike brukere jobbe på *samme data* samtidig (eks. et bestillingssystem med flere selgere på samme data: en kino, en flyavgang,... Dette skjer på "servere" (tjenermaskiner))
- "Dual-core", "8-core", "N-core": Ekte parallellitet



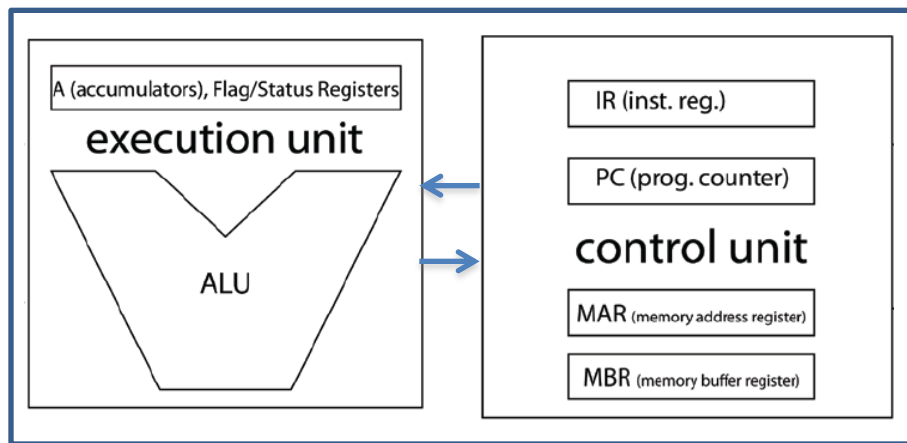
# Von Neumann Arkitektur Block Diagram



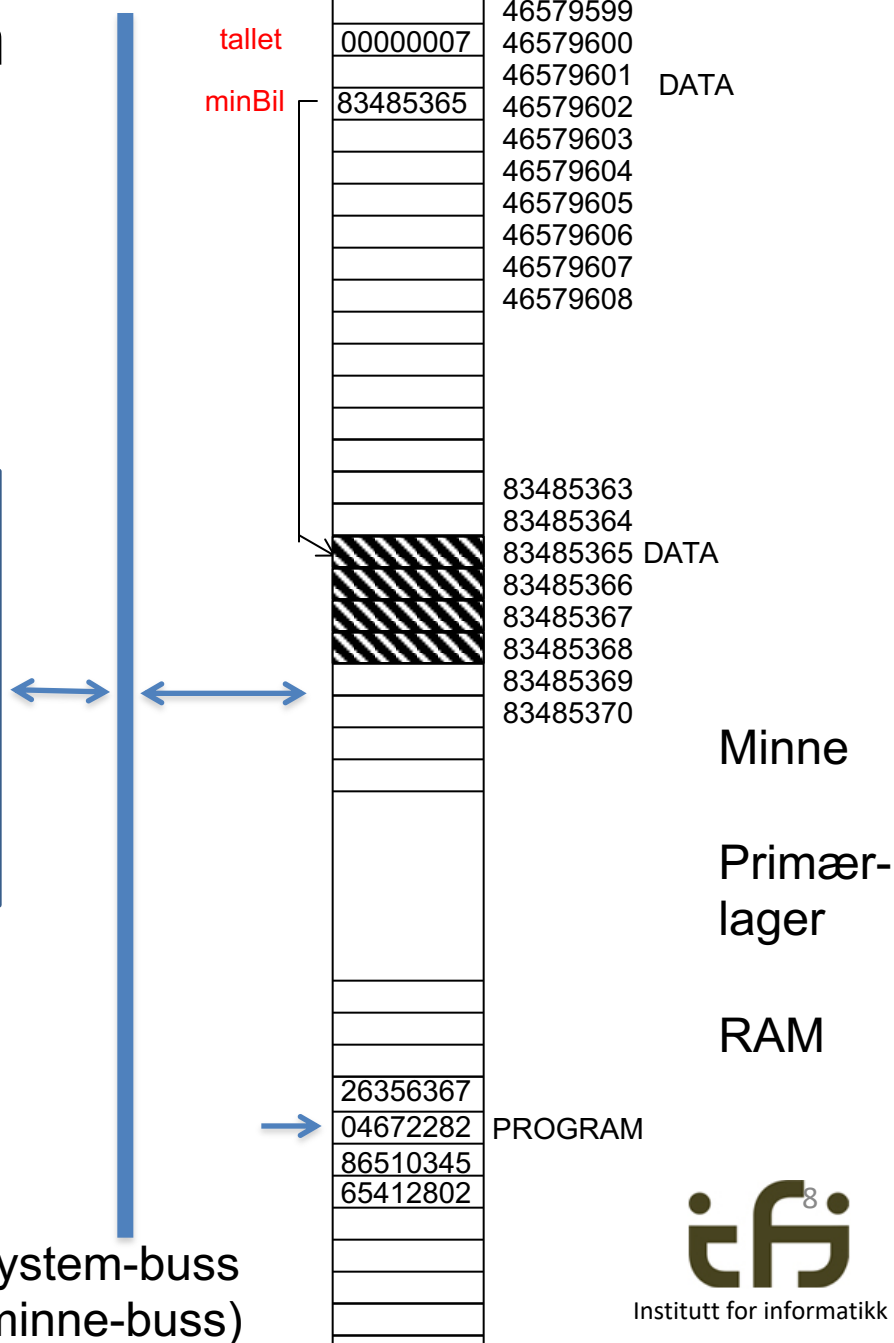
Omid Mirmotahari



# Innmaten i en datamaskin (datamaskinarkitektur)



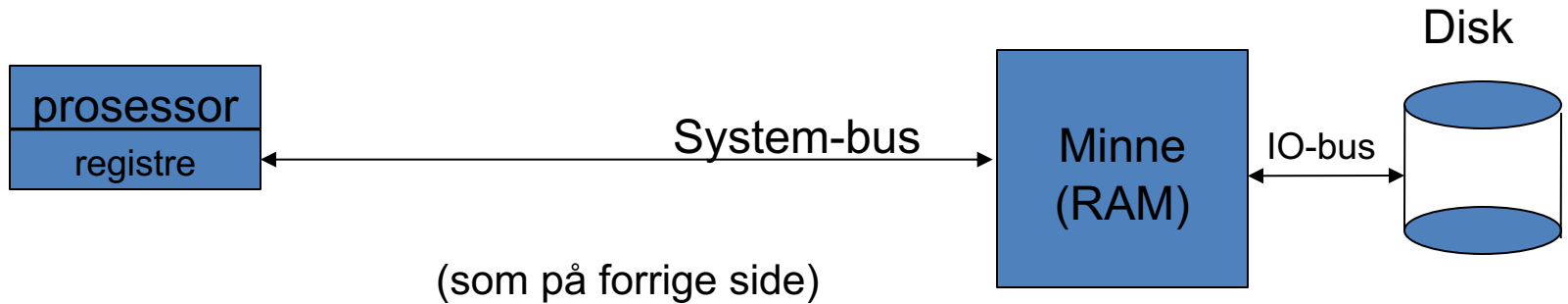
CPU - Pro세서 - Kjerne



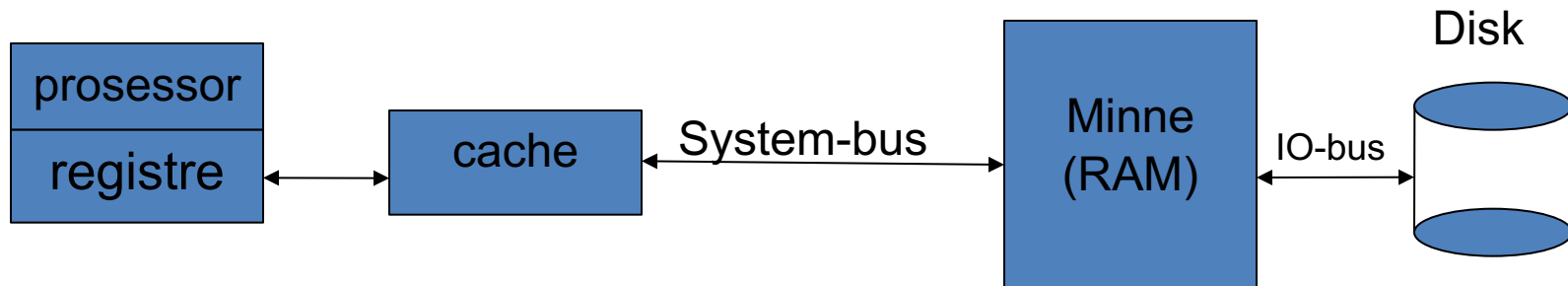


# Datamaskinarkitektur (Computer Architecture)

1. I gamle dager =  
(nesten) fremdeles dagens abstrakte modell:

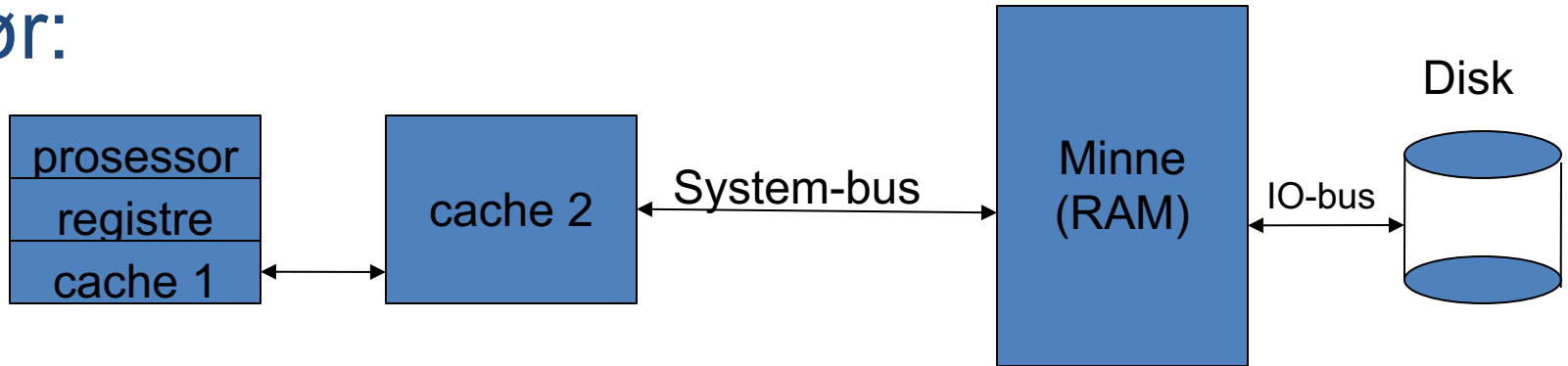


2. Før:

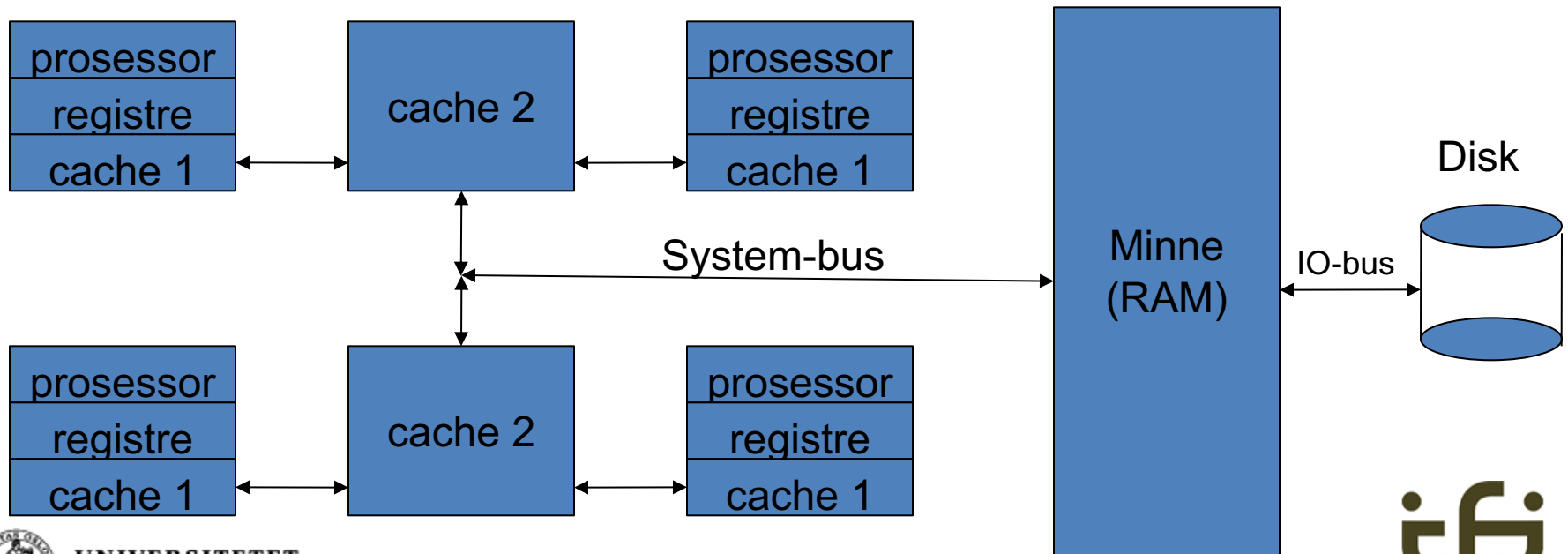


# Datamaskinarkitektur (Computer Architecture)

## 3. Før:

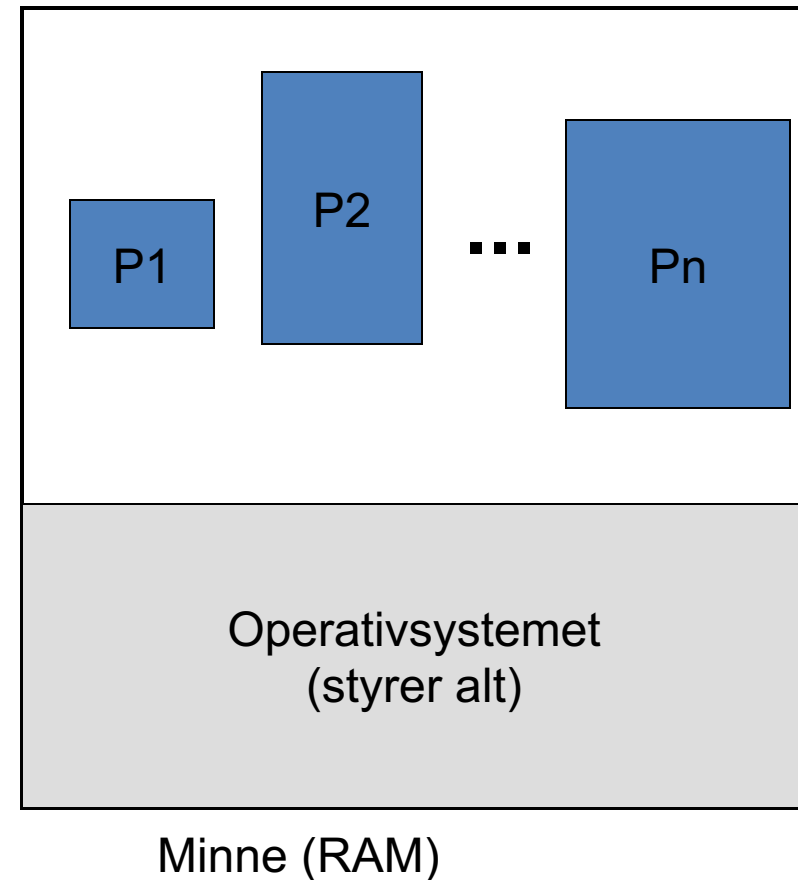


## 4. Nå, f.eks:



# Programmer, prosesser og tråder

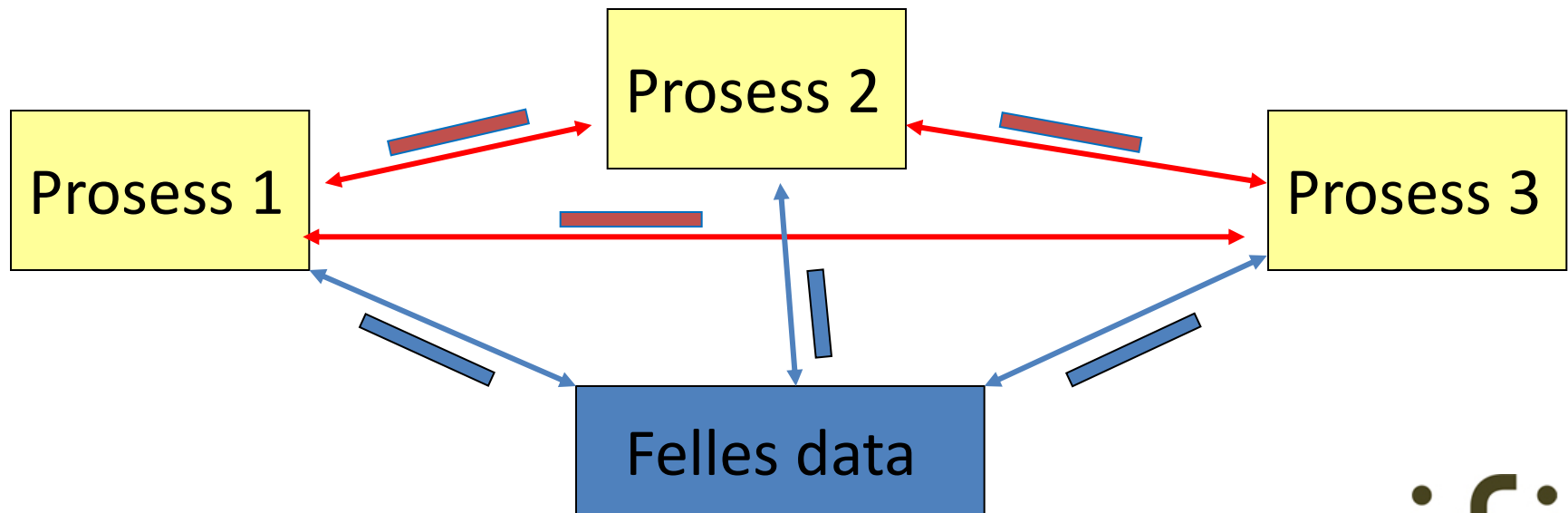
- Operativsystemet administrerer
  - Prosesser
  - (og delvis et antall tråder i hver prosess)
- **Prosesser** (P1, P2, ... , Pn)
  - En prosess er utføringen av et program
  - Er isolert fra hverandre, kan i utgangspunktet bare snakke til operativsystemet
  - Kan sende meldinger til andre prosesser via operativsystemet
  - Eier hver sin del av hukommelsen
  - Eier hver sine filer,...
- Et program
  - Startes som én prosess (kan så evt. starte andre prosesser)
- **En tråd**
  - Er parallelle eksekveringer **inne i én prosess**
  - Alle tråder i en prosess deler prosessens del av hukommelsen (ser de samme variable og programkode)
  - Tråder er som ”små”-prosesser inne i en vanlig ”stor” prosess
  - Tråder kan også gå i ekte parallell



# Parallellprogrammering

Parallellprogrammering vil si å løse en oppgave ved hjelp av programmer (eller programbiter) som utføres samtidig.

**Samarbeidende prosesser** sender meldinger til hverandre (røde piler)\* **eller** leser og skriver i felles primærlager (blå piler) (men vanligvis ikke begge deler).



# Operativsystemets oppgave (Anbefalt kurs: INF3151)

- Operativsystemet velger *hver gang* det kjører, blant de mange prosessene som er klare til å kjøre, hvilke som skal kjøres nå.
- ”Hver gang” betyr:
  - Det kommer noe data utenfra (tastatur, disk, nettverk, . . . )
  - Den innebygde klokka tikker (50 ganger i sekundet)
  - En prosess er (midlertidig) ferdig

Da ’ vekkes ’ operativsystemet opp og overtar kontrollen fra (avbryter) den prosessen/ de prosessene som kjørte. Må huske hva de avbrutte prosessene var i ferd med å gjøre.

- Så bestemmer operativsystemet ’ rettferdig ’ hvilke prosesser som nå skal overta og kjøre litt:
  - De med høyest prioritet må slippe til oftest og mest
  - Ingen må vente ’ alt for lenge ’
  - Dersom ingen prosess ønsker å kjøre, har operativsystemet en egen tomgangsprosess som kan kjøre (går bare rundt i en tom løkke).



# Hvorfor fant man på tråder ?

- Vi har prosesser – hvorfor ikke bare bruke dem?
  - Det går greit, men litt tregt
  - Å skifte fra at en prosess til en annen tar om lag 20 000 instruksjoner
- Prosesser ble funnet på omlag 1960, tråder minst 20 år seinere.
- Tråder er som små prosesser inne i én prosess, og det er langt raskere å skifte fra en tråd til en annen tråd (kalles ofte lettvektsprosesser)
- Prosesser kommuniserer via operativsystemet
- Tråder kommuniserer via felles data (inne i samme prosess)
- Ellers har tråder og prosesser omlag samme muligheter og problemer når man lager programmer
- Parallellprogrammering
  - = bruke **flere** prosesser og/eller **flere** tråder for å løse en programmeringsoppgave
  - Programmer med tråder er mye vanskeligere å skrive og teste/feilsøke enn bare én tråd i én prosess (som vi har gjort til nå)



# Hva bruker vi tråder til?

- Dele opp oppgavene i en prosess i naturlig del-oppgaver
- Alle slags tunge beregninger
  - Kunstig intelligens - maskinlæring - big data
- Vi har et program som skal betjene flere brukere (f.eks. web-sider som Facebook, Google, flybestillinger, . . . ).
- Tolking og visning av video (animasjon) , tolking av tale, biler, . . . .
- Simuleringsmodeller: Etterape prosesser i virkeligheten for å finne ut statistikk/utfallet for f.eks. køer på et motorvei, variasjoner i biologiske systemer (simulere 100 000 laks fra fødsel til død) , datanettverk, hva skjer med et olje eller gassreservoar når oljen/gassen pumpes ut, krigføring (slag med tanks og fly),.... (men kvasiparallellitet er også vanlig (og enklere))
- Med flere prosessorer kan programmer med tråder gå raskere ("Dual/Quad/8/16/ .... -core"-prosessorer).
  - Også mulig med grafiske prosessorer
- Om å programmere med tråder og prosesser:
  - Prøv å finne det som naturlig er parallelt og uavhengig i oppgaven
  - Lag tråder eller prosesser som kommuniserer minst mulig med hverandre.



```

public class Restaurant {
    int antBestilt;
    int antLaget = 0, antServert = 0; // tallerkenrørter
    int antKokker = 5, antServitører = 50;

    Restaurant(int ant) {
        antBestilt = ant;
        for (int i = 0; i < antKokker; i++) {
            Kokk k = new Kokk(this, "Kokk nr. " + i);
            k.start();
        }
        for (int i = 0; i < antServitører; i++) {
            Servitor s = new Servitor(this, "S" + i);
            s.start();
        }
    }

    public static void main(String[] args) {
        new Restaurant(Integer.parseInt(args[0]));
    }

    synchronized boolean kokkFerdig() {
        return antLaget == antBestilt;
    }

    synchronized boolean servitørFerdig() {
        return antServert == antBestilt;
    }

    synchronized boolean putTallerken(Kokk k) {
        // Kokketråden blir eier av låsen.
        while (antLaget - antServert > 2) {
            /* så lenge det er minst 2 tallerkner
            * som ikke er servert, skal kokken vente. */
            try {
                wait(); /* Kokketråden gir fra seg
                * låsen og sover til den
                * blir vekket */
            } catch (InterruptedException e) {}
            // Kokketråden blir igjen eier av låsen
        }

        boolean ferdig = kokkFerdig();
        if (!ferdig) {
            antLaget++;
            System.out.println(k.getName() + " laget nr: " + antLaget);
        }

        notify(); /* Si ifra til servitøren. */
        return !ferdig;
    }

    synchronized boolean getTallerken(Servitor s) {
        // Servitørtråden blir eier av låsen.
        while (antLaget == antServert && !servitørFerdig()) {
            /* så lenge kokken ikke har plassert
            * en ny tallerken. Dermed skal
            * servitøren vente. */
            try {
                wait(); /* Servitørtråden gir fra seg
                * låsen og sover til den
                * blir vekket */
            } catch (InterruptedException e) {}
            // Servitørtråden blir igjen eier av låsen.
        }

        boolean ferdig = servitørFerdig();
        if (!ferdig) {
            antServert++;
            System.out.println(s.getName() + " serverer nr: " + antServert);
        }

        notify(); /* si ifra til kokken */
        return !ferdig;
    }
}

class Kokk extends Thread {
    Restaurant rest;

    Kokk(Restaurant rest, String navn) {
        super(navn); // Denne tråden heter nå <navn>
        this.rest = rest;
    }

    public void run() {
        while (rest.putTallerken(this)) {
            // levert tallerken

            try {
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }
        // Kokken er ferdig
    }
}

class Servitor extends Thread {
    Restaurant rest;

    Servitor(Restaurant rest, String navn) {
        super(navn); // Denne tråden heter nå <navn>
    }

    public void run() {
        while (rest.getTallerken(this)) {
            // hent tallerken

            try {
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }
        // Servitøren er ferdig
    }
}

```

## Tankemodell for tråder

```

public class Restaurant {
    int antBestilt;
    int antLaget = 0, antServert = 0; // tallerkenrørter
    int antKokker = 5, antServitører = 50;

    Restaurant(int ant) {
        antBestilt = ant;
        for (int i = 0; i < antKokker; i++) {
            Kokk k = new Kokk(this, "Kokk nr. " + i);
            k.start();
        }
        for (int i = 0; i < antServitører; i++) {
            Servitor s = new Servitor(this, "Servitør nr. " + i);
            s.start();
        }
    }

    public static void main(String[] args) {
        new Restaurant(Integer.parseInt(args[0]));
    }

    synchronized boolean kokkFerdig() {
        return antLaget == antBestilt;
    }

    synchronized boolean servitørFerdig() {
        return antServert == antBestilt;
    }

    synchronized boolean putTallerken(Kokk k) {
        // Kokketråden blir eier av låsen.
        while (antLaget - antServert > 2) {
            /* så lenge det er minst 2 tallerkner
            * som ikke er servert, skal kokken vente. */
            try {
                wait(); /* Kokketråden gir fra seg
                * låsen og sover til den
                * blir vekket */
            } catch (InterruptedException e) {}
            // Kokketråden blir igjen eier av låsen
        }

        boolean ferdig = kokkFerdig();
        if (!ferdig) {
            antLaget++;
            System.out.println(k.getName() + " laget nr: " + antLaget);
        }

        notify(); /* Si ifra til servitøren. */
        return !ferdig;
    }

    synchronized boolean getTallerken(Servitor s) {
        // Servitørtråden blir eier av låsen.
        while (antLaget == antServert && !servitørFerdig()) {
            /* så lenge kokken ikke har plassert
            * en ny tallerken. Dermed skal
            * servitøren vente. */
            try {
                wait(); /* Servitørtråden gir fra seg
                * låsen og sover til den
                * blir vekket */
            } catch (InterruptedException e) {}
            // Servitørtråden blir igjen eier av låsen.
        }

        boolean ferdig = servitørFerdig();
        if (!ferdig) {
            antServert++;
            System.out.println(s.getName() + " serverer nr: " + antServert);
        }

        notify(); /* si ifra til kokken */
        return !ferdig;
    }
}

class Kokk extends Thread {
    Restaurant rest;

    Kokk(Restaurant rest, String navn) {
        super(navn); // Denne tråden heter nå <navn>
        this.rest = rest;
    }

    public void run() {
        while (rest.putTallerken(this)) {
            // levert tallerken

            try {
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }
        // Kokken er ferdig
    }
}

class Servitor extends Thread {
    Restaurant rest;

    Servitor(Restaurant rest, String navn) {
        super(navn); // Denne tråden heter nå <navn>
    }

    public void run() {
        while (rest.getTallerken(this)) {
            // hent tallerken

            try {
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }
        // Servitøren er ferdig
    }
}

```

## En tråd i programmet

## To tråder i programmet



# Tråder i Java

- Er innebygget i språket
- I ethvert program er det minst én tråd (den som starter og kjører **main**)
- GUI kjører en egen tråd (Event Dispatch Thread, EDT) (etter påske)
- Fra en tråd kan vi starte flere andre, nye tråder
- En tråd starter enten som:
  - I. Lag et objekt av en klasse som også implementerer **interface Runnable** (kap 20.1 i Horstmann)
    - `run()` er eneste metode i grensesnittet
    - Gi dette objektet som parameter til klassen `Thread`, og kall `start()` på dette `Thread`-objektet
      - Litt mer komplisert enn metoden nedenfor.
      - Litt mer fleksibelt (kan da bytte ut objektet som inneholder `run`-metoden)
  - II. Et objekt av en subklasse av **class Thread** som inneholder en polymorf metode **run()** som du skriver i subklassen (Programming tip 20.1). Slik gjør du:
    - Lag først et objekt av en subklasse til klassen `Thread` (`new ...`)
    - Kall så metoden `start()` i objektet (ikke laget av deg, men arvet fra `Thread`). `start()` sørger for at `run()`, som du selv har skrevet, vil bli kalt.
    - `Thread` implementerer grensesnittet `Runnable` (se over)



compact1, compact2, compact3

java.lang

## Interface Runnable

### All Known Subinterfaces:

[RunnableFuture<V>](#), [RunnableScheduledFuture<V>](#)

### All Known Implementing Classes:

[AsyncBoxView.ChildState](#), [ForkJoinWorkerThread](#), [FutureTask](#), [RenderableImageProducer](#), [SwingWorker](#), [Thread](#), [TimerTask](#)

### Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

### @FunctionalInterface

```
public interface Runnable
```

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

In addition, Runnable provides the means for a class to be active while not subclassing Thread. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

### Since:

JDK1.0

### See Also:

[Thread](#), [Callable](#)

## Method Summary

All Methods Instance Methods Abstract Methods

Modifier and Type	Method and Description
void	<b>run()</b> When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

## Method Detail

### run

```
void run()
```

When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

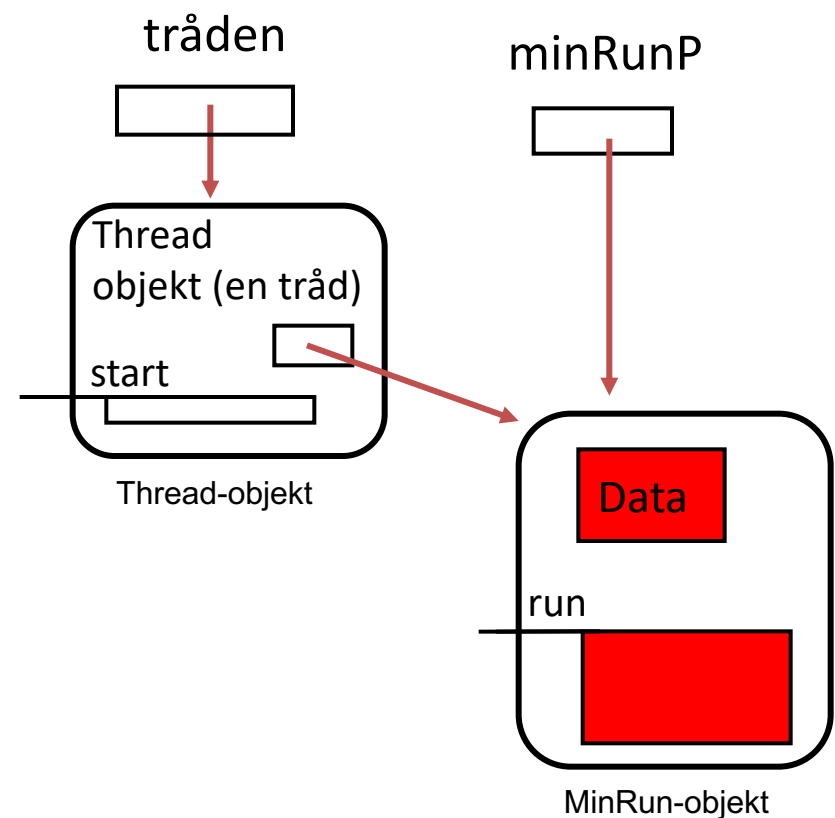
The general contract of the method run is that it may take any action whatsoever.

### See Also:

[Thread.run\(\)](#)

# Tråder i Java I

```
class MinRun implements Runnable {  
    <datastruktur>  
    public void run( ) {  
        while (<mer å gjøre>) {  
            <gjør noe>;  
            ...  
        }  
    }  
}
```



En tråd lages og startes opp slik:

```
Runnable minRunP = new MinRun();  
Thread tråden = new Thread(minRunP);  
tråden.start( );
```

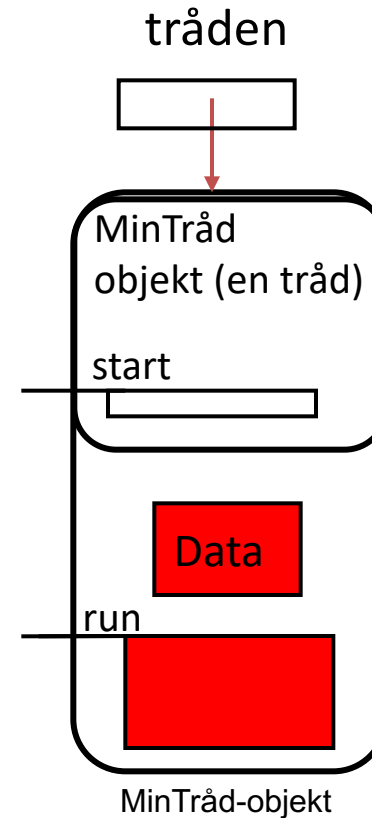


Her går den nye og den gamle tråden (dette programmet), videre hver for seg

start( ) er en metode i Thread som må kalles opp for å få startet tråden. start-metoden vil igjen kalle metoden run (som vi selv programmerer).

# Tråder i Java II

```
class MinTråd extends Thread {  
    <datastruktur>  
    public void run( ) {  
        while (<mer å gjøre>) {  
            <gjør noe>;  
            ...  
        }  
    }  
}
```



En tråd lages og startes opp slik:

```
MinTråd tråden;  
tråden = new MinTråd( );  
tråden.start( );
```



Her går den nye og den gamle tråden (dette programmet), videre hver for seg

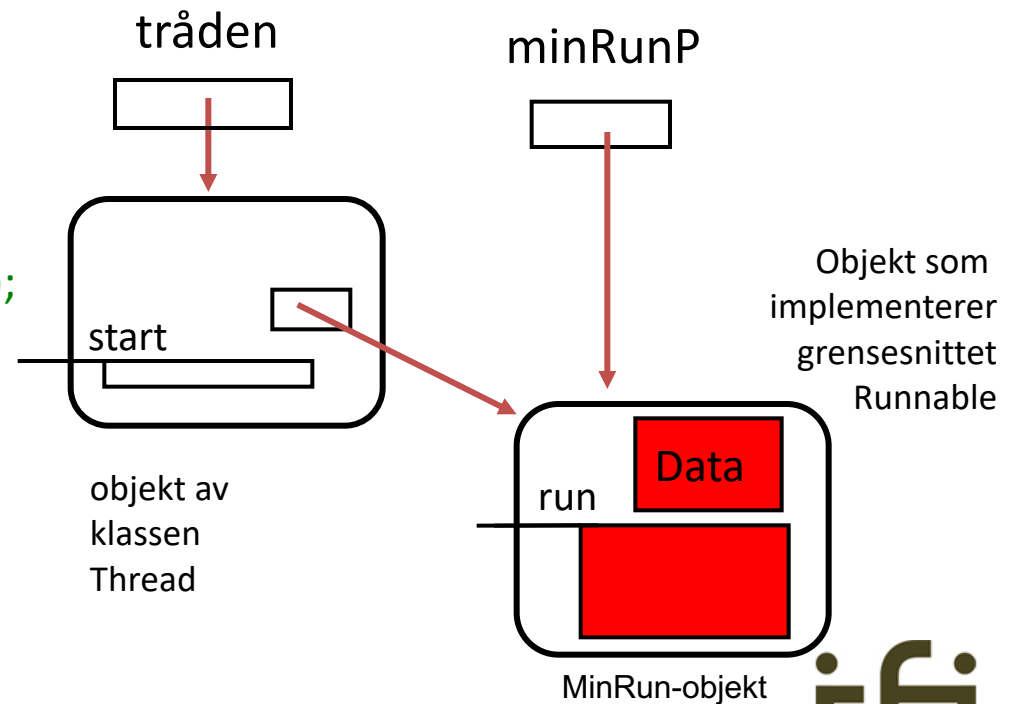
start( ) er en metode i Thread som må kalles opp for å få startet tråden. start-metoden vil igjen kalle metoden run (som vi selv programmerer).

# Det er det vi trenger å vite om tråder

- Nå er resten bare opp til fantasien 😊

(og å passe på at de samarbeider riktig, mer om det senere)

```
Runnable minRunP = new MinRun();  
Thread tråden = new Thread(minRunP);  
tråden.start( );
```



Delegering



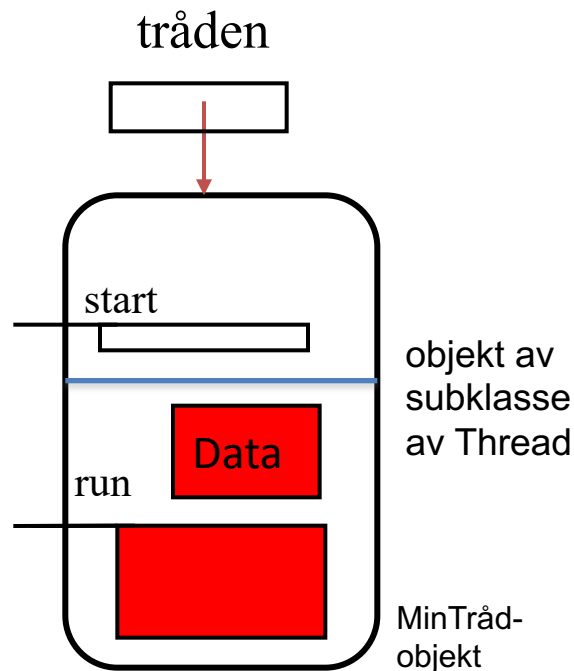
# Det er det vi trenger å vite om tråder

- Nå er resten bare opp til fantasien



(og å passe på at de samarbeider riktig, mer om det senere)

```
MinTråd tråden;  
tråden = new MinTråd( );  
tråden.start( );
```



# Sove og våkne

- En tråd kan sove et antall milli- (og nano) sekunder; metode i Thread:
  - static void `sleep` (long millis)  
"Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds."
    - Andre tråder slipper da til (får kjøre)
    - Hvis noen avbryter tråden mens den sover skjer et unntak
      - Inne i Runnable:

```
try { Thread.sleep(1000); } // sover ett sekund  
catch (InterruptedException e) { behandle stoppet-avbrudd }
```
      - Inne i et objekt av klassen Thread:

```
try { sleep(1000); } // sover ett sekund  
catch (InterruptedException e) { behandle stoppet-avbrudd }
```



# Om å stoppe en tråd

- Kall metoden
  - `interrupt();` i den tråden du ønsker å stoppe
- Hvis tråden ligger og venter / sover, vil den våkne opp av et `InterruptedException`
  - NB! Som resetter stoppemodus til "ikke stoppet"
- En tråd kan teste om den er stoppet ved å kalle på
  - `interrupted()`
  - da blir også stoppemodus resatt.





```
import java.util.Scanner;
```

```
class Klokke {  
    public static void main(String[] args) throws IOException {  
        System.out.println("Trykk s <retur> for å starte/stoppe");  
        Scanner tastatur= new Scanner (System.in);  
        tastatur.next();  
  
        // Her lages stoppeklokke-objektet:  
        Stoppeklokke stoppeklokke = new Stoppeklokke();  
        Thread mintråd = new Thread(stoppeklokke);  
        // og her settes den nye tråden i gang:  
        mintråd.start();  
  
        tastatur.next();  
        mintråd.interrupt();  
    }  
}
```



```
class Stoppeklokke implements Runnable {  
    // run blir kalt opp av start-metoden.
```

```
    public void run() {  
        int tid = 0;  
        try {  
            while (! Thread.interrupted()) {  
                System.out.println(tid++);  
                Thread.sleep(1 * 1000); // ett sekund  
            }  
        }  
        catch (InterruptedException e) { }  
    }  
}
```

```
}
```

# Avbrytbare eller ikke-avbrytbare prosesser og tråder.

- Hvis en **prosess** (eller en tråd) prøver å gjøre en så lang beregning at andre prosesser (tråder) ikke slipper til, må den kanskje avbrytes.
- Alle skikkelige operativsystemer (Windows, MAC OS og Unix/linux) greier å avbryte både prosesser og tråder – f.eks. vha. klokka som sender avbruddssignal 50 ganger per sek.
- Gamle/enkle operativsystemer greide ikke det
  - Og fremdeles ikke enkle mobiltelefoner/lesebrett i dag
- Java-definisjonen sier at det er opp til hver implementasjon av kjøresystemet 'java' om tråder skal være avbrytbare eller ikke.
- Selv om vi har ”mange-core” prosessorer vil en tråd ikke gjøre fornuftig arbeid hele tide. Vi bør ha 2 – 8 ganger så mange tråder som prosessorer (”core-er”), avhengig av hvor mye hver tråd trenger å vente (på andre tråder eller I/O)



# Tråder kan avbryte seg selv

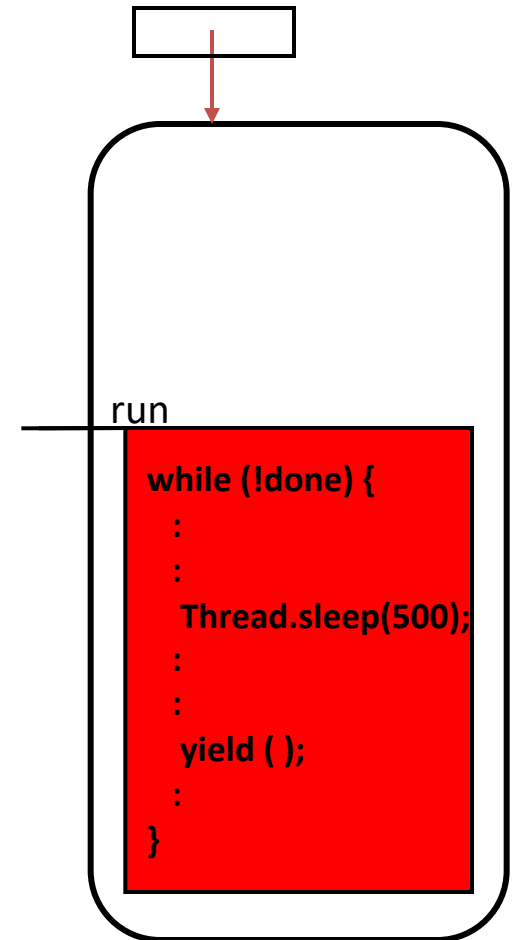
(ikke veldig viktig)

Vanligvis vil operativsystemet dele prosessoren mellom alle aktive tråder (tidsdeling - “time-slicing”), og altså kaste en tråd ut av prosessoren etter en tid (på engelsk: pre-emption).

run-metoden kan eksplisitt la andre tråder slippe til ved å si `yield( )`;

Noen ganger det kan være ønskelig å la andre aktiviteter komme foran.

`yield( )`; slipper andre tråder til, og lar tråden som utfører `yield` vente midlertidig (men denne tråden blir igjen startet opp når det på ny er den sin tur).



Sove og vike

# NYTT OG VIKTIG: Oppdateringsproblemet:

Om å passe på at tråder samarbeider riktig  
Horstmann kap 20.3

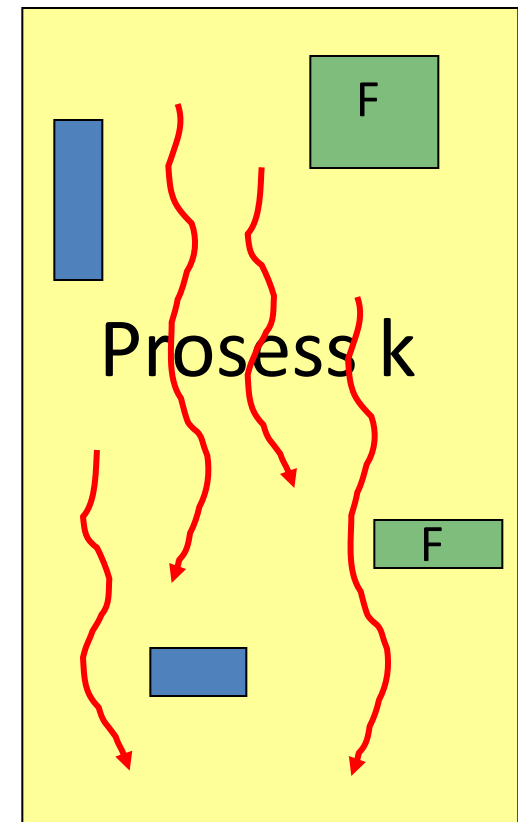
- **Samtidig oppdatering av felles data**
- Løsningen: Kritiske regioner
- Først: Mer intro og aller enkleste eksempel
- Så et større program (Kokk og servitør)



# Felles data

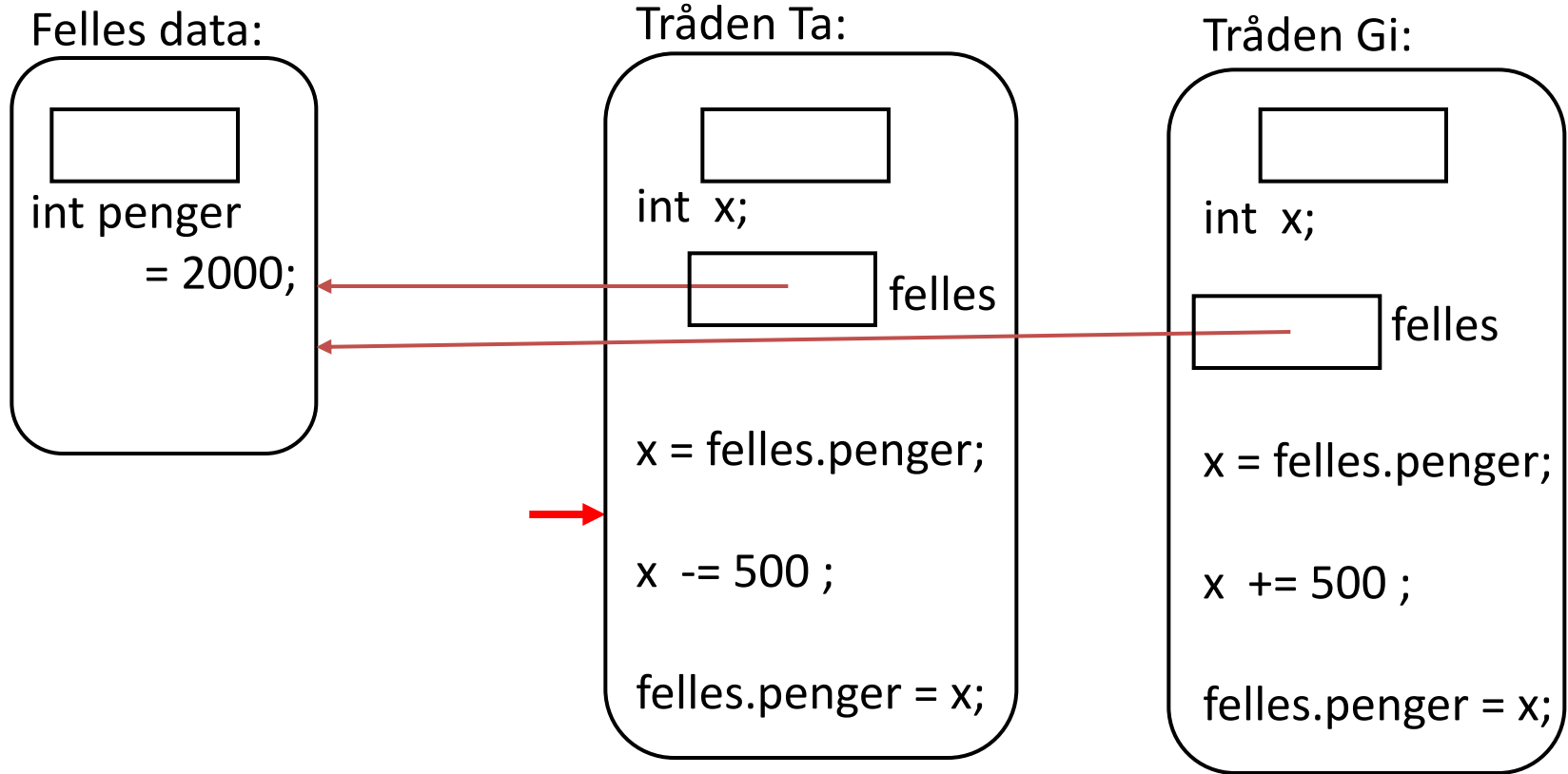
Felles data (grønne felt) må vanligvis bare aksesseres (lese og skrives i) av en tråd om gangen. Hvis ikke blir det kluss i dataene.

På figuren vår er det to områder vi har problemer med (dvs. at to eller flere tråder kan risikere å manipulere data i disse områdene samtidig). Disse to områdene er markert med F (for Felles). Et slikt område (et objekt) kalles en monitor.



De andre to områdene inneholder data som vi enten vet bare kan leses ("immutable" data), eller vi vet at bare en tråd om gangen skriver i disse dataene. Slikt resonement er imidlertid farlige og ofte feilaktig. Hvis du er det minste i tvil så beskytte alle delte data som om de kan bli oppdatert samtidig.

# Enkelt eksempel på at to tråder kan ødelegge felles data.



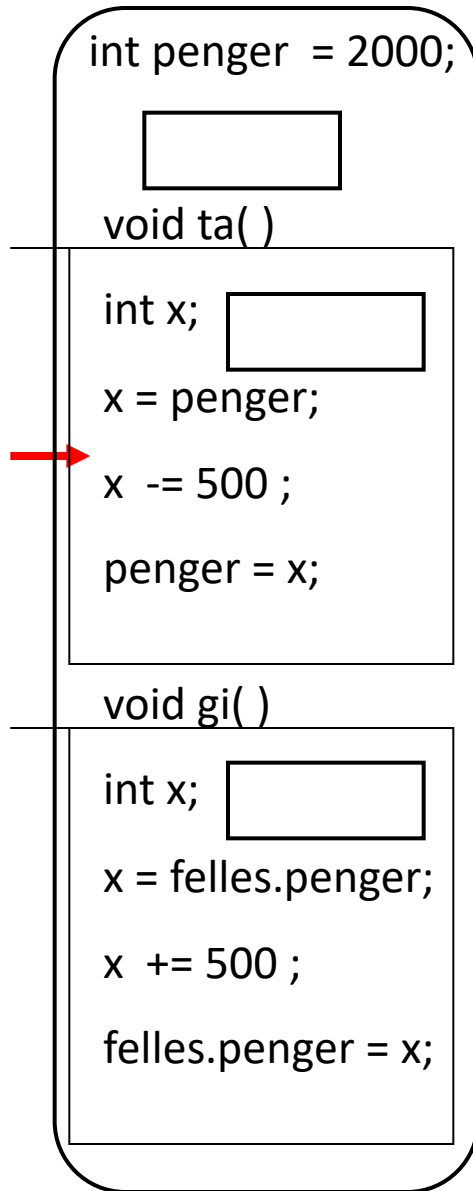
La oss se hva som skjer hvis tråden Ta først utføres litt, og stopper opp ved pilen. Deretter overtar tråden Gi, og hele denne tråden utføres ferdig. Til slutt utføres resten av tråden Ta.



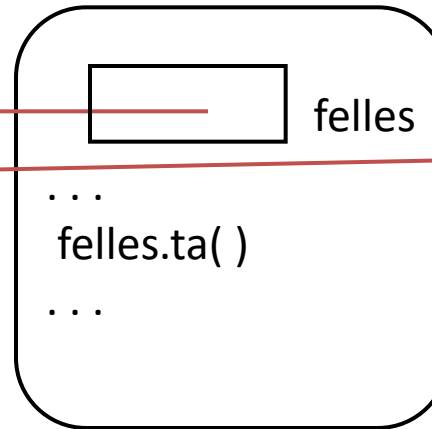
Hva med ekte parallellitet ?

# Samme – men objektorientert

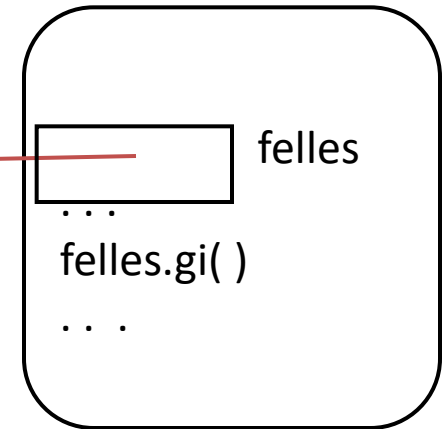
Felles data:



Tråden Ta:



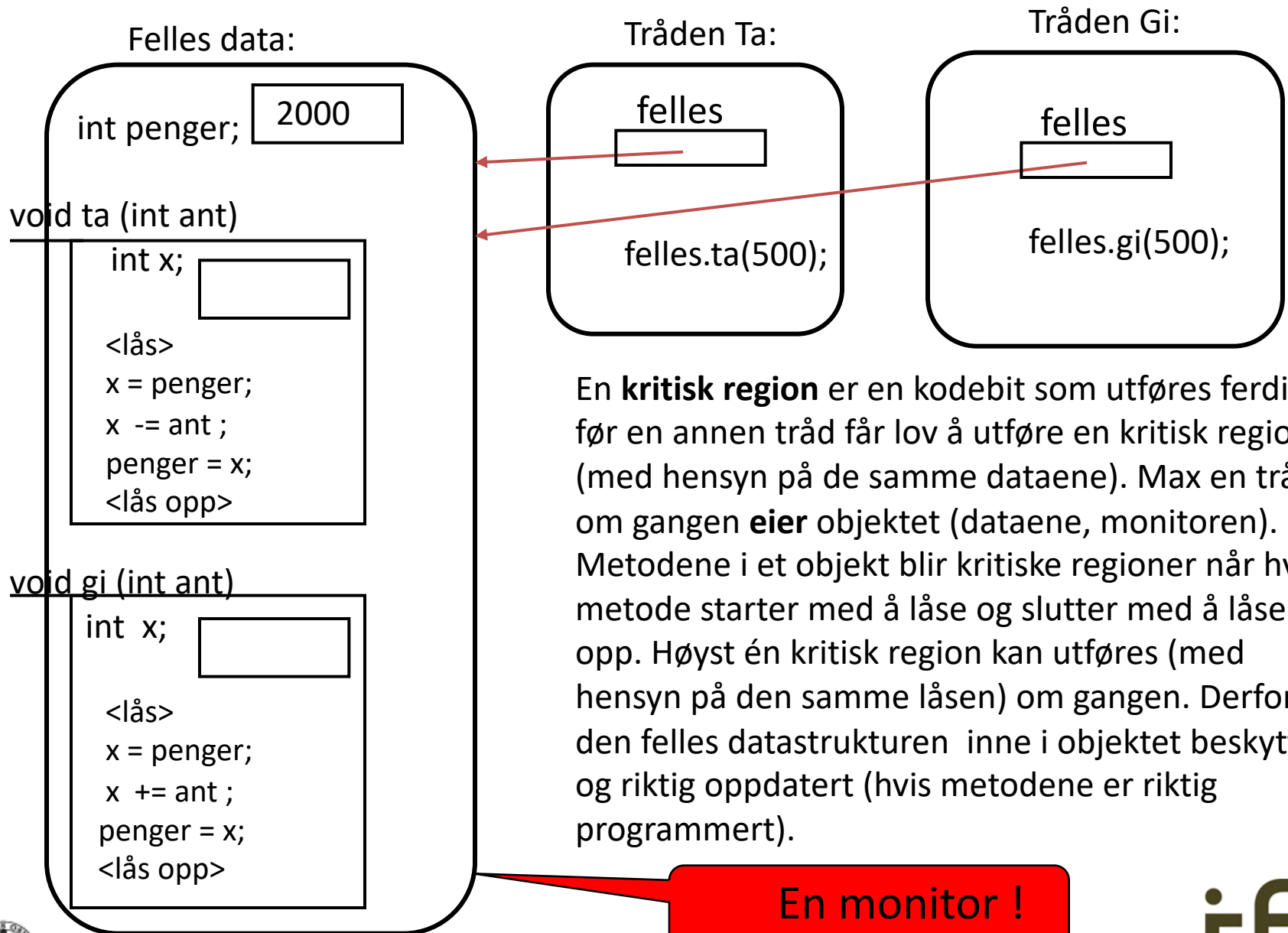
Tråden Gi:



Trådene utfører metodene `ta` og `gi` som om koden inne i metodene var en del av trådenes kode. På samme måte som på forrige side kan vi se hva som skjer hvis tråden Ta først utfører metoden `ta` litt, og så stopper opp ved pilen. Deretter overtar tråden Gi, og hele denne tråden utføres ferdig (og utfører hele metoden `gi`). Til slutt utføres resten av tråden Ta (metoden `ta`).



# Vi ordner dette med **kritiske regioner**.





# Mer om kritiske regioner / monitorer

- Alle tråder har felles adresserom
- Hvis flere tråder forsøker samtidig å
  - først lese en variabel ' a '
  - så oppdatere (endre) ' a ' basert på den verdien den leste, kan det gå galt (jfr. eksemplet side 30-31)
  - FORDI:
    - En tråd X kan først lese verdien av ' a ', og så bli avbrutt.
    - Så kan andre tråder Y, Z komme inn og endre ' a '
    - Når X igjen får kjøre, vil den oppdatere ' a ' ut fra ' a ' s gamle verdi, og ikke det den nå er
- Vi må beskytte slik lesing og etterfølgende skriving av samme data i metoder som blir **kritisk regioner** i et objekt som vi kaller **en monitor**
- Til dette kan vi bruke en LÅS
  - Høyst en tråd er inne i noen av de låste metodene (låst med samme lås i samme objekt) samtidig.
  - En tråd får slippe til, de andre trådene må vente
  - Når den ene er ferdig, slipper de som venter til (en etter en)
  - Alle data i objektet blir skrevet skikkelig ned i variablene i primærlageret (RAM)

Monitor  
er ikke  
noe ord  
i Java



33



# Slik programmeres dette i Java

```
Lock laas = new ReentrantLock();
```

```
Int penger; 2000
```

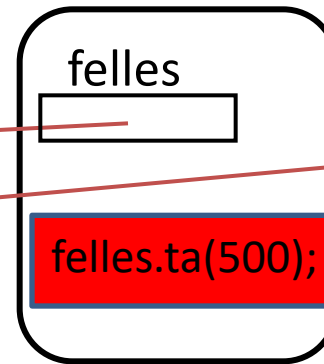
```
public void gi ( int verdi ) throws InterruptedException
```

```
    laas.lock();  
    try {  
        penger = penger + verdi;  
    } finally {  
        laas.unlock()  
    }  
}
```

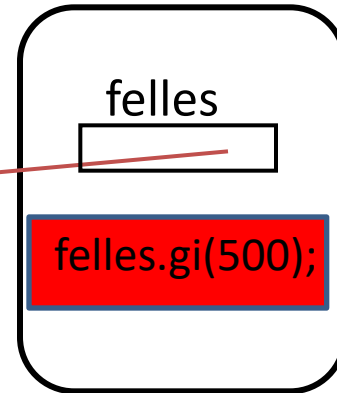
```
public void ta ( int verdi ) throws InterruptedException
```

```
    laas.lock();  
    try {  
        penger = penger - verdi;  
    } finally {  
        laas.unlock()  
    }  
}
```

Tråden Ta:



Tråden Gi:



En lås (laas) slik at bare en tråd  
kommer inn i monitoren om gangen

```
(import java.concurrent.locks.*)
```

# Legg merke til bruken av **finally**

```
void putInn (int verdi) throws InterruptedException {  
    laas.lock();  
    try {  
        :  
        :  
    } finally {  
        laas.unlock();  
    }  
}
```

Da blir laas.unlock() **alltid** utført !!



# Men hva om det ikke er penger igjen på konto?

```
Lock laas = new ReentrantLock();
```

```
int penger; 2000
```

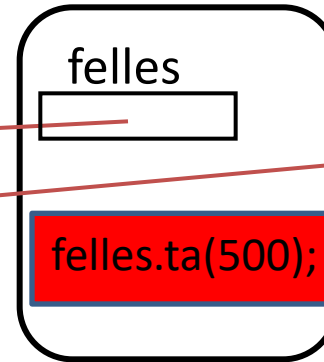
```
public void gi ( int verdi ) throws InterruptedException
```

```
    laas.lock();  
    try {  
        penger = penger + verdi;  
    } finally {  
        laas.unlock();  
    }  
}
```

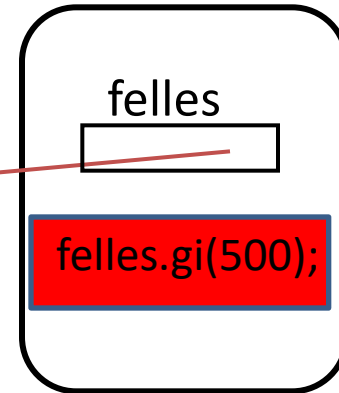
```
public void ta ( int verdi ) throws InterruptedException
```

```
    laas.lock();  
    try { if (penger >= verdi)  
        penger = penger - verdi;  
    } finally {  
        laas.unlock(;  
    }  
}
```

Tråden Ta:



Tråden Gi:



Et litt annet scenario:

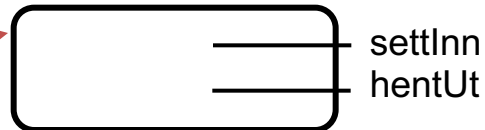
# En monitor som er/beskytter en beholder

```
Lock laas = new ReentrantLock();
```

```
Beholder behold;
```



Objekt av klassen Beholder



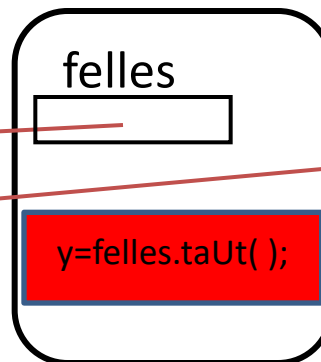
```
public void settInn ( Ting tingen ) throws InterruptedException
```

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

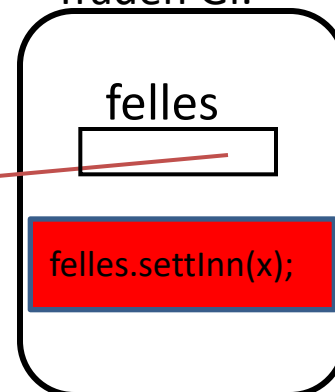
```
public Ting taUt ( ) throws InterruptedException
```

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

Tråden Ta:



Tråden Gi:



Beholderen kan også skrives direkte inne i monitoren



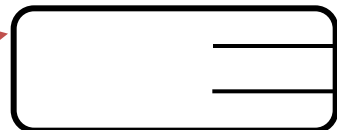
# Prøver å ta ut, men beholderen er tom

```
Lock laas = new ReentrantLock();
```

```
Beholder behold;
```



Objekt av klassen Beholder



settInn  
hentUt

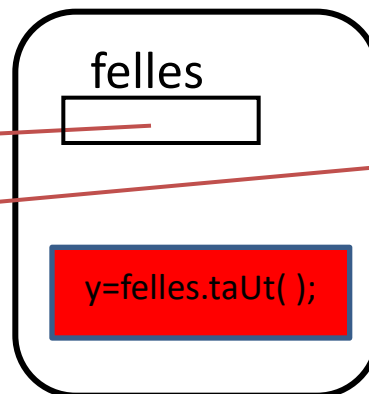
```
public void settInn ( Ting tingen ) throws InterruptedException
```

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

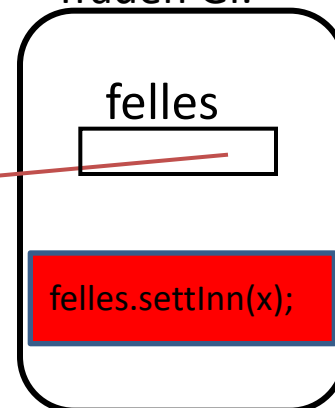
```
public Ting taUt ( ) throws InterruptedException
```

```
    laas.lock();  
    try { if (!beholder.tom)  
        {return (behold.hentUt());}  
        else return null;  
    } finally {  
        laas.unlock();  
    }
```

Tråden Ta:



Tråden Gi:



- Men kanskje det at beholderen er tom er veldig midlertid
- Kanskje kan vi vente på et en annen tråd legger inn noe i beholderen



# Aktiv venting

```
Lock laas = new ReentrantLock();
```

```
Beholder behold;
```



Objekt av klassen Beholder



settInn  
hentUt

```
public void settInn ( Ting tingen ) throws InterruptedException
```

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

```
public Ting taUt ( ) throws InterruptedException
```

```
    laas.lock();  
    try { if (!beholder.tom)  
        {return (behold.hentUt());}  
        else {return null;}  
    } finally {  
        laas.unlock()  
    }
```

Tråden Ta:

felles



```
y= null;  
while (y == null)  
    { y=felles.taUt( ); }  
<bruk y>
```

Tråden Gi:

felles



```
felles.settInn(x);
```





# Passiv venting er bedre enn aktiv venting

- Aktiv venting er vanligvis ikke smart
- I verste fall tar det opp ressurser som kunne vært brukt til noe nyttigere
  - CPU-kraft
  - Monitoren blokeres hver eneste gang tråden går inn i monitoren for å teste



# Passiv venting

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

```
Beholder behold; 
```

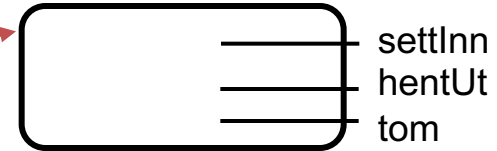
```
public void settInn ( Ting tingen ) throws InterruptedException
```

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

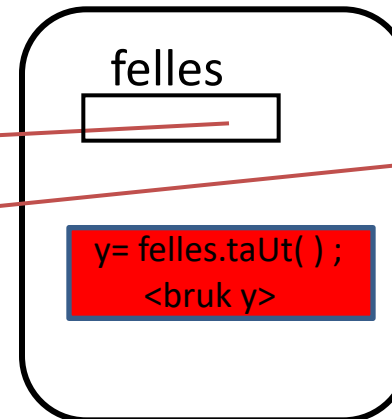
```
public Ting taUt ( ) throws InterruptedException
```

```
    laas.lock();  
    try {  
        if (behold.tom())  
            { ikkeTom.await(); }  
        return (behold.hentUt());  
    } finally {  
        laas.unlock()  
    }  
}
```

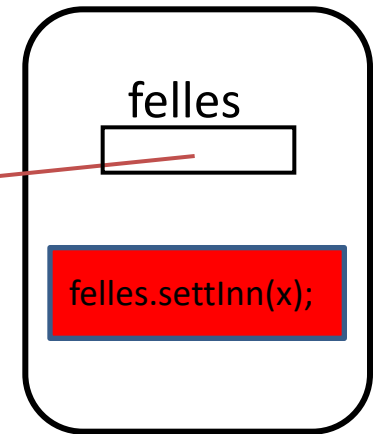
Objekt av klassen Beholder



Tråden Ta:



Tråden Gi:



Men hvem skal vekke opp tråden som venter ?



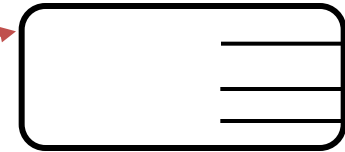
# Passiv venting

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

```
Beholder behold;
```



Objekt av klassen Beholder



settInn  
hentUt  
tom

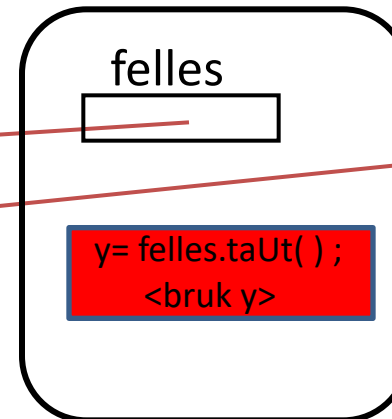
```
public void settInn ( Ting tingen ) throws InterruptedException
```

```
laas.lock();  
try {  
    behold.settInn(tingen);  
    ikkeTom.signal();  
} finally {  
    laas.unlock()  
}
```

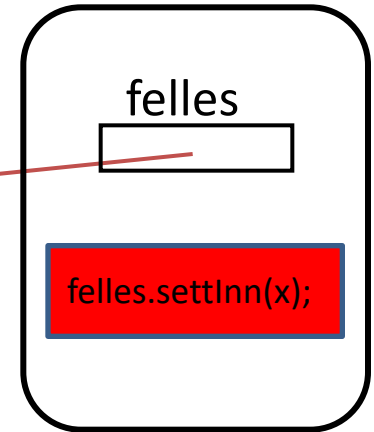
```
public Ting taUt ( ) throws InterruptedException
```

```
laas.lock();  
try {  
    if (behold.tom())  
        { ikkeTom.await(); }  
    return (behold.hentUt());  
} finally {  
    laas.unlock()  
}
```

Tråden Ta:



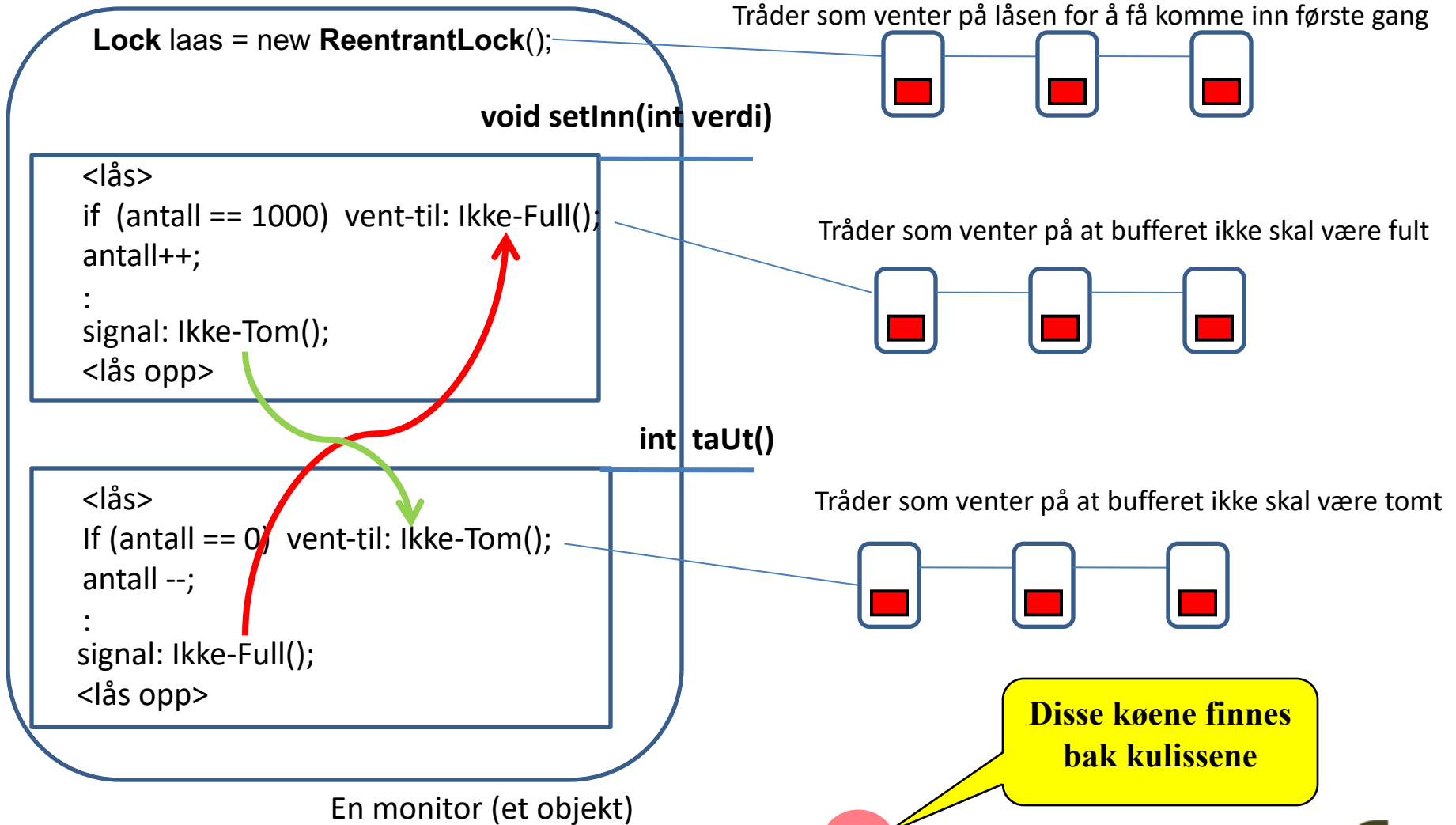
Tråden Gi:



**Når noe settes inn  
vil bufferen  
ikke lenger være tom**



# Ofte har vi flere grunner til å vente i en monitor:



# settlInn med Java-kode

Beholder behold;

```
Lock laas = new ReentrantLock();
```

```
Condition ikkeFull = laas.newCondition();
```

```
Condition ikkeTom = laas.newCondition();
```

```
void settInn ( Ting tingen) throws InterruptedException
```

```
laas.lock();
```

```
try {
```

```
    while (full) { ikkeFull.await(); }
```

```
    // nå er det helst sikkert ikke fullt
```

```
    behold.settlInn(tingen);
```

```
    // det er lagt inn noe, så det er
```

```
    // helt sikkert ikke tomt:
```

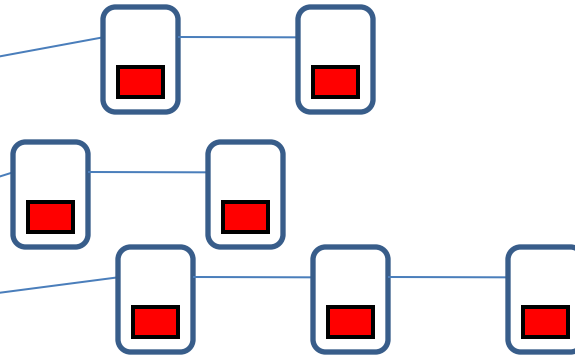
```
    ikkeTom.signalAll();
```

```
} finally {
```

```
    laas.unlock()
```

```
}
```

```
Ting taUt ( )
```



En kø for selve låsen  
og en kø for hver  
betingelse (for hver  
condition-variabel)

```
(import java.concurrent.locks.*)
```

# Java-biblioteket `java.concurrent`

## Interface Condition

- void [await\(\)](#)  
Causes the current thread to wait on this Condition until it is signalled (or [interrupted](#))
- void [signal\(\)](#)  
Wakes up one waiting thread (on this Condition).
- void [signalAll\(\)](#)  
Wakes up all the waiting threads (on this Condition)
- Class `ReentrantLock` er en lås og en fabrikk som lager objekter som implementerer grensesnittet `Condition` (på denne låsen)



Beholder behold;

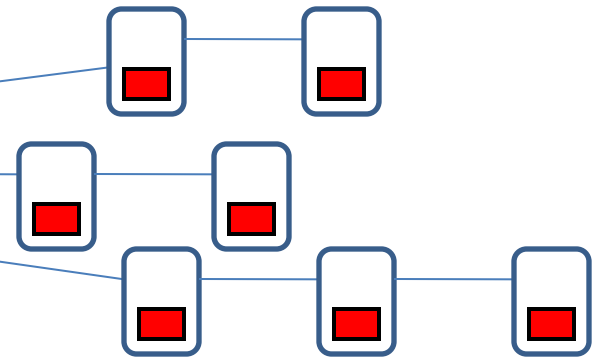
```
Lock laas = new ReentrantLock();  
Condition ikkeFull = laas.newCondition();  
Condition ikkeTom = laas.newCondition();
```

```
void settInn ( Ting tingen) throws InterruptedException
```

```
laas.lock();  
try {  
    while (full) ikkeFull.await();  
    // nå er det helst sikkert ikke fullt  
    behold.settInn(tingen);  
    // det er lagt inn noe, så det er helt sikkert ikke tomt:  
    ikkeTom.signalAll();  
} finally {  
    laas.unlock();  
}
```

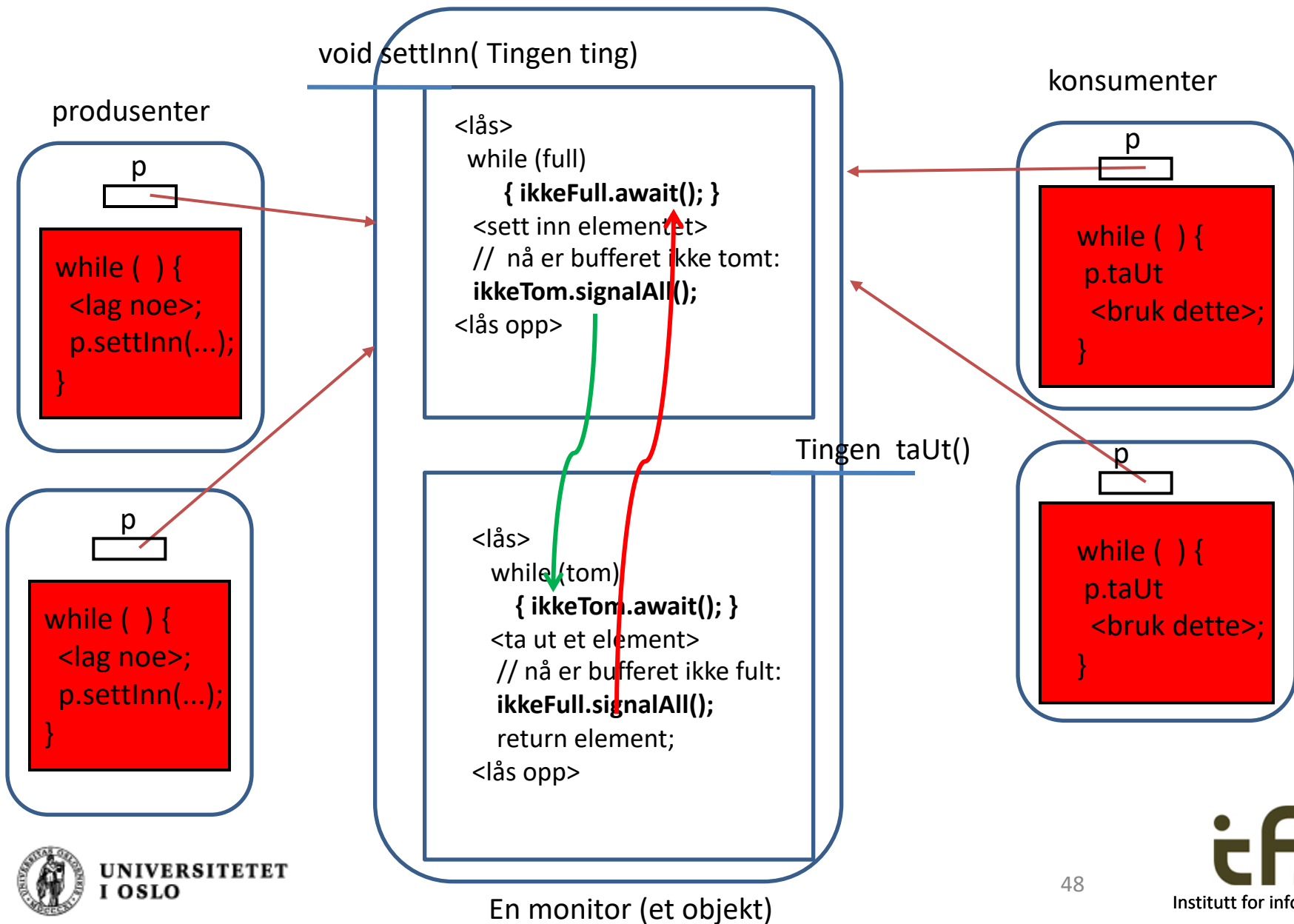
```
Tingen taUt ( ) throws InterruptedException
```

```
laas.lock();  
try {  
    while (tom) ikkeTom.await();  
    // nå er det helst sikkert ikke tomt;  
    Tingen tmp = beholder.hentUt();  
    // det er det tatt ut noe, så det er helt sikkert ikke fullt:  
    ikkeFull.signalAll();  
    return (tmp);  
} finally {  
    laas.unlock();  
}
```



# Med begge metodene skrevet ut

# Produsenter og konsumenter





# Et større litt større eksempel – kokk og servitør

- Kokken lager mat og setter en og en tallerken på et bord
- Servitøren tar en og en tallerken fra bordet og serverer
- Kokken må ikke sette mer enn BORD\_KAPASITET tallerkener på bordet (bordet er fullt og maten blir kald)
- Servitøren kan selvsagt ikke servere mat som ikke er laget (bordet er tomt)
- Her: en kokk og en servitør – Oppgave: lag flere av hver.

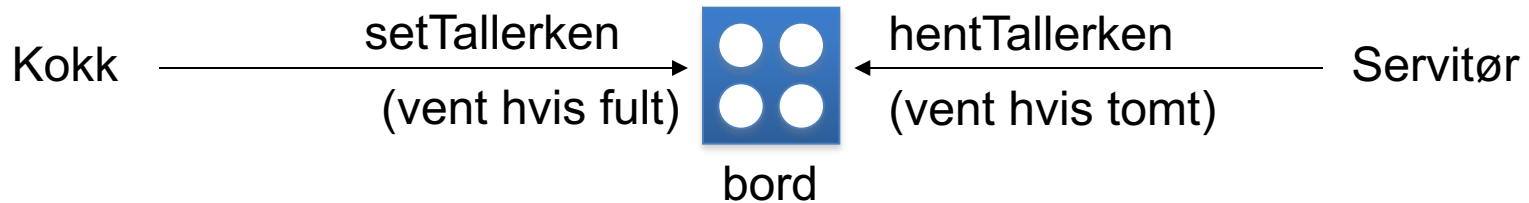


```
Terminal — bash ...
Antall paa bordet: 4
Kokken lager tallerken nr: 44
Servitør serverer nr:40
Antall paa bordet: 4
Kokken lager tallerken nr: 45
Servitør serverer nr:41
Antall paa bordet: 4
Kokken lager tallerken nr: 46
Servitør serverer nr:42
Antall paa bordet: 4
Kokken lager tallerken nr: 47
Servitør serverer nr:43
Antall paa bordet: 4
Kokken lager tallerken nr: 48
Servitør serverer nr:44
Antall paa bordet: 4
Kokken lager tallerken nr: 49
Servitør serverer nr:45
Antall paa bordet: 4
Kokken lager tallerken nr: 50
Servitør serverer nr:46
Antall paa bordet: 4
Servitør serverer nr:47
Servitør serverer nr:48
Servitør serverer nr:49
Servitør serverer nr:50
kubix:Trader-1 steing$
```



# await(); signal();

- Kokken må vente når det allerede er fire tallerkener på bordet
- Servitøren må vente når det ikke er laget noe mat (ingen tallerkener på bordet)
- Kokken må starte opp kelneren igjen når han har satt tallerken nr. 1 på bordet (eller alltid når han har satt en tallerken på bordet ?)
- Servitøren må starte opp kokken igjen når han tar tallerken nr. 4 fra bordet (eller alltid når han tar en tallerken fra bordet ?)



Kokk:

Servitør:

```
ikkeFulltBord.await();  
ikkeTomtBord.signal();
```

```
ikkeTomtBord.await();  
ikkeFulltBord.signal();
```



Tror du at programmet nedenfor er feil ?

Hvis ja: Påvis en kjøring som gir en feilsituasjon

Tror du dette programmet er riktig ?

Hvis ja: Begrunn hvorfor det er riktig ?

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class RestaurantC {
    private Lock bordlas;
    private Condition ikkeTomtBord;
    private Condition ikkeFulltBord;

    RestaurantC(String[] args) {
        bordlas = new ReentrantLock();
        ikkeTomtBord = bordlas.newCondition();
        ikkeFulltBord = bordlas.newCondition();

        int antall = Integer.parseInt(args[0]);

        FellesBord bord = new FellesBord();

        Kokk kokk = new Kokk(bord, antall);
        new Thread(kokk).start();

        Servitor servitor = new Servitor(bord, antall);
        new Thread(servitor).start();
    }

    public static void main(String[] args) {
        new RestaurantC(args);
    }
}
```

```

class FellesBord { // En monitor

    private final int BORD_KAPASITET = 4;
    private int antallPaBordet = 0;
    /* Invariant: 0 <= antallPaBordet <= BORD_KAPASITET */

    void settTallerken() throws InterruptedException {
        bordlas.lock();
        try {
            while (antallPaBordet >= BORD_KAPASITET) {
                /* sa lenge det er BORD_KAPASITET tallerkner
                pa bordet er det ikke lov a sette pa flere. */
                ikkeFulltBord.await();
            } // Na er antallPaBordet < BORD_KAPASITET
            antallPaBordet++;
            System.out.println("Antall paa bordet: " + antallPaBordet);
            ikkeTomtBord.signal(); /* Si fra til kelneren. */
        }
        finally { bordlas.unlock(); }
    }

    void hentTallerken() throws InterruptedException {
        bordlas.lock();
        try {
            while (antallPaBordet == 0) {
                /* Sa lenge det ikke er noen tallerkner pa
                bordet er det ikke lov a ta en */
                ikkeTomtBord.await();
            } // Na er antallPaBordet > 0
            antallPaBordet--;
            ikkeFulltBord.signal(); /* si fra til kokken. */
        }
        finally { bordlas.unlock(); }
    }
}

```

```

class Kokk implements Runnable {
    private FellesBord bord;
    private final int ANTALL;
    private int laget = 0;
    Kokk(FellesBord bord, int ant) {
        this.bord = bord;
        ANTALL = ant;
    }
    public void run() {
        try {
            while(ANTALL != laget) {
                laget ++;
                System.out.println("Kokken lager tallerken nr: " + laget);
                bord.settTallerken(); // lag og lever tallerken
                Thread.sleep((long) (500 * Math.random()));
            }
        } catch (InterruptedException e) {System.out.println("Stopp 1");}
    }
}

```

```

class Servitor implements Runnable {
    private FellesBord bord;
    private final int ANTALL;
    private int servert = 0;
    Servitor(FellesBord bord, int ant) {
        this.bord = bord;
        ANTALL = ant;
    }
    public void run() {
        try {
            while (ANTALL != servert) {
                bord.hentTallerken(); /* hent tallerken og server */
                servert++;
                System.out.println("Kelner serverer nr:" + servert);
                Thread.sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) { System.out.println("Stopp 2");}
    }
}

```

# Enda mer om monitorer, kritiske regioner og Big Java

- Kritiske regioner (eksklusiv tilgang til felles data) kan løses på mange måter
- Det er bl. a. noe som heter *semaforer* (P og V)
  - og aktiv venting (spin locks), OK ved meget kort venting
- I IN1010 bruker vi *monitorer* fordi det er mest objektorientert.
- Venting og signalering i monitorer kan gjøres på forskjellige måter.
  - I dag: En god måte, og slik det gjøres i Horstmann
  - Kanskje senere i vår: Slik det gjøres innebygget i Java (notify og wait, ikke pensum i IN1010)



# Oppsummering

## Vi har lært:

- Hva tråder brukes til
- Hvordan vi lager tråder i Java
- Hvordan tråder kommuniserer seg imellom ved hjelp av monitorer
- Hvordan programmet venter i en monitor – og starter opp igjen de som venter

