IN1010 våren 2025
Parallellitet og tråder – del 2
27. mars 2025

og litt om innlevering 5

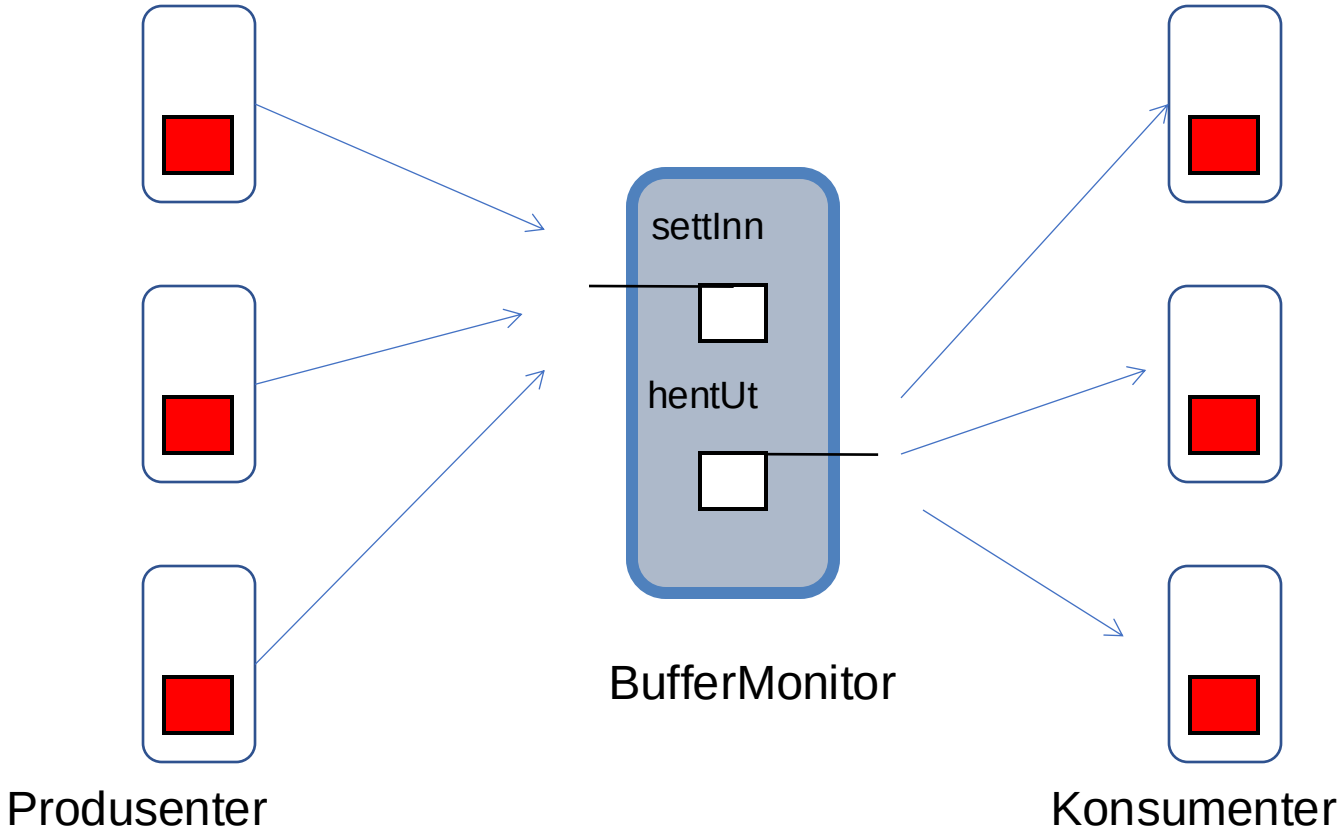
Stein Gjessing

Horstmann, hele kap 20

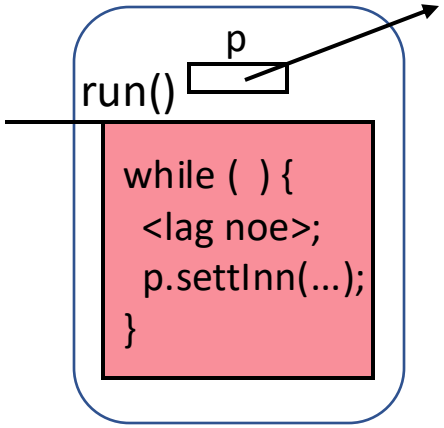
Agenda

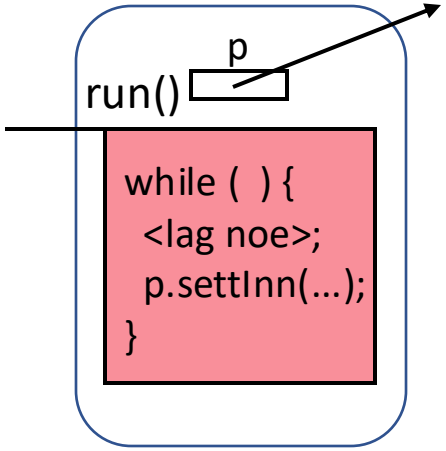
- Ørlite repetisjon med peker tilbake på invarianter i objekter.
  - Hva venter vi på (og hvorfor) i en monitor.
- Klassisk eksempel: Kokk og servitør
- Amdahls lov
- Barrierer (i Java)
  - Eksempel: Finn minste tall
- "Løsning" av innlevering 5
- Vranglås (Deadlock)

Kommunikasjon mellom tråder: En felles beholder = en monitor



Huskeregler for tråder





NB! Trådene (run()-metodene) kaller bare metodene i monitorene som om monitoren var et vanlig objekt.

Ingen låsing, venting eller signalering i trådene utenom monitorene!

Alle låsing, venting og signalering skjer i inne monitorene

Men en tråd kan si Thread.sleep().
Og en tråd kan kalle interrupt() og join() i andre tråder
(og senere i dag skal vi se at de kan kommunisere via barrierer)

En tråd kan stoppe opp en stund når som helst.

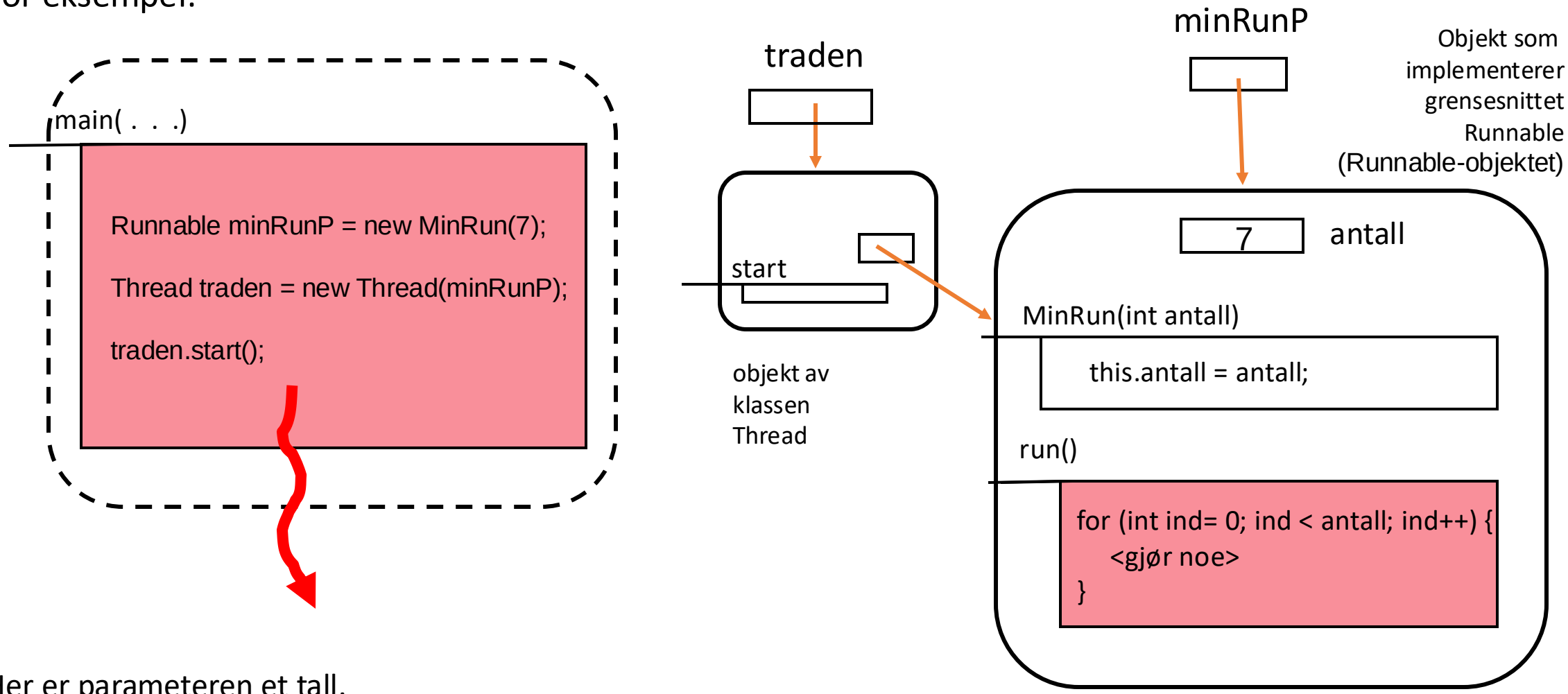
Huskeregler for programmering

- Å programmere betyr å følge 1000-vis (?) av små enkle regler.
- Du må kunne (og helst skjønne) disse reglene.
- Hvis du lurer på noe: Gå tilbake til reglene.
  - F.eks.: Hvordan virker parameteroverføring i Java?
    1. Beregen verdien av uttrykket som er aktuell parameter på kallstedet
    2. La denne verdien være startverdien til variabelen som er formelle parameter når metoden starter
- Det finne ikke noe hokkus-pokkus – bare enkle regler
- ALT ! lar seg forklare ut fra disse enkle reglene
  - Om det ikke lar seg forklare er det ikke mulig

- En helt annen ting er at god programmering er å bruke disse reglene på en god måte.
  - **Dette er mye vanskeligere og tar lang tid å lære**

Parametre til tråder – Bruk konstruktøren til «Runnable-objektet»

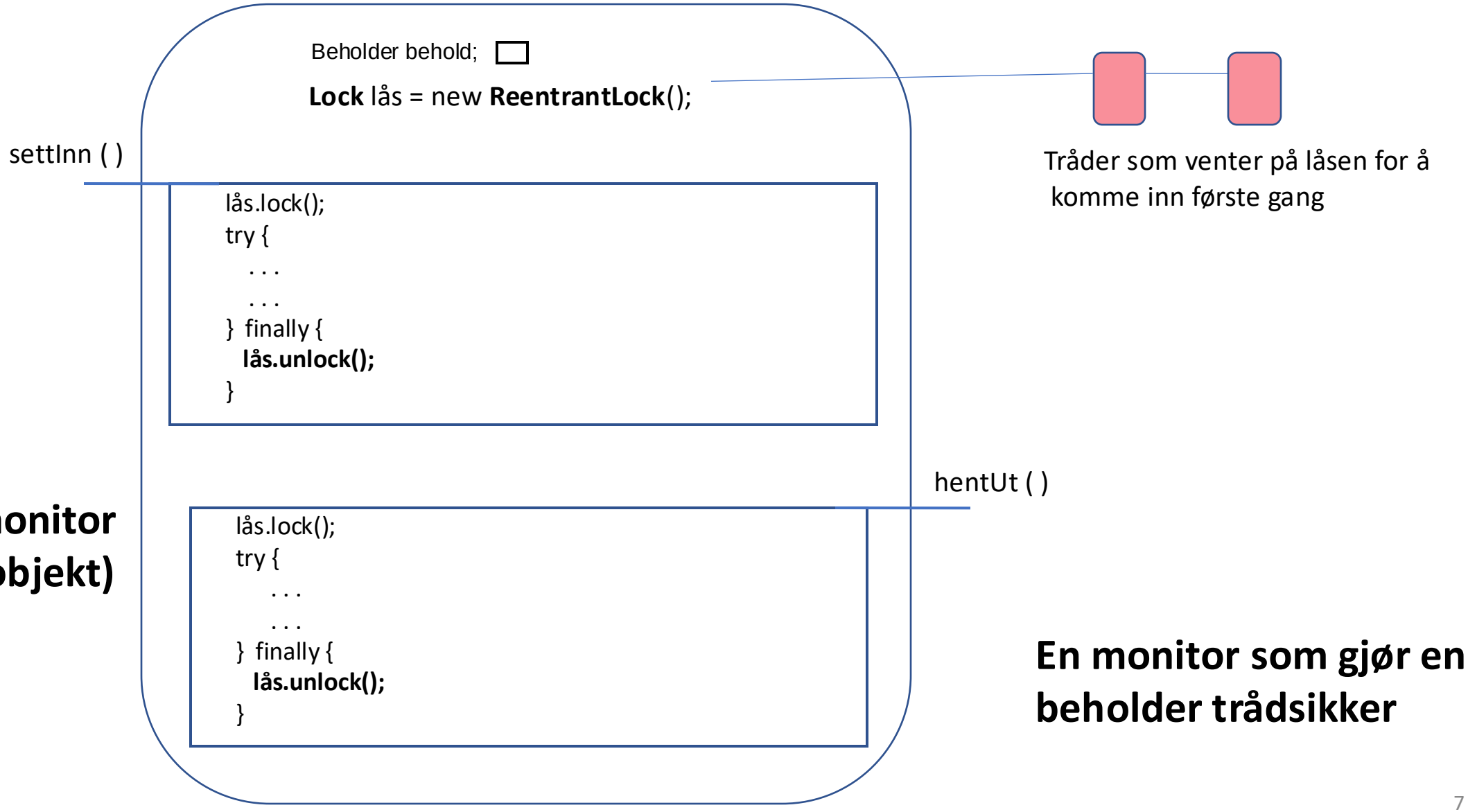
For eksempel:



Her er parameteren et tall.
Også vanlig: En referanse til en monitor

Repetisjon:

Vente på å komme inn i en kritisk region



En monitor (et objekt)

En monitor som gjør en beholder trådsikker

Nå kombinerer vi forskjellige ting vi har lært før i IN1010:

Vi tar opp tråden (☺) fra forelesningen om invarianter på data i objekter:

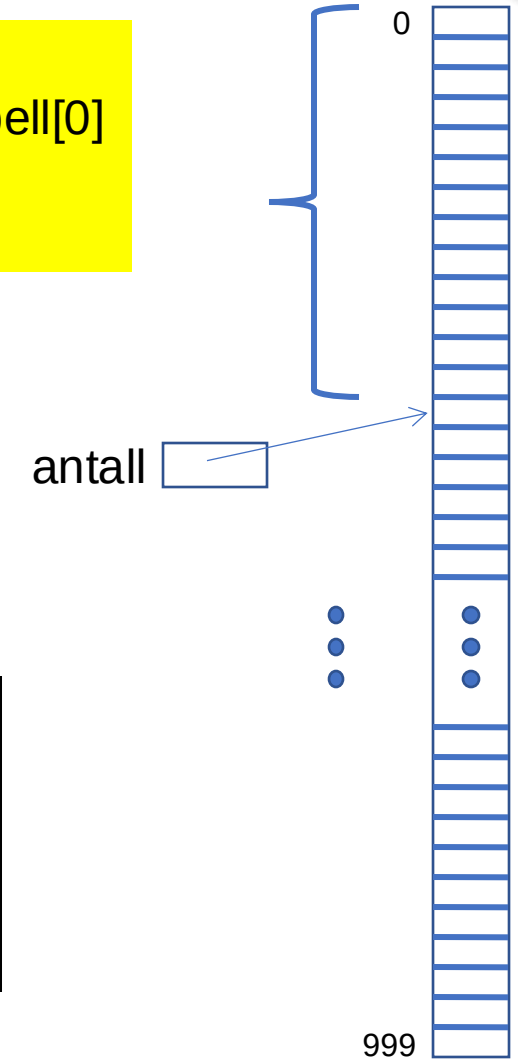
Da var det bare én tråd som opererte på beholderen

Hvis operasjonen ville bryte invarianten måtte vi returnere med uforrettet sak ☹

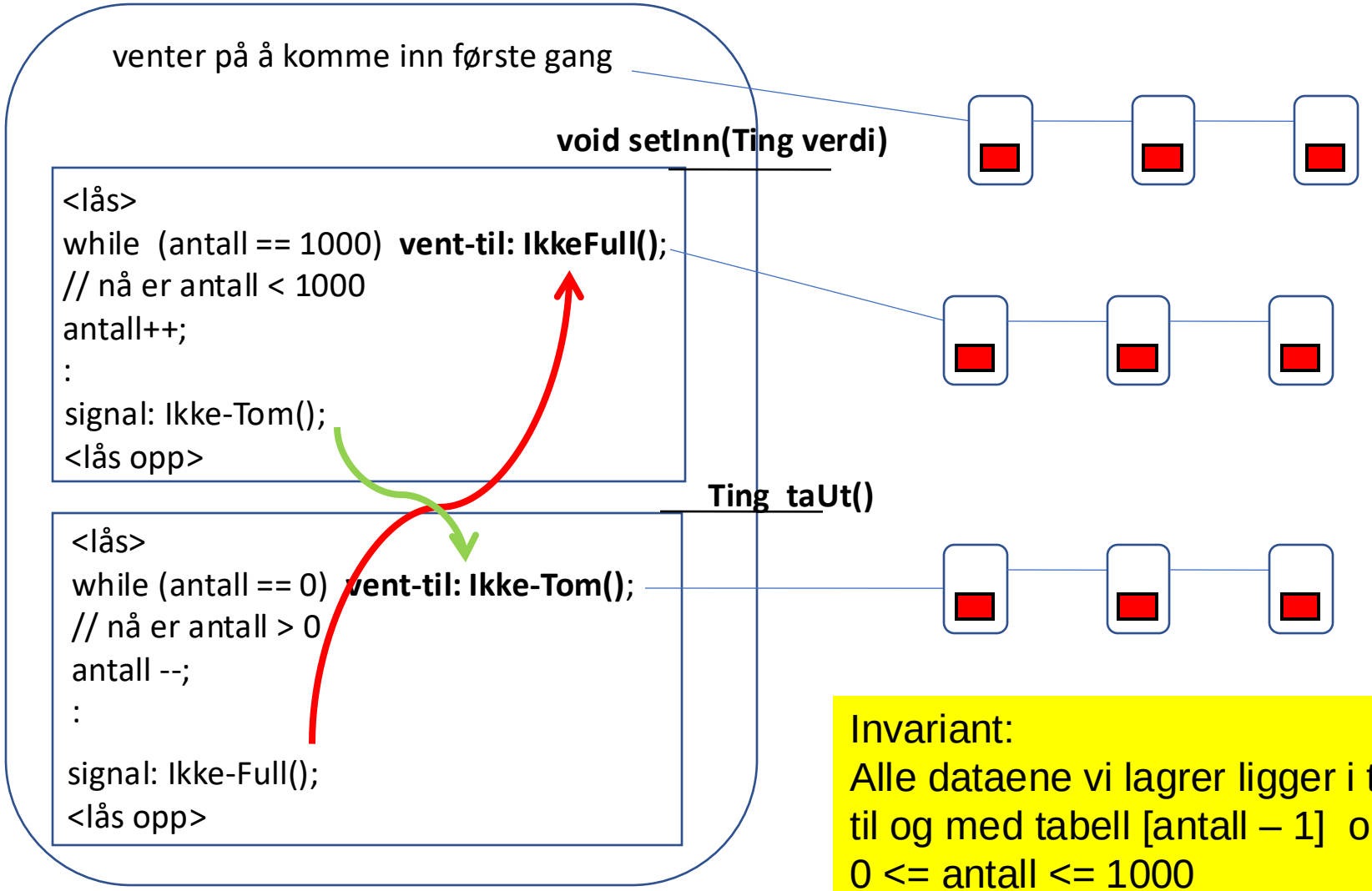
Invariant:
Alle dataene vi lagrer ligger i tabell[0] til og med tabell [antall – 1] og
0 <= antall <= 1000

```
settInn(x) {
    if (antall == 1000) return ;
    antall ++;
    tabell[antall-1] = x;
}
```

```
taUt ( ) {
    if (antall == 0) return null;
    antall --;
    return (tabell[antall]);
}
```



Hvordan bevare invarianter på data i objekter når vi ikke har ansvaret alene. Svar: *Vi venter ofte på at andre skal gjøre objektets (monitorens) tilsand hyggeligere.*



En monitor (et objekt)

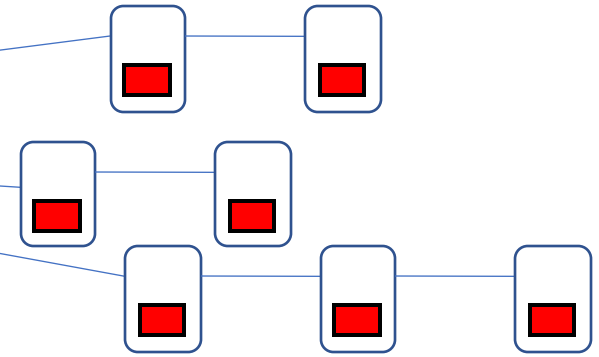
Invariant:
Alle dataene vi lagrer ligger i tabell[0] til og med tabell [antall – 1] og
0 <= antall <= 1000

Beholder bhld;

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

```
void settInn ( Ting verdi) throws InterrupedException
    lås.lock();
    try {
       while  (bhld.full)  ikkeFull.await();
       // nå er beholderen helt sikkert ikke full
       :
        // det er lagt inn noe, så den er helt sikkert ikke tom:
      ikkeTom.signal();
    }  finally {
      lås.unlock();
    }
```

```
Ting taUt ( )  throws InterrupedException
     lås.lock();
     try {
        while  (bhld.tom)  ikkeTom.await();
        // nå er beholderen helt sikkert ikke tom;
        :
         // det er det tatt ut noe, så det er helt sikkert ikke fult:
       ikkeFull.signal();
     }  finally {
       lås.unlock();
     }
```



Litt mer Java

Venter
så vi
ikke bryter
invarianten

Volatile variable

En variabel som er deklarert volatile caches ikke
(og oppbevares ikke i registre
(lenger enn helt nødvending))

- En volatil variabel skrives helt tilbake i primærlageret.




```
boolean stopp = false;

// En tråd:
stopp = true;

// En annen tråd:
  while(! stopp) {
       . . .
   }
```

Må da deklarere:

```
volatile boolean stopp=false;
```

Klassisk eksempel: Kokk og servitør
(fra ekstrastoffet forrige uke)

En (eller to) kokk(er) som lager pizzaer og legger dem på bordet.
Må vente hvis bordet er fullt.

Et bord med plass til 5 pizzaer

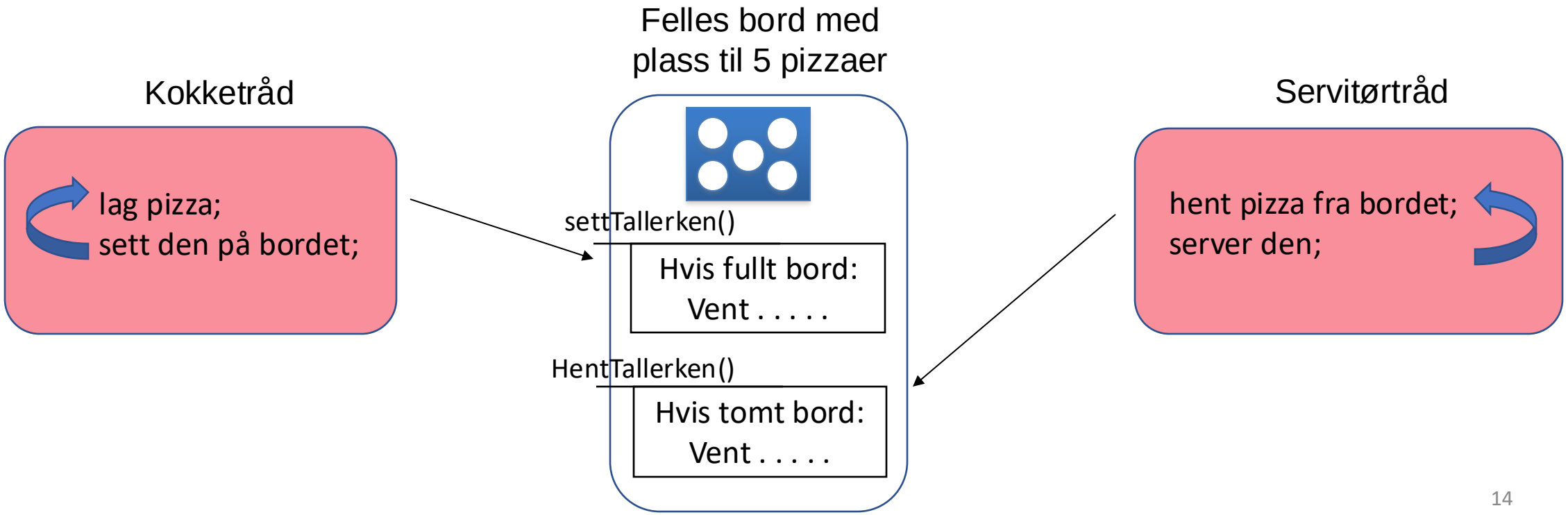
En servitør som henter pizzaer på bordet og serverer til gjesten.
Må vente hvis bordet er tomt.

To produsenter (kokkene) og en konsumet (servitøren



Kokk og servitør (se program på IN1010-siden)

- Kokken må vente når det allerede er fem tallerkener på bodet
- Servitøren må vente når det ikke er laget noe mat (ingen tallerkener på bordet)
- Kokken må starte opp servitøren igjen når han setteren en tallerken på borde
- Servitøren må starte opp kokken igjen når hun tar en tallerken fra bordet



Nytt eksempel

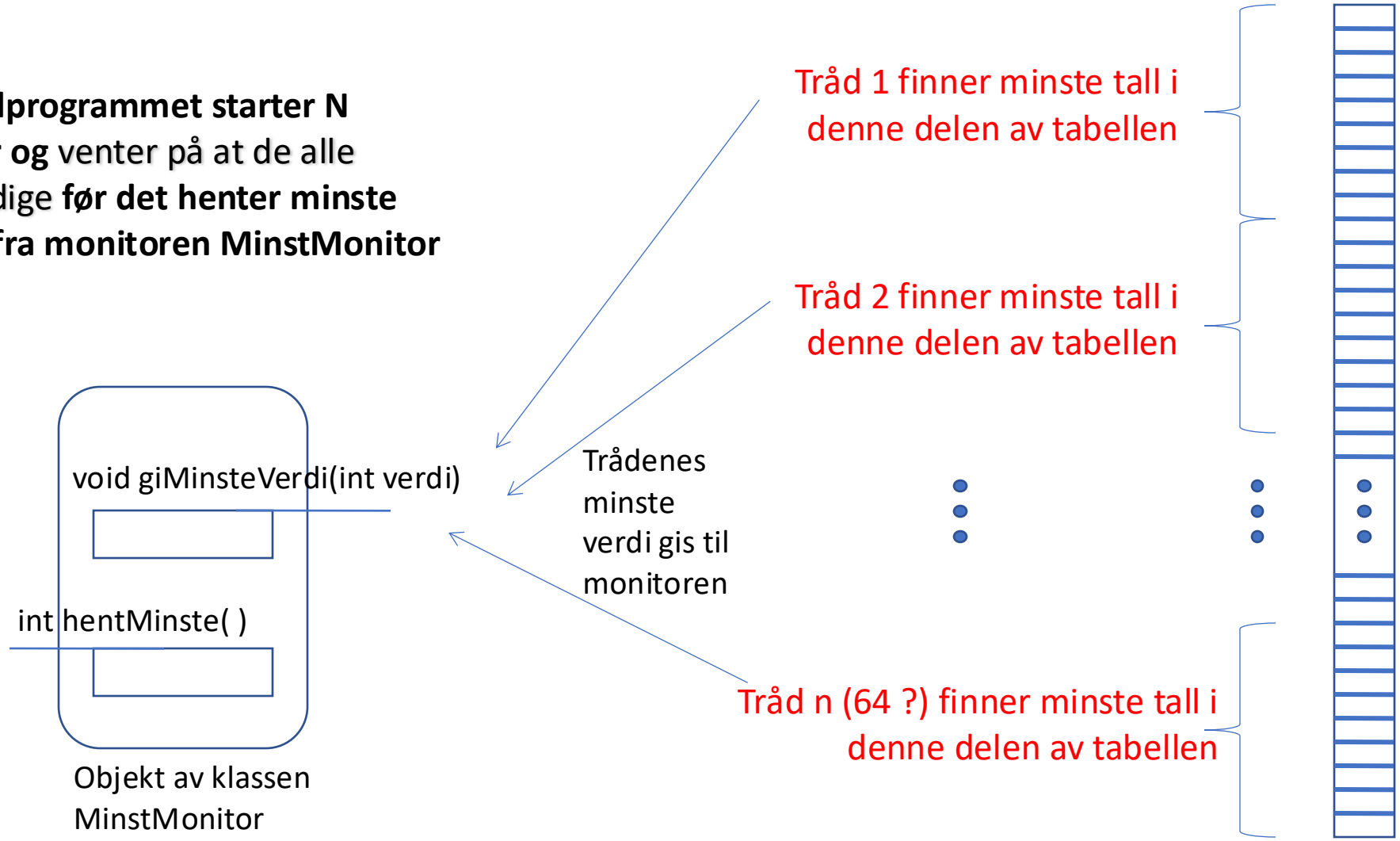
- Finne minst tall i en array



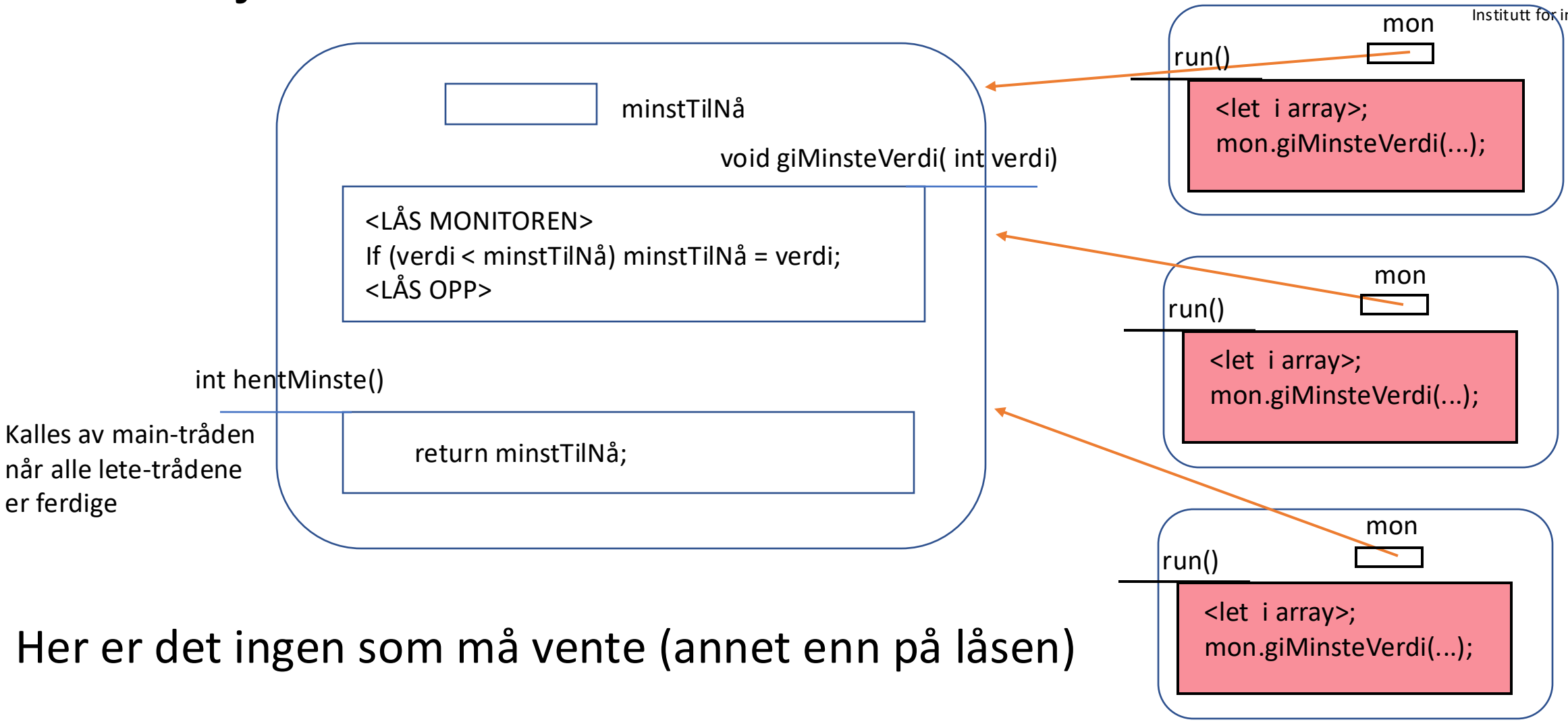
Eksempel på parallellisering: Finn minste tall i en tabell

(samme teknikk for å finne sum / gjennomsnitt)

Hovedprogrammet starter N tråder og venter på at de alle er ferdige **før det henter minste verdi fra monitoren MinstMonitor**



Objekt av klassen MinstMonitor

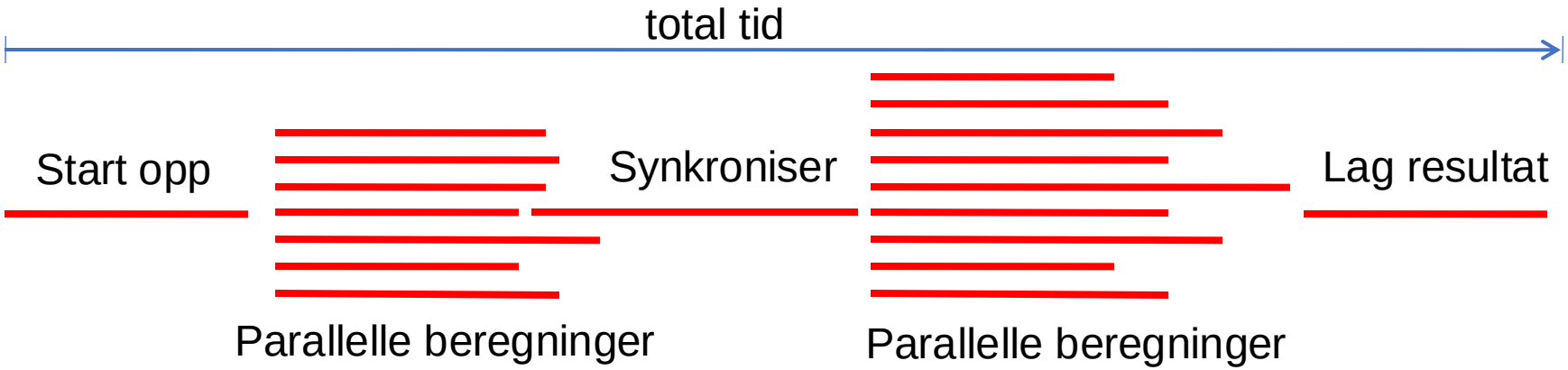


Her er det ingen som må vente (annet enn på låsen)

Men hvordan skal vi vite når alle lete-trådene er ferdige ?

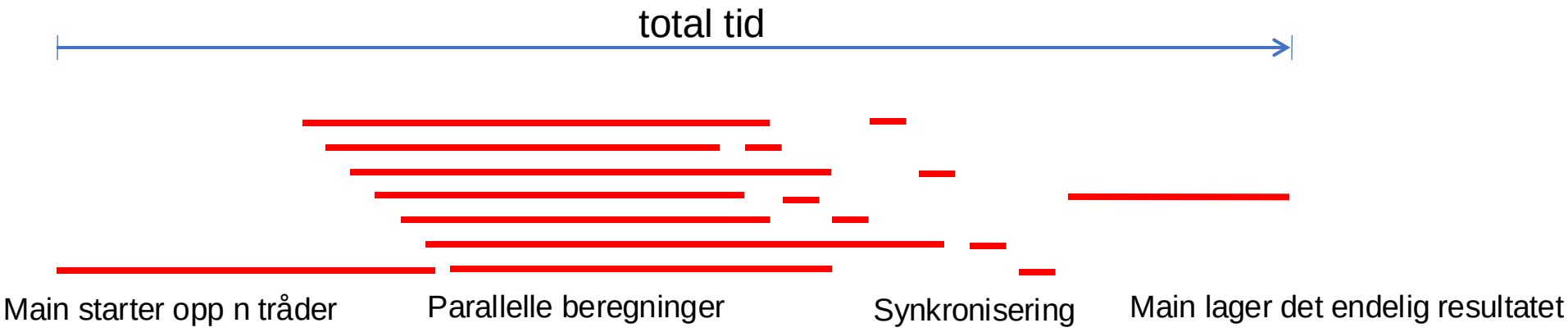
Amdahls lov

- En beregning delt opp i parallell går fortere jo mer uavhengig delene er
- **Amdahls lov**:
  - Totaltiden er
    - tiden i parallell +
    - tiden det tar å kommunisere / synkronisere/ gjøre felles oppgaver
  - Tiden det tar å synkronisere er ikke parallelliserbar (hjelper ikke med flere prosessorer)
  - Men du kan være smart og lage synkroniseringen så kort eller mellom så få tråder som mulig



Amdahls lov og ”finn minste tall”

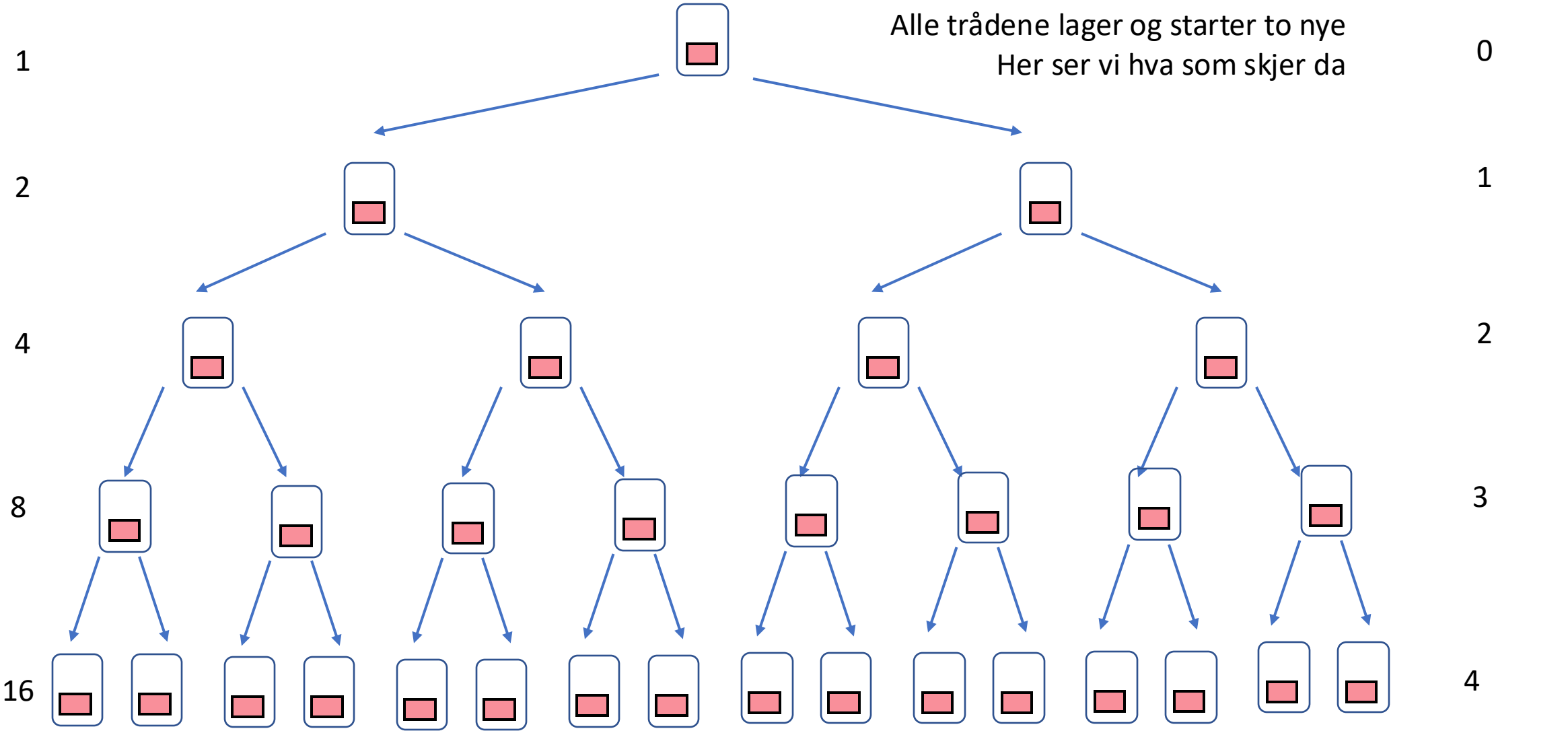
- Totaltid:
  - Tiden det tar å lage og sette i gang 64 tråder
    - (kan det gjøres i parallell ?) ($\log_2 64 = 6$) *
  - Tiden det tar å finne et minste tall i min del av tabellen
  - Til slutt i monitoren: En og en tråd må teste sitt resultat
    - (Kan det gjøres i parallell ?)



* Start med én tråd, vha. doblinger 6 ganger har vi 64 tråder, dvs, $2^6 = 64$, og $\log_2 64 = 6$

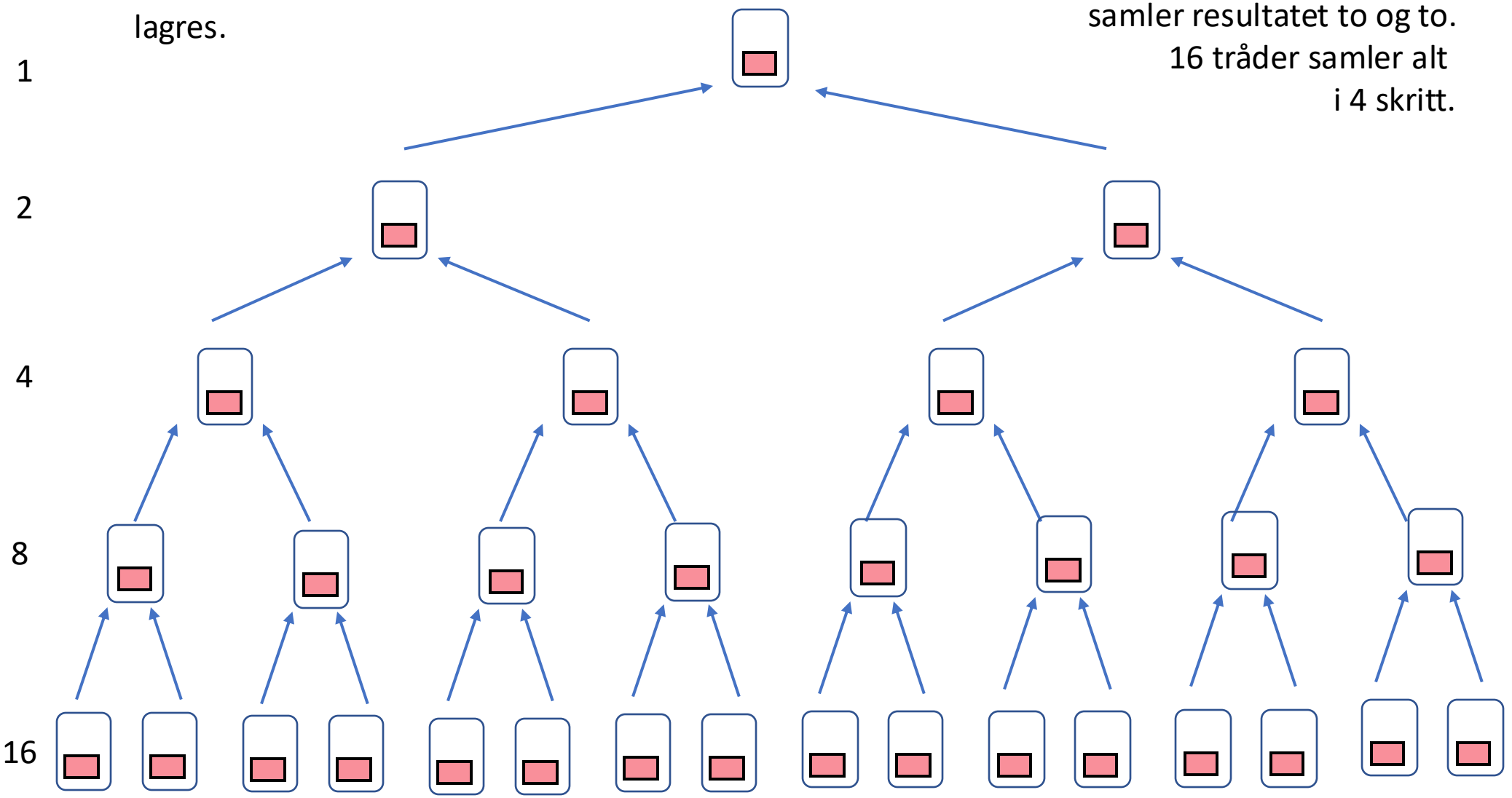
2, 4, 8, 16, 32, 64

Trær er populære i informatikk (men dette eksempelet er litt på siden i IN1010):



Men her mangler det mye bl.a. om hvor resultatene skal lagres.

Vi kunne tenkt oss at trådene samler resultatet to og to. 16 tråder samler alt i 4 skritt.

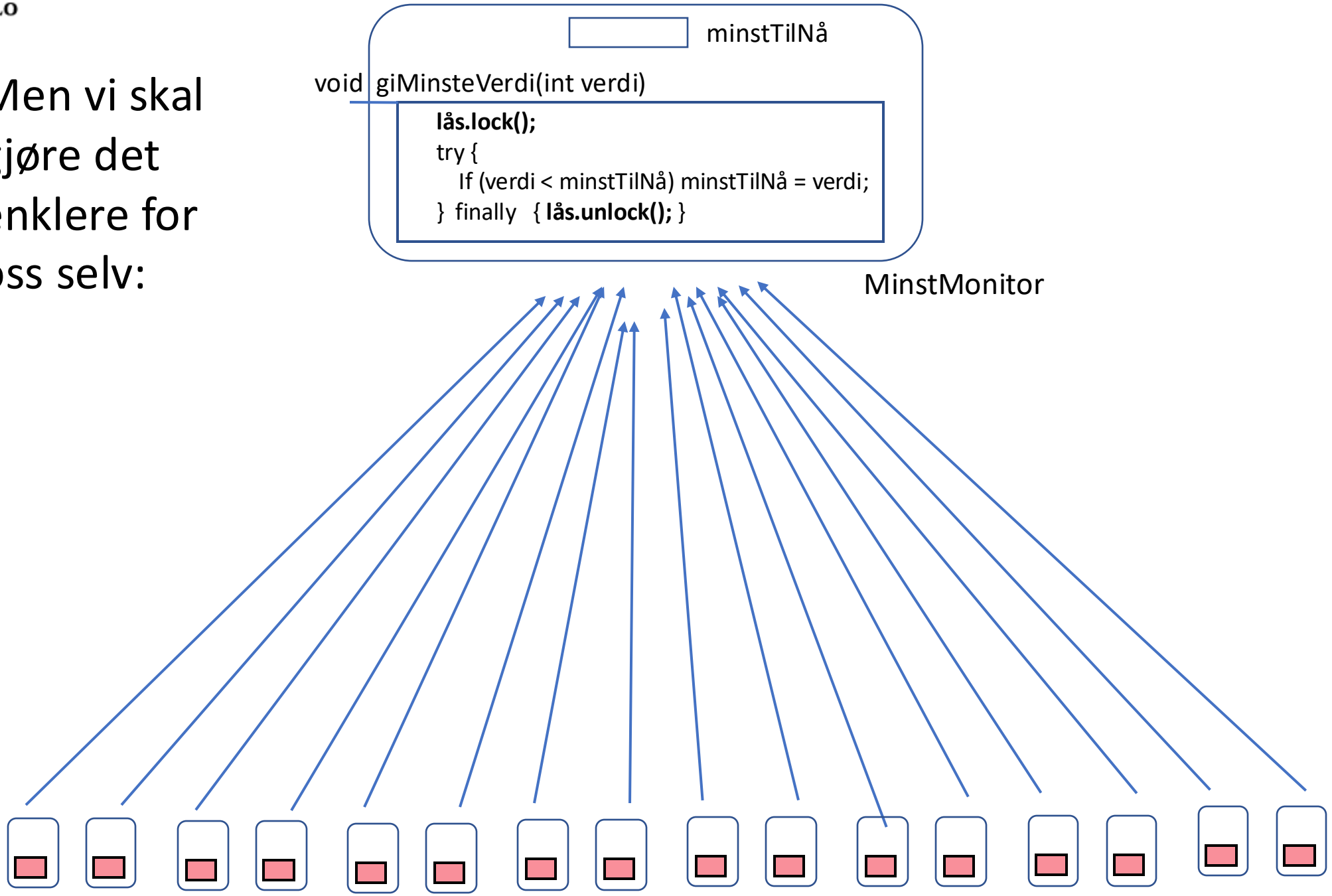


På forrige bilde ("Her mangel det mye")

- For spesielt interesserte (meget avansert bruk av pensum i IN1010):
  - Det er mulig for en klasse som implementerer Runnable å være en monitor også
  - Når treet lages kan en nye tråd få med en referanse til sine mor.
  - Når resultatet er laget av hver av de to barna kan resultatet sendes til moren, men siden det er to barn må metoden som tar i mot resultatet være en kritisk region og objektet må være en monitor.
  - Moren må vente på at begge barna har lagt inn et resultat før hun sender sitt resultate til sin mor igjen, osv.

- Enda "lettere" tråder: Virtuelle tråder (Java versjon 21)
  - Tråder som hovedsakelig bare administreres av kjøretidsystemet

Men vi skal gjøre det enklere for oss selv:

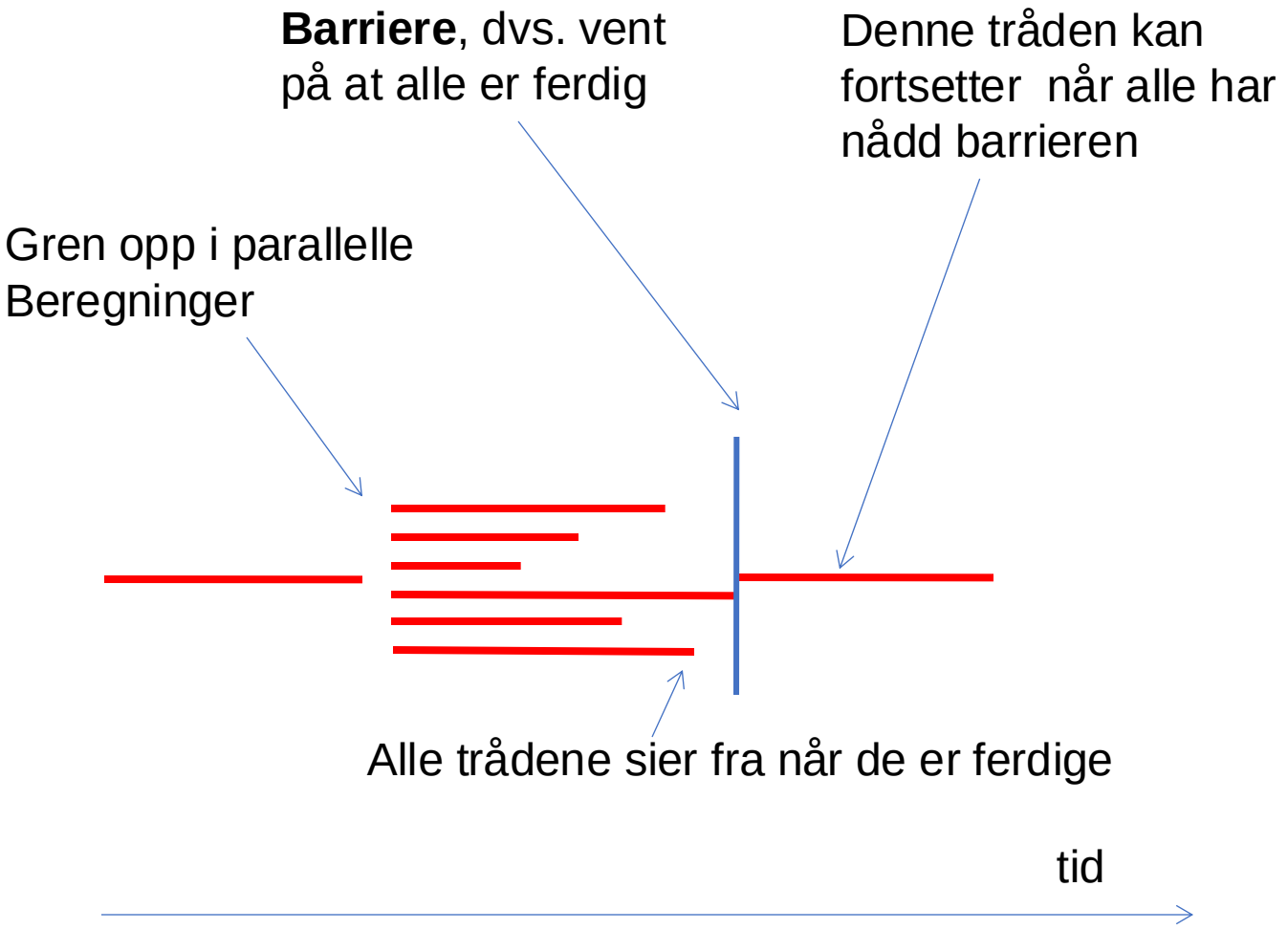


«Finn minste tall i en stor array» ved hjelp av mange (64?) tråder

- Hvordan kan vi vite når alle lete-trådene er ferdige og resultatet ligger ferdig i monitoren?

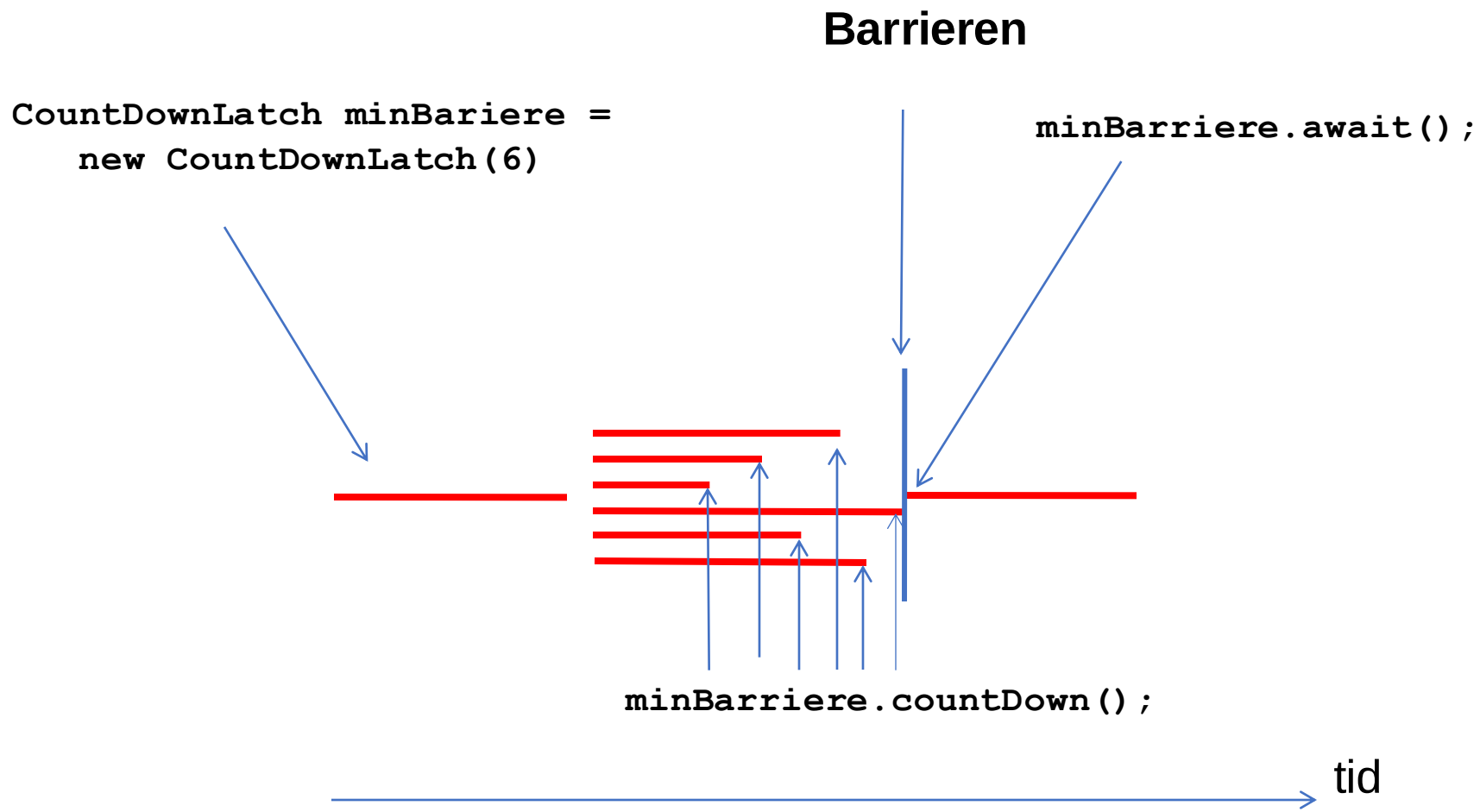
- Se programmet Minst.java på IN1010-sidene

Barrierer i parallellprogrammering

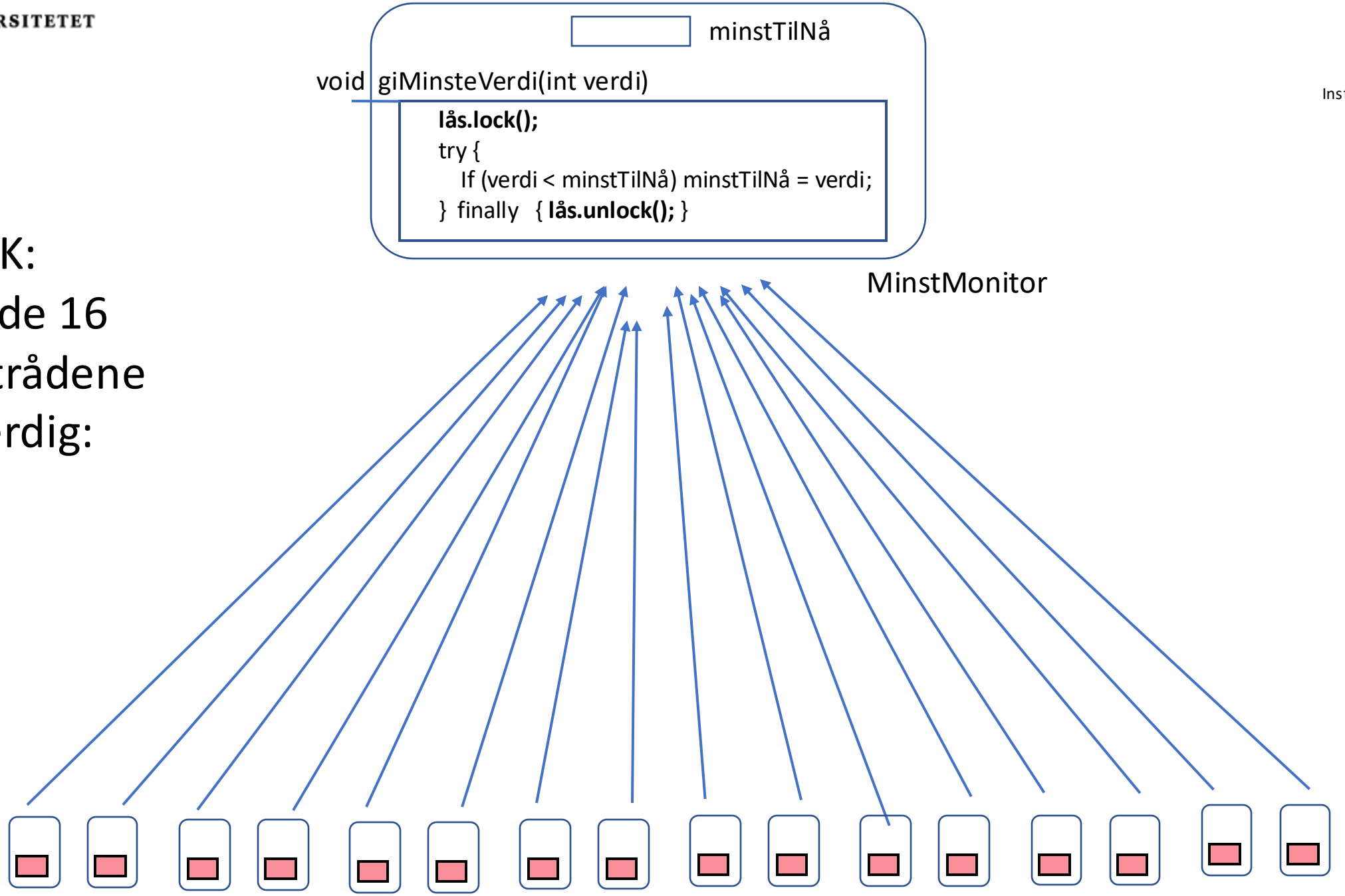


Barrierer i Java

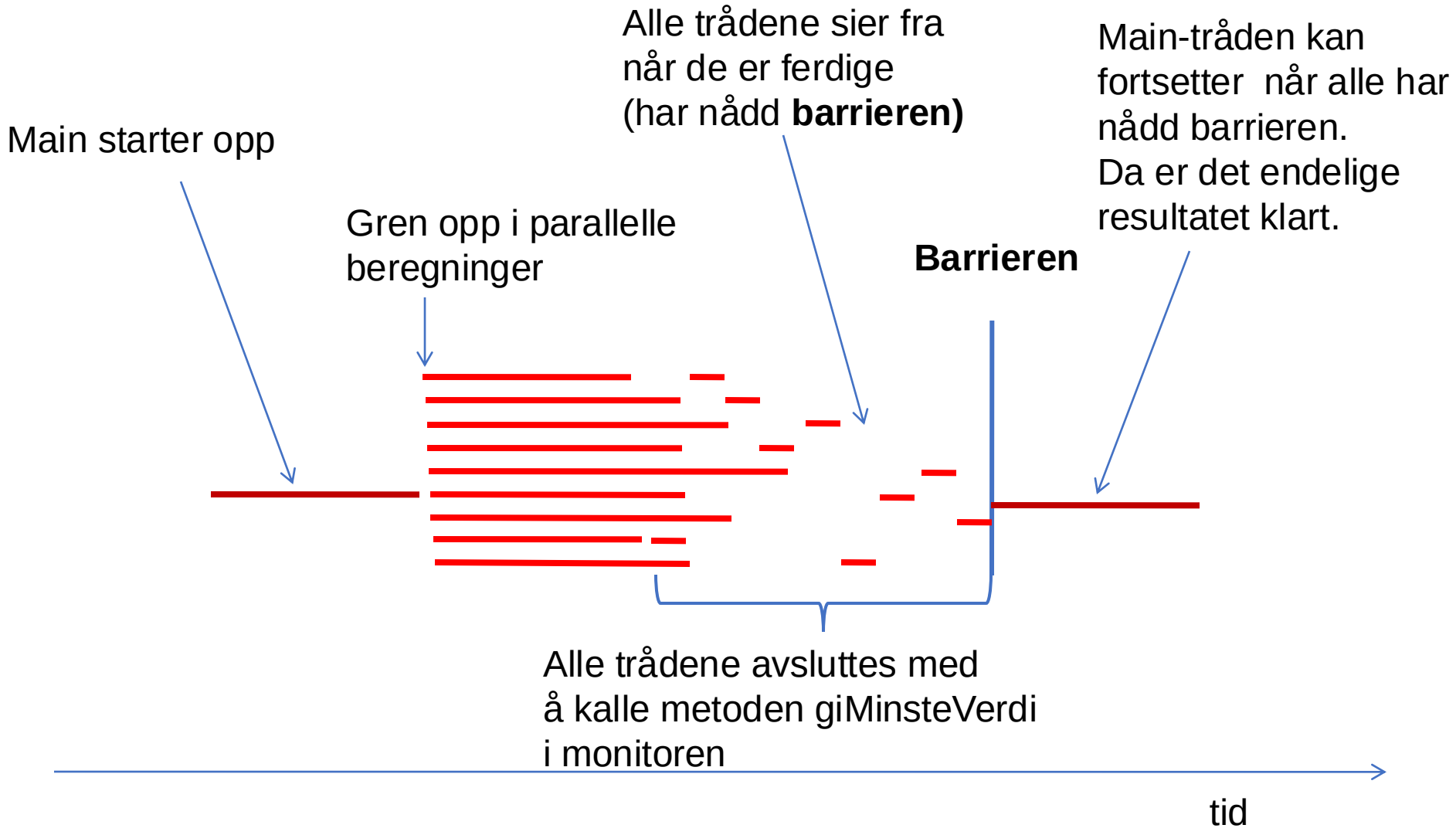

```
import java.util.concurrent.*;
```



HUSK:
Når de 16 letetrådene er ferdig:



Barrieren i dette eksemplet (Minst.java)

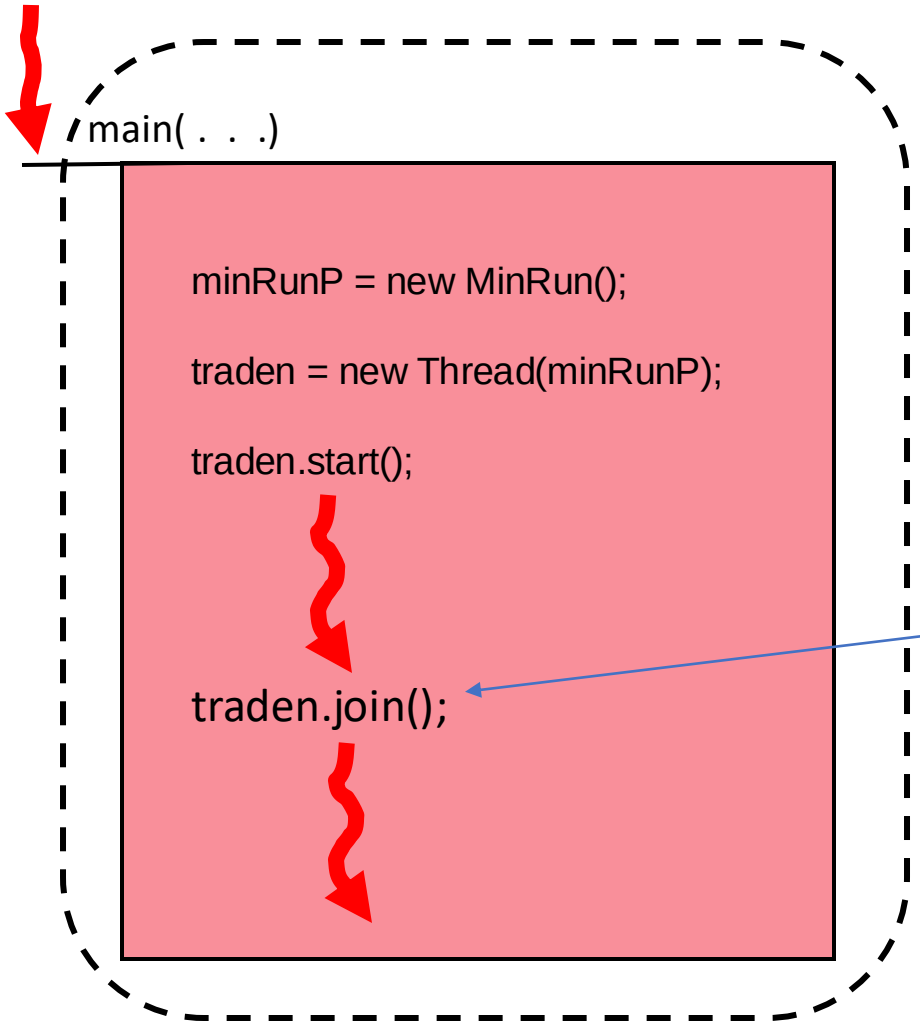


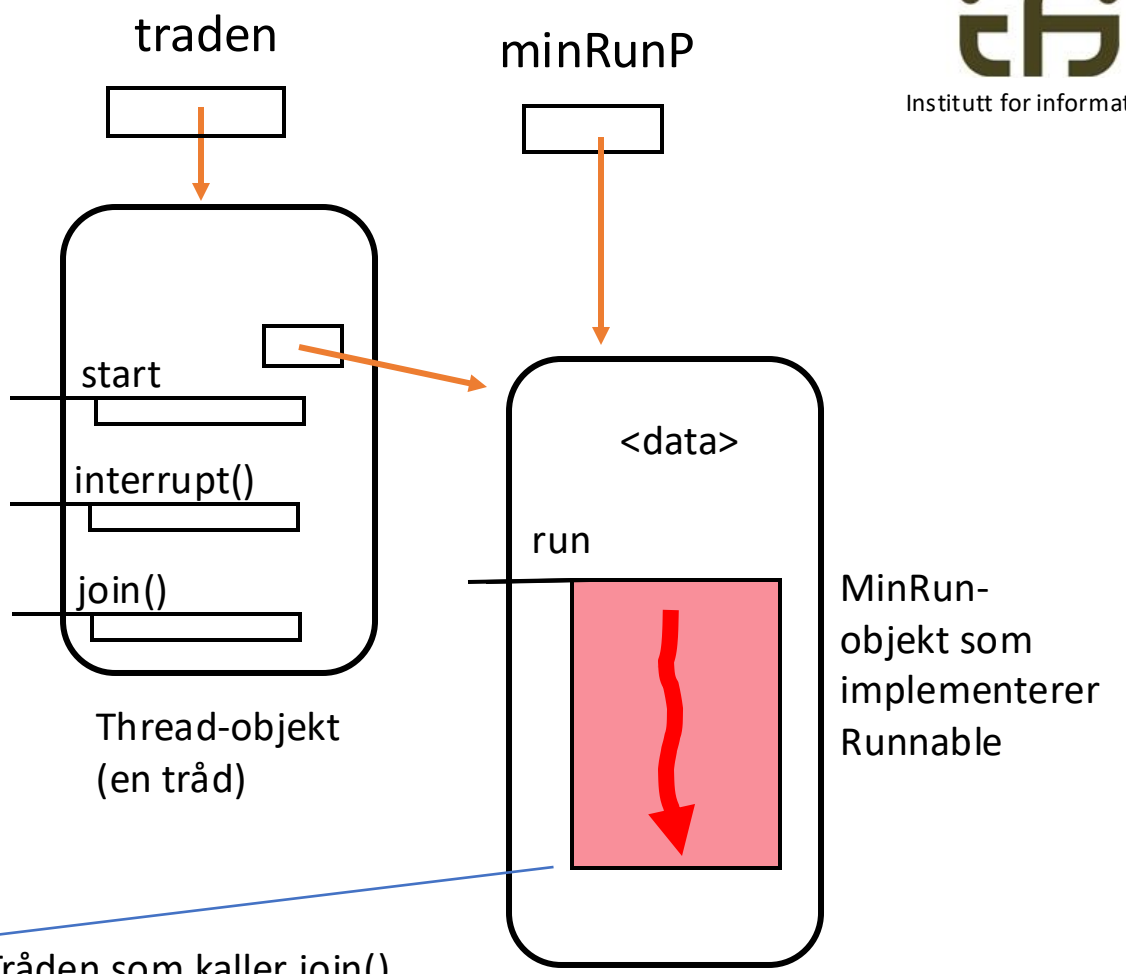
Gafling og møting (fork and join)

- I parallellprogrammering og på engelsk brukes "fork" om å lage en ny tråd eller en ny prosess
- Jeg tror ikke det brukes noen norsk oversettelse (å gafle? forgrene?)
- Det finnnes ikke noen fork-mekanisme i Java
  - Du må selv lage en ny tråd, og eventuelt gå i løkke for å lage flere
  - Når tråder terminerer kalles det gjerne join (neste side)

join()

F.eks. kjøretidsystemet kaller main()





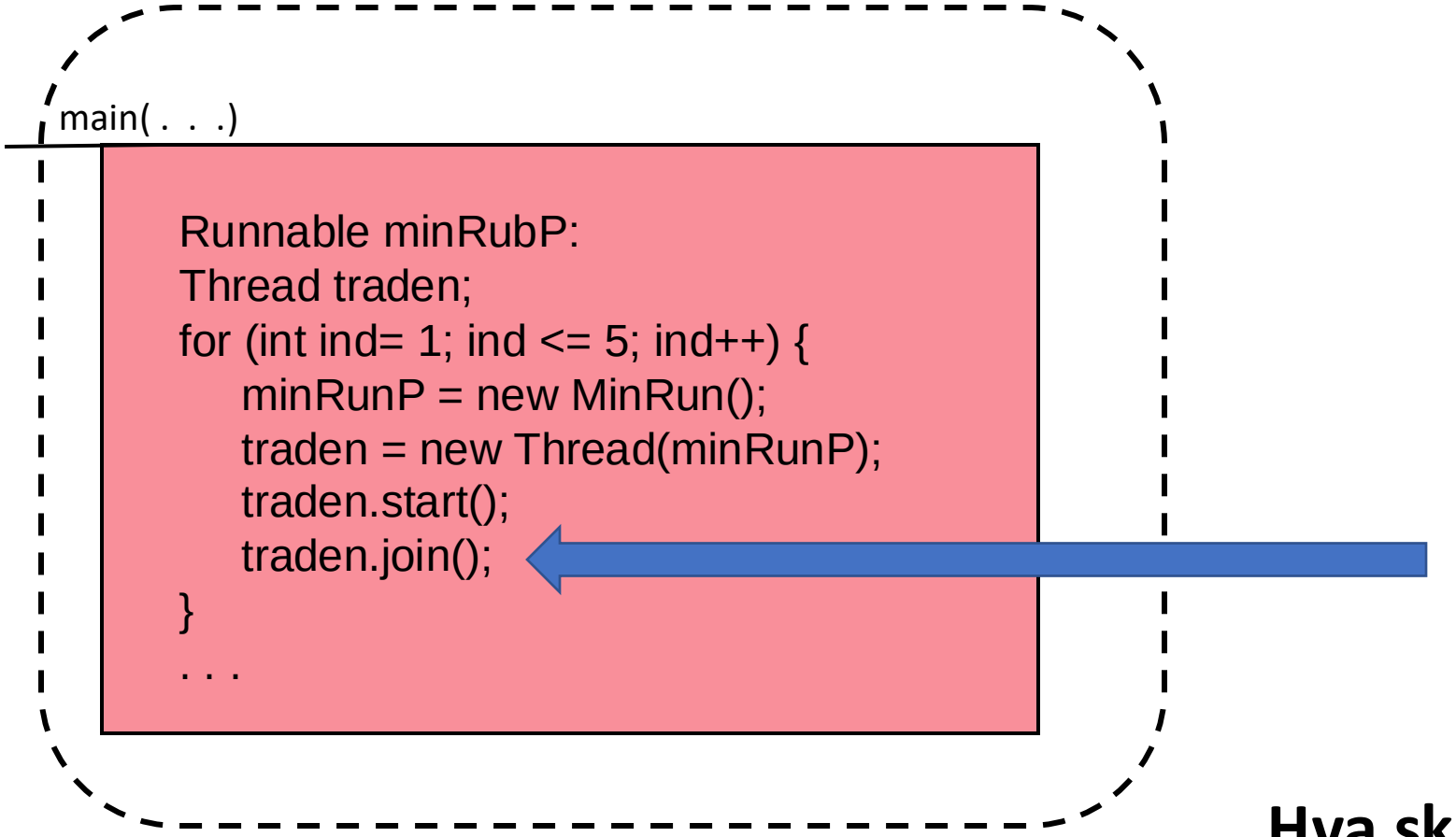
Thread-objekt (en tråd)

MinRun-objekt som implementerer Runnable

Tråden som kaller join() venter på at den andre tråden skal avslutte run()-metoden

Pass på, hvis du starter og venter på mange tråder:

Ikke gjør slik:

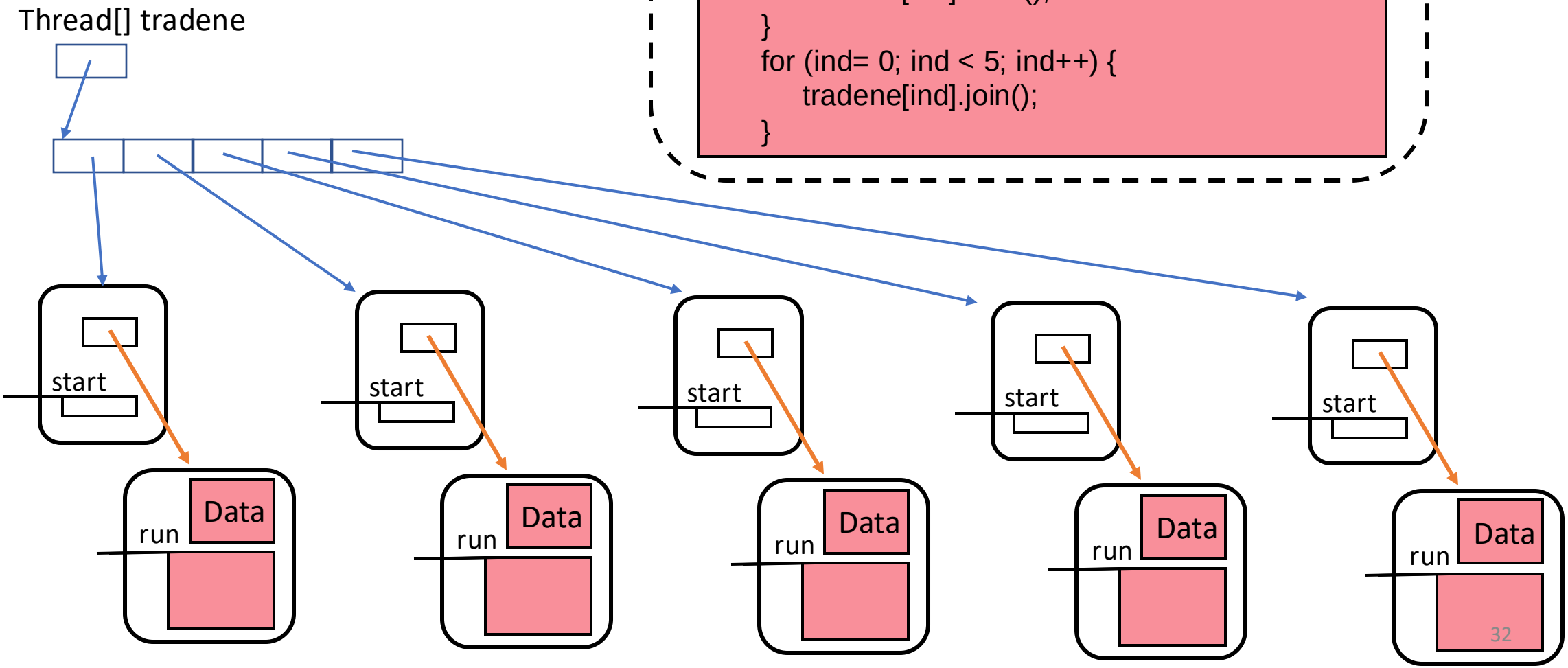


Hva skjer?

Men slik:

main( )

```
Runnable minRunP;
Thread[ ]  tradene = new Thread[5];
int ind;
for (ind= 0; ind < 5; ind++) {
    minRunP = new MinRun();
    traden[ind] = new Thread(minRunP);
    tradene[ind].start();
}
for (ind= 0; ind < 5; ind++) {
    tradene[ind].join();
}
```



Innlevering 5

- Oppgave 1: Frekvenstabell
- Oppgave 2: Subsekvensregister
- Oppgave 3: Monitor
- Oppgave 4: Lesetråd
- Oppgave 5: Utvidet Monitor
- Oppgave 6: Flettetråd
- Oppgave 7: Klargjør data: Hovedprogram

Innlevering 5

- Du skal lage et Subsekvensregister-objekt (oppgave 2)
- Du skal lage en Monitor (et Monitor-objekt) som beskytter dette (oppgave 3 og 5)

- Du skal lage N lesetråder som leser N filer, lager N frekvenstabeller og legger dem inn i Monitoren (oppgave 4)

- Du skal lage 8 flettetråder der hver tråd går i løkke og henter ut to frekvenstabeller fra Monitoren, fletter dem sammen og legger resultatet tilbake i Monitoren.
  Løkken (og alle disse trådene) avsluttes når alle de N frekvenstabellene er flettet sammen til èn. (oppgave 6)

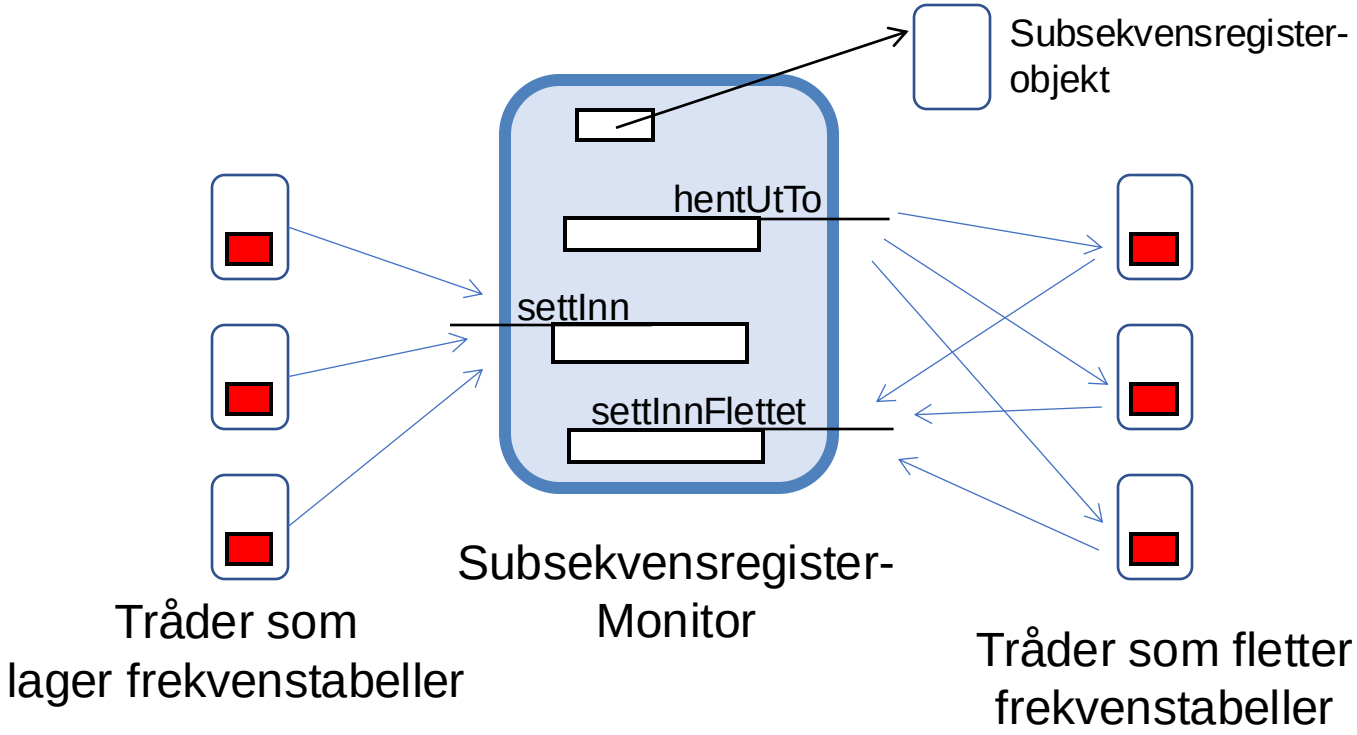
Innlevering 5

- Vi skal lage et Taustump-register-objekt som tar vare på mange taustumper
- Vi skal lage en Monitor (et Monitor-objekt) som beskytter Taustump-registeret

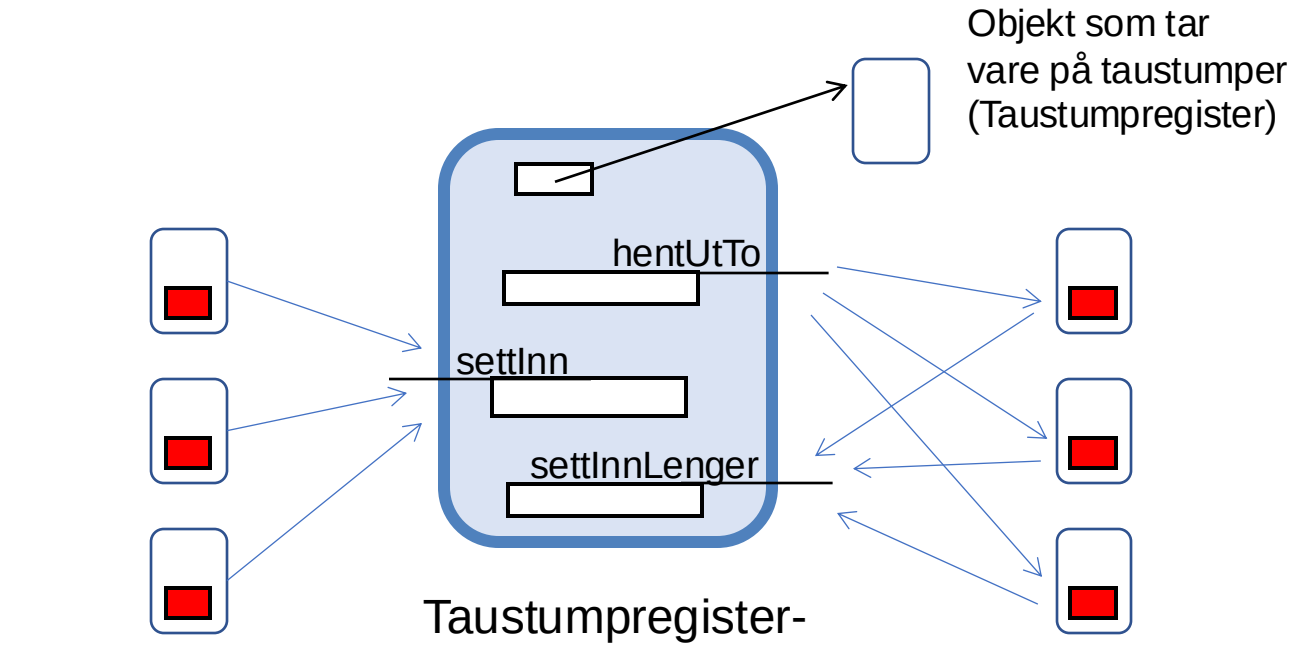
- Vi skal lage 16 tråder som lager 16 taustumper og legger dem inn i Monitoren (og derved inn i Taustump-registeret)

- Vi skal lage 4 flettetråder der hver tråd går i løkke og henter ut to taustumper fra Monitoren, fletter eller binder dem sammen og legger reultatet tilbake i Monitoren.
  Løkken (og alle disse trådene) avsluttes når alle de 16 taustumpene er flettet eller bundet sammen til én.

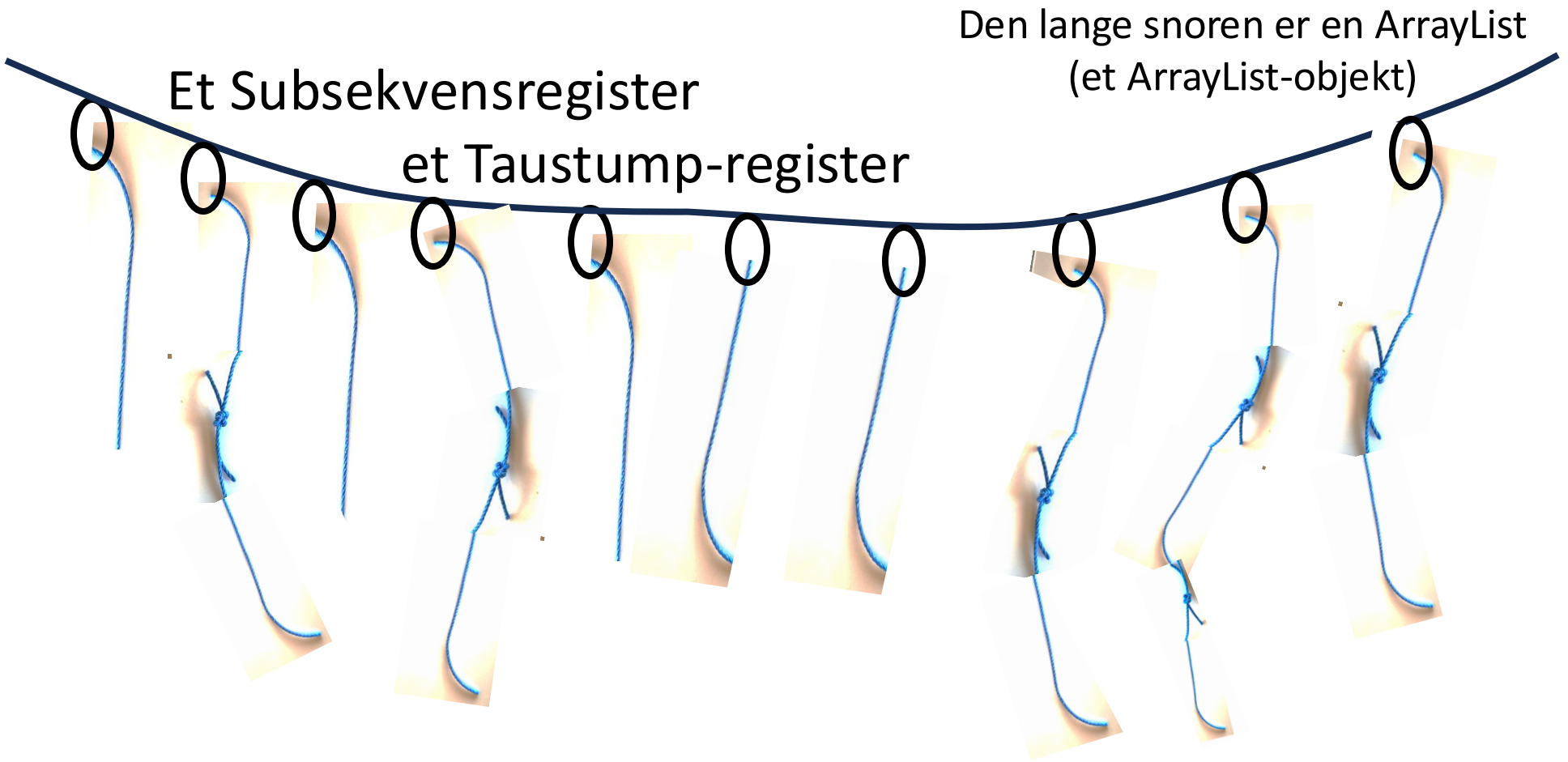
I innlevering 5 skal du lage:



Vi skal nå lage:

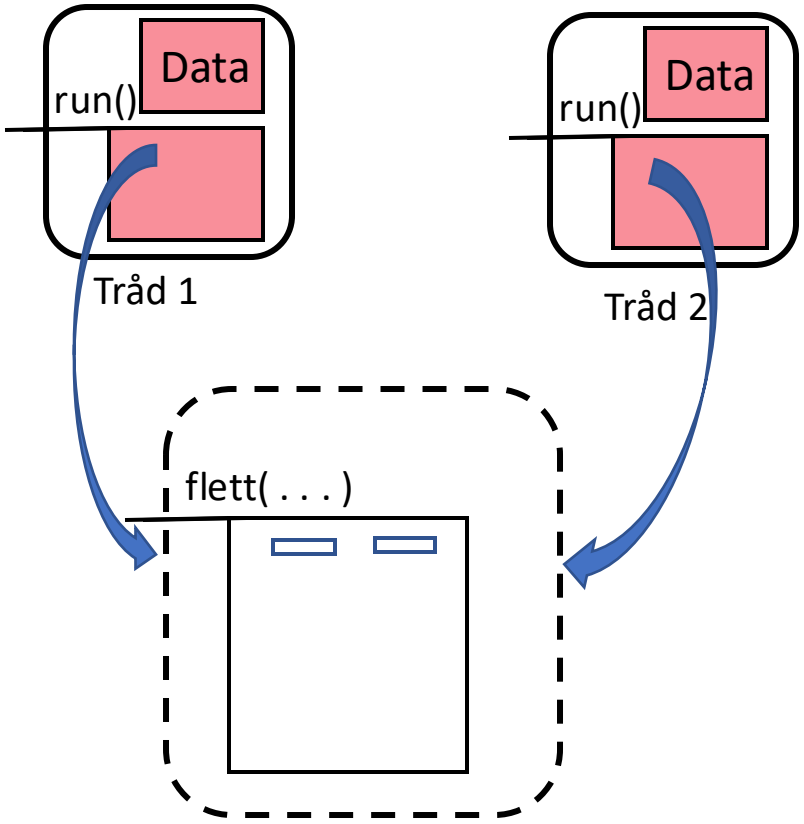


Innlevering 5



Hver taustump er en frekvenstabell
(eller en flettet frekvenstabell)

Innlevering 5 og bruk av samme metode



Om du programmerer flettingen av frekvenstabeller i en statisk metode som bare bruker lokale variabler, kan alle tråder som vil kalle og bruke denne metoden samtidig gå i ekte parallell.

Derfor kan en static flett-metode brukes av alle trådene dine uten å gå i bena på hverandre.

Husk: Hver eneste tråd har sin egen stabel av metodekall ("the call stack")

Nytt tema
(men fortsatt tråder)

Vranglås
Engelsk: Deadlock

Horstmann kap. 20.5

Vranglås (deadlock)

- Vranglås skjer når flere tråder venter på hverandre i en sykel:
- Eksempel
  - Veikryss:
    alle bilene skal stoppe for biler fra høyre

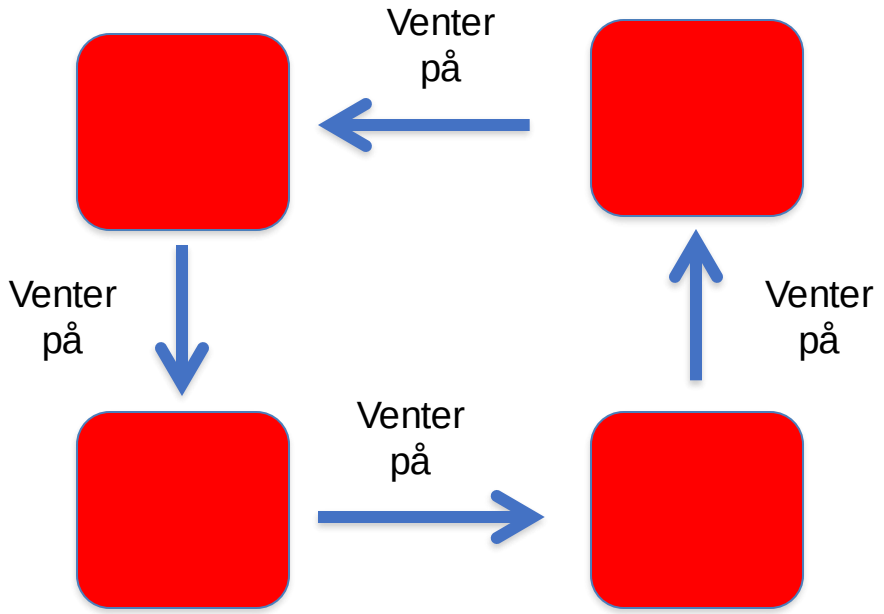
    ->

    Alle stopper = VRANGLÅS

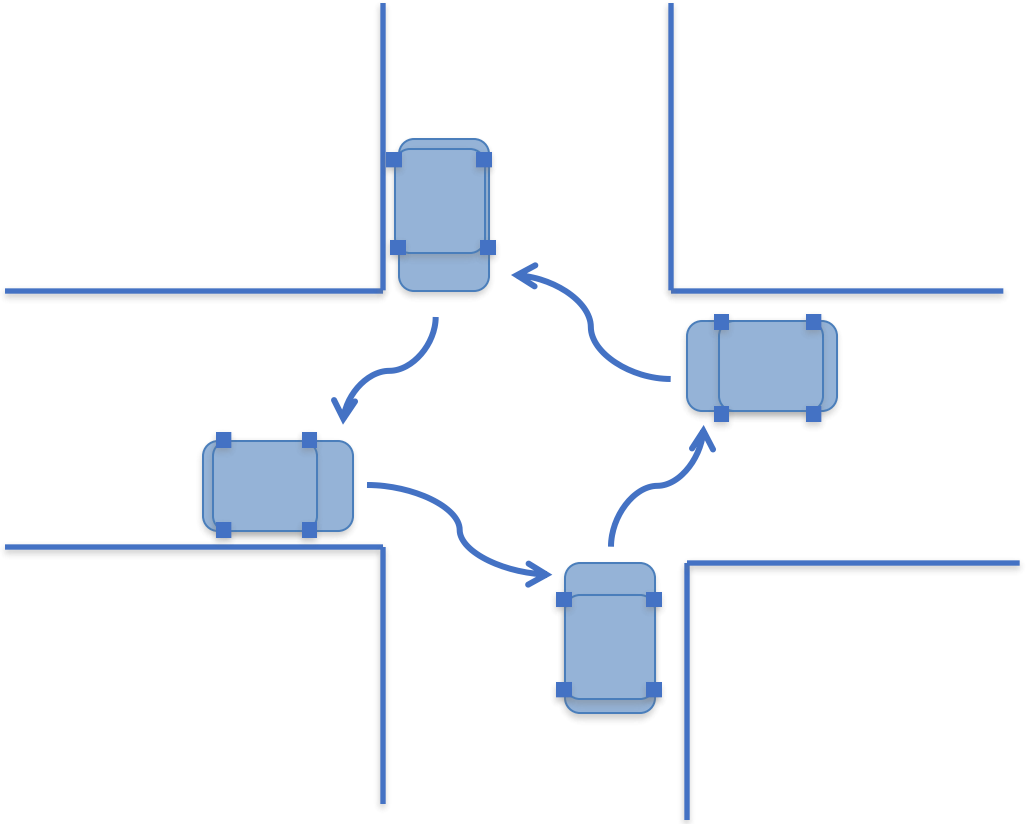


Vranglås betyr

- Flere tråder venter på hverandre
- Syklisk venting



Hvordan kan dette skje?



Vranglås kan oppstå når flere tråder kjemper om felles ressurser

- En eller flere felles ressurser ønskes av mer enn en tråd (f.eks. i et OS)
- Hvis en tråd først tar en ressurs og deretter en annen . . .

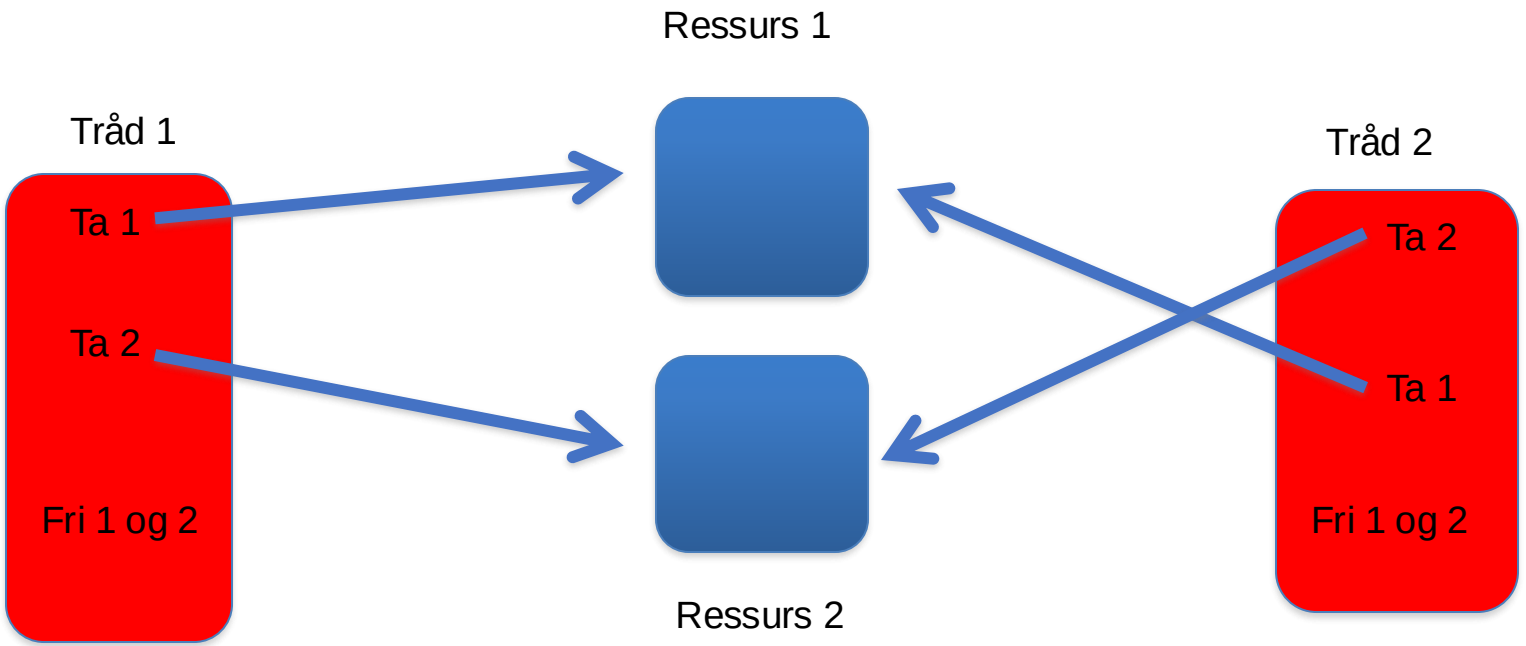
Unngå vranglås

1. Ta bare én ressurs
2. Ta alle på en gang, eller ingen
3. Alle tråder tar alle ressurser i samme rekkefølge

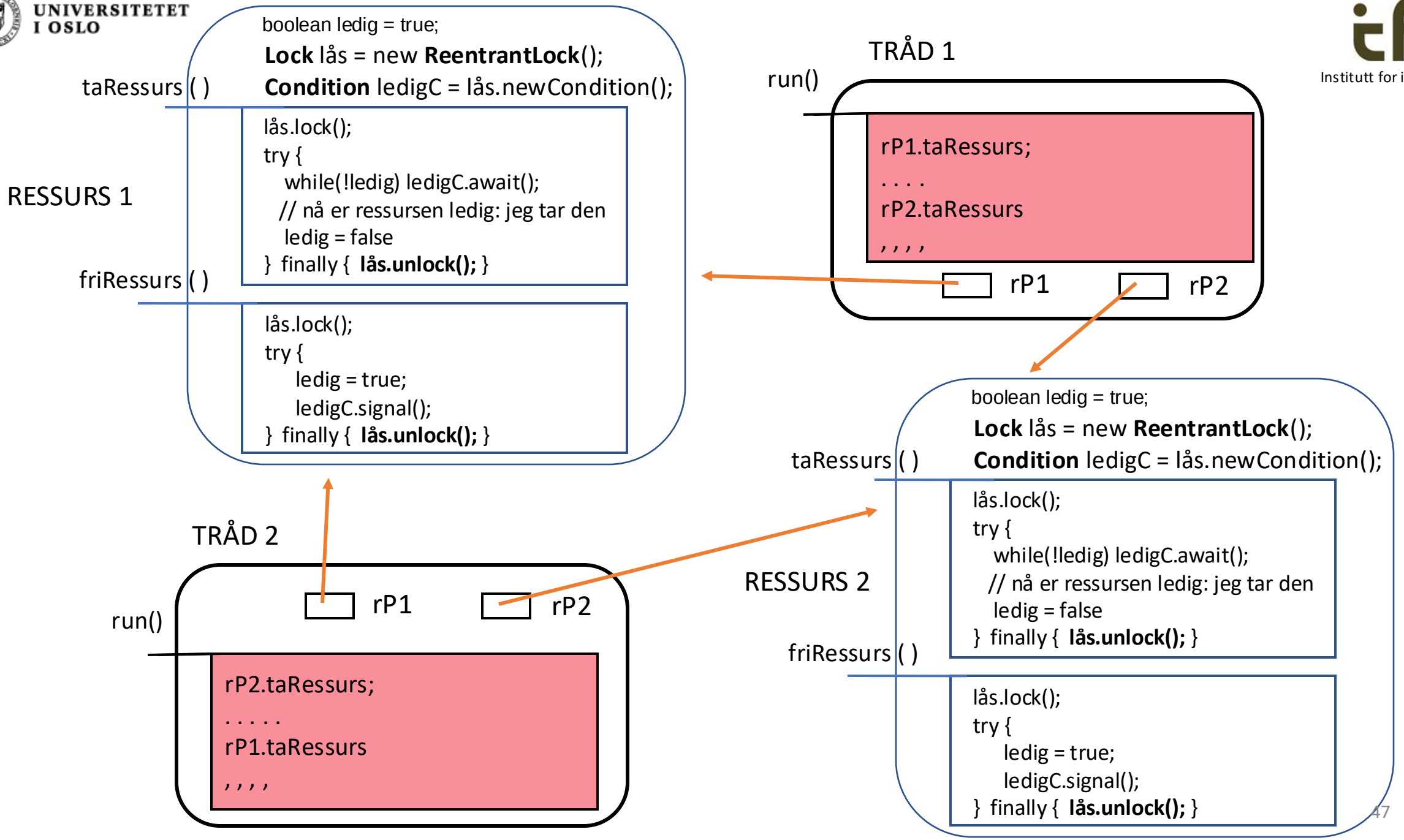
- Hvis vranglås har oppstått:
  - Fri en og en ressurs til det ikke lenger er vranglås

Enkleste eksempel på vranglås

- To tråder
- To ressurser som tas i forskjellig rekkefølge



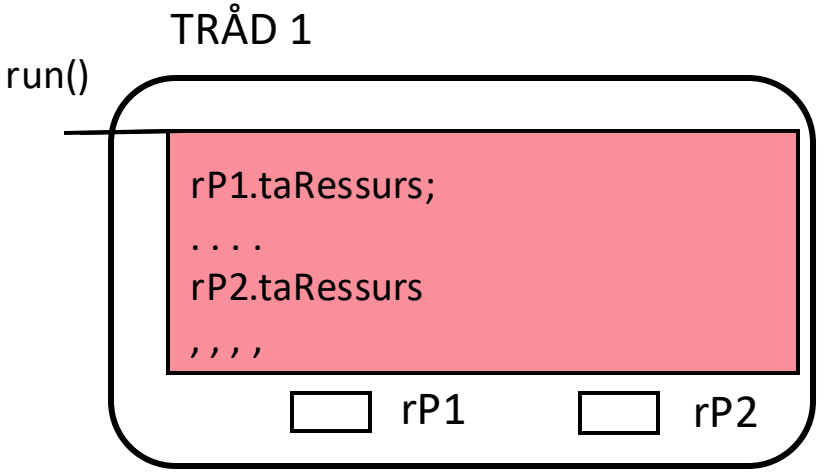
Detaljert beskrivelse i Java på neste side

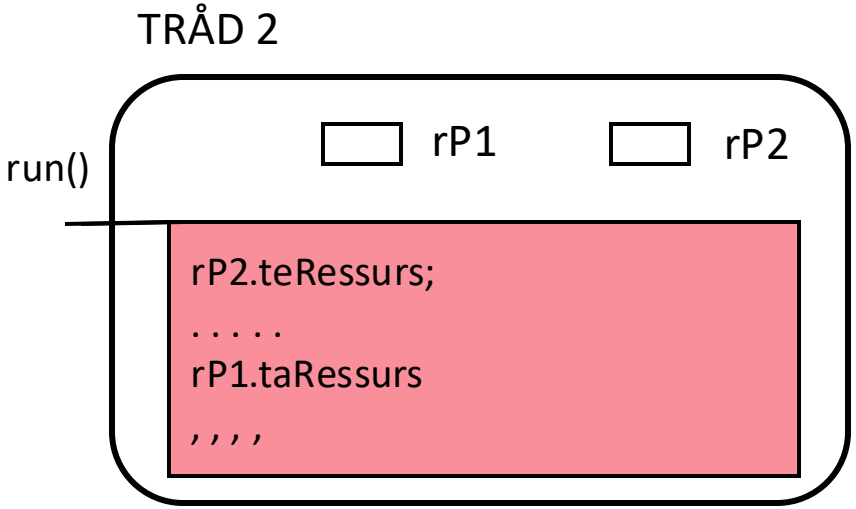

boolean ledig = true;
Lock lås = new ReentrantLock();
taRessurs ( )
Condition ledigC = lås.newCondition();
lås.lock();
try {
while(!ledig) ledigC.await();
// nå er ressursen ledig: jeg tar den
ledig = false
} finally { lås.unlock(); }
RESSURS 1
friRessurs ( )
lås.lock();
try {
ledig = true;
ledigC.signal();
} finally { lås.unlock(); }
TRÅD 1
run()
rP1.taRessurs;
. . . .
rP2.taRessurs
, , , ,
rP1
rP2
TRÅD 2
run()
rP1
rP2
rP2.taRessurs;
. . . . .
rP1.taRessurs
, , , ,
RESSURS 2
boolean ledig = true;
Lock lås = new ReentrantLock();
taRessurs ( )
Condition ledigC = lås.newCondition();
lås.lock();
try {
while(!ledig) ledigC.await();
// nå er ressursen ledig: jeg tar den
ledig = false
} finally { lås.unlock(); }
friRessurs ( )
lås.lock();
try {
ledig = true;
ledigC.signal();
} finally { lås.unlock(); }

VRANGLÅS:

Tråd 1 tar rP1
Tråd 2 tar rP2

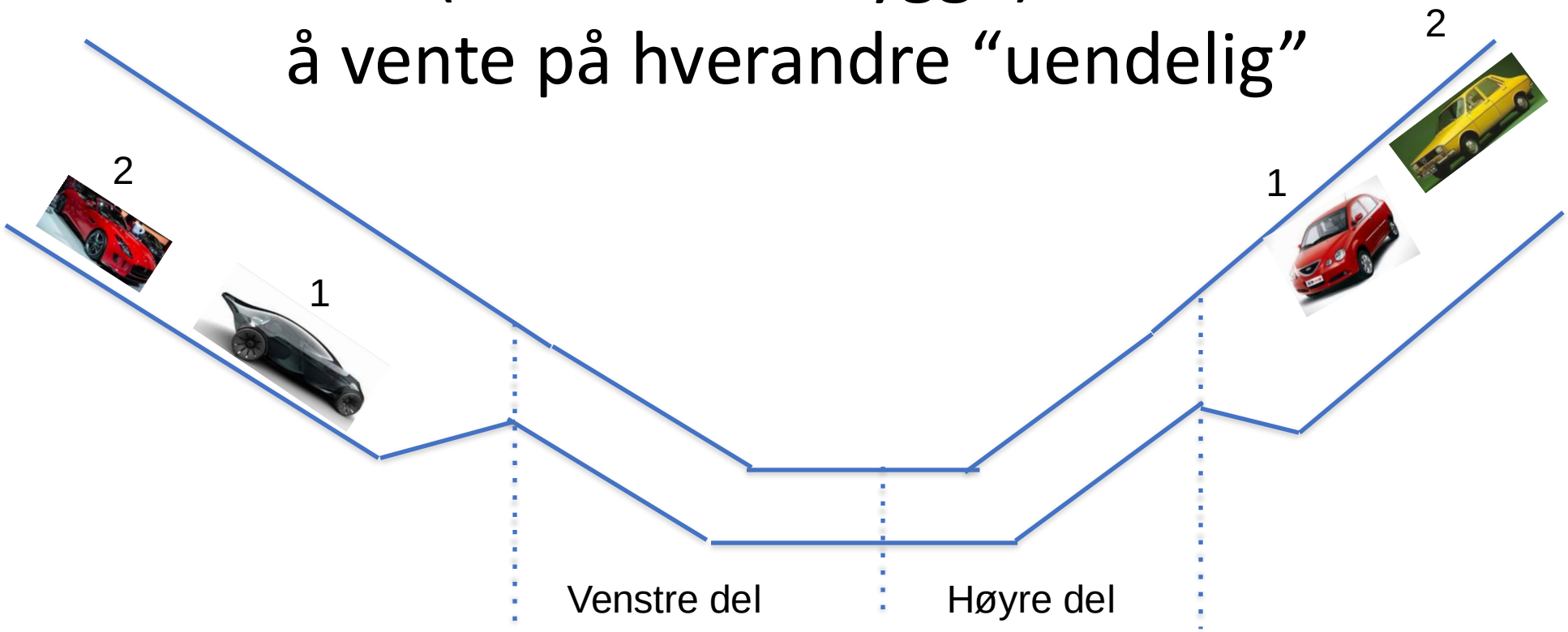
Tråd 1 ønsker å ta rP2 men må vente (i ledig.await())
Tråd 2 ønsker å ta rP1 men må vente (i ledig.await())





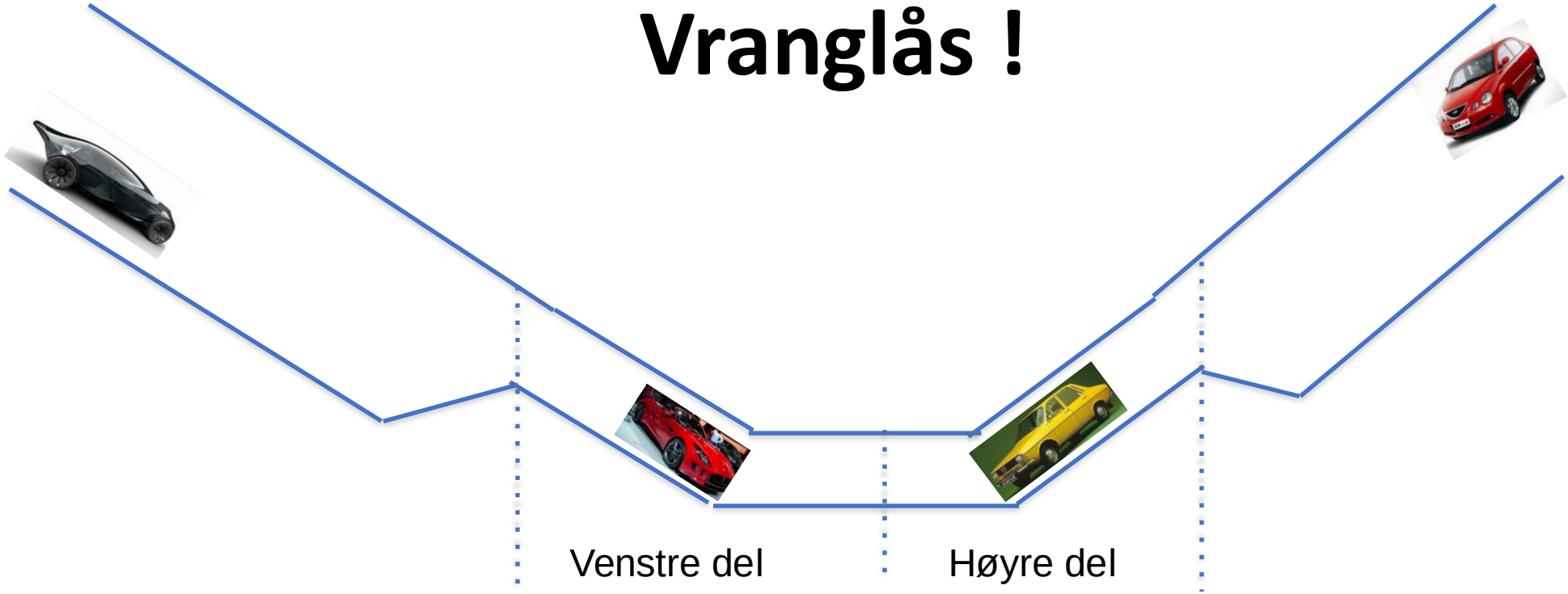
Samme eksempel på vranglås:

To biler (som ikke vil rygge) kan risikere å vente på hverandre "uendelig"



Smalt veistykke, to biler
kan ikke passere hverandre.
Bilene kan bare se den første delen.

Vranglås !

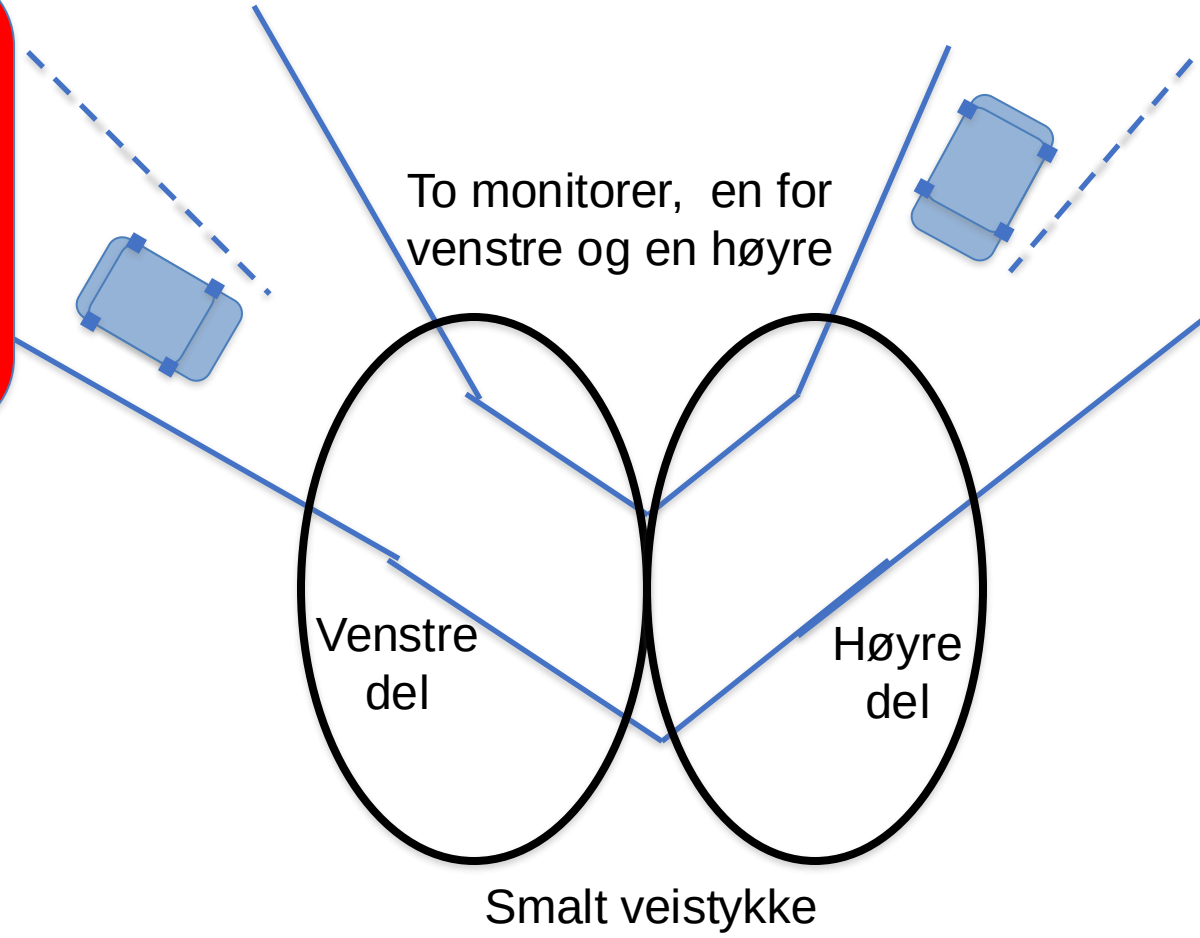


Det smale veistykket består av to ressurser, og begge bilene venter på at den andre bilen skal bli ferdig med å bruke sin ressurs
De har tatt ressursene i forskjellig rekkefølge!

Eksempel Smaltveistykke v 1.java

Se IN1010-siden

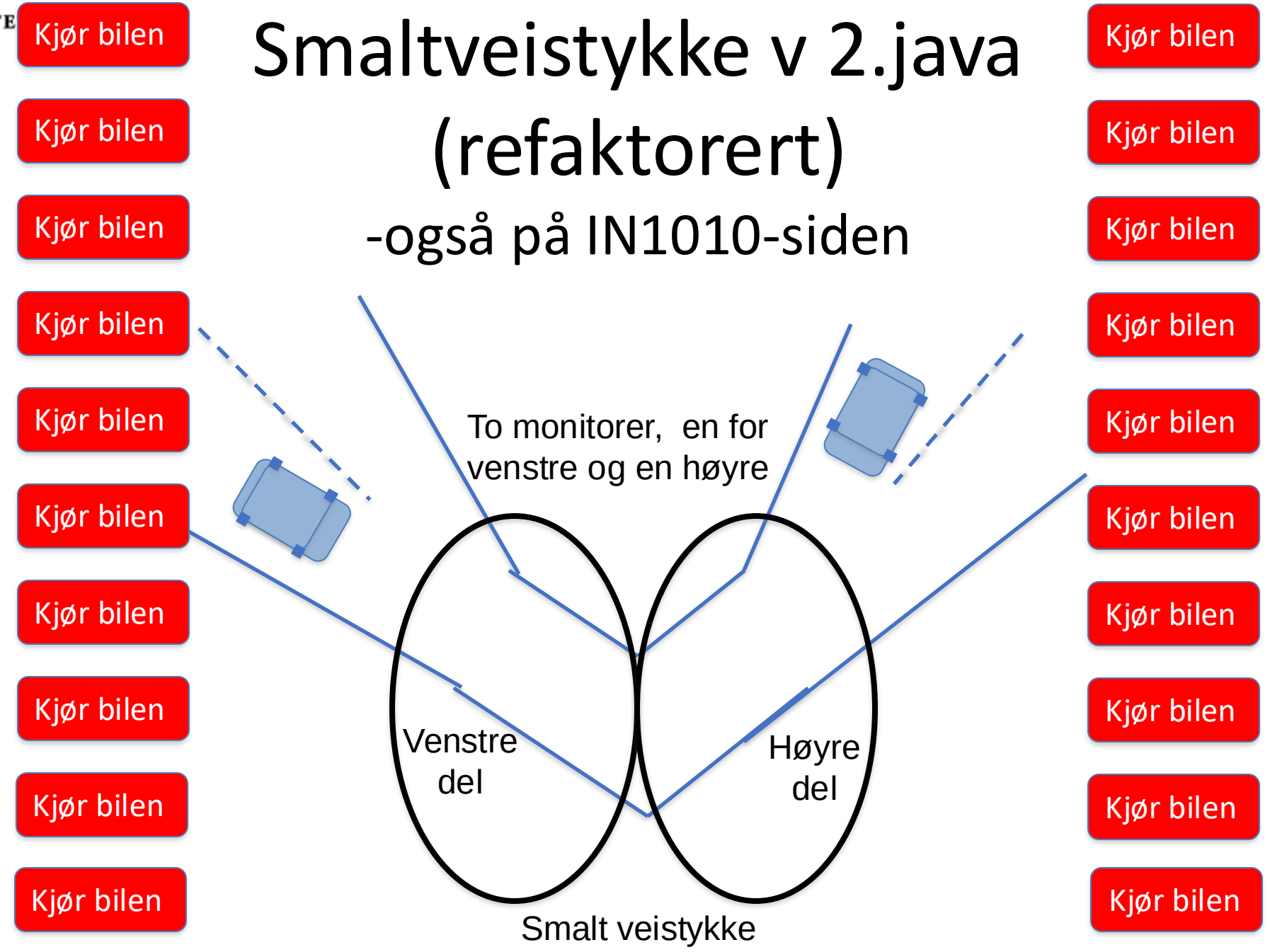
```
For (i=0 ; i < 10;  i++) {
   kjør bil fra
   venstre  til
   høyre;
   vent litt;
}
```




```
For (i=0 ; i < 10;  i++) {
   kjør bil fra
   høyre til
   venstre
   vent litt;
}
```

Smaltveistykke v 2.java (refaktorert)

-også på IN1010-siden



Disse to ukene har vi lært:

- Hvordan tråder kommuniserer seg imellom ved hjelp av monitorer
- Hvordan programmer (tråder) venter i en monitor
  - og starter opp igjen de som venter
  - Låsing og venting og signalering (på betingelser) skjer bare inne i monitorer.
- Hvordan tråder oppstår og kan vente på hverandre
- At det er utfordrende å programmere parallelle aktiviteter
  - Viktig å resonere om programmets tilstand.
    - Bruk invarianter
  - Nesten umulig å teste nok tilfeller
    - Tidsavhengige feil kan vise seg først etter mange år.
- Amdahls lov sier at mer parallellitet er bedre, og at synkronisering koster
- Vranglås kan forekomme i parallellprogrammering.
  - Vranglås forekommer særlig når ressurser tas i forskjellig rekkefølge

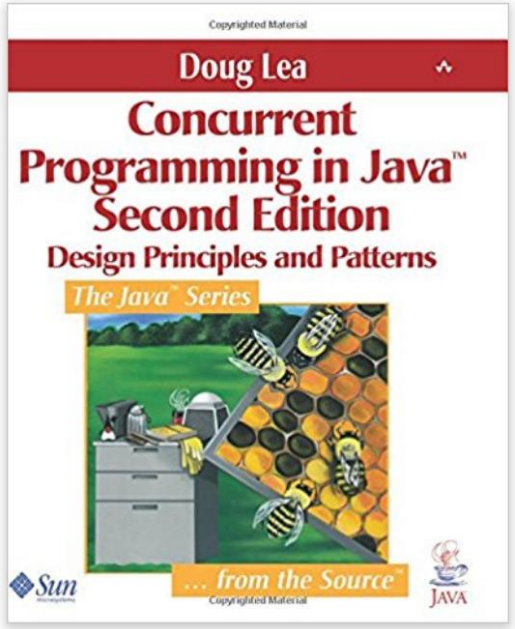
Ekstrastoff

(litt repetisjon fra ekstrastoffet forrige uke)

Monitorer i Java

- Opprinnelig ble Java laget med et innebygget monitor-begrep (ikke pensum i IN1010):
  - Alle objekter har en lås
  - synchronized foran en metode gjør metoden til en kritisk region (med hensyn på data i dette objektet)
  - wait() og signal() er metoder i class Object

- Doug Lea forbedret dette med java.util.concurrent-biblioteket



Andre alternativer for tråder og monitorer
Ikke pensum i IN1010

- Lage tråder ved hjelp av **subklasser** istedenfor delegering
  - Horstmann: Programming Tip 20.1
- Lage **monitorer** ved hjelp av **Javas originale mekanismer** istedenfor Doug Leas concurrent-bibliotek.
  - Horstmann: Special Topic 20.2
  - Se class Object (wait(), notify(), notifyAll()) og nøkkelordet synchronized
    - Alle objekter i Java har en iboende lås

Generelt om funksjonell programmering og parallellitet

- Funksjonell programmering betyr at all kode består av kall på funksjoner / metoder
- To paradigmer
  - Imperativ programmering (Python, Java, . . .)
    - Imperativ: Gjør noe, skritt for skritt
  - Funksjonell programmering (Lisp, . . .)
    - Kall en funksjon som gjør noe for deg
- Parallell programmering er naturlig når flere funksjoner kan evalueres samtidig
  - Ingen delte data ☺

- Kunklusjon: Funksjonell programmering egener seg godt for parallellitet

Parellellitet ved meldingsutvekling

- Et meget enkelt paradigme:
- Trådene / prosessene har ingen felles variable
- All kommunikasjon skjer ved å sende meldinger mellom prosessene/trådene

- Enkelt å programmere (?)
- Lettere å finne feil (?)
- Men neppe mer effektivt

- Eksempel: MPI - Message Passing Interface