IN3030 F09, våren 2024

Eric Jul
PT
Inst. for informatikk
UiO

Review F08v24

I. Moore's Law
II. Performance improvements in processor power
III. Speed of light
IV. Why distribution?
V. How to present your timing results
VI. Hva er PRAM modellen - og hvorfor er den ubrukelig for oss

Plan for F09

1. Synchronization revisited
2. Cooks and waiters
3. 3 solutions
4. Hoare Monitors

Oblig 3 Prime Numbers

Questions?

Problemet vi nå skal løse: En restaurant med kokker og kelnere og med et varmebord hvor maten står

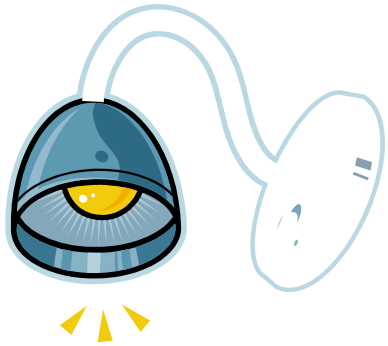
- Vi har **c** Kokker som lager mat og
  w Kelnere som server maten (tallerkenretter)
- Mat som kokkene lager blir satt fra seg på et **varmebord** (med TABLE_SIZE antall plasser til tallerkener)
- Kokkene kan ikke lage flere tallerkener hvis varmebordet er fullt
- Kelnerne kan ikke servere flere tallerkener hvis varmebordet er tomt
- Det skal lages og serveres NUM_TO_BE_MADE antall tallerkener

Restaurant versjon 1:

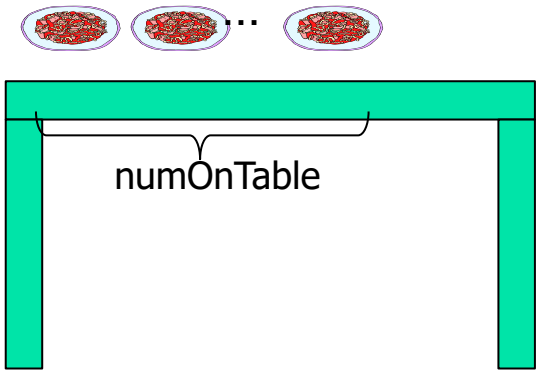
Lager tallerken-retter

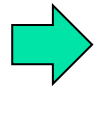
c Kokker (produsenter)

varmelampe



TABLE_SIZE





Serverer tallerken

w Kelnere (konsumenter)

3) Om monitorer og køer (tre eksempler på concurrent programming). Vi løser synkronisering mellom to ulike klasser.

- **Først** en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java).
  - Aktiv venting i en løkke i hver Kokk og Kelner
    + at de er i køen på å slippe inn i en synkronisert metode
- **Så** en løsning med bruk av monitor slik det var tenkt fra starten av i Java (Restaurant2.java).
  - Kokker og Kelnere venter i samme wait()-køen
    + i køen på å slippe inn i en synkronisert metode.
- **Til siste** en løsning med monitor med Lock og Condition-køer (flere køer – en per ventetilstand (Restaurant9.java)
  - Kelnere og Kokker venter i hver sin kø
    + i en køen på å slippe inn i de to metoder beskyttet av en Lock

.

Felles for de tre løsningene

```
import java.util.concurrent.locks.*;
class Restaurant {

  Restaurant(String[] args) {
     <Leser inn antall Kokker, Kelnere og  antall retter>
      <Oppretter Kokkene og Kelnerne og starter dem>
   }

    public static void main(String[] args) {
          new Restaurant(args);
       }
    } // end main
} // end class Restaurant

class HeatingTable{  // MONITOR
     int numOnTable  = 0,
         numProduced = 0,
         numServed=0;
     final int MAX_ON_TABLE  =3;
     final int NUM_TO_BE_MADE;
     // Invarianter:
     //   0 <= numOnTable  <=  MAX_ON_TABLE
     //          numProduced  <= NUM_TO_BE_MADE
     //          numServed      <= NUM_TO_BE_MADE

    < + ulike data i de tre eksemplene>
```

```
public xxx boolean putPlate(Kokk c)
   <Leggen tallerken til på bordet
   (true) ellers (false) Kokk må vente)
} // end put

public xxx boolean getPlate(Kelner w)  {
   <Hvis bordet tomt (false) Kelner venter
     ellers (true) - Kelner tar da en
     tallerken og serverer den>
  } // end get
} // end class HeatingTable

class Kokk extends Thread {
    HeatingTable tab;
    int ind;
    public void run() {
          while/if (tab.putPlate(..))
         < Ulik logikk i eksemplene>
    }
} // end class Kokk

class Kelner extends Thread {
    HeatingTable tab;
    int ind;
    public void run() {
          while/if (tab.getPlate()){
           <ulik logikk i eksemplene>
        }
} // end class Kelner
```

Invariantene på felles variable

- Invariantene må **alltid holde** (være sanne) utenfor monitor-metodene.
- Hva er de felles variable her:
  - MAX_ON_THE_TABLE
  - NUM_TO_BE_MADE
  - numOnTable
  - numProduced
  - numServed = numProduced – numOnTable
- Invarianter:
  1. **0 ≤ numOnTable ≤ TABLE_SIZE**
  2. **numProduced ≤ NUM_TO_BE_MADE**
  3. **numServed ≤ numProduced**

Invariantene viser 4 tilstander vi må ta skrive kode for

Invarianter:

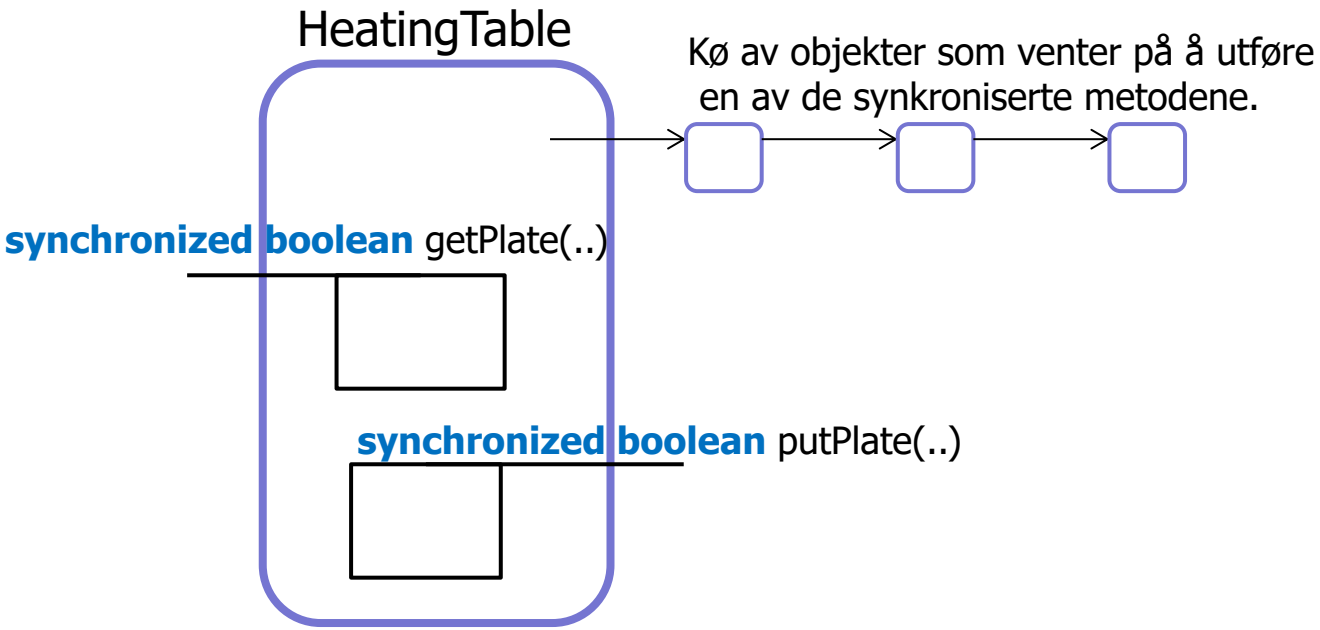
0 <= numOnTable <= MAX_ON_TABLE

numServed <= numProduced <= NUM_TO_BE_MADE

1. **numOnTable == MAX_ON_TABLE**
   → **Kokker venter**
2. **0 == numOnTable**
   → **Kelnere venter**
3. **numProduced == NUM_TO_BE_MADE**
   → **Kokkene ferdige**
4. **numServed == NUM_TO_BE_MADE**
   → **Kelnerene ferdige**

Først en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java).

- Dette er en løsning med **en kø**, den som alle tråder kommer i hvis en annen tråd er inne i en synkronisert metode i samme objekt.
- Terminering ordnes i hver kokk og kelner (i deres run-metode)
- Den køen som nyttes er en felles kø av alle aktive objekter som evt. samtidig prøver å kalle en av de to synkroniserte metodene **get** og **put.** Alle objekter har en slik kø.



Restaurant løsning 1

```
class Kokk extends Thread {
....
    public void run() {
       try {
           while (tab.numProduced < tab.NUM_TO_BE_MADE) {
              if (tab.putPlate(this) ) {
                 // lag neste tallerken
              }
              sleep((long) (1000 * Math.random()));
            }
       } catch (InterruptedException e) {}
       System.out.println("Kokk "+ind+" ferdig: " );
   }
} // end Kokk

class Kelner extends Thread {
.............
     public void run() {
           try {
             while ( tab.numServed< tab.NUM_TO_BE_MADE) {
                  if ( tab.getPlate(this)) {
                     // server tallerken
                  }
                  sleep((long) (1000 * Math.random()));
               }
           } catch (InterruptedException e) {}
             System.out.println("Kelner " + ind+" ferdig");
     }
} // end Kelner
```

```
synchronized boolean  putPlate(Kokk c) {
      if (numOnTable == TABLE_SIZE) {
                return false;
       }
       numProduced++;
      // 0 <= numOnTable  <  TABLE_SIZE
      numOnTable++;
      // 0 < numOnTable  <=  TABLE_SIZE
      System.out.println("Kokk no:"+c.ind+",
        laget  tallerken no:"+numProduced);
     return true;
} // end putPlate

synchronized boolean  getPlate(Kelner w) {
     if  (numOnTable == 0)     return false;
     // 0 < numOnTable  <=  TABLE_SIZE
     numServed++;
     numOnTable--;
     // 0 <= numOnTable <  TABLE_SIZE
     System.out.println("Kelner no:"+w.ind+
     ",  serverte  tallerken no:"+numServed);
    return true;
}
```

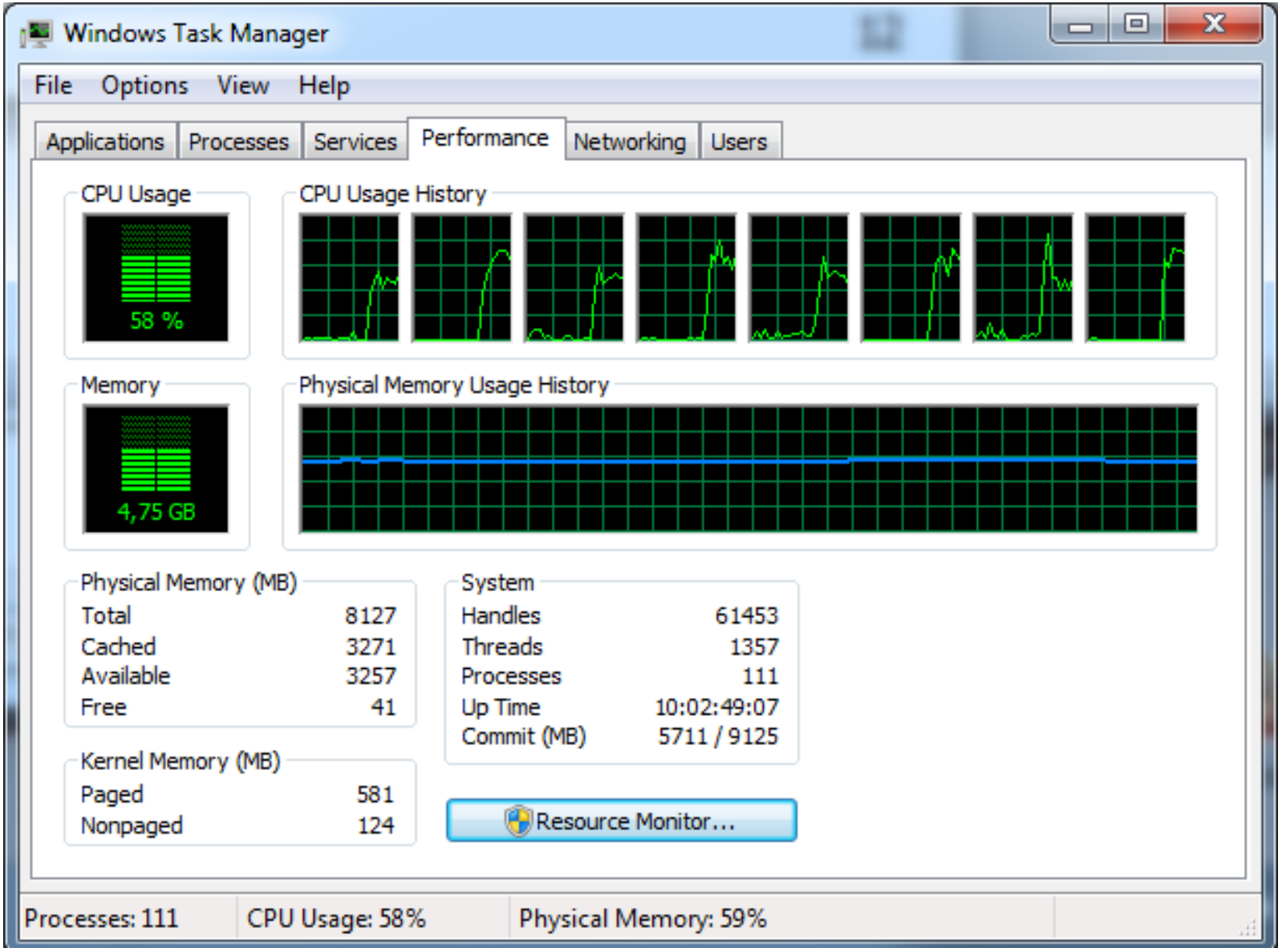
```
M:Restaurant1>java Restaurant1 11 8 8
Kokk no:8, laget  tallerken no:1
Kokk no:4, laget  tallerken no:2
Kokk no:6, laget  tallerken no:3
Kelner no:5, serverte  tallerken no:1
Kokk no:3, laget  tallerken no:4
Kelner no:2, serverte  tallerken no:2
Kokk no:1, laget  tallerken no:5
Kelner no:5, serverte  tallerken no:3
Kelner no:4, serverte  tallerken no:4
Kokk no:7, laget  tallerken no:6
Kokk no:2, laget  tallerken no:7
Kelner no:7, serverte  tallerken no:5
Kokk no:4, laget  tallerken no:8
Kelner no:3, serverte  tallerken no:6
Kelner no:3, serverte  tallerken no:7
Kokk no:1, laget  tallerken no:9
Kelner no:2, serverte  tallerken no:8
Kokk no:6, laget  tallerken no:10
Kokk no:3, laget  tallerken no:11
Kokk 8 ferdig:
```

```
Kelner no:8, serverte  tallerken no:9
Kelner no:7, serverte  tallerken no:10
Kelner no:6, serverte  tallerken no:11
Kokk 3 ferdig:
Kokk 5 ferdig:
Kelner 1 ferdig
Kokk 1 ferdig:
Kokk 4 ferdig:
Kelner 5 ferdig
Kokk 7 ferdig:
Kelner 2 ferdig
Kokk 2 ferdig:
Kelner 4 ferdig
Kelner 3 ferdig
Kelner 7 ferdig
Kelner 6 ferdig
Kokk 6 ferdig:
Kelner 8 ferdig
```

Problemer med denne løsningen er aktiv polling

- Alle Kokke- og Kelner-trådene går aktivt rundt å spør:
  - Er der mer arbeid til meg? Hviler litt, ca.1 sec. og spør igjen.
  - Kaster bort mye tid/maskininstruksjoner.
- Spesielt belastende hvis en av trådtypene (Produsent eller Konsument) er klart raskere enn den andre,
  - Eks . setter opp 18 raske Kokker som sover bare 1 millisek mot 2 langsomme Kelnere som sover 1000 ms.
  - I det tilfellet tok denne aktive ventingen/masingen 58% av CPU-kapasiteten til 8 kjerner
- Selv etter at vi har testet i run-metoden at vi kan greie en tallerken til, må vi likevel teste på om det går OK
  - En annen tråd kan ha vært inne og endret variable
- Utskriften må være i get- og put-metodene. Hvorfor?

Løsning1 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser maskinen med stadige mislykte spørsmål hvert ms. om det nå er plass til en tallerken på varmebordet . CPU-bruk = 58%.



Løsning 2: Javas originale opplegg med monitorer og **to køer**.

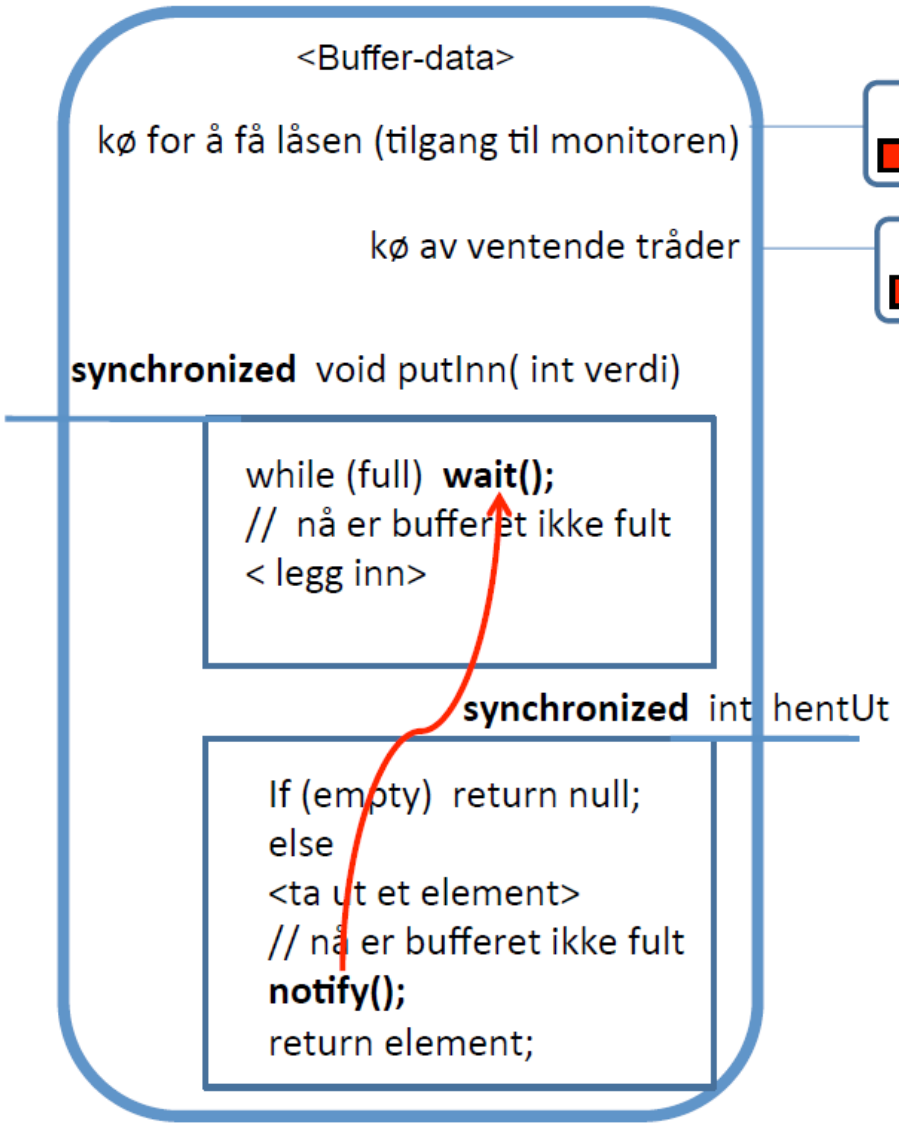
- Den originale Java løsningen med synkroniserte metoder og en rekke andre metoder og følgende innebygde metoder:
- **sleep(t):** Den nå kjørende tråden sover i 't' millisek.
- **notify():** (arvet fra klassen Object som alle er subklasse av). Den vekker opp **en** av trådene som venter på låsen i inneværende objekt. Denne prøver da en gang til å få det den ventet på.
- **notifyAll():** (arvet fra klassen Object). Den vekker opp **alle de** trådene som venter på låsen i inneværende objekt. De prøver da alle en gang til å få det de ventet på.
- **wait():** (arvet fra klassen Object). Får nåværende tråd til å vente til den enten blir vekket med notify() eller notifyAll() for dette objektet.

Å lage parallelle løsninger med en Java 'monitor'

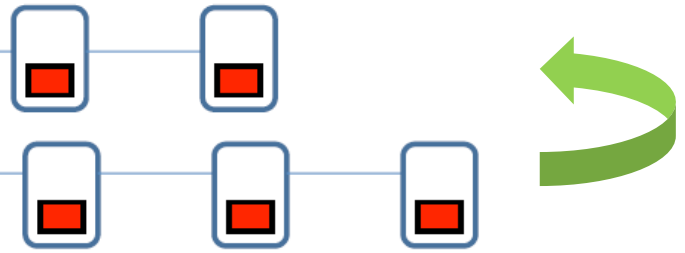
- En Java-monitor er et objekt av en vilkårlig klasse med synchronized metoder
- Det er to køer på et slikt objekt:
  - En kø for de som venter på å komme inn/fortsette i en synkronisert metode
  - En kø for de som her sagt **wait()** (og som venter på at noen annen tråd vekker dem opp med å si notify() eller notifyAll() på dem)
    - wait() sier en tråd inne i en synchronized metode.
    - notify() eller notifyAll() sies også inne i en synchronized metode.

Monitor-ideen er sterkt inspirert av Tony Hoare (mannen bak Quicksort)

To køer i en basal Java monitor:



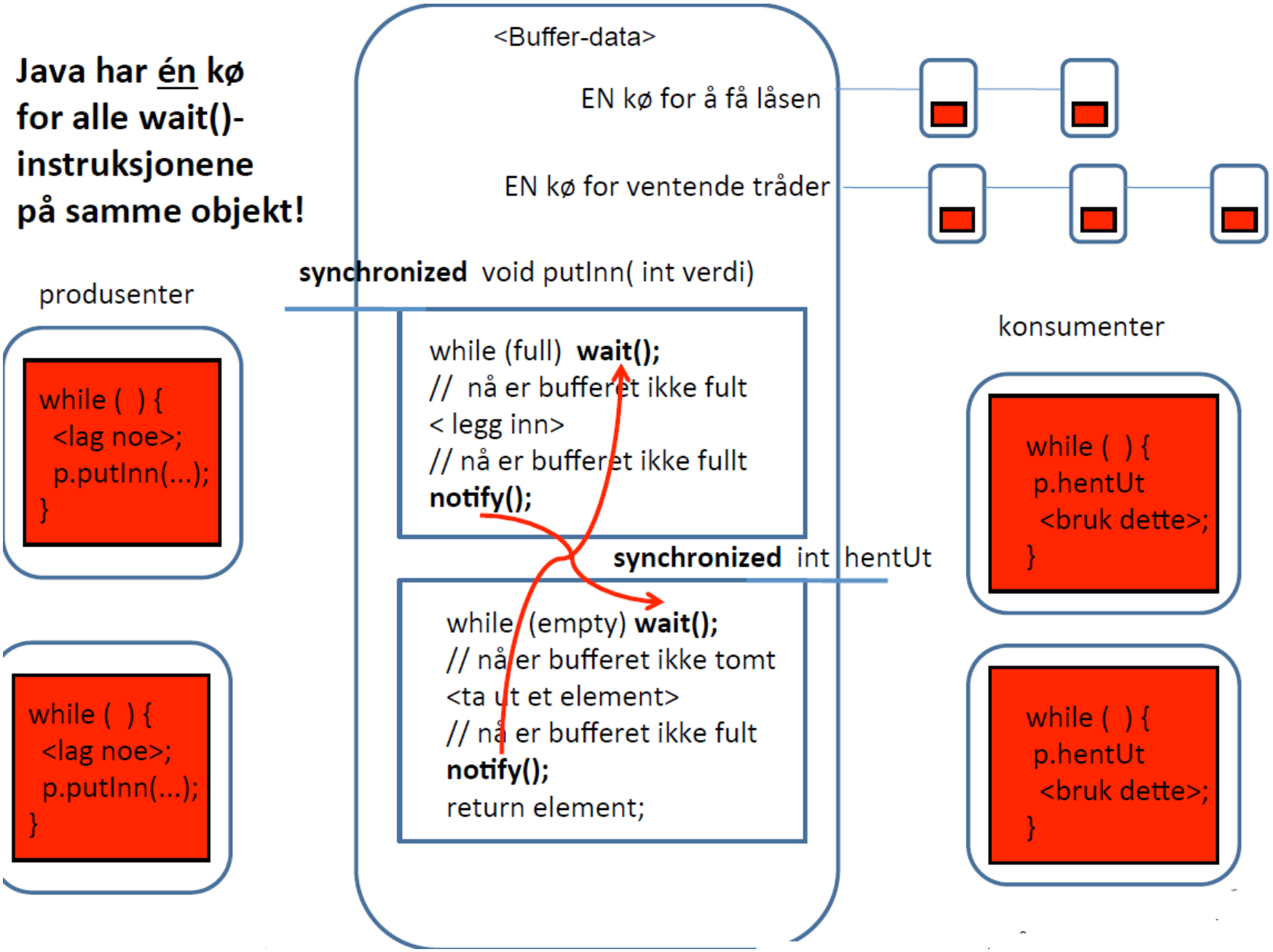
En kø av ventende tråder
på hele monitoren



notify()

En kø av ventende tråder på
"wait"-instruksjoner
(wait-set).
Startes av notify () og/eller
notifyAll()
Legges da i den andre køen
(først ? (Nei, ingen garanti))
Derfor er det nødvendig med
"while ..."

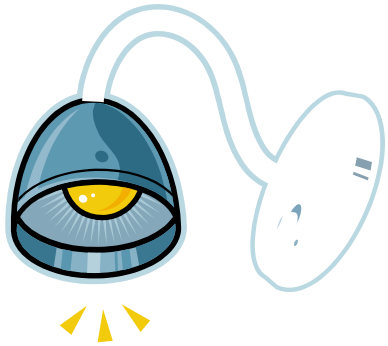
Java har <u>én</u> kø for alle wait()-instruksjonene på samme objekt!



Restauranten (2):

numProduced ≤ NOS_TO_BE_MADE

c Kokker (producers)



MAX_ON_THE_TABLE

numOnTable

numServed = = numProduced - numOnTable

w Kelnere (consumers)

En kø: Kokker og Kelnere venter samme wait-køen

Løsning 2, All venting er inne i synkroniserte metoder i en to køer.

- All venting inne i to synkroniserte metoder
- Kokker and Kelnere venter på neste tallerken i wait-køen
- Vi må vekke opp alle i wait-køen for å sikre oss at vi finner en av den typen vi trenger (Kokk eller Kelner) som kan drive programmet videre
- Ingen testing på invariantene i run-metodene

Begge løsninger 2) og 3):
run-metodene prøver en gang til hvis siste operasjon lykkes:

Kokker:

```
public void run() {
        try {
            while (tab.putPlate(this)) {
                sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) {}
        // Denne Kokken er ferdig
    }
```

Kelnere:

```
public void run() {
        try {
            while (tab.getPlate(this) ){
                sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) {}
          // Denne Kelneren er ferdig
    }
```

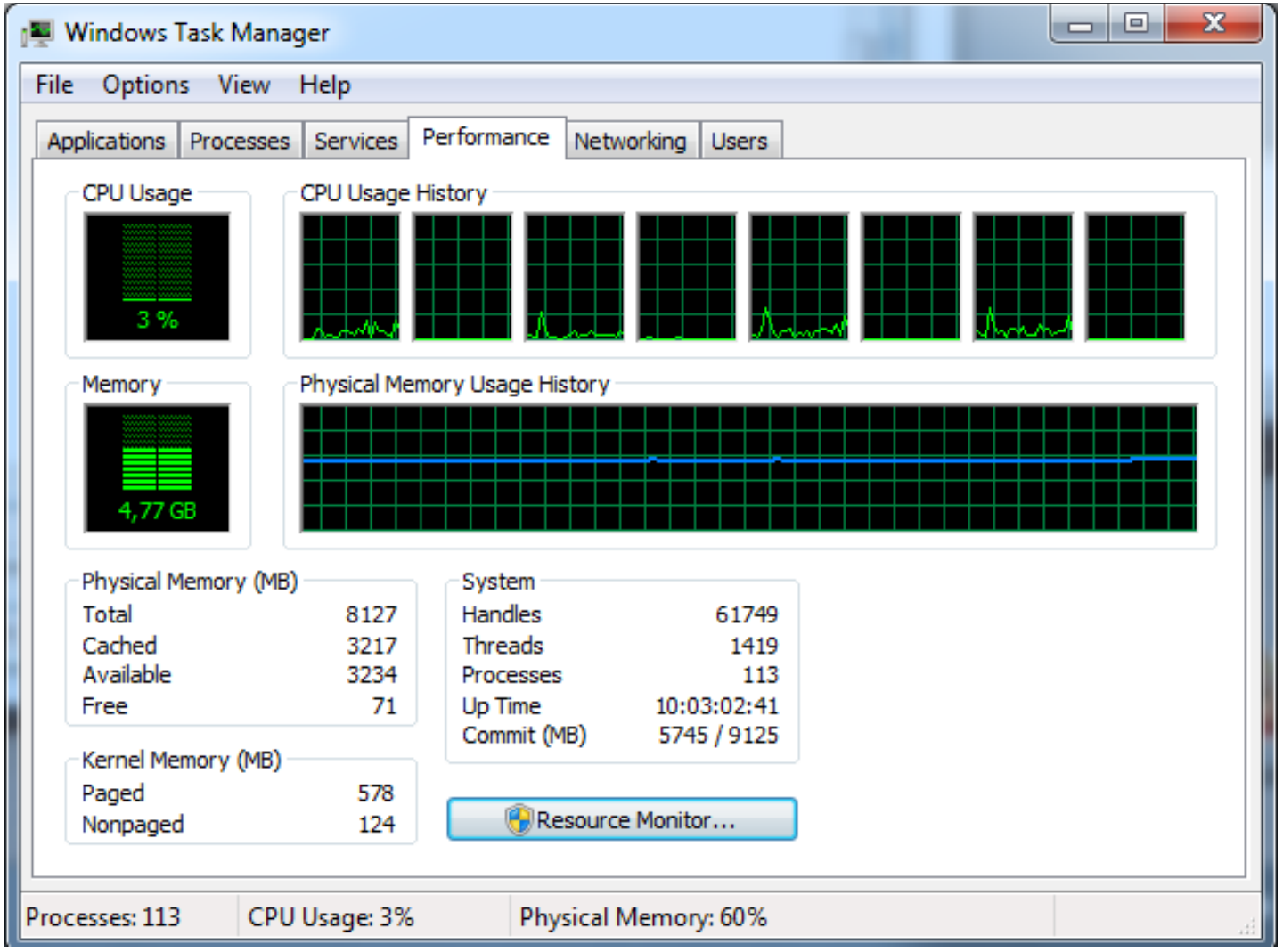
Løsning2

```
public synchronized boolean putPlate (Kokk c) {
    while (numOnTable == TABLE_SIZE &&
           numProduced <   NUM_TO_BE_MADE) {
        try { // The while test holds here meaning  that a Kokk should
              // but can not make a dish, because the table is full
            wait();
        } catch (InterruptedException e) { // Insert code to handle interrupt }
    }
    // one or both of the loop conditions are now false
    if (numProduced < NUM_TO_BE_MADE) {
        // numOnTable < TABLE_SIZE
        // Hence OK to increase numOnTable
        numOnTable++;
        // numProduced < NUM_TO_BE MADE
        // Hence OK to increase numProduced:
        numProduced++;
        // numOnTable > 0 , Wake up a waiting
        // waiter, or all if
        //numProduced ==NUM_TO_BE_MADE
        notifyAll();  // Wake up all waiting
        if (numProduced ==
            NUM_TO_BE_MADE) {
            return false;
        } else{ return true; }
    } else {
        // numProduced ==
        //NUM_TO_BE_MADE
        return false;}
} // end putPlate
```

Løsning2

```
public synchronized boolean getPlate (Kelner w) {
      while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
            try { // The while test holds here the meaning that the table
                  // is empty and there is more to serve
                  wait();
            } catch (InterruptedException e) { // Insert code to handle interrupt }
       }
      //one or both of the loop conditions are now false
      if (numOnTable > 0) {
            // 0 < numOnTable <= TABLE_SIZE
            // Hence OK to decrease numOnTable:
            numOnTable--;
            // numOnTable < TABLE_SIZE
            // Must wake up a sleeping Kokker:
            notifyAll();   // wake up all queued Kelnere and Kokker
            if (numProduced == NUM_TO_BE_MADE && numOnTable == 0) {
                  return false;
            }else{ return true;}
      } else { // numOnTable == 0 && numProduced == NUM_TO_BE_MADE
            return false;}
} // end getPlate
```

Løsning2 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser ikke maskinen med stadige mislykte spørsmål , men venter i kø til det er plass til en tallerken til på varmebordet . CPU-bruk = 3%.



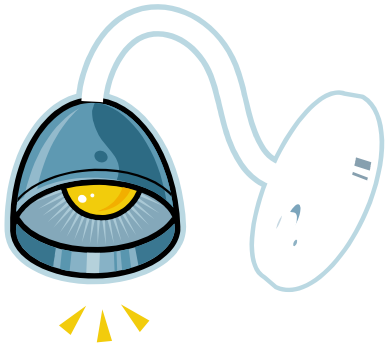
3) En parallell løsning med Conditions.

- Bruker to køer:
  - En for Kokker som venter på en tallerkenplass på bordet
  - En for Kelnere som venter på en tallerken
- Da trenger vi ikke vekke opp alle trådene, **Bare en** i den riktige køen.
  - Kanskje mer effektivt
  - Klart lettere å programmere

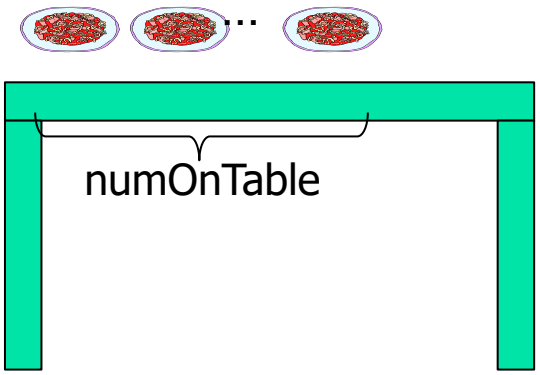
Restauranten (3):

numProduced ≤ NUM_TO_BE_MADE

c Kokker (produsenter)



MAX_ON_THE_TABLE



numServed = = numProduced – numOnTable

w Kelnere (konsumenter)

To køer:
a) Kokker venter på en plass på bordet

b) Kelnere venter på flere tallerkener

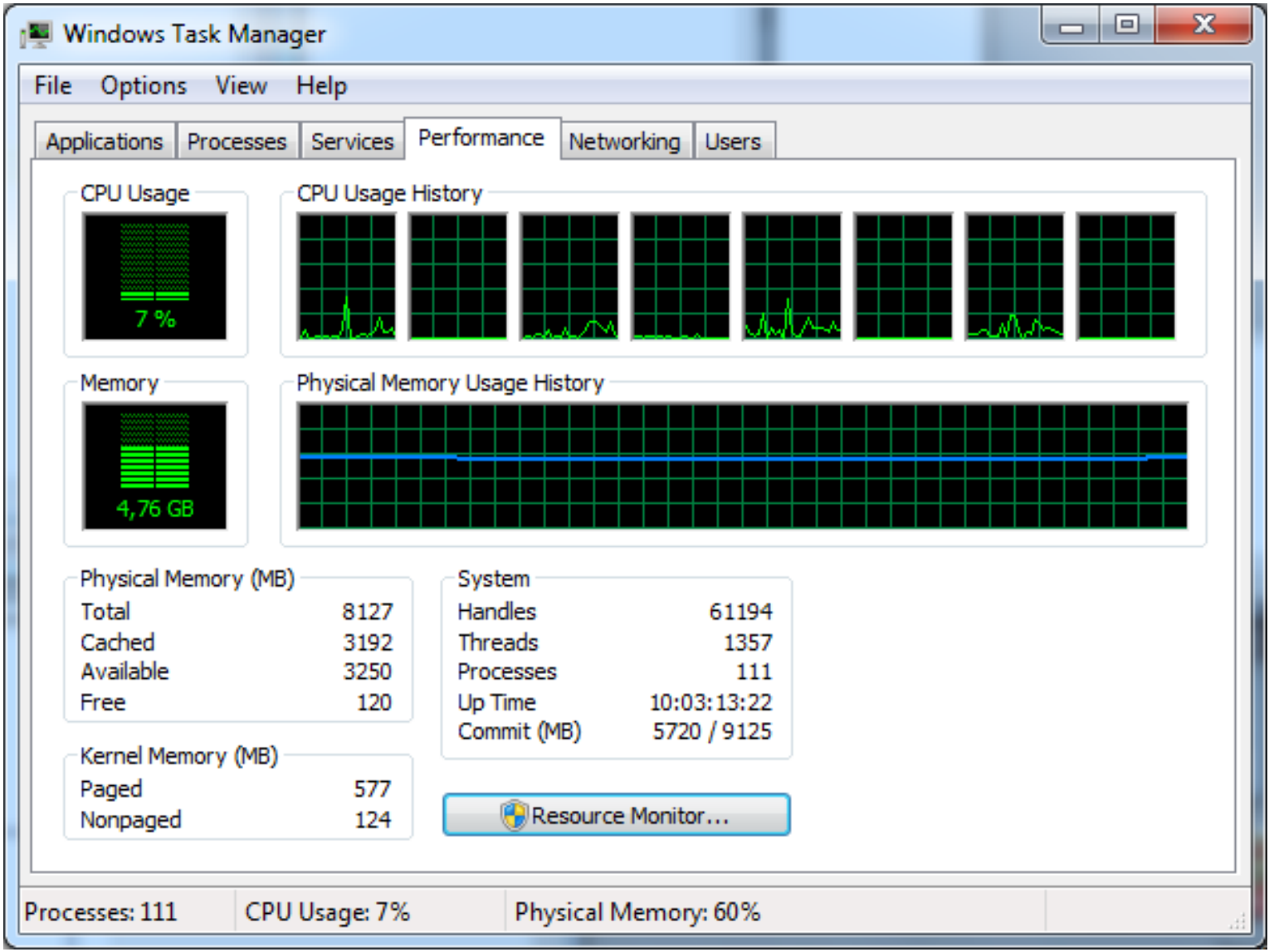
```
final Lock lock = new ReentrantLock();
final Condition notFull  = lock.newCondition();     // kø for Kokker
final Condition notEmpty = lock.newCondition(); // kø for Kelnere

public boolean putPlate (Kokker c) throws InterruptedException {
     lock.lock();
      try {while (numOnTable == MAX_ON_TABLE && numProduced < NUM_TO_BE_MADE){
                notFull.await(); // waiting for a place on the table
             }
            if (numProduced < NUM_TO_BE_MADE) {
                  numProduced++;
                  numOnTable++;;
                  notEmpty.signal(); // Wake up a waiting Kelner to serve
                  if (numProduced == NUM_TO_BE_MADE) {
                        //  I have  produced the last plate,
                       notEmpty.signalAll();   // tell Kelnere to stop waiting, terminate
                      notFull.signalAll();       // tell Kokker to stop waiting and terminate
                       return false;
                   }
                 return true;
             } else {  return false;}
      } finally {
         lock.unlock();
  }  } // end putPlate
```

```
public boolean getPlate (Kelnerw) throws InterruptedException {
    lock.lock();
    try {
        while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
            notEmpty.await();      // This Kelner is waiting for a plate
        }
        if (numOnTable > 0) {
            numOnTable--;
            notFull.signal();  // Signal to one Kokk in the Kokker's waiting queue
            return true;
        } else {
            return false;}
    } finally {
        lock.unlock();
    }
} // end getPlate
```

En Kelner eller en Kokk blir signalisert av to grunner:
- for å behandle (lage eller servere) en tallerken til
- ikke mer å gjøre, gå hjem (tøm begge køene)

Løsning3 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser ikke maskinen med stadige mislykte spørsmål , men venter i kø til det er plass til en tallerken til på varmebordet . CPU-bruk = 7%.



Vurdering av de tre løsningene

- **Løsning 1**: Enkel, men kan ta for mye av CPU-tiden. Særlig når systemet holder av andre grunner å gå i metning vil typisk en av trådene da bli veldig treige, og da tar denne løsningen plutselig ½-parten av CPU-tiden.
- **Løsning 2:** God, men vanskelig å skrive
- **Løsning 3:** God, nesten like effektiv som løsning 2 og lettere å skrive.

Avsluttende bemerkninger til Produsent-Konsument problemet

- Invarianter brukes av alle programmerere (ofte ubevisst)
  - program, loop or metode (sekvensiell eller parallell)
  - Å si dem eksplisitt hjelper på programmeringen
- HUSK: synchronized/lock virker bare når alle trådene synkroniserer på samme objektet.
  - Når det skjer er det **sekvensiell tankegang** mellom wait/signal
- Når vi sier notify() eller wait() på en kø, vet vi ikke:
  - Hvilken tråd som starter
  - Får den tråden det er signalisert på kjernen direkte etter at den som sa notify(), eller ikke ??. Ikke definert
- Debugging ved å spore utførelsen (trace) – System.out.println(”..”)
  - Skriving utenfor en Locket/synkronisert metode/del av metode, så lag en:
    - synchronized void println(String s) {System.out.println(s);}
  - Ellers kan utskrift bli blandet eller komme i gal rekkefølge.

Hoare Monitors

- The concept of Monitors was originally presented by Per Brinch Hansen who put the concept into his language Concurrent Pascal
- Brinch Hansen's Monitors were refined and presented in a nice, clean version in his seminal paper: *Monitors: An Operating System Structuring Concept* published in 1974.