INF2440 Uke 6, våren 2018

Eric Jul
PSE
Inst. for informatikk

Hva så vi på i uke 5 (kun første forelesningstime)

1. Enda bedre Matrise-multiplisering
2. Modell2-kode for sammenligning av kjøretider på (enkle) parallelle og sekvensielle algoritmer.
3. Hvordan lage en parallell løsning – ulike måter å synkronisere skriving på felles variable
4. Vranglås - et problem vi lett kan få (og lett unngå)

Gemt til uke 6:

1. Ulike strategier for å dele opp et problem for parallellisering
2. Litt om Oblig 3 ☺
3. Hvorfor lage en avkryssingstabell over alle oddetall for å finne alle primtall (Eratosthenes sil) – steg 1 i Oblig 3

Å dele opp algoritmen

- Koden består en eller flere steg; som oftest i form av en eller flere samlinger av løkker (som er enkle, doble, triple..)
- Vi vil parallellisere med k tråder, og hver slikt steg vil få hver sin parallellisering med en CyclickBarrier-synkronisering mellom hver av disse delene + en synkronisert avslutning (join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1) -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke

Å dele opp data – del 2

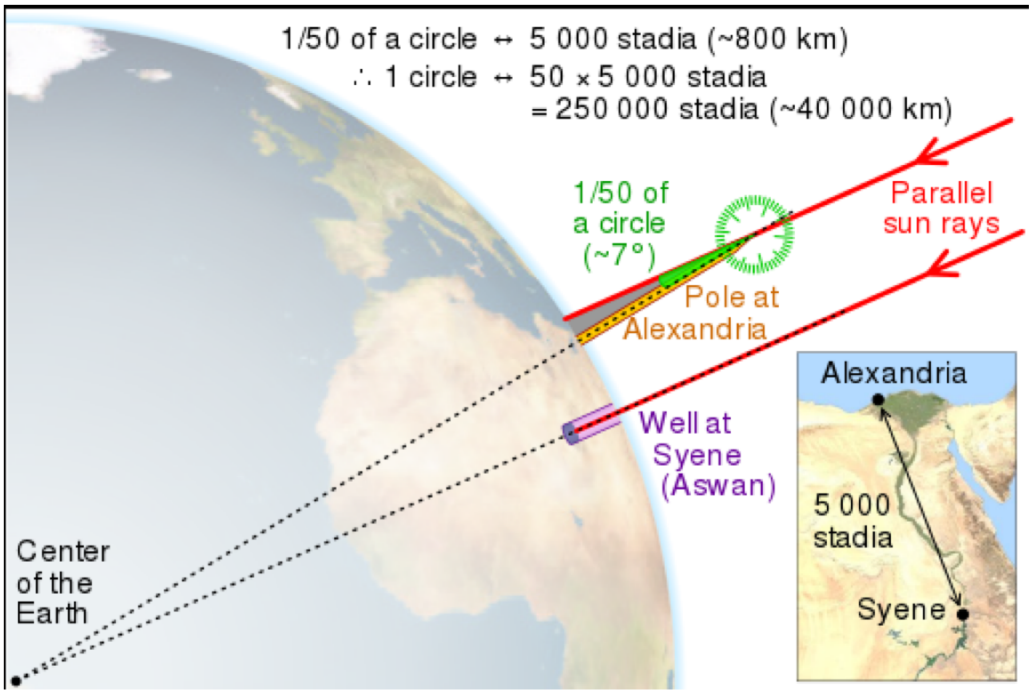
- For å planlegge parallellisering av ett slikt steg må vi finne:
  - Hvilke data i problemet er lokale i hver tråd?
  - Hvilke data i problemet er felles/delt mellom trådene?
- Viktig for effektiv parallell kode.
  - Hvordan deler vi opp felles data (om mulig)
  - Kan hver tråd beregne hver sin egen, disjunkte del av data
  - Færrest mulig synkroniseringer (de tar 'mye' tid)

5) Om primtall – og om Eratosthenes sil (oblig 3)

- Et primtall er :
  Et heltall som bare lar seg dividere med 1 og seg selv.
  - 1 er ikke et heltall (det mente mange på 1700-tallet, og noen mener det fortsatt)
- Ethvert tall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig; dvs. den eneste faktoriseringen av N (unntagen rekkefølge – entydig hvis sortert!)
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall.

Litt mer om Eratosthenes

Eratosthenes, matematikker, laget også et estimat på jordas radius som var < 15% feil, grunnla geografi som fag, fant opp skuddårsdagen og at han var sjef for Biblioteket i Alexandria (den tids største forskningsinstitusjon).



2 måter å lage primtallene op til N

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

- Dividere alle tall < N med alle oddetall < $\sqrt{N}$
  - Divisjonsmetoden

Hvordan lage og lagre primtall

- A) Med Eratosthenes sil:

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa     18 949 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

- Med gjentatte divisjoner

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa   1 577 302  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle oddetall < SQRT(p) p = 2,3,4,..) er ca. 100 ganger langsommere enn Eratosthenes avkryssings-tabell (kalt Eratosthenes sil).

Å lage og lagre primtall (Erotosthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i jernalderen av Eratosthenes (ca. 200 f.kr)
  - Man skal finne alle primtall < M
  - Man finner da de første primtallene og krysser av alle multipla av disse (N.B. dette forbedres/endres senere):
    - Eks: 3 er et primtall, da krysses 6, 9,12,15,.. Av fordi de alle er ett-eller-annet-tall (1,2,3,4,5,..) ganger 3 og følgelig selv ikke er et primtall. 6=2*3, 9 = 3*3,
      12 =2*2*3, 15 = 5*3, ..osv
    - De tallene som *ikke blir* krysset av , når vi har krysset av for alle primtallene vi har, er primtallene
- Vi finner 5 som et primtall fordi, etter at vi har krysset av for 3, finner første ikke-avkryssete tall: 5, som da er et primtall (og som vi så krysser av for, …finner så 7 osv)

Litt mer om Eratothenes sil

- Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
- Har vi funnet et nytt primtall p, for eksempel. 5, starter vi avkryssingen for dette primtallet først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall t < p*p slik det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p.
- Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare å krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer svært mye tid.

Vise at vi trenger bare primtallene <10 for å finne alle primtall < 100, avkryssing for 3 (3*3, 9+2*3,9+4*3, ….)

1	**3**	**5**	**7**	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	**3**	**5**	**7**	**9**
11	13	**15**	17	19
21	23	25	**27**	29
31	**33**	35	37	**39**
41	43	**45**	47	49
51	53	55	**57**	59
61	**63**	65	67	**69**
71	73	**75**	77	79
81	83	85	**87**	89
91	**93**	95	97	**99**

Avkryssing for 5 (starter med 25, så 25+2*5, 25+4,5,..):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Avkryssing for 7 (starter med 49, så 49+2*7,49+4*7,.):

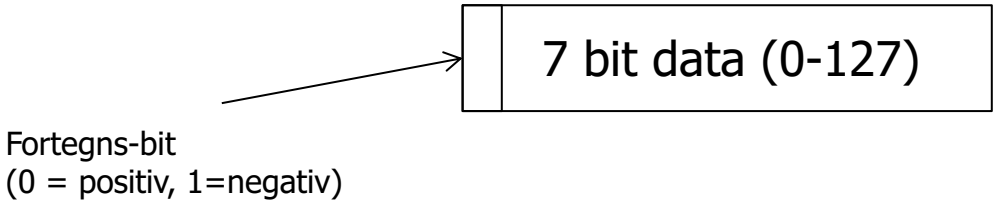
1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63 63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Er nå ferdig fordi neste primtall vi finner: 11, så er 11*11=121 utenfor tabellen

1) Hvordan bruke 8 eller 7 bit i en **byte-array** for å representere primtallene

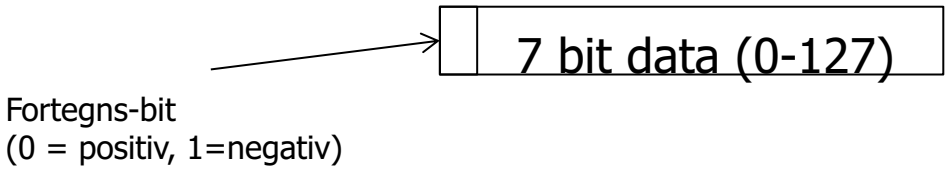
En byte = 8 bit heltall:



- Vi representer alle oddetallene (1,3,5,,,) som ett bit (0= ikke-primtall, 1 = primtall)
- Bruke alle 8 bit :
  - Fordel: mer kompakt lagring og litt raskere(?) adressering
  - Ulempe: Kan da ikke bruke verdien i byten direkte (f.eks som en indeks til en array), heller ikke +,-,* eller /-operasjonene på verdien
- Bruke 7 bit:
  - Fordel: ingen av ulempene med 8 bit
  - Ulempe: Tar litt større plass og litt langsommere(?) adressering

1) Hvordan representere 8 (eller 7) bit i en byte-array

byte = et 8 bit heltall



- Bruker alle 8 bitene til oddetallene:
  - Anta at vi vil sjekke om tallet ***k*** er et primtall, sjekk først om ***k*** er 2, da ja, hvis det er et partall (men ikke 2) da nei – ellers sjekk så tallets bit i byte-arrayen
    - Byte nummeret til ***k*** i arrayen er da:
      - Enten: ***k***/16, eller: ***k*** >>>4 (shift 4 høyreover uten kopi av fortegns-bitet er det samme som å dele med 16)
    - Bit-nummeret er i denne byten er da enten (***k*** %16)/2 eller (***k*** &15)>>1
  - Hvorfor dele på 16 når det er 8 bit
    - fordi vi fjernet alle partallene – egentlig 16 tall representert i første byten, for byte 0: tallene 0-15
  - Om så å finne bitverdien – se neste lysark.

Bruke 7 bit i hver byte i arrayen

- Anta at vi vil sjekke om tallet ***k*** er et primtall sjekk først om ***k*** er 2, da ja, ellers hvis det er et partall (men ikke 2) da nei – ellers:
- Sjekk da tallets bit i byte-arrayen
  - Byte nummeret til ***k*** i arrayen er da: ***k*** /14
  - Bit-nummeret er i denne byten er da: (***k*** %14)/2
- Nå har vi byte nummeret og bit-nummeret i den byten. Vi kan da ta AND (&) med det riktige elementet i en av de to arryene som er oppgitt i skjelett-koden og teste om svaret er 0 eller ikke.
- Hvordan sette alle 7 eller 8 bit == 1 i alle byter)
  - 7 bit: hver byte settes = 127 (men bitet for 1 settes =0)
  - 8 bit: hver byte settes = -1 (men bit for 1 settes = 0)

- Konklusjon: bruk 8 eller 7 bit i hver byte (valgfritt) i Oblig3

2) Faktorisering av et tall M i sine primtallsfaktorer

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle (unntatt 2) primtall < N i en bit-array over alle odde-tallene.
  - 1 = primtall, 0=ikke-primtall
  - Vi har krysset ut de som ikke er primtall
- Hvordan skal vi så bruke dette til å faktorisere et tall M < N*N ?
- **Svar:** Divider M med alle primtall $p_i < \sqrt{M}$ (p_i =2,3,5,..), og hver gang en slik divisjon $M\%p_i == 0$, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere ett mindre tall $M' = M/p_i$.
- Faktoriseringen av $M = p_i * .. * p_k$ er da produktet av alle de primtall som dividerer M uten rest.
- HUSK at en p_i kan forekommer flere ganger i svaret. eks: 20= 2*2*5, 81 = 3*3*3*3, osv
- Finner vi ingen faktorisering av M, dvs. ingen $p_i \leq \sqrt{M}$ som dividerer M med rest == 0, så er M selv et primtall.

Hvordan parallellisere faktorisering ?

1. Gjennomgås neste uke - denne uka viktig å få på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2 000 000
Genererte primtall <= 2000000  paa        15.56 millisek
med Eratosthenes sil (  0.00004182 millisek/primtall)
......
3999998764380 = 2*2*3*5*103*647248991
3999998764381 = 37*108108074713
3999998764382 = 2*271*457*1931*8363
3999998764383 = 3*19*47*1493093977
3999998764384 = 2*2*2*2*2*7*313*1033*55229
3999998764385 = 5*13*59951*1026479
3999998764386 = 2*3*3*31*71*100964177
3999998764387 = 1163*1879*1830431
3999998764388 = 2*2*11*11*17*23*293*72139
 100 faktoriseringer beregnet paa:     422.0307ms -
 dvs:      4.2203ms. per faktorisering
```

Faktorisering av store tall med 18-19 desimale sifre

```
Uke5>java PrimtallESil 2140000000
max primtall m:2 140 000 000
bitArr.length:133 750 001
Genererte primtall <= 2 140 000 000  paa    11030.36 millisek
med Eratosthenes sil ( 0.00010530 millisec/primtall)
 antall primtall < 2 140 000 000 er: 104 748 779, dvs:  4.89% ,
 og det største primtallet er: 2 139 999 977

4 579 599 999 999 999 900 = 2*2*3*5*5*967*3673*19421*221303
4 579 599 999 999 999 901 = 45795999999999901
4 579 599 999 999 999 902 = 2*22897999999999951
4 579 599 999 999 999 903 = 3*31*13188589*3733758839
4 579 599 999 999 999 904 = 2*2*2*2*2*19*71*106087842846553
4 579 599 999 999 999 905 = 5*7*1308457142857142 83
.............
4 579 599 999 999 999 997 = 11*4163272727272727
4 579 599 999 999 999 998 = 2*121081*18911307306679
4 579 599 999 999 999 999 = 3*17*19*6625387*713333333
100 faktoriseringer beregnet paa:  333481.4427ms
 dvs:   3334.8144ms. per faktorisering

 largestLongFactorizedSafe: 4 579 599 841 640 001 173= 2139999949*2139999977
```

Hva har vi sett på i uke 5 – annen forelesning

1. Ulike strategier for å dele opp et problem for parallellisering:
2. Hvorfor lage en avkryssingstabell over alle oddetall for å finne alle primtall (Eratosthenes sil) – steg 1 i Oblig 2

Dette skal vi se på i uke 6

1. Hva er raskest: Modell2 eller Modell3 – kode?
2. Kort om noen mulige måter å parallellisere Eratosthenes Sil og faktoriseringen i Oblig2 (mer neste uke).
3. **Concurrency:** Tre måter å programmere monitorer i Java eksemplifisert med tre løsninger av problemet: Kokker og Kelnere (eller generelt: produsenter og konsumenter)
    1. med sleep()og aktiv polling.
    2. med synchronized methods , wait>() og notify(),..
    3. med Lock og flere køer (Condition-køer)

2) Kortfattet om parallellisere Eratosthenes Sil (mer neste uke)

- Vi har M store tall som skal faktoriseres og skal nå lage n = $\sqrt{M}$ primtall
- Del opp noe:
  a. Tall-linjen (oddetallene) vi skal krysse av med –dvs. tallene mellom 3 og $\sqrt{M}$ i k deler (like store?) – kryss av hver del med **alle** primtallene vi har
     a. Men vi starter jo med bare to primtall :2 og 3 - alle?
  b. Dele opp primtallene vi skal krysse av med – f.eks
     a. Hver tråd tar neste primtall vi har funnet (3,5,..)
     b. Vi deler opp de første primtallene til tråd0, neste til tråd1,..
- Last-ballansering ?
  - Like mye arbeid til hver tråd – gir god speedup
- Er alle disse alternativene riktige ? (to skriver samtidig?)
- Skal noe kopieres lokalt og så samles når trådene er ferdige?

Modell3 10 10 2000000000 11 TEST AV i++ med synchronized oppdatering som eksempel (8 kjerner , og 8 traader, Median av:11 iterasjoner)

n	sekv.tid(ms)	para.tid(ms)	Speedup
2000000000	34.325	20.824	1.6484
200000000	3.437	2.746	1.2519
20000000	0.360	0.913	0.3945
2000000	0.036	0.422	0.0854
200000	0.004	0.385	0.0094
20000	0.001	0.421	0.0014
2000	0.000	0.444	0.0007
200	0.000	0.416	0.0007
20	0.000	0.419	0.0007

Konklusjon: Lage og starte tråder hver gang (Modell3) tar ca **0.40 ms mer tid** enn å starte de en gang for alle og ha 2 CyclicBarrier for start/stop (Modell2)

Modell2 Samme test :

n	sekv.tid(ms)	para.tid(ms)	Speedup
2000000000	34.384	20.540	1.6740
200000000	3.470	2.373	1.4623
20000000	0.404	0.453	0.8907
2000000	0.034	0.074	0.4615
200000	0.004	0.052	0.0690
20000	0.000	0.050	0.0061
2000	0.000	0.050	0.0060
200	0.000	0.053	0.0056
20	0.000	0.048	0.0000

3) Flere (synkroniserings-) metoder i klassen Thread.

- **getName()** Gir navnet på tråden (default: Thread-0, Thread-1,..)
- **join():** Du venter på at en tråd terminerer hvis du kaller på dens join() metode.
- **sleep(t):** Den nå kjørende tråden sover i minst 't' millisek.
- **yield():** Den nå kjørende tråden pauser midlertidig og overlater til en annen tråd å kjøre på den kjernen som den første tråden ble kjørt på..
- **notify():** (arvet fra klassen Object, som alle er subklasse av). Den vekker opp **én** 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.

Et program som tester join(),yield() og getname() :

```
import java.util.ArrayList;

public class YieldTest {
    public static void main(String args[]){
        ArrayList<YieldThread> list = new ArrayList<YieldThread>();
        for (int i=0;i<20;i++){
            YieldThread et = new YieldThread(i+5);
            list.add(et);
            et.start();
        }
        for (YieldThread et:list){
            try {
                et.join();
            } catch (InterruptedException ex) { }
}}} // end class YielsTest

class YieldThread extends Thread{
   int stopCount;
    public YieldThread(int count){
        stopCount = count;
    }
    public void run(){
        for (int i=0;i<30;i++){
            if (i%stopCount == 0){
                System.out.println(«Stopper thread: "+getName());
                yield();
            }
}}} // end class YieldThread
```

```
M:\INF2440Para\Yield>java YieldTest
Stopper thread: Thread-0
Stopper thread: Thread-0
Stopper thread: Thread-5
Stopper thread: Thread-5
Stopper thread: Thread-9
Stopper thread: Thread-9
Stopper thread: Thread-12
Stopper thread: Thread-4
Stopper thread: Thread-3
Stopper thread: Thread-3
Stopper thread: Thread-2
Stopper thread: Thread-2
Stopper thread: Thread-2
Stopper thread: Thread-1
Stopper thread: Thread-1
Stopper thread: Thread-1
Stopper thread: Thread-3
Stopper thread: Thread-19
Stopper thread: Thread-18
Stopper thread: Thread-18
Stopper thread: Thread-17
Stopper thread: Thread-16
Stopper thread: Thread-16
```

Vi ser at de 20 trådene gir fra seg kontrollen et ulike antall ganger > 0

```
Stopper thread: Thread-4
Stopper thread: Thread-4
Stopper thread: Thread-15
Stopper thread: Thread-12
Stopper thread: Thread-14
Stopper thread: Thread-14
Stopper thread: Thread-13
Stopper thread: Thread-13
Stopper thread: Thread-11
Stopper thread: Thread-11
Stopper thread: Thread-10
Stopper thread: Thread-8
Stopper thread: Thread-7
Stopper thread: Thread-6
Stopper thread: Thread-0
Stopper thread: Thread-6
Stopper thread: Thread-10
Stopper thread: Thread-15
Stopper thread: Thread-17
Stopper thread: Thread-19
Stopper thread: Thread-7
Stopper thread: Thread-8
```

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

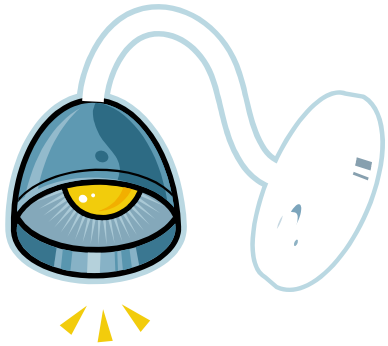
- Vi har **c** Kokker som lager mat og
  w Kelnere som server maten (tallerkenretter)
- Mat som kokkene lager blir satt fra seg på et **varmebord** (med TABLE_SIZE antall plasser til tallerkener)
- Kokkene kan ikke lage flere tallerkener hvis varmebordet er fullt
- Kelnerne kan ikke servere flere tallerkener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

Restaurant versjon 1:

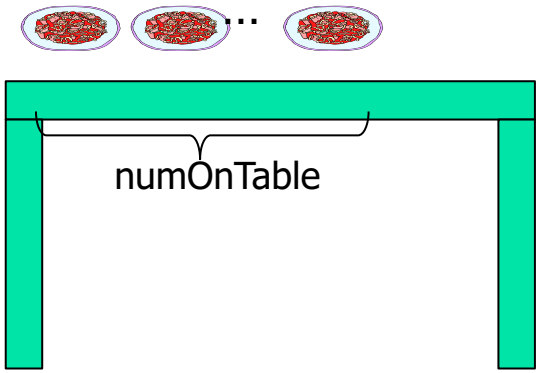
Lager tallerken-retter

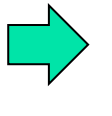
c Kokker (produsenter)

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:

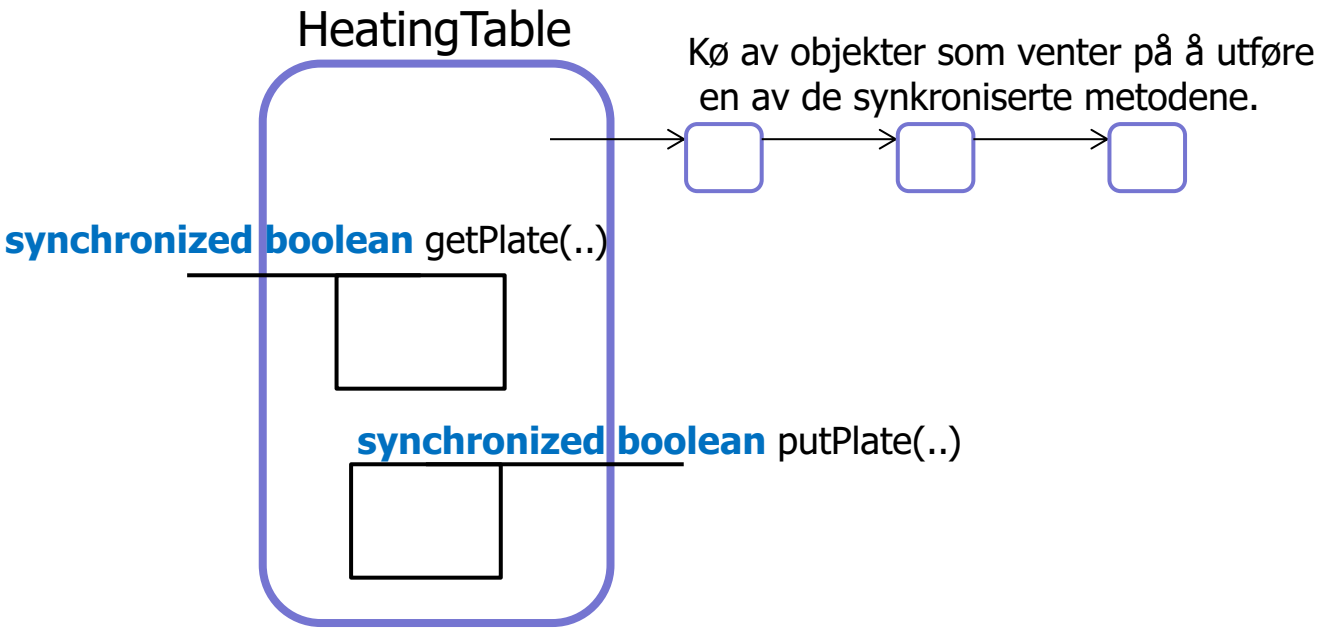
0 <= numOnTable <= MAX_ON_TABLE

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;
}
```

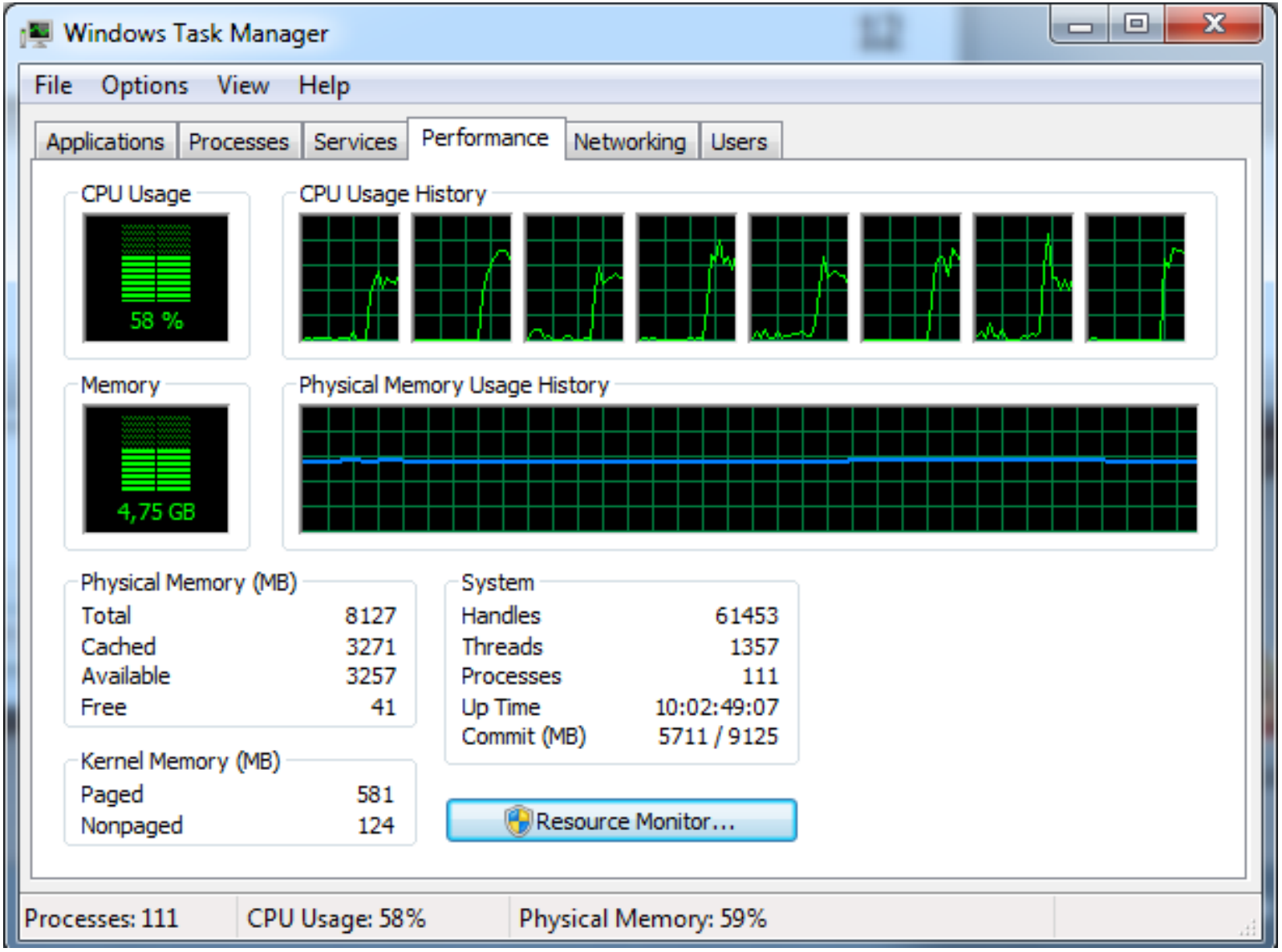
```
M:Restaurant1>java Restaurant1 11 8 8
Kokk no:8, laget  tallerken no:1
Kokk no:4, laget  tallerken no:2
Kokk no:6, laget  tallerken no:3
Kelner no:5, serverte  tallerken no:1
Kokk no:3, laget  tallerken no:4
Kelner no:2, serverte  tallerken no:2
Kokk no:1, laget  tallerken no:5
Kelner no:5, serverte  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**.

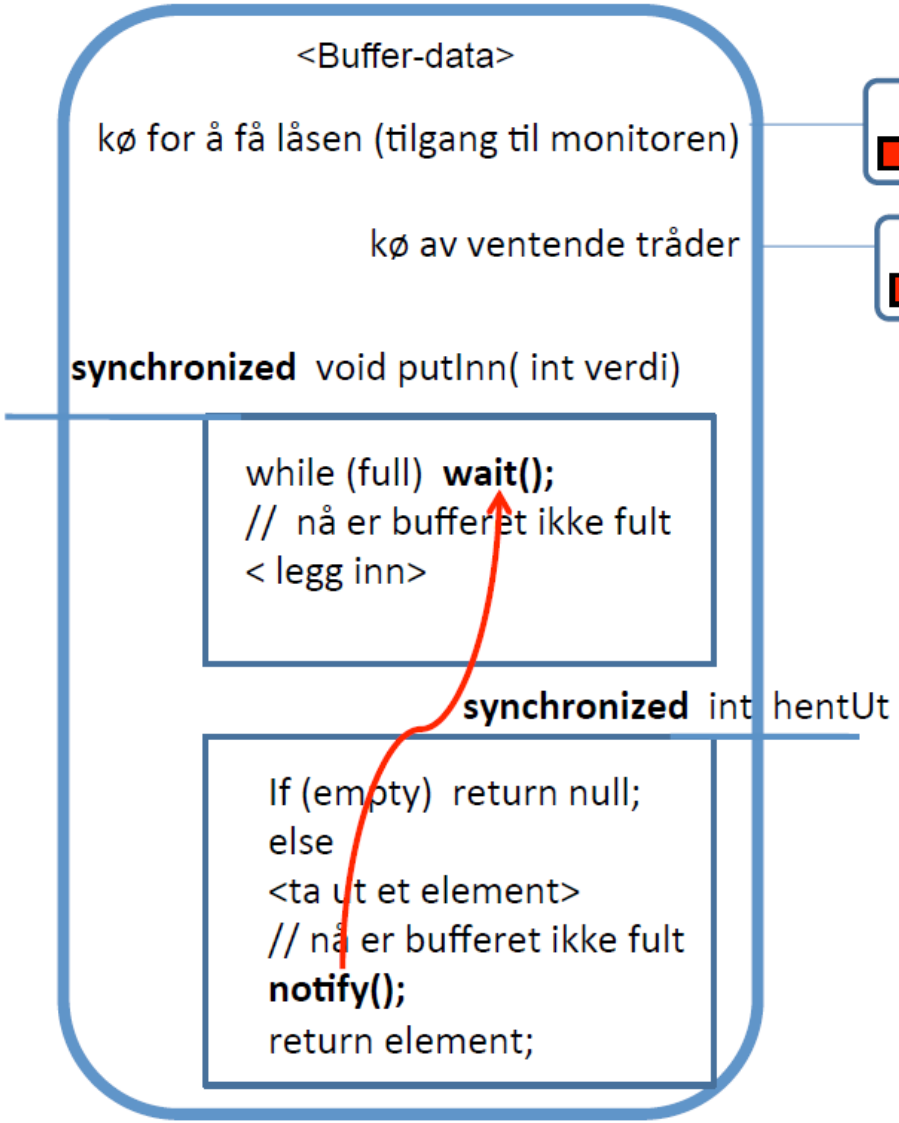
- Den originale Java løsningen med synkroniserte metoder og en rekke andre metoder og følgende innebygde metoder:
- **sleep(t):** Den nå kjørende tråden sover i 't' millisek.
- **notify():** (arvet fra klassen Object som alle er subklasse av). Den vekker opp **en** av trådene som venter på låsen i inneværende 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'

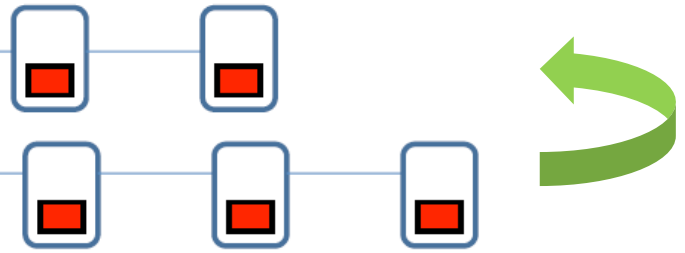
- En Java-monitor er et objekt av en vilkårlig klasse med synchronized metoder
- Det er to køer på et slikt objekt:
  - En kø for de som venter på å komme inn/fortsette i en synkronisert metode
  - En kø for de som her sagt **wait()** (og som venter på at noen annen tråd vekker dem opp med å si notify()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



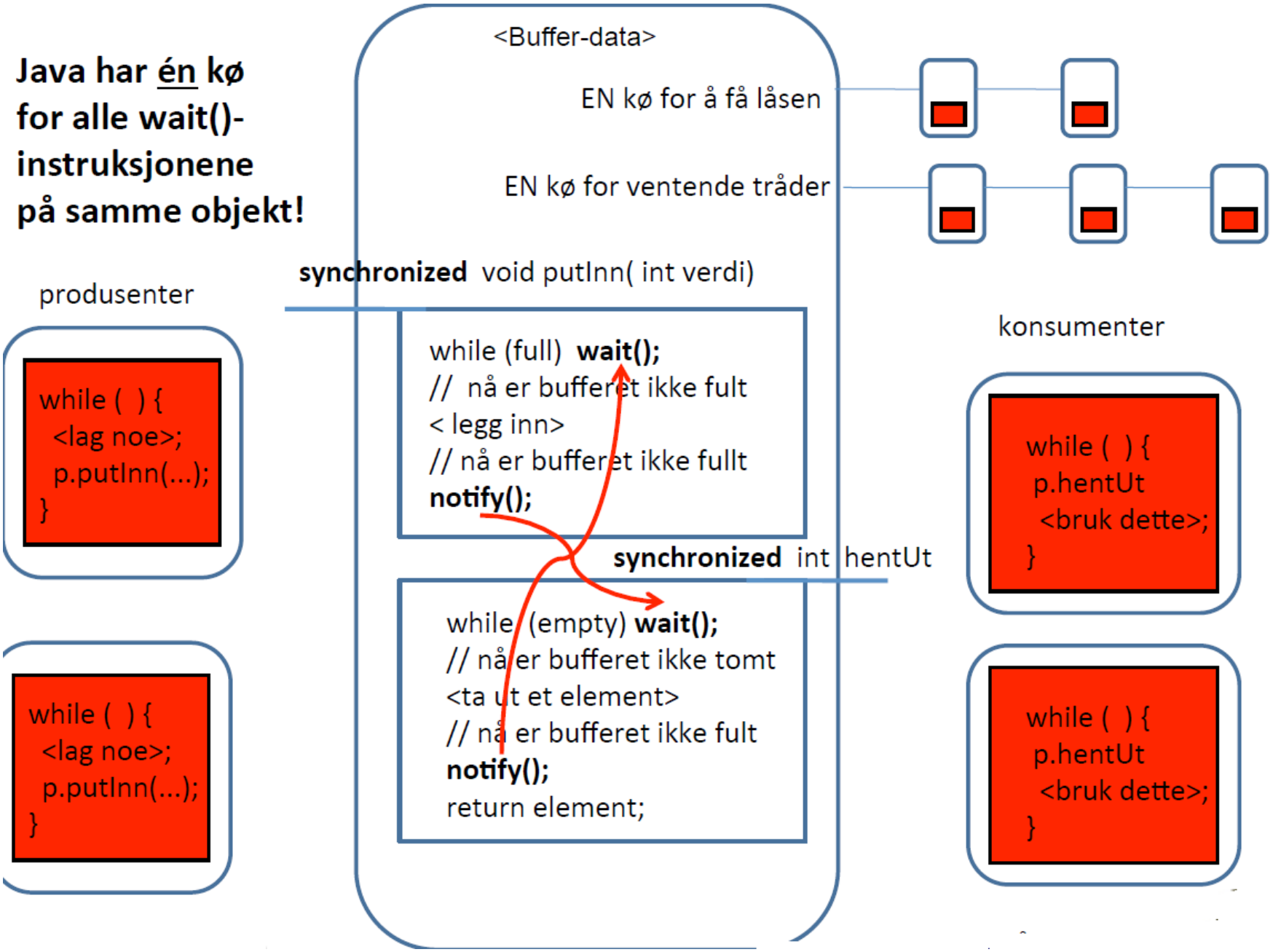
En kø av ventende tråder
på hele monitoren



notify()

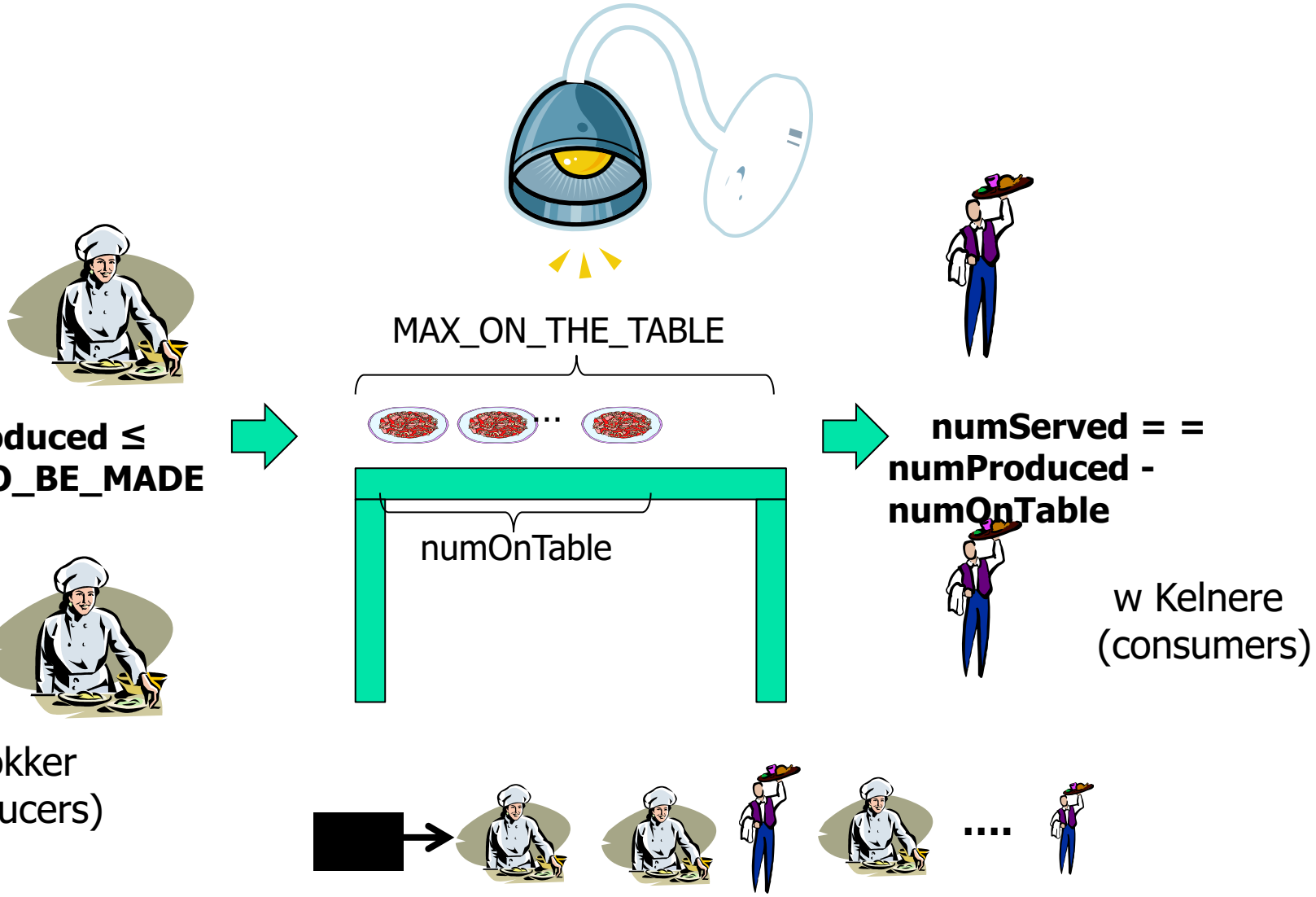
En kø av ventende tråder på
"wait"-instruksjoner
(wait-set).
Startes av notify () og/eller
notifyAll()
Legges da i den andre køen
(først ? (Nei, ingen garanti))
Derfor er det nødvendig med
"while ..."

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



Pass på å unngå vranglås

Restauranten (2):



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

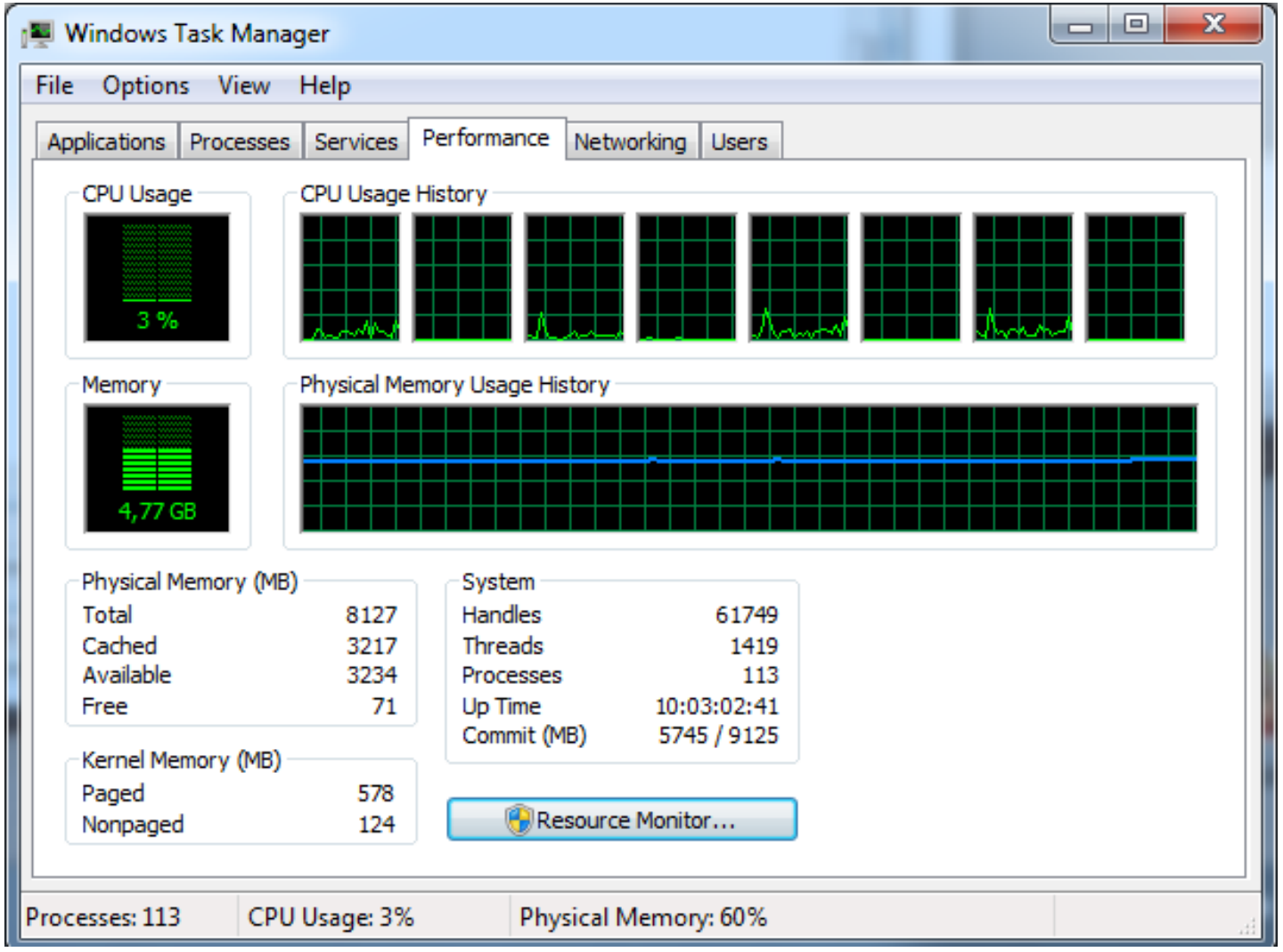
```
public synchronized boolean putPlate (Kokk c) {
    while (numOnTable == TABLE_SIZE &&
           numProduced <   NUM_TO_BE_MADE) {
        try { // The while test holds here meaning  that a Kokk should
              // but can not make a dish, because the table is full
            wait();
        } 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%.



End of second lecture, uke 6