

IN3030 L09, spring 2026
Synchronization
2026-03-18

Eric Jul and Paulo Ferreira
PT
Department of Computer Science
University of Oslo

1

1

Review L08v26

- Convex hull

2

2

Plan for L09

1. Chefs and waiters
2. 3 solutions
3. Hoare Monitors
4. Andes trains.

3

3

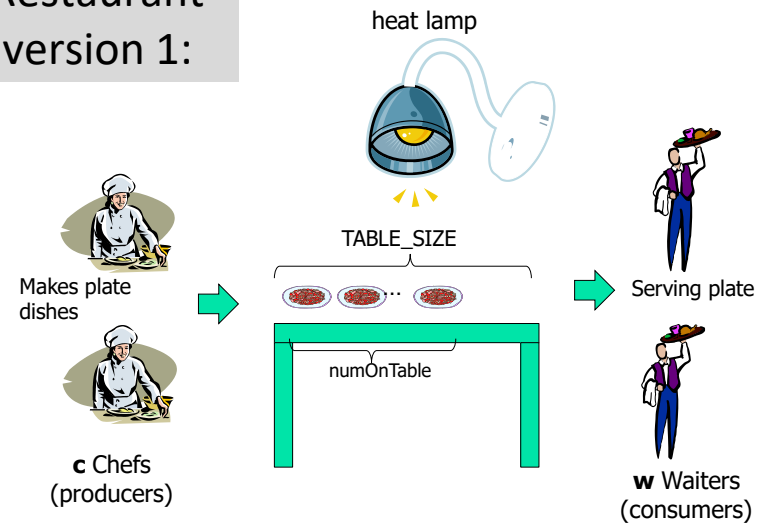
Chefs and Waiters

- The problem we are now going to solve.
- A restaurant with chefs and waiters and a warming table where the food is placed.
- We have **c** Chefs who cook and **w** Waiters serving the food (plate dishes).
- Food that the chefs prepare is placed on a **warming table** (with TABLE_SIZE number of places for plates).
- The chefs cannot make more plates if the hot table is full.
- The waiters cannot serve more plates if the hot plate is empty.
- NUM_TO_BE_MADE number of plates.

4

4

Restaurant version 1:



5

5

Chefs and Waiters

- About monitors and queues (three examples of concurrent programming.)
- We solve synchronization between two different classes.
- First, an actively polling (busy) solution with synchronized methods (Restaurant1.java).
  - Actively waiting in a loop in each Chef and Waiter
    + that they are in the queue to be allowed into a synchronized method
- So a solution using a monitor as it was intended from the start in Java (Restaurant2.java).
  - Chefs and Waiters wait in the same wait () queue
    + in the queue to be allowed into a synchronized method.
- Finally, a solution with a monitor with Lock and Condition queues (several queues – one per waiting condition (Restaurant9.java)
  - Waiters and Chefs wait in their own queue
    + in a queue to be let into the two methods protected by a Lock

6

6

Common to the three solutions

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

Restaurant(String[] args ) {
<Entering the number of Chefs, Waiters and number of
dishes>
<Creates the Chefs and Waiters and starts them>
}

    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;
     // Invariants:
// 0 <= numOnTable <= MAX_ON_TABLE
// numProduced <= NUM_TO_BE_MADE
// numServed <= NUM_TO_BE_MADE

< + different data in the three examples>
```

```
public xxx boolean putPlate (Chef c)
   <Put another plate on the table
   (true) otherwise (false) Chef must wait)
} // end put

public xxx boolean getPlate (Waiter w) {
<If the table is empty (false) Waiter waits
otherwise (true) - Waiter then takes a
plate and serves it>
   } // end get
} // end class HeatingTable

class Chef extends Thread {
    HeatingTable tab;
    int ind ;
    public void run() {
         while / if ( tab .putPlate (..))
         <Different logic in the examples>
    }
} // end class Chef

class Waiter extends Thread {
    HeatingTable tab;
    int ind ;
    public void run() {
          while / if ( tab. getPlate ()){
         <different logic in the examples>
    }
} // end class Waiter
```

7

Invariants on common variables

- The invariants must **always hold** (be true) outside of the monitor methods.
- What are the common variables here:
  - MAX_ON_THE_TABLE
  - NUM_TO_BE_MADE
  - numOnTable
  - numProduced
  - numServed = numProduced – numOnTable
- Invariants:
  1. **0 ≤ numOnTable ≤ TABLE_SIZE**
  2. **numProduced ≤ NUM_TO_BE_MADE**
  3. **numServed ≤ numProduced**

8

8

The invariants shows 4 states we must take write code for

Invariants :

0 <= numOnTable <= MAX_ON_TABLE

numServed <= numProduced <= NUM_TO_BE_MADE

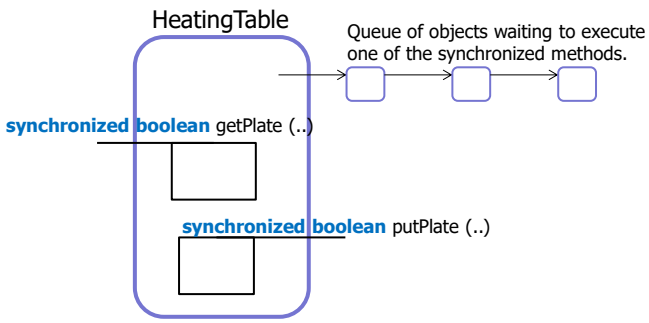
1. **numOnTable == MAX_ON_TABLE**
   → **Chefs are waiting**
2. **0 == numOnTable**
   → **Waiters are waiting**
3. **numProduced == NUM_TO_BE_MADE**
   → **The chefs finished**
4. **numServed == NUM_TO_BE_MADE**
   → **The waiters finished**

9

9

First, an actively polling (busy) solution with synchronized methods (Restaurant1.java).

- This is a solution with **a queue** , the one that all threads enter if another thread is inside a synchronized method in the same object.
- Termination is arranged in each chef and waiter (in their run method)
- The queue used is a common queue of all active objects that may simultaneously try to call one of the two synchronized methods. **goat** and **put .** All objects have such a queue.



10

10

Restaurant solution 1

```
class Chef extends Thread {
....
    public void run() {
       try {
          while ( tab.numProduced < tab.NUM_TO_BE_MADE ) {
            if ( tab.putPlate ( this ) ) {
              // create the next plate
            }
            sleep ((long) (1000 * Math.random ()));
          }
       } catch ( InterruptedException e ) {}
      System.out.println("Cook "+ ind +" done: " );
    }
} // end Chef

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

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

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

11

11

```
M:Restaurant1>java Restaurant1 11 8 8
Chef no:8, made dish no:1
Chef no:4, made dish no:2
Chef no:6, made dish no:3
Waiter no:5, served plate no:1
Chef no:3, made dish no:4
Waiter no:2, served plate no:2
Chef no:1, made dish no:5
Waiter no:5, served plate no:3
Waiter no:4, served plate no:4
Chef no:7, made dish no:6
Chef no:2, made dish no:7
Waiter no:7, served plate no:5
Chef no:4, made dish no:8
Waiter no:3, served plate no:6
Waiter no:3, served plate no:7
Chef no:1, made dish no:9
Waiter no:2, served plate no:8
Chef no:6, made dish no:10
Chef no:3, made dish no:11
Cook 8 done:
```

```
Waiter no:8, served plate no:9
Waiter no:7, served plate no:10
Waiter no:6, served plate no:11
Cook 3 done:
Cook 5 done:
Waiter 1 finished
Cook 1 done:
Cook 4 done:
Waiter 5 finished
Cook 7 finished:
Waiter 2 finished
Cook 2 finished:
Waiter 4 finished
Waiter 3 finished
Waiter 7 finished
Waiter 6 finished
Cook 6 done:
Waiter 8 finished
```

12

12

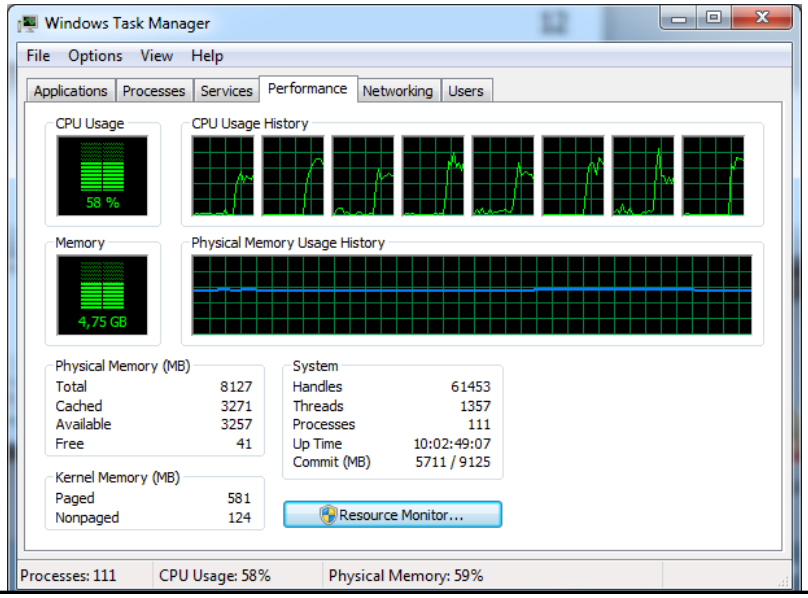
Problems with this solution are active polling

- All the Chef and Waiter threads are actively going around asking:
  - Is there more work for me? Rest a little, about 1 second, and ask again.
  - Wastes a lot of time/machine instructions.
- Especially stressful if one of the thread types (Producer or Consumer) is clearly faster than the other,
  - E.g. set up 18 fast Chefs who sleep only 1 millisecond against 2 slow Waiters who sleep 1000 ms.
  - In that case, this active waiting/masing took 58% of the CPU capacity of 8 cores
- The print must be in the get and put methods. Why?

13

13

Solution 1 with 18 fast Chefs (waiting 1 ms) and 2 slow Waiters (waiting 1000 ms). The Chefs stress the machine with constant failed queries every ms. whether there is room for a plate on the warming table. CPU usage = 58%.



14

14

Solution 2: Java's original scheme with monitors and **two queues** .

- The original Java solution with synchronized methods and a number of other methods and the following built-in methods:
  - **sleep (t):**
    - The currently running thread sleeps for 't' milliseconds.
  - **notify ():**
    - Inherited from the class Object of which all are subclasses.
    - It wakes up one of the threads waiting for the lock on the current object. It then tries again to get what it was waiting for.
  - **notifyAll ():**
    - Inherited from class Object.
    - It wakes up all threads waiting for the lock on the current object. They then all try once to get what they were waiting for.
  - **wait ():**
    - Inherited from class Object.
    - Causes the current thread to wait until it is either woken up by notify () or notifyAll () for this object.

15

15

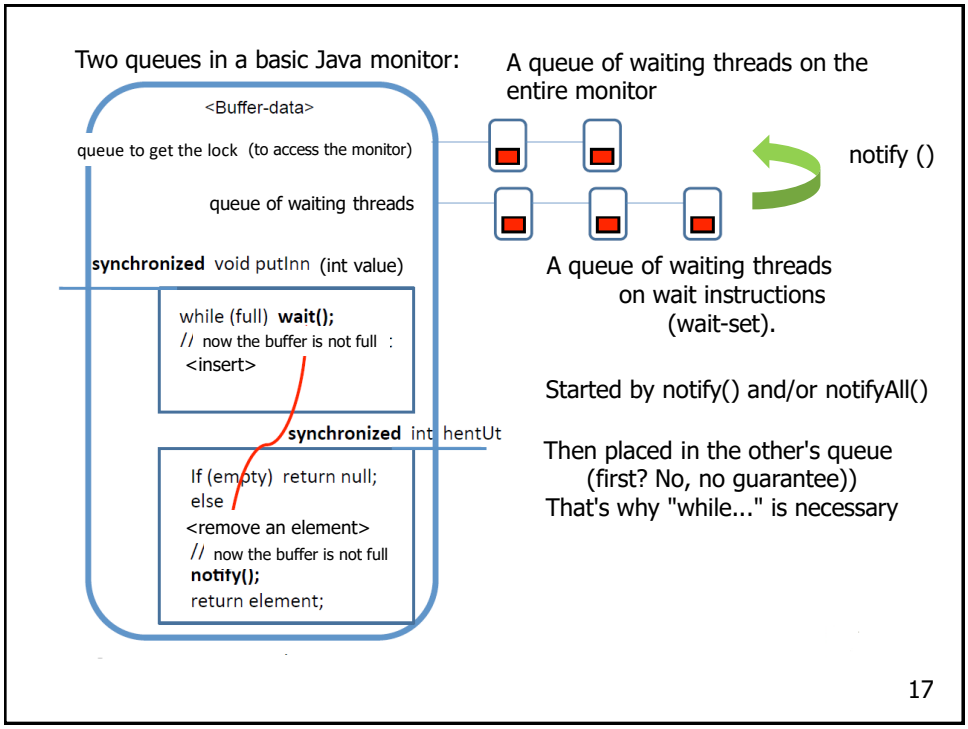
Creating parallel solutions with a Java 'monitor'

- A Java monitor is an object of any class with synchronized methods.
- There are two queues on such an object:
  - A queue for those waiting to enter/continue in a synchronized method
  - A queue for those who said **wait ()** here (and who are waiting for some other thread to wake them up by saying notify () or notifyAll () on them)
    - wait () calls a thread inside a synchronized method.
    - notify () or notifyAll () is also called inside a synchronized method.

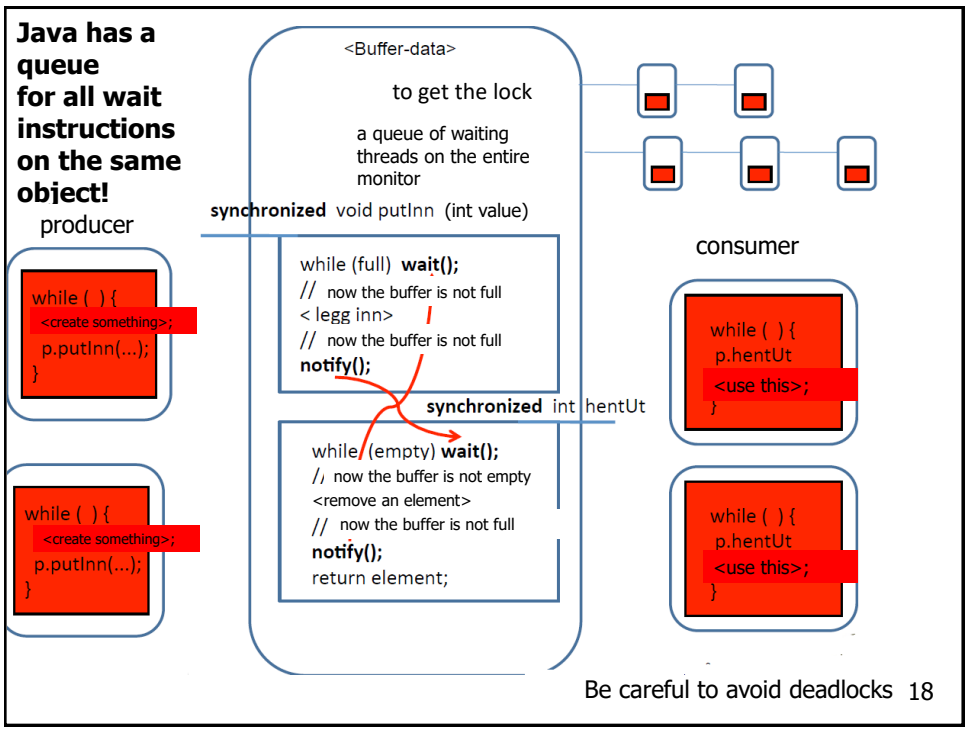
The Monitor idea is heavily inspired by Tony Hoare (the man behind Quicksort)

16

16

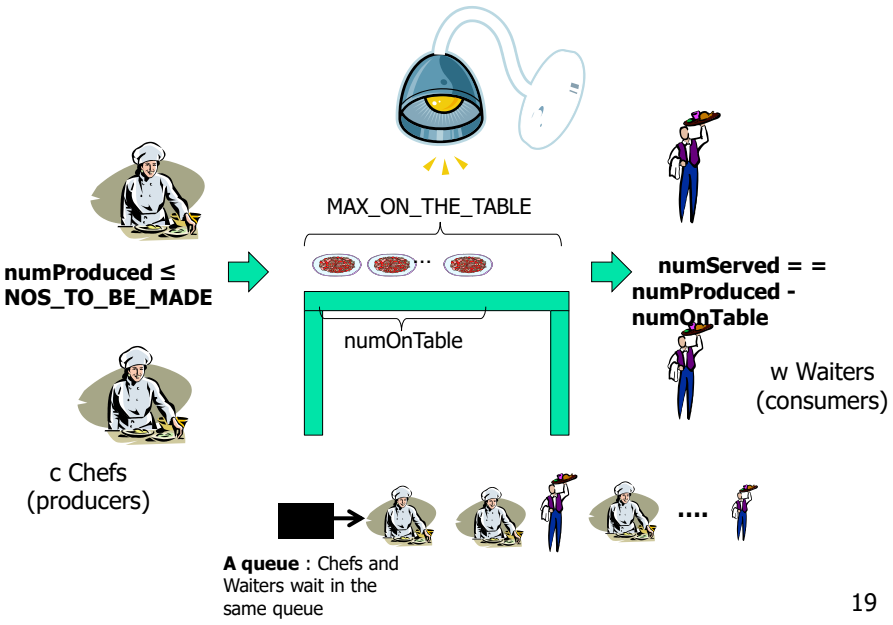

Two queues in a basic Java monitor:
<Buffer-data>
queue to get the lock (to access the monitor)
queue of waiting threads
synchronized void putInn (int value)
while (full) wait();
// now the buffer is not full :
<insert>
synchronized int hentUt
If (empty) return null;
else
<remove an element>
// now the buffer is not full
notify();
return element;
A queue of waiting threads on the entire monitor
notify ()
A queue of waiting threads on wait instructions (wait-set).
Started by notify() and/or notifyAll()
Then placed in the other's queue (first? No, no guarantee))
That's why "while..." is necessary
17

17


Java has a queue for all wait instructions on the same object!
producer
while ( ) {
<create something>;
p.putInn(...);
}
while ( ) {
<create something>;
p.putInn(...);
}
<Buffer-data>
to get the lock
a queue of waiting threads on the entire monitor
synchronized void putInn (int value)
while (full) wait();
// now the buffer is not full
< legg inn>
// now the buffer is not full
notify();
synchronized int hentUt
while (empty) wait();
// now the buffer is not empty
<remove an element>
// now the buffer is not full
notify();
return element;
consumer
while ( ) {
p.hentUt
<use this>;
}
while ( ) {
p.hentUt
<use this>;
}
Be careful to avoid deadlocks
18

18

The restaurant (2):



19

19

Second Solution

- All waiting is inside synchronized methods in one of two queues.
- All waiting inside two synchronized methods
- Chefs and Waiters waiting for the next plate in the wait queue
- We have to wake up everyone in the wait queue to make sure we find someone of the type we need (Chef or Waiter) who can run the program.
- No testing on the invariants in the run methods

20

20

Both solutions 2) and 3):
The run methods try again if the last operation succeeds:

Chefs :

```
public void run() {
      try {
         while ( tab.putPlate ( this )) {
         sleep((long) (1000 * Math.random()));
         }
      } catch (InterruptedException e) {}
      // This Chef is finished
}
```

Waiters:

```
public void run() {
      try {
         while ( tab.getPlate ( this )){
         sleep((long) (1000 * Math.random()));
         }
      } catch (InterruptedException e) {}
      // This Waiter is finished
}
```

21

Solution2

```
public synchronized boolean putPlate ( Chef c) {
    while ( numOnTable == TABLE_SIZE &&
            numProduced < NUM_TO_BE_MADE) {
        try { // The while test holds here meaning that a Chef should
            // but can't 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
```

22

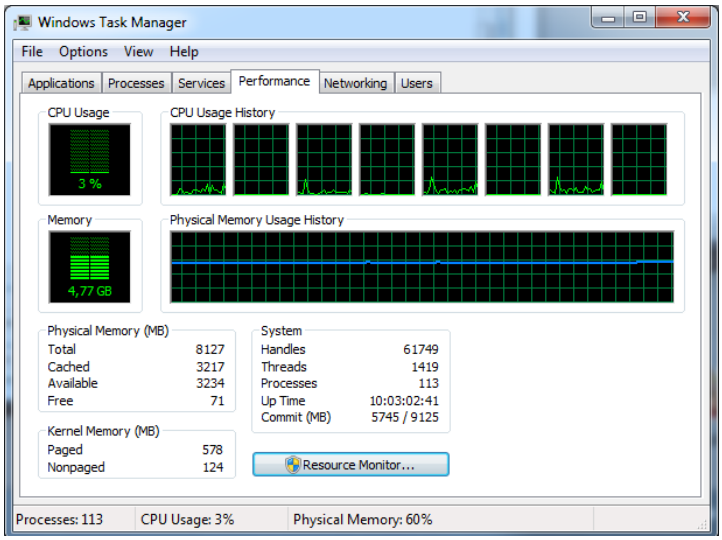
Solution2

```
public synchronized boolean getPlate ( Waiter 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 no 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 Cooks :
        notifyAll (); // wake up all queued Waiters and Chefs
        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
```

23

23

Solution 2 with 18 fast Chefs (waiting 1 ms) and 2 slow Waiters (waiting 1000 ms). The Chefs do not stress the machine with constant failed queries, but wait in line until there is room for another plate on the warming table. CPU usage = 3%.



24

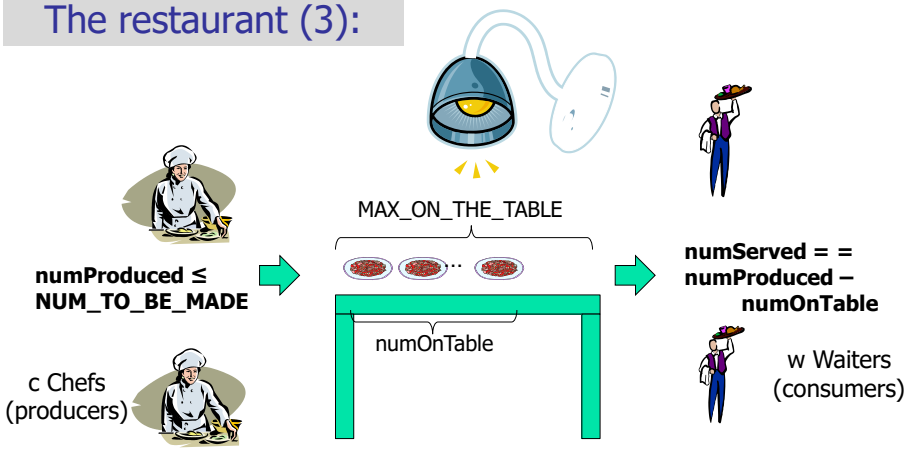
24

3) A parallel solution with Conditions

- Using two queues:
  - One for Chefs waiting for a plate space at the table
  - One for Waiters waiting for a plate
- Then we don't need to wake up all the threads, just one in the right queue.
  - Maybe more efficient
  - Clearly easier to program

25

The restaurant (3):



Two queues:
a) Chefs waiting for a place at the table

b) Waiters are waiting for more plates



26

```
final Lock lock = new ReentrantLock();
final Condition notFull = lock.newCondition(); // queue for Chefs
final Condition notEmpty = lock.newCondition(); // queue for Waiters

public boolean putPlate (Cooks 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 waiter to serve
            if (numProduced == NUM_TO_BE_MADE ) {
                // I have produced the last plate,
                notEmpty.signalAll (); // tell Waiters to stop waiting, terminate
                notFull.signalAll (); // tell Chefs to stop waiting and terminate
                return false ;
            }
            return true ;
        } else { return false ;}
    } finally {
        lock.unlock();
} } // end putPlate
```

27

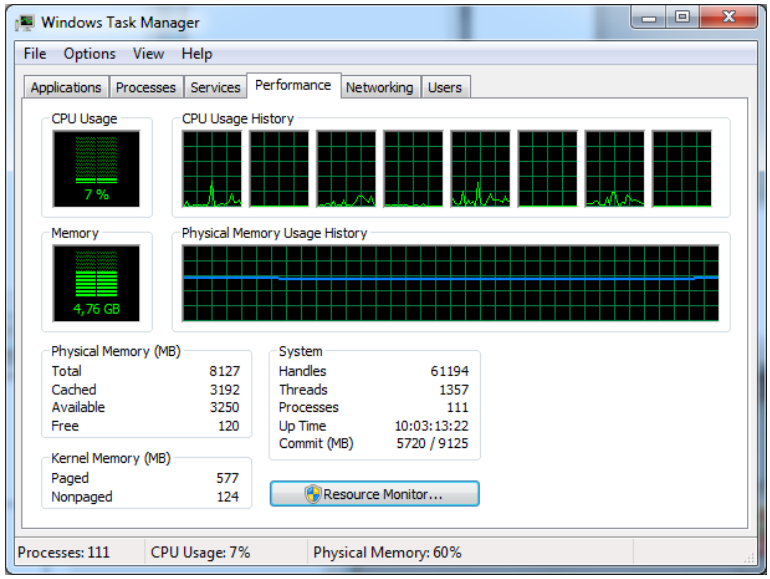
```
public boolean getPlate ( Waiter ) throws InterruptedException {
  lock.lock();
  try {
    while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
        notEmpty.await (); // This Waiter is waiting for a plate
    }
    if (numOnTable > 0) {
        numOnTable-- ;
        notFull.signal ();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

A Waiter or a Chef is signaled for two reasons:
- to treat (make or serve) another dish
- nothing more to do, go home (clear both queues)

28

28

Solution 3 with 18 fast Chefs (waiting 1 ms) and 2 slow Waiters (waiting 1000 ms). The Chefs do not stress the machine with constant failed queries, but wait in line until there is room for another plate on the warming table. CPU usage = 7%.



29

29

Assessment of the three solutions

- **Solution 1** : Simple, but can take up too much CPU time. Especially when the system is getting saturated for other reasons, typically one of the threads will become very slow, and then this solution suddenly takes up half of the CPU time.
- **Solution 2:** Good, but difficult to write
- **Solution 3:** Good, almost as effective as solution 2 and easier to write.

30

30

Concluding remarks on the Producer-Consumer problem

- Invariants are used by all programmers (often unconsciously)
  - program, loop or method (sequential or parallel)
  - saying them explicitly helps with programming
- REMEMBER: synchronized/lock only works when all threads synchronize on the same object.
  - when that happens, there is **sequential thinking** between wait /signal
- Debugging by tracing execution (trace) – System.out.println("..")
  - writing outside a Lock/synchronized method/part of method, then create one:
    - synchronized void println (String s) {System.out.println(s);}
  - otherwise, printing may be mixed up or out of order.

31

31

Problem

- When we say notify () or wait () on a queue, we don't know:
  - which thread starts
  - does the thread that is signaled get on the kernel directly after the one that said notify (), or not??
  - Not defined
- Even after we have tested in the run method that we can handle another plate, we still need to test if it works OK:
  - Another thread may have been inside and changed the variable

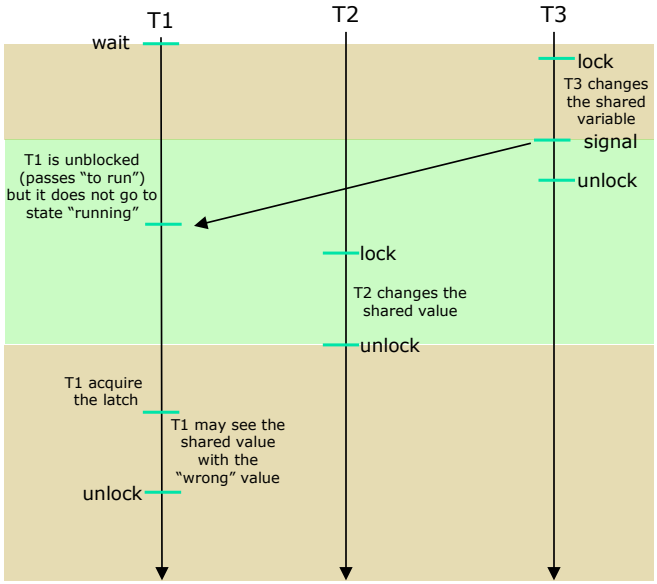
32

32

Problem

- Attention:
  - During the time that passes between the signal (T3) and a thread be unblocked (T1) acquiring the latch
  - The condition variable can be changed by another thread (T2)

Semantics Mesa (here is assumed) vs Semantics Hoare (see monitors)



33

Mesa and Hoare Semantics

- Most modern languages use **Mesa semantics**
- **Hoare monitor:** signal immediately transfers control to the waiting thread.
- **Mesa monitor:** signal only wakes the waiting thread, but the notifying/signalling thread keeps running.
- In **Mesa monitors**, when a thread calls notify()/signal():
  - The waiting thread is only moved to the ready queue.
  - The signalling thread continues executing.
  - Other threads may enter the monitor and modify the shared state before the waiting thread resumes.
  - So, when the waiting thread finally runs again, the condition might no longer be true.

34

34

Hoare Monitors

- The concept of Monitors was originally presented by Per Brinch Hansen who put the concept into his language Concurrent Pascal, in 1970`s
- 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 .

- Monitors paper
- Andes train

35

35

End of lecture L09v25

36

36