http://cs.lth.se/EDA040

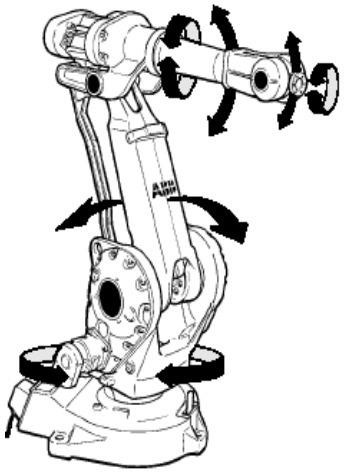
Lectures during HT 2012: EDA040

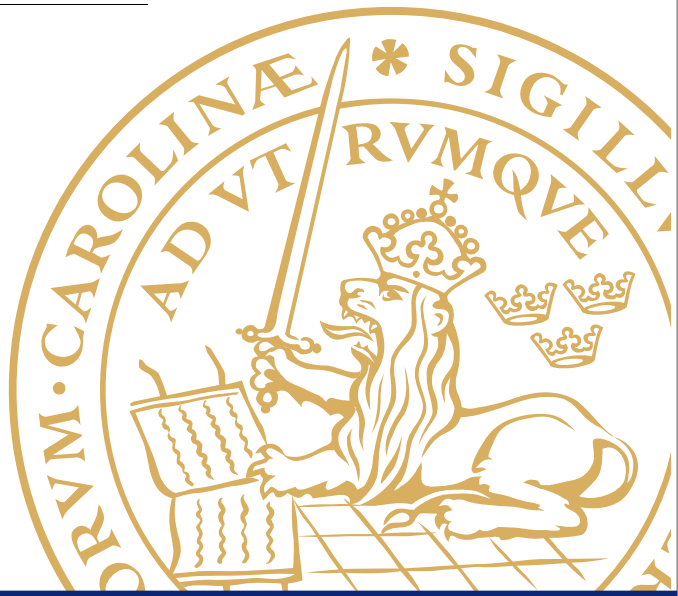
Concurrent and Real-time programming

Department of Computer Science, Lund

klas@cs.lth.se,

Lecture 3 [F3]: Monitors: `synchronized`, `wait` and `notify`





Monitors; Synchronized methods

Monitors

- Language construct for synchronization of threads
- More powerful than semaphores
- Fits in well with object orientation
- Supported by `synchronized` in Java

Mutual exclusion (exercise 1)

In-line use of semaphores for mutual exclusion
Disadvantage: `take`/`give` tends to get spread out through the entire program (exercise 1).

Abstract data-types for mutual exclusion
Principle: `take`/`give` part of (mutually exclusive) methods that are kept together with the hidden data.

- Such a data-type with mutually exclusive access-functions is called a **Monitor**.

Monitors (objects & concept)

- In OOP we use classes as a (more powerful) mean to accomplish abstract data-types.
- Objects with such mutually exclusive methods are then monitor objects.

- With methods like

```
class Account {
  // ...
  void deposit(int a){
    mutex.take();
    balance += a;
    mutex.give();
  }
}
```

the monitor concept is implemented by using semaphores.

Language support for Monitors

Problem: Using semaphores requires (too much) discipline.

- Idea: Provide support via language constructs.
- Degree of language support:
  - **None** [C/C++]: Manual calls (as with mutex; take/give) using library functions. Object-orientation may simplify usage.
  - **Explicit** per method[Java]: Declared property of methods (language and run-time support).
  - **Implicit** per task [Ada]: Declared property of class (implicitly applies to all methods and data).
- **None** results in more complicated programming. **Implicit** language support safest and simplest but can limit applicability (resources). The Java approach (**explicit** and optional declaration of mutually exclusive methods) is a pragmatic solution.

Java-supported monitors

- In Java: Critical region/block/method is declared using the keyword **synchronized**
- Unfortunately: Neither classes nor attributes can be declared synchronized; discipline required.
- The monitor concept by use of Java:

```
class Account {
  // ...
  synchronized void deposit(int a){
    balance += a;
  }
}
```

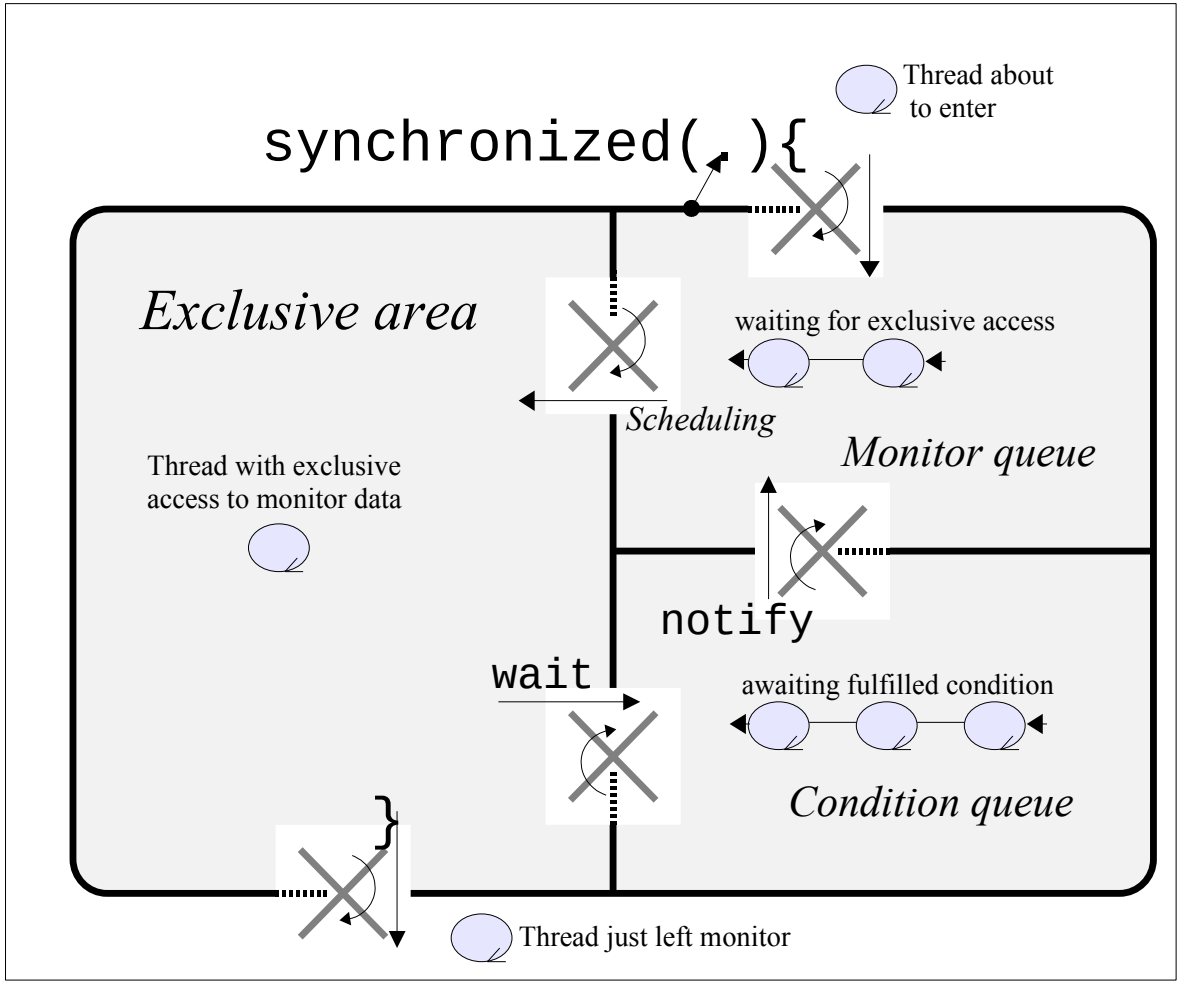
Object categories

- Plain passive object:
  - Thread safe by reentrant methods (`java.lang.Math`)
  - Explicitly thread unsafe; to be used by a single thread (`java.util.HashSet`)
  - Implicitly thread unsafe; has to be assumed if not documented.
- Monitor object:
  - Mutually exclusive methods, e.g., by using synchronized.
  - Should be passive; do not mix monitors and threads!
- Thread object:
  - Active object (if started but not terminated); drives execution.
  - Don't call me, I'll call you!

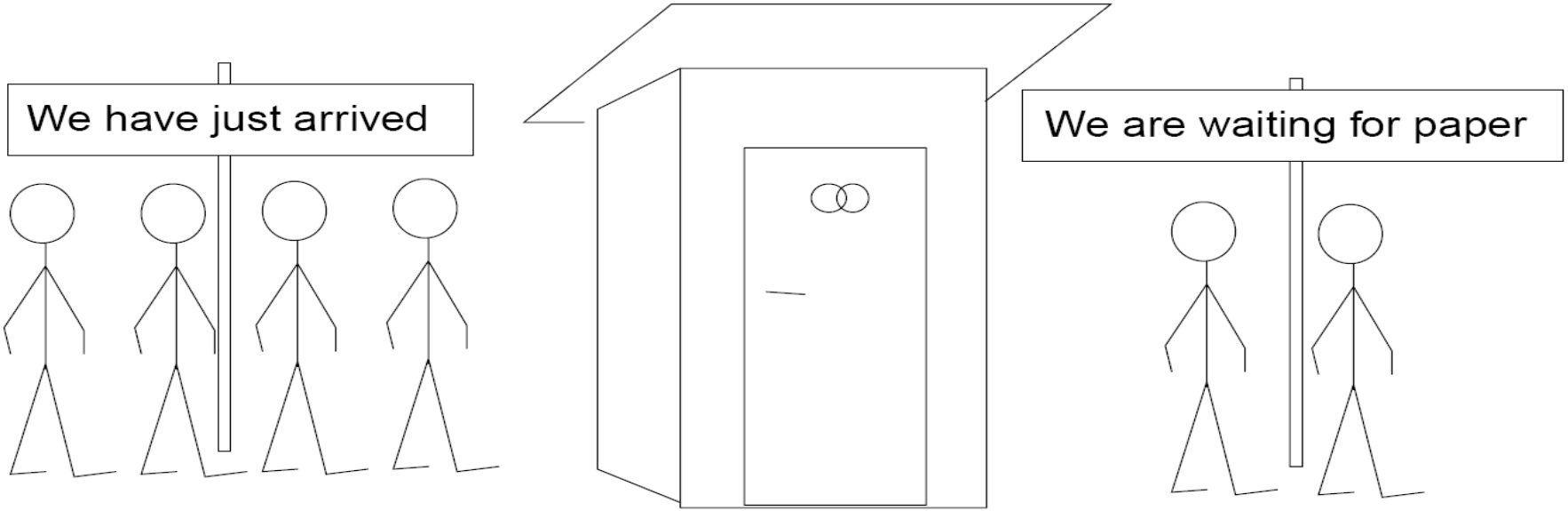
`synchronized` - `wait` - `notify`

Condition queue

In addition to locking the object for exclusive access (*mutex*): Temporarily unlock until someone *signals* that the state has changed:

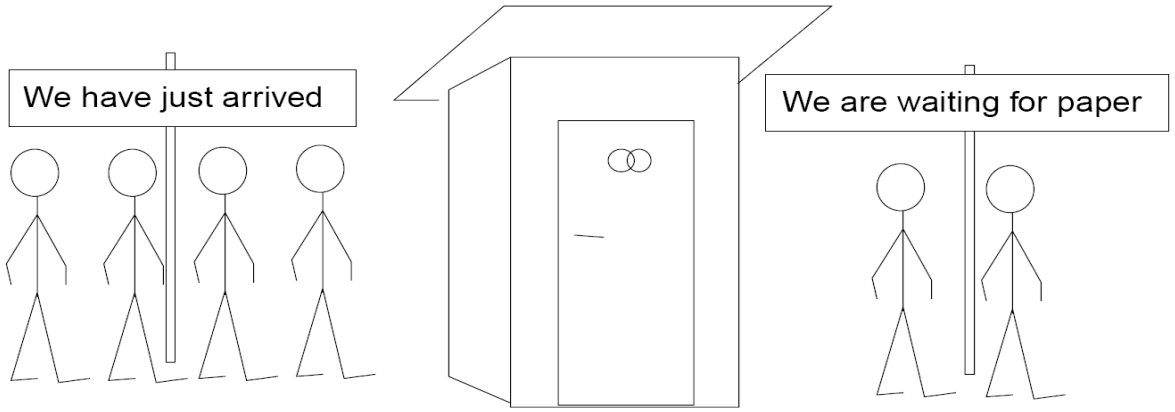


Monitor conditions - analogy



- Assume shared resource providing three operations: `opA`, `opB`, and `addPaper`.
- Only one can enter at a time, entrance means exclusive access.
- The `opB` requires that paper is available, discovered after entrance.
- Two queues, one for entrance (left) and one for conditions (right).

Monitor conditions - analogy/scenario



It is the responsibility of the one performing `addPaper` to inform the waiting persons that the state of the object has changed.

Scenario:

- The persons (threads) entering the monitor to do `opB`, but discovers that there is no paper aborts the operation, exits, and waits in a special queue until the condition 'paper is available' will be true.
- Even though there is no paper, other persons are let in to perform `opA`. Eventually someone arrives who changes the roll of paper after which the waiting persons can be let in again.

Notifications are stateless

Semaphore has state

Compare the case

P1	P2
•	•
s.take();	•
•	s.give();
•	•

with

P1	P2
•	•
•	s.give();
s.take();	•
•	•

P1 continues in both cases since the internal state reflects the give.

Notification is stateless

Compare the case

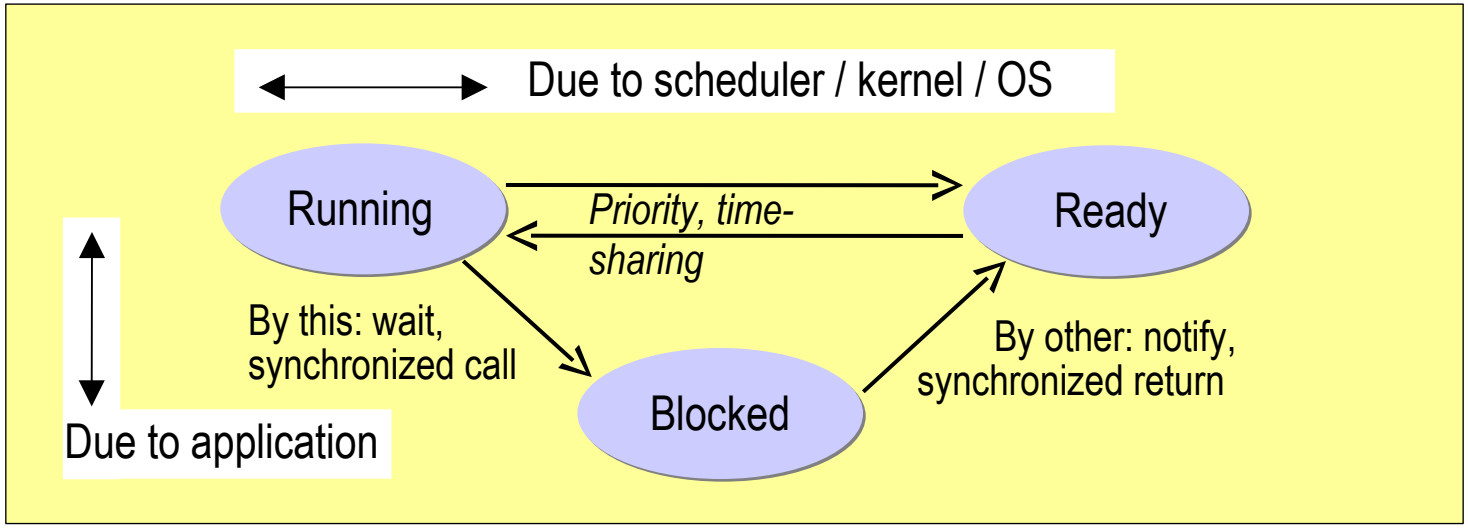
P1	P2
•	•
wait();	•
•	notify();
•	•

with

P1	P2
•	•
•	notify();
wait();	•
•	•

P1 waits until next notification; The notify is forgotten, unless appropriate state variables exist in the monitor (i.e., in your code).

Execution states, revisited



Scheduling state:

- Running
- Ready
- Blocked

Context:

- Current PC
- Application data
- Machine registers (SP, ..)

Original Hoare Monitor (1974)

Originally defined monitor properties:

- Immediate Resumption; the awakened thread takes control immediately
- The `notify` must be performed last, one thread only is awakened.
- The condition for waiting could be coded:
  `if (!ok) wait();`
- The notifying thread guarantees that the condition being waited for is true.
- Easier to prove that starvation can not occur.
- Does not handle priority for blocked threads.
- The enter queue can be FIFO or (preferably) a priority queue.

Real-time Monitor

We assume these monitor properties:

- High priority threads should be given precedence, even to threads which have been waiting longer (desired starvation risk).
- Immediate resumption not guaranteed (depends on OS/scheduler)
- The condition being waited for might not be true anymore when a blocked thread resumes execution.
- Waiting for a condition must be coded: `while (!ok) wait();`
- Use '`notifyAll`' to avoid problems (practical - wakes all).
- The notify not necessarily called last in the method.

When previously blocked threads precedes those with same priority + notify last + one level of priority: equivalent with Hoare Monitor.

Basic rules

- Do not mix a thread and a monitor in the same object/class *[so you can get assistance from the compiler concerning proper access, which should go over visible methods]*.
- All public methods should be synchronized *[and that is not inherited so redo in subclass]*.
- Wrap thread-unsafe classes by monitor *[if possibly used by multiple threads]*.
- Do not use (spread-out) synchronized blocks *[which are more for limited GUI concurrency]*.

Details (on board and in book)

- Atomic access
- Keyword volatile.
- Attribute for locking: private and final.
- The monitor property (synchronized) is not inherited.
- The internal lock can be exposed for external synchronization.

Badly implemented buffer

```
class Producer extends
Thread
{
  public void run()
  {
    prod = source.get();
    buffer.post(prod);
  }
}

class Consumer extends
Thread
{
  public void run()
  {
    cons = buffer.fetch();
    sink.put(cons);
  }
}
```

```
class Buffer
{
  synchronized void post(Object obj)
  {
    if (buff.size()==maxSize) wait();
    if (buff.isEmpty()) notify();
    buff.add(obj);
  }

  synchronized Object fetch()
  {
    if (buff.isEmpty()) wait();
    if (buff.size()==maxSize) notify();
    buff.remove(buff.size());
  }
}
```

The **`if (...) wait();`** *makes the buffer fragile: additional calls of notify or additional interacting threads could course the buffering to fail.*

Better buffers

```
class Buffer // Inefficient!!
{
  synchronized void post(Object obj
)
  {
    while (buff.size()>=maxSize) {
      wait();
    }
    buff.add(obj);
    notifyAll();
  }

  synchronized Object fetch()
  {
    while (buff.isEmpty()) {
      wait();
    }
    buff.remove(buff.size());
    notifyAll();
  }
}
```

```
class Buffer // Well done.
{
  synchronized void post(Object obj)
  {
    while (buff.size()>=maxSize) {
      wait();
    }
    if (buff.isEmpty()) notifyAll();
    buff.add(obj);
  }

  synchronized Object fetch()
  {
    while (buff.isEmpty()) {
      wait();
    }
    if (buff.size()>=maxSize) notifyAll();
    buff.remove(buff.size());
  }
}
```

The **`while (...) wait();`** *makes the buffer robust with respect to other threads that can access the buffer and change the conditions.*

Monitor == Semaphore

Semaphores and Monitors are equivalent since:

- Semaphores (for threads but not for interrupt routines) can be (and are in standard Java) implemented by a monitor (with methods take and give).
- Monitors can be implemented by semaphores, for a given set of threads (using one **`MutexSem`** per monitor, and one **`CountingSem`** for each thread per monitor).

Thus, a specific implementation using one mechanism can (even if hard) always be reimplemented using the other.

Use the right technique depending on the problem to solve!

`synchronized` - `wait` - `notify`

Make sure you understand how threads are interacting via monitors in Java;
do understand this figure: