Software Foundations in JAVA (JSF)

Week 7

Monitors

Peter Boots
p.boots@fontys.nl

Erik van der Schriek
e.vanderschriek@fontys.nl

Planning

Week 1	Introduction JSF, Operating systems, Linux
Week 2	Linux shell, interoperabiliteit JAVA
Week 3	Processen en threads
Week 4	Multithreading (MT) en performance
Week 5	MT & concurrency
Week 6	Advanced MT mechanisms
Week 7	Monitors
Week 8	Roundup
Week 9	Exam
Week 10	Repair time

Overview

- ReentrantLock
- Condition
- Monitor

Drawbacks of synchronized

- In week 5, we solved the Critical Section (CS) problem by using `synchronized`
- We could wait inside a `synchronized` section with `wait`/`notify`/`notifyAll`

- Problems:
  - we don't know which thread is woken up by `notify`
  - `notifyAll` will wake up all threads

General tools

- Standard synchronization tools are not always sufficient
- There are tools to solve any synchronization problem
  - Semaphores:
    - A low level mechanism
    - Difficult to get a 100% correct solution
    - We will not use them
  - Monitors
    - High level
    - Easier to get correct solution

ReentrantLock

- Monitors in Java are made with `ReentrantLock`
- This can be used to solve the Critical Section problem:
  - `Lock monLock = new ReentrantLock();`

  -
    ```
    monLock.lock();
    ...... // Critical Section
    monLock.unlock();
    ```
- Alternative for `synchronized`

ReentrantLock

- In order to make sure that the lock is <u>always</u> unlocked:

  -
    ```
    monLock.lock();
    try {
      ...... // Critical Section
    } finally {
      monLock.unlock();
    }
    ```

Condition

- Inside such a lock, we can wait for different ‘conditions’
- For this we have class `Condition`
  - every `Condition` has its own waiting queue
  - can only be used <u>inside</u> a lock
  - methods:
    - `void await()`
      - replaces the `wait` method
    - `void signal()`
      - replaces the `notify` method
    - `void signalAll()`
      - replaces the `notifyAll` method

How to use a `Condition`

- Usually, a thread checks a certain condition
  → if not satisfied, thread calls `await`
  - There can be a number of threads waiting, on different or the same condition

- Another thread calls `signal` or `signalAll` when condition is satisfied
  - `signal` will wake up one waiting thread
    - unknown which one
  - `signalAll` will wake up all waiting threads

How to use a `Condition`

- General usage guidelines for using `Conditions`:
  - **Specify what threads are waiting for**: <expression>
    - Eg: if threads are waiting for `x` to become positive, the <expression> is: `x>0`
  - **Choose meaningful name**
    - Eg: `xPos`
  - **Use !<expression> in a `while` that contains `await`**
    - Eg: `while(!(x>0)) { xPos.await(); }`
  - **Use <expression> in an `if` that contains `signal`**
    - Eg: `if(x>0) xPos.signal();`
    - `if` is not always necessary

Example

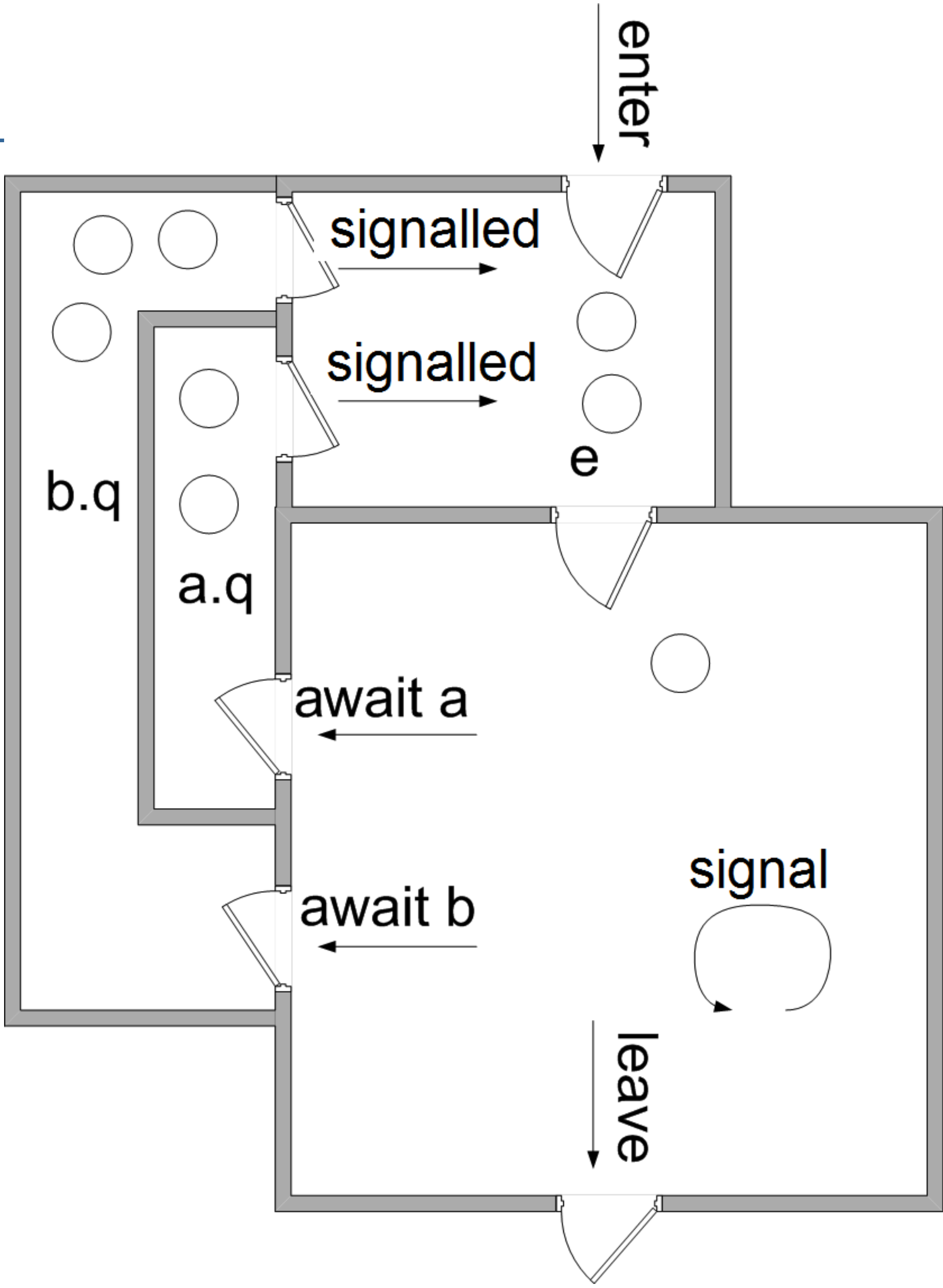
- So if we want to wait until a variable x is positive:
  - `Lock monLock = new ReentrantLock();`
  - `Condition xPos = monLock.newCondition();`
  -
```
monLock.lock();
try {
  ......
  while (!(x > 0)) { xPos.await(); }
  ......
} finally {
  monLock.unlock();
}
```
  -
```
monLock.lock()
try {
  ......
  if (x > 0) xPos.signal();
  ......
} finally {
  monLock.unlock();
}
```

Monitor

- How do we create a <u>monitor</u> out of this?
  - There is no class with name `Monitor`
  - A monitor is a ‘normal’ class, with some extra features
    - all methods are synchronized with a `Lock`
    - methods can contain `Conditions`

- All synchronization issues are <u>concentrated</u> in a monitor:
  - it contains all shared variables (private)
  - it contains all synchronization code

- Dentist with multiple waiting rooms
  - Room a:
    - to wait for sedation to kick in
  - Room b:
    - to wait for result of a test



`await` in `while`

- Why use `while`-statement for `await` (and not `if`)?
  - Thread first has to wait its turn before it can continue running
  - State can be changed in the meantime

Readers-Writers Problem

- A data set is shared among a number of threads

- Two types of threads:
  - Readers
    - only read the data set; do <u>not</u> perform any updates
  - Writers
    - both read <u>and</u> update the data set

- Requirements:
  - allow multiple readers at the same time
  - allow only one writer at the same time
  - allow no readers and writers together

Readers-Writers Monitor

- Make a monitor class RW with 4 methods:
  - `enterReader`
  - `exitReader`

  - `enterWriter`
  - `exitWriter`

Structure of a reader/writer

- General structure of a reader:

  -
    ```
    while(true) {
       RW.enterReader();
       ... // Read dataset (CS)
       RW.exitReader();
    }
    ```
- General structure of a writer:

  -
    ```
    while(true) {
       RW.enterWriter();
       ... // Read/update dataset (CS)
       RW.exitWriter();
    }
    ```
- More complicated if reader/writer can be interrupted in CS

Inside the monitor

- In this monitor, we use the following private variables:
  - `readersActive` → Number of readers in CS
  - `writersActive` → Number of writers in CS

- We also use these `Conditions`:
  - **`okToRead`**: readers wait here until they can read:
    - `writersActive==0`
  - **`okToWrite`**: writers wait here until they can write:
    - `writersActive==0` and `readersActive==0`

Reader part of monitor

-
  ```
  public void enterReader() throws InterruptedException{
    monLock.lock();
    try {
      while (writersActive > 0) { okToRead.await(); }
      readersActive++;
    }
    finally {
      monLock.unlock();
    }
  }
  ```

-
  ```
  public void exitReader() {
    monLock.lock();
    try {
      readersActive--;
      if (readersActive == 0) okToWrite.signal();
    }
    finally {
      monLock.unlock();
    }
  }
  ```

Writer part of monitor

-
  ```
  public void enterWriter() throws InterruptedException {
    monLock.lock();
    try {
      while (writersActive > 0 || readersActive > 0)
        okToWrite.await();
      writersActive++;
    }
    finally {
      monLock.unlock();
    }
  }
  ```

Writer part of monitor

- There is a problem when a writer leaves the CS, and there are both readers and writers waiting:
  - does he wake up a reader, or
  - does he wake up a writer?

- Suppose we want to give the readers precedence:
  - count number of waiting readers: `readersWaiting`

- Replace in `enterReader` the `while` statement by:

  -
    ```
    while (writersActive != 0) {
      readersWaiting++;
      okToRead.await();
      readersWaiting--;
    }
    ```

-
  ```
  public void exitWriter() {
    monLock.lock();
    try {
      writersActive--;
      if(readersWaiting > 0) okToRead.signal();
        else okToWrite.signal();
      }
    finally {
      monLock.unlock();
    }
  }
  ```

- Links:
  - http://www.baptiste-wicht.com/2010/09/java-concurrency-part-5-monitors-locks-and-conditions/
- From Java Core, Volume 1 (ed 8):
  - Chapter 14 “Multithreading”, paragraphs:
    - “Synchronization”, especially:
      - “Lock Objects”
      - “Condition Objects”