Java concurrency (lecture 7)
INF4140 - Models of concurrency
Java concurrency, lecture 7

Høsten 2013

14. 10. 2013
1. Monitors: review

2. Threads in Java:
   - Thread classes and Runnable interfaces
   - Interference and Java threads
   - Synchronized blocks and methods: (atomic regions and monitors)

3. Example The ornamental garden

4. Thread communication & Condition synchronization (wait and signal)

5. Example: Mutual exclusion

6. Example: Readers/writers
Short recap of monitors

- Monitor **encapsulates** data, which can only be **observed** and **modified** by the monitor’s procedures
  - Contains variables that describe the **state**
  - Variables can be accesses/changed only through the available **procedures**
- Implicit mutex: Only a procedure may be active at a time.
  - 2 procedures in the same monitor: never executed concurrently
- **Condition synchronization**: block a process until a particular condition holds) is given by **condition variables**.

Signaling disciplines

- **Signal and Wait (SW)**: the signaller waits, and the signalled process gets to execute immediately
- **Signal and Continue (SC)**: the signaller continues, and the signalled process executes later
From Wikipedia:\(^1\)

"... Java is a general-purpose, \textit{concurrent}, class-based, object-oriented language ..."

\(^1\text{But it’s correct nonetheless ...}\)
A thread in Java

- unit of concurrency\(^2\)
- identity, accessible via static method `Thread.currentThread()`\(^3\)
- has its own stack / execution context
- access to shared state
- shared mutable state: heap structured into objects
  - privacy restrictions possible
  - what are private fields?
- may be created (and deleted) dynamically

\(^2\) as such, roughly corresponding to the concept of “processes” from previous lectures

\(^3\) What’s the difference to this?
The **Thread class** executes instructions from its method `run()`. The actual code executed depends on the implementation provided for `run()` in a derived class.

```java
class MyThread extends Thread {
    public void run() {
        // .......
    }
}
```

// Creating a thread object:
Thread a = new MyThread();
a.start();
Since Java does not permit multiple inheritance, we often implement the `run()` method in a class not derived from `Thread` but from the interface `Runnable`.

```java
class MyRun implements Runnable {
    public void run() {
        // .....
    }
}
```

// Creating a thread object:
Runnable b = new MyRun();
new Thread(b).start();
steps to \textit{create} a thread in Java and get it running:

1. Define class that
   \begin{itemize}
   \item \textit{extends} the Java Thread class or
   \item \textit{implements} the Runnable interface
   \end{itemize}

2. define \textit{run} method inside the new class\footnote{overriding, late-binding.}

3. create an instance of the new class.

4. \textit{start} the thread.
class Store {
    private int data = 0;
    public void update() { data++; }
}

// in a method:
Store s = new Store(); // the threads below have access to s
FooThread t1 = new FooThread(s); t1.start();
FooThread t2 = new FooThread(s); t2.start();

t1 and t2 execute s.update() concurrently!
Interference between t1 and t2 ⇒ may lose updates to data.
Synchronization

Avoid interference \( \Rightarrow \) threads “synchronize” access to shared data

1. One **unique lock** for each object \( o \).
2. Mutex: at most one thread \( t \) can lock \( o \) at any time.\(^5\)
3. 2 “flavors”

“synchronized block”

\[
\text{synchronized} (o) \{ B \}
\]

Synchronized method

Whole method body of \( m \) “protected”\(^a\):

\[
\text{synchronized} \quad \text{Type} \quad m(\ldots) \quad \{ \ldots \}
\]

\(^{a}\)Assuming that other methods play according to the rules as well etc.

\(^{5}\)But: in a re-entrant manner!
Protecting the initialization

Solution to earlier problem: lock the Store objects before executing problematic method:

class Store {
    private int data = 0;

    public synchronized void update() {
        data++;
    }
}

or

class Store {
    private int data = 0;

    public synchronized void update() {data++; }
}

...  

// inside a method:
Store s = new Store();
Book:
Concurrency: State Models & Java Programs, 2nd Edition
Jeff Magee & Jeff Kramer
Wiley

Examples in Java:
http://www.doc.ic.ac.uk/~jnm/book/
Ornamental garden problem

- people enter an ornamental garden through either of two turnstiles.
- problem: the number of people present at any time.

The concurrent program consists of:

- 2 threads
- shared counter object
The Turnstile thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the `increment()` method of the counter object.
class Counter {

    int value = 0;
    NumberCanvas display;

    Counter(NumberCanvas n) {
        display = n;
        display.setvalue(value);
    }

    void increment() {
        int temp = value; // read [v]
        Simulate.HWInterrupt();
        value = temp + 1;   // write [v+1]
        display.setvalue(value);
    }
}

class Turnstile extends Thread {
  NumberCanvas display;  // interface
  Counter people;        // shared data

  Turnstile(NumberCanvas n, Counter c) {   // constructor
    display = n;
    people = c;
  }

  public void run() {
    try {
      display.setValue(0);
      for (int i = 1; i <= Garden.MAX; i++) {
        Thread.sleep(500);  // 0.5 second
        display.setValue(i);
        people.increment();  // increment the counter
      }
    }
    catch (InterruptedException e) { }
  }
}
The **Counter** object and **Turnstile** threads are created by the `go()` method of the **Garden** applet:

```java
private void go() {
    counter = new Counter(counterD);
    west = new Turnstile(westD, counter);
    east = new Turnstile(eastD, counter);
    west.start();
    east.start();
}
```
After the **East** and **West** turnstile threads have each incremented its counter **20** times, the garden people counter is **not the sum** of the counts displayed. **Counter** increments have been lost. **Why?**
Avoid interference by synchronization

class SynchronizedCounter extends Counter {

    SynchronizedCounter(NumberCanvas n) {
        super(n);
    }

    synchronized void increment() {
        super.increment();
    }
}
Mutual Exclusion: The Ornamental Garden - DEMO

**Go**

WEST 20

COUNTER 40

EAST 20

DEM0
Monitors

- each object
  - has attached to it a unique lock
  - and thus: can act as monitor

- 3 important monitor operations
  - `o.wait()`: release lock on `o`, enter `o`’s wait queue and wait
  - `o.notify()`: wake up one thread in `o`’s wait queue
  - `o.notifyAll()`: wake up all threads in `o`’s wait queue

- executable by a thread “inside” the monitor represented by `o`
- executing thread must hold the lock of `o`/executed within synchronized portions of code
- typical use: `this.wait()` etc.
- note: notify does not operate on a thread-identity

```java
Thread t = new MyThread();
...
  t.notify(); // mostly to be nonsense
```

---

6there are more
7technically, a thread identity is represented by a “thread object” though.
Condition synchronization, scheduling, and signaling

- quite simple/weak form of monitors in Java
- only one (implicit) condition variable per object: availability of the lock. threads that wait on \( o \) (\( o\text{.wait()} \)) are in this queue.
- ordering of wait “queue”: implementation-dependent (usually FIFO)
- signaling discipline: S & C
- awakened thread: no advantage in competing for the lock to \( o \).
- no built-in support for general-purpose condition variables.
- note: monitor-protection not enforced
  - private field modifier \( \neq \) instance private
  - not all methods need to be synchronized\(^8\)
  - besides that: there’s re-entrance!

\(^8\)remember: find of oblig-1.
/ * down() = P operation
  * up() = V operation

```java
public class Semaphore {
    private int value;

    public Semaphore(int initial) {
        value = initial;
    }

    synchronized public void up() {
        ++value;
        notifyAll();
    }

    synchronized public void down() throws InterruptedException {
        while (value == 0) wait();
        --value;
    }
}
```

- cf. also `java.util.concurrent.Semaphore` (acquire/release + more methods)
Mutual exclusion with semaphores

Thread 1

Thread 2

Thread 3

Mutex

0

Run  Pause  Run  Pause  Run  Pause
class MutexLoop implements Runnable {

    Semaphore mutex;

    MutexLoop (Semaphore sema) {mutex=sema;}

    public void run() {
        try {
            while (true) {
                while (!ThreadPanel.rotate());
                // get mutual exclusion
                mutex.down();
                while (ThreadPanel.rotate()); // critical section
                //release mutual exclusion
                mutex.up();
            }
        } catch (InterruptedException e) {} 
    }
}
A shared database is accessed by two kinds of processes. Readers execute transactions that examine the database while Writers both examine and update the database. A Writer must have exclusive access to the database; any number of Readers may concurrently access it.
interface ReadWrite {
    public void acquireRead() throws InterruptedException;
    public void releaseRead();
    public void acquireWrite() throws InterruptedException;
    public void releaseWrite();
}
class Reader implements Runnable {

    ReadWrite monitor_;

    Reader(ReadWrite monitor) {
        monitor_ = monitor;
    }

    public void run() {
        try {
            while (true) {
                while (!ThreadPanel.rotate());
                // begin critical section
                monitor_.acquireRead();
                while (ThreadPanel.rotate());
                monitor_.releaseRead();
            }
        } catch (InterruptedException e) {
        }
    }
}
class **Writer** implements Runnable {

    ReadWrite **monitor**_;

    **Writer**(ReadWrite **monitor**) {
        **monitor**_ = **monitor**;
    }

    **public** **void** run() {
        try {
            **while**(true) {
                **while**(=!ThreadPanel.rotate());
                // begin critical section
                **monitor**_.acquireWrite();
                **while**(ThreadPanel.rotate());
                **monitor**_.releaseWrite();
            }
            **catch**(InterruptedException **e**){
            }
        }
    }
}
class ReadWriteSafe implements ReadWrite {
    private int readers = 0;
    private boolean writing = false;

    public synchronized void acquireRead() throws InterruptedException {
        while (writing) wait();
        ++readers;
    }

    public synchronized void releaseRead() {
        --readers;
        if (readers == 0) notifyAll();
    }

    public synchronized void acquireWrite() {...}
    public synchronized void releaseWrite() {...}
}
```java
class ReadWriteSafe implements ReadWrite {
    private int readers = 0;
    private boolean writing = false;

    public synchronized void acquireRead() throws InterruptedException {
        while (writing) wait();
        ++readers;
    }

    public synchronized void releaseRead() {
        --readers;
        if (readers == 0) notifyAll();
    }

    public synchronized void acquireWrite() {
        ...}

    public synchronized void releaseWrite() {
        ...}
}
```
class ReadWriteSafe implements ReadWrite {
    private int readers = 0;
    private boolean writing = false;

    public synchronized void acquireRead() {
        // ...
    }

    public synchronized void releaseRead() {
        // ...
    }

    public synchronized void acquireWrite() throws InterruptedException {
        while (readers > 0 || writing) wait();
        writing = true;
    }

    public synchronized void releaseWrite() {
        writing = false;
        notifyAll();
    }
}
ReadWriteFair
readers = 0  writing = true
class ReadWriteFair implements ReadWrite {

    private int readers = 0;
    private boolean writing = false;
    private int waitingW = 0; // no of waiting Writers.
    private boolean readersturn = false;

    synchronized public void acquireRead() throws InterruptedException {
        while (writing || (waitingW > 0 && !readersturn)) wait();
        ++readers;
    }

    synchronized public void releaseRead() {
        --readers;
        readersturn = false;
        if (readers == 0) notifyAll();
    }

    synchronized public void acquireWrite() { ... }
    synchronized public void releaseWrite() { ... }
}
"Fairness": regulating writers

```java
class ReadWriteFair implements ReadWrite {

    private int readers = 0;
    private boolean writing = false;
    private int waitingW = 0;  // no of waiting Writers.
    private boolean readersturn = false;

    synchronized public void acquireRead() {...}
    synchronized public void releaseRead() {...}

    synchronized public void acquireWrite() throws InterruptedException {
        ++waitingW;
        while (readers > 0 || writing) wait();
        --waitingW; writing = true;
    }

    synchronized public void releaseWrite() {
        writing = false; readersturn = true;
        notifyAll();
    }
}
```
Readers and Writers problem

DEMO
there’s (much) more to it than what we discussed (synchronization, monitors) (see java.util.concurrent)
Java’s memory model: since Java 1: loooong, hot debate
connections to
  - GUI-programming (swing/awt/events) and to
  - RMI etc.
major clean-up/repair since Java 5
better “thread management”
Lock class (allowing new Lock() and non block-structured locking)
one simplification here: Java has a (complex!) weak memory model (out-of-order execution, compiler optimization)
not discussed here volatile
shared, mutable state is more than a bit tricky,\(^a\) watch out!
- work thread-local if possible
- make variables immutable if possible
- keep things local: encapsulate state
- learn from tried-and-tested concurrent design patterns

\(^a\)and pointer aliasing and a weak memory model makes it worse.

golden rule

never, ever allow (real, unprotected) races

- unfortunately: no silver bullet
- for instance: “synchronize everything as much as possible”: not just inefficient, but mostly nonsense
  ⇒ concurrent programming remains a bit of an art

see for instance [GPB\(^+\)06] or [Lea99]
*Foundations of Multithreaded, Parallel, and Distributed Programming.*
Addison-Wesley, 2000.

[GPB+06] Briand Goetz, Tim Peierls, Joshua Bloch, Josheph Bowbeer, David Holmes, and Doug Lea.  
*Java Concurrency in Practice.*
Addison-Wesley, 2006.

*Concurrent Programming in Java: Design Principles and Patterns.*

[MK99] Jeff Magee and Jeff Kramer.  
*Concurrency: State Models and Java Programs.*