INF4140 - Models of concurrency

Fall 2016

October 7, 2016

Abstract

This is the "handout" version of the slides for the lecture (i.e., it's a rendering of the content of the slides in a way that does not waste so much paper when printing out). The material is found in [Andrews, 2000]. Being a handout-version of the slides, some figures and graph overlays may not be rendered in full detail, I remove most of the overlays, especially the long ones, because they don't make sense much on a handout/paper. Scroll through the real slides instead, if one needs the overlays.

This handout version also contains more remarks and footnotes, which would clutter the slides, and which typically contains remarks and elaborations, which may be given orally in the lecture.

1 Java concurrency

10. 10. 2016

1.1 Threads in Java

Outline

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/notify)
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 accessed/changed only through the available *procedures*
- Implicit mutex: Only one 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, achieved through *condition variables*.

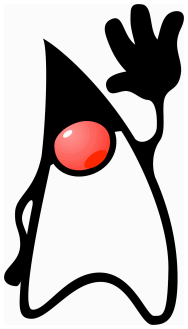
  Signaling disciplines

  - 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

Java

From Wikipedia:[1]

" ... Java is a general-purpose, concurrent, class-based, object-oriented language ..."

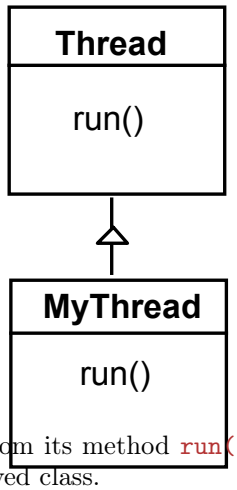


Threads in Java

A thread in Java

- unit of concurrency[2]
- originally "green threads"
- 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

Thread class



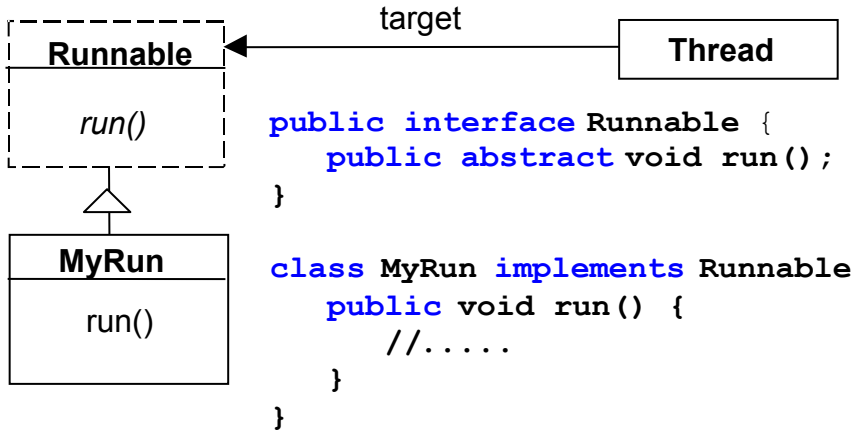
The `Thread` class executes instructions from its method `run()`. The actual code executed depends on the implementation provided for `run()` in a derived class.

```
class MyThread extends Thread {
  public void run() {
    //......
  }
}
  // Creating a thread object:
  Thread a = new MyThread();
  a.start();
```

[1] But it's correct nonetheless ...
[2] as such, roughly corresponding to the concept of "processes" from previous lecctures.
[3] What's the difference to `this`?

Runnable interface

no multiple inheritance ⇒, often implement the `run()` method in a class not derived from `Thread` but from the interface `Runnable`.




```
public interface Runnable {
    public abstract void run();
}

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

```
// Creating a thread object:
Runnable b = new MyRun();
new Thread(b).start();
```

Threads in Java

steps to *create* a thread and get it running:

1. Define class that
   - *extends* the Java Thread class or
   - *implements* the Runnable interface
2. define *run* method inside the new class[4]
3. create an instance of the new class.
4. *start* the thread.

Interference and Java threads

```
...
class Store {
  private int data = 0;
  public void update() { data++; }
}
...

// in a method:
Store s = new Store(); // the threads below have access to s
t1 = new FooThread(s); t1.start();
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 ⇒ 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. two "flavors" of synchronization:

"synchronized block"

```
synchronized (o) { B }
```

[4] overriding, late-binding.
[5] but: in a re-entrant manner!

synchronized method

whole method body of m "protected"[6]:

```
synchronized Type m(...) { ... }
```

Protecting the initialization

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

```
class Store {
  private int data = 0;

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

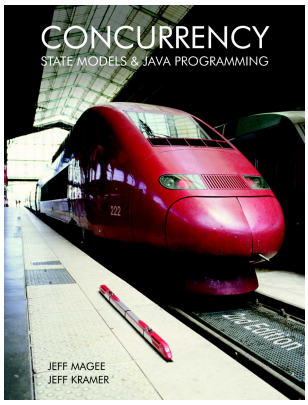
or

```
class Store {
  private int data = 0;

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

// inside a method:
Store s = new Store();
```

Java Examples

Book:

Concurrency: State Models & Java Programs, 2^{nd} Edition
Jeff Magee & Jeff Kramer
Wiley



`http://www.doc.ic.ac.uk/~jnm/book/`

Examples in Java:

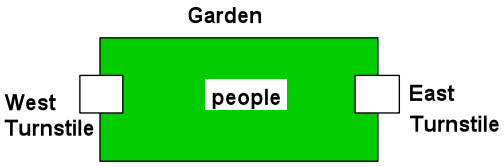
`http://www.doc.ic.ac.uk/~jnm/book/book_applets`

[6] assuming that other methods play according to the rules as well etc.

1.2 Ornamental garden

Ornamental garden problem

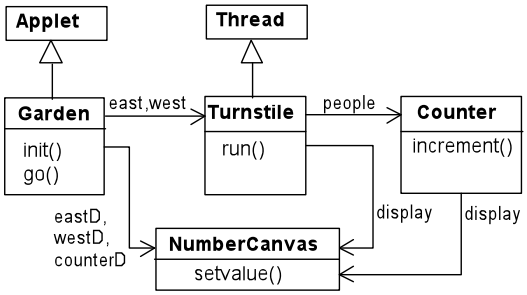
- people enter an ornamental garden through either of 2 turnstiles.
- problem: the number of people present at any time.



The concurrent program consists of:

- 2 threads
- shared counter object

Ornamental garden problem: Class diagram



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.

Counter

```
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);
      }
 }
```

Turnstile

```
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) { }
    }
}
```

Ornamental Garden Program

The Counter object and Turnstile threads are created by the go() method of the Garden applet:

```
private void go() {
  counter = new Counter(counterD);
  west = new Turnstile(westD,counter);
  east = new Turnstile(eastD,counter);
  west.start();
  east.start();
}
```

Ornamental Garden Program: DEMO

DEMO

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?

`http://www.doc.ic.ac.uk/~jnm/book/book_applets/Garden.html`

Avoid interference by synchronization

```
class SynchronizedCounter extends Counter {

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

  synchronized void increment() {
    super.increment();
  }
}
```

Mutual Exclusion: The Ornamental Garden - DEMO

DEMO

1.3 Thread communication, monitors, and signaling

Monitors

- *each* object
  - has attached to it a unique *lock*
  - and thus: can act as monitor
- 3 important monitor operations[7]
  - *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[8]

⇒

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

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*.wait()) are in this queue
- no built-in support for general-purpose condition variables.
- ordering of wait "queue": implementation-dependent (usually FIFO)
- signaling discipline: S & C
- awakened thread: no advantage in competing for the lock to *o*.
- note: monitor-protection not enforced (!)
  - `private` field modifier ≠ instance private
  - not all methods need to be synchronized[9]
  - besides that: there's re-entrance!

[7] there are more

[8] technically, a thread identity is represented by a "thread object" though. Note also : `Thread.suspend()` and `Thread.resume()` are deprecated.

[9] remember: `find` of oblig-1.

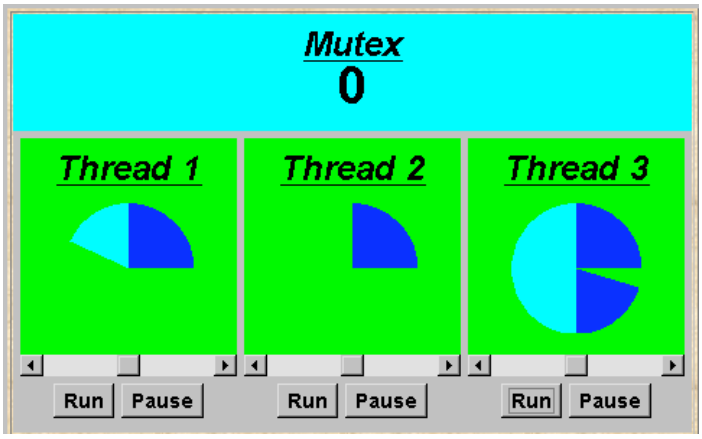
A semaphore implementation in Java

```
  // down() = P operation
  // up()   = V operation

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();//the well-known while-cond-wait pattern
    --value;}
}
```

- cf. also `java.util.concurrency.Semaphore` (acquire/release + more methods)

1.4 Semaphores

Mutual exclusion with sempahores



The graphics shows waiting and active phases, plus value of mutex. **Note:** Light blue for active phase, other colours for waiting.

Mutual exclusion with sempahores

```
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){}
    }
}
```

DEMO

Panel is an (old) AWT class (applet is a subclass). It's the simplest container class. The function `rotate` returns a *boolean*. It's a `static` method of the thread subclass `DisplayThhread`.

`http://www.doc.ic.ac.uk/~jnm/book/book_applets/Garden.html`

1.5 Readers and writers

Readers and writers problem (again...)

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 R/W

```
interface ReadWrite {

   public void acquireRead() throws InterruptedException;

   public void releaseRead();

   public void acquireWrite() throws InterruptedException;

   public void releaseWrite();
 }
```

Reader client code

```
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){}
    }
}
```

Writer client code

```
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){}
    }
}
```

R/W monitor (regulate readers)

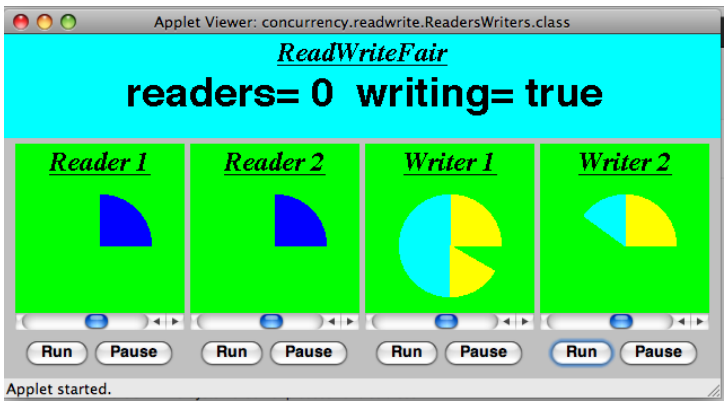
```
class ReadWriteSafe implements ReadWrite {
  private int readers =0;
  private boolean writing = false;

  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() {...}
}
```

R/W monitor (regulate writers)

```
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();
  }
}
```

DEMO

Fairness



http://www.doc.ic.ac.uk/~jnm/book/book_applets/ReadWriteFair.html

"Fairness": regulating readers

```
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

```
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

`http://www.doc.ic.ac.uk/~jnm/book/book_applets/ReadersWriters.html`

Java concurrency

- there's (much) more to it than what we discussed (synchronization, monitors) (see `java.util.concurrency`)
- 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`

General advice

shared, mutable state is more than a bit tricky,[10] 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

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 programmig remains a bit of an art

see for instance [Goetz et al., 2006] or [Lea, 1999]

References

[Andrews, 2000] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley.

[Goetz et al., 2006] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. (2006). *Java Concurrency in Practice.* Addison-Wesley.

[Lea, 1999] Lea, D. (1999). *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 2d edition.

[Magee and Kramer, 1999] Magee, J. and Kramer, J. (1999). *Concurrency: State Models and Java Programs.* John Wiley & Sons Inc.

[10] and pointer aliasing and a weak memory model makes it worse.