Limits of von-Neumann (thread-based) computing

Jian-Jia Chen
(Slides are based on
Peter Marwedel)
TU Dortmund
Informatik 12

These slides use Microsoft clip arts.
Microsoft copyright restrictions apply.

Why not use von-Neumann (thread-based) computing (C, C++, Java, …) ?

Potential race conditions (☞inconsistent results possible)
☞ Critical sections = sections at which exclusive access to resource r (e.g. shared memory) must be guaranteed.

```
thread a {
  ..
  P(S)  //obtain lock
  ..     // critical section
  V(S)  //release lock
}
```

```
thread b {
  ..
  P(S)  //obtain lock
  ..     // critical section
  V(S)  //release lock
}
```

Race-free access to shared memory protected by S possible

This model may be supported by:

- mutual exclusion for critical sections
- special memory properties

Why not just use von-Neumann computing (C, Java, …) (2)?

Problems with von-Neumann Computing

- Thread-based multiprocessing may access global variables
- We know from the theory of operating systems that
  - access to global variables might lead to race conditions,
  - to avoid these, we need to use mutual exclusion,
  - mutual exclusion may lead to deadlocks,
  - avoiding deadlocks is possible only if we accept performance penalties.
- Other problems (need to specify total orders, …)

Consider a Simple Example

"*The Observer pattern defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically*."

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, Addision-Wesley, 1995

Example: Observer Pattern in Java

```
public void addListener(listener) {…}

public void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```

Would this work in a multithreaded context?

Thanks to Mark S. Miller for the details of this example.

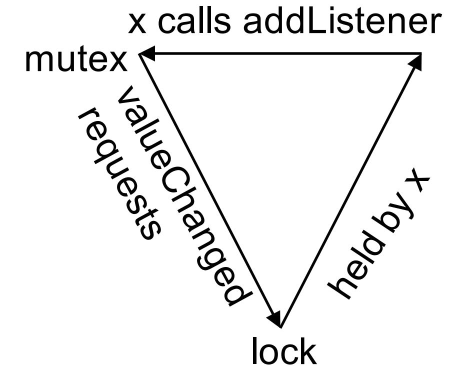
Example: Observer Pattern with Mutual Exclusion (mutexes)

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```

Javasoft recommends against this.
What's wrong with it?

Mutexes using monitors are minefields

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```



valueChanged() may attempt to acquire a lock on some other object and stall. If the holder of that lock calls addListener(): deadlock!

Simple Observer Pattern becomes not so simple

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
   synchronized (this) {
      myValue=newValue;
      listeners=myListeners.clone();
   }
   for (int i=0; i<listeners.length; i++) {
     listeners[i].valueChanged(newValue)
   }
}
```

while holding lock, make a copy of listeners to avoid race conditions

notify each listener outside of the synchronized block to avoid deadlock

This still isn't right.
What's wrong with it?

Simple Observer Pattern: How to Make it Right?

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
      listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value-changes in the wrong order!

Why are deadlocks possible?

We know from the theory of operating systems, that deadlocks are possible in a multi-threaded system if we have

- Mutual exclusion
- Holding resources while waiting for more
- No preemption
- Circular wait

Conditions are met for our example

© JJ Chen and P.Marwedel, Informatik 12, 2019

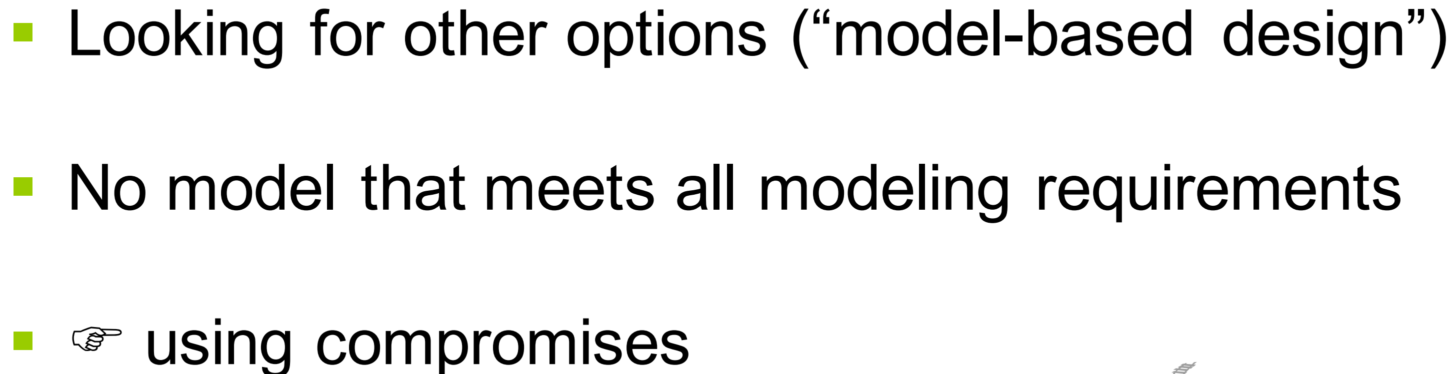
A stake in the ground …

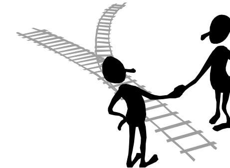
Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans.

*"… **threads as a concurrency model are a poor match for embedded systems**. … they work well only … where best-effort scheduling policies are sufficient."*

Edward Lee: Absolutely Positively on Time, *IEEE Computer*, July, 2005

Ways out of this problem





© JJ Chen and P.Marwedel, Informatik 12, 2019