Java Concurrency - Part 1

CS-E4110 Concurrent Programming

Keijo Heljanko

Department of Computer Science
Aalto University School of Science

September 28th, 2016

Slides by Jan Lönnberg, Teemu Kiviniemi, Jaakko Harjuhahto and Keijo Heljanko

On the previous lecture... 1/3

Computational model

- States and state transitions
- Concluded that the model applies to multitasking and multiprocessing systems
- Took a quick look at how statements are executed on the CPU
- Noted that assignment must satisfy the Limited Critical Reference (LCR) -condition in order to be atomic for our model
- This abstract model is preferable to real life systems due to its simplicity and clarity. We can force concrete systems to implement our model.

On the previous lecture... 2/3

Correctness

- Sequential correctness relative to pre- and post-conditions
- A concurrent program is *safe* with regards to *invariant* which holds always.
- A concurrent program satisfies a *liveness criteria* T w.r.t to a state S, if started in S it will eventually reach a state satisfying T.
- Weak fairness assumption: a statement that is continuously ready, will be executed eventually.
- Formal logic (propositional and temporal) is an the preferred tool for defining invariants and discussing programs in general

On the previous lecture... 3/3

Critical section

- Executing an arbitrary sequence of actions atomically
- Mutual exclusion for a block of code
- Applied state diagrams and traces to analyse some (flawed) sample algorithms for the critical section problem
- Dekker's Algorithm is one of many known solutions to the critical section problem

Section 1

Threads

Threads

What are threads?

- **Threads** in Java are concurrent streams of execution within a Java Virtual Machine (JVM).
- Depending on implementation, they may run on separate processors or through time-slicing of one or more processors.
- Java threads may or may not correspond to the underlying operating system's notion of threads.
- The VM may create additional **daemon** threads for its own housekeeping needs.

Threads

How do I use threads?

- The `main` method is invoked in the main thread, which can create more threads.
- Threads can be manipulated using `Thread` objects.
- The static method `Thread.currentThread()` can be used to get the current thread.
  - This is mostly useful for debugging or demonstrating thread behaviour.
  - Can be bad practice for a production environment
- `Threads` contain a `run()` method that contains the code executed in the thread.
  - This is typically where you put your own code.

Threads

Creating threads (one way)

- Extend the `java.lang.Thread` class with your own `run()` method and other thread-local stuff:

```
1 public class ThreadDemo extends Thread {
2     private int value = 1; /* Each ThreadDemo gets its own value. */
3     public void run() {
4         /* This method is executed in the new thread. */
5         while(value < 10000) {
6             System.out.println(value++);
7         }
8     }
```

Starting threads

- When the `start()` method of a `Thread` is called, a new thread is started and the `run()` method is executed in the new thread.
- Start two instances of the `ThreadDemo` thread:

```
10      public static void main(String[] args) {
11          Thread demo1 = new ThreadDemo();
12          Thread demo2 = new ThreadDemo();
13          demo1.start();
14          demo2.start();
15      }
16  }
```

Threads

Creating and starting threads (another way)

- Alternatively, create a `java.lang.Runnable` with the `run()` method.
- Use the `Thread(Runnable)` constructor to use the `Runnable`'s `run()` method instead of the `Thread`'s.

```
1 public class RunnableDemo implements Runnable {
2     private int value = 1;
3     public void run() {
4         /* This method is executed in the new thread. */
5         while(value < 10000) {
6             System.out.println(value++);
7         }
8     }
```

Threads

Creating and starting threads (another way)

- Alternatively, create a `java.lang.Runnable` with the `run()` method.
- Use the `Thread(Runnable)` constructor to use the `Runnable`'s `run()` method instead of the `Thread`'s.

```
10        public static void main(String[] args) {
11            Thread demo1 = new Thread(new RunnableDemo());
12            Thread demo2 = new Thread(new RunnableDemo());
13            demo1.start();
14            demo2.start();
15        }
16    }
```

Letting other threads execute

- A running thread may want to give the other threads a chance to run
- `Thread.yield()` lets the current thread pause and let other threads run.
  - "The scheduler is free to ignore this hint." This method may do nothing.
  - Using this often leads to **busy-waiting** (continuously rechecking a wait condition) on some platforms.
  - Busy-waiting threads may prevent threads that could actually do something useful from getting a chance to run.
  - Even if every thread gets to execute, it is tremendously wasteful of CPU power.
    - Other processes and threads now have less time for actual work.
    - Power-saving (and cooling) measures such as underclocking idle CPUs do not work if the CPU is busy-waiting.
  - This method is mostly useful for testing and debugging.

Threads

Letting other threads execute

- `Thread.sleep(long)` makes a thread sleep for a time given in milliseconds.
  - This still does not guarantee that other threads get a chance to execute.
    - If there are lots of threads waiting like this, the system may spend all its time switching back and forth between a few of them.
    - Other processes can prevent your program from running at all for a while.
  - Increasing the time to sleep introduces unnecessarily long delays.
  - `sleep`ing can be a good idea when doing animation or similar tasks.
- Clearly, we need a way to tell the system that a thread does not need to run until another thread has done its job.

Waiting for threads to end

- When the `join()` method of a `Thread` is called, the calling thread waits until the `run()` method is completed in the referenced thread.
- Start two instances of the `ThreadDemo` thread and wait for them to complete:

```
3       Thread demo1 = new ThreadDemo();
4       Thread demo2 = new ThreadDemo();
5       demo1.start();
6       demo2.start();
7       try {
8           demo1.join();
9           demo2.join();
10      } catch(InterruptedException ie) { /* Unreachable in this example */}
```

Threads

Waiting for threads to end

- The end of a thread **happens-before** the completion of the `join()`.
- Similarly, the call to `start()` happens-before the start of a thread in the Java memory model.
- Hence, you can safely write data for a helper thread to use, start it, wait for it to end and then read any value the helper thread has written.
- This can be used to pass processing over to a helper thread for simple parallel processing.
- The happens-before relationship is part of the Java memory model, basics of which will be explained in the following.

Section 2

Threads and Memory

Threads and Memory

Sharing variables between threads

- A thread can access all the variables visible in its current scope.
- Local variables (including parameters) are only accessible to the thread running the method.
- Threads share other variable values (fields in objects and classes, array components) in main memory.

Threads and Memory

Sharing variables between threads

- A thread may cache variable values in its own local working memory.
    - A thread may avoid writing its data back to main memory or reading new values from main memory unless forced to do so.
    - The order in which the shared main memory is accessed is not specified unless syncronization actions are used (the compiler or runtime environment may reorder operations to speed up execution); unexpected combinations of writes may be visible.

Threads and Memory

A thread may cache variable values in its own local working memory.

- There are tons of possible reasons for this, including different coherence policies on different multiprocessor systems due to, e.g., different compilers, different JVMs, different hardware memory models, and so on.
- Even if your SMP system guaranteed sequential consistency memory model memory coherence (which slow and thus not implemented by commercial microprocessors), the JIT compiler can take your bytecode and generate optimised machine code that does things in a different order.

Synchronisation actions

- **Synchronisation actions** can be used to ensure one thing **happens-before** another in another thread. If *a* happens-before *b*, the effects of *a* will be visible to *b*. If not, *b* may or may not see what *a* did.
  - Operations in a thread happen-before following operations in the same thread.
  - The happen-before relation is a partial order
  - As noted previously, starting a thread happens-before the thread starts and a thread ending happens-before another thread notices that it has ended (through e.g. `join()`).
  - Default initialisation of variables (the `0`, `false` or `null` automatically written to new non-local variables) happens-before all threads start.

The *happens-before* Relationship

Happens-before relationship is...

- Transitive: $hb(a, b) \wedge hb(b, c) \Rightarrow hb(a, c)$
- Antisymmetric: $hb(a, b) \Rightarrow a = b \vee \neg hb(b, a)$
- Naturally visualized as a directed, acyclic graph
- Not the same as "A happens before B" in real time! The logical relationship and the order in which instructions are executed on the CPU are two different things.

The HB-relationship is defined in the Java Language Specification

- `http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5`

Threads and Memory

Bad things can happen

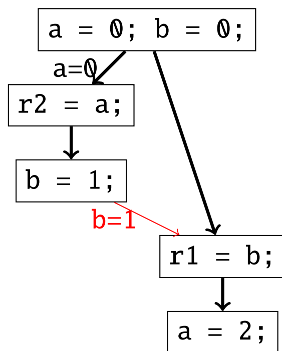
- If proper synchronisation is not done:
  - "Normal" non-deterministic behaviour can occur (interleaving of operations).
  - Other threads may not see changes to shared variables.
  - `double` and `long` variables may have incorrect values (these reads and writes are not guaranteed to be atomic).
- If there are two operations on a variable such that one of them is a write and which are not ordered by **happens-before**, we have a **data race**.
- Correct synchronisation (freedom from data races) for shared variables is always required on assignments of this course.
  - . . . unless you can prove that the program behaves correctly wrt. the quite subtle Java memory model.

Threads and Memory - Two Threads with Races

Example

```
2        public static int a, b;

7                  int r2 = a;
8                  b = 1;
9                  System.out.println("r2␣=␣" + r2);

14                 int r1 = b;
15                 a = 2;
16                 System.out.println("r1␣=␣" + r1);
```

Output:
r2 = 0
r1 = 1

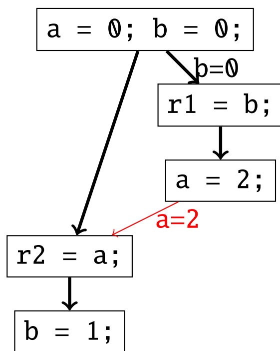


Threads and Memory - Two Threads with Races

Example

```
2        public static int a, b;

7                  int r2 = a;
8                  b = 1;
9                  System.out.println("r2␣=␣" + r2);

14                 int r1 = b;
15                 a = 2;
16                 System.out.println("r1␣=␣" + r1);
```

Output:
r1 = 0
r2 = 2



Threads and Memory - Two Threads with Races

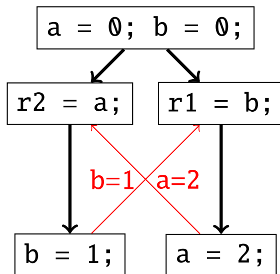
Example

```
2        public static int a, b;

7                int r2 = a;
8                b = 1;
9                System.out.println("r2␣=␣" + r2);

14               int r1 = b;
15               a = 2;
16               System.out.println("r1␣=␣" + r1);
```

Output:

r2 = 2

r1 = 1



Section 3

Volatile variables

`volatile` variables

Ordering of reads and writes

- All reads and writes to the same `volatile` variables are totally ordered.
- Every access to a `volatile` variable uses the shared main memory; each write to the variable happens-before the following reads of the volatile variable.
- By transitivity on happens-before: This means that all writes of any other variables of the thread writing the volatile variable also happen-before any reads of a thread done after reading the volatile variable.

`volatile` variables

Non-atomic reads and writes

- Declaring a `long`/`double` variable `volatile` makes its reads & writes atomic.
- `long`/`double` read/write operations may otherwise consist of two operations.
    - 32-bit machines often read and write 64-bit values in two steps.
    - A read operation may see half of one value and half of the other.
    - Irrespective of their size, references are always read and written atomically.

`volatile` variables

Declaring

- Variables can be declared volatile with the `volatile` keyword.

```
private volatile int a = 10;
```

- Declaring a reference variable `volatile` affects only the reference to the object, **not the object's contents**.
- Volatile variables can be useful as flag variables, but are of limited use in practice.

Section 4

Mutual exclusion

Protecting variables

Problem

A variable is written to in one thread and read or written in another without synchronization.

Avoidance

- Look for **all** variables that can be modified in one thread and read or modified in another.
- Declare these shared variables `volatile`.
- Alternatively, make the variables immutable (`final`).

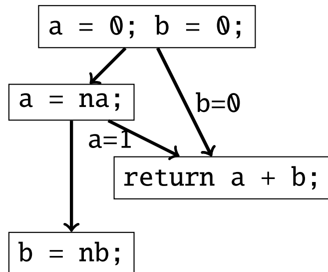
Protecting variables

Problem

`volatile` is not enough to ensure a consistent state for several variables.

```
1  public class BadExample {
2      private volatile int a, b;
3      void setValues(int na, int nb) {
4          a = na;
5          b = nb;
6      }
7      int getSum() {
8          return a + b;
9      }
10 }
```

`getSum()` during `setValue(1,1)` from initial value of `(0,0)` results in 1.



Mutual exclusion

Mutual exclusion

- Mutual exclusion is necessary to ensure multiple variables are read and written atomically.
- Java provides **locks** that can be used to solve this problem.

. . . in Java

- Each `Object` has an associated **intrinsic lock**.
  - Locking is done by declaring parts (methods or blocks) of the code `synchronized`.
  - The parts are mutually excluded from each other, i.e. executed as object-specific critical sections, since entering a `synchronized` block or method acquires the lock and leaving releases it.
  - The feature is inherited by all classes from the `Object` root class.

Mutual exclusion

synchronized blocks

- To ensure ownership of `myObject`'s lock:

```
14    public void exampleMethod() {
15        synchronized(myObject) {
16            /* code here will be executed only
17               by one thread at a time */
18        }
19    }
```

Mutual exclusion

synchronized methods

- A whole method may be declared synchronised by using the `synchronized` keyword.

```
9      public synchronized void myMethod() {
10         /* code here will be executed only
11             by one thread at a time */
12     }
```

Monitors

synchronized methods versus blocks

- A `synchronized` method works exactly as if the body was in a `synchronized` block on `this`:

```
2    public void myMethodAlternate() {
3        synchronized(this) {
4            /* code here will be executed only
5               by one thread at a time */
6        }
7    }
```

Monitors

How the `synchronized` keyword works

- Before a thread exits a `synchronized` block (normally, or by throwing a `Throwable`), it releases the lock.
- Before a thread starts executing a `synchronized` block, it acquires the object's lock (waiting if not available).
- Any variable values written before a lock is released are guaranteed to be visible after the lock is then acquired; the release happens-before the next time the lock is acquired.
- In other words, changes written in a `synchronized` block are guaranteed to be visible to other code `synchronized` on the same monitor; `volatile` is not necessary in this case.

Mutual exclusion

How the `synchronized` keyword works

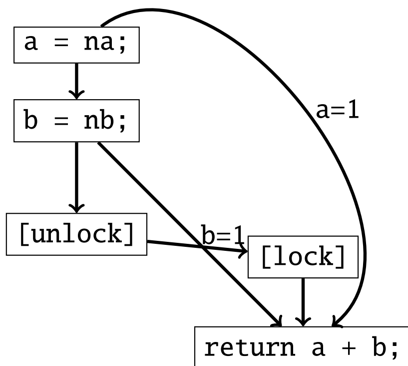
- Locks are **re-entrant**; a thread that already owns the lock can acquire it again without deadlocking.
  - Example: A `synchronized` block is called from another `synchronized` block of the same object.
  - Example: `a.m1()` calls `b.m1()`, which calls `a.m2()` when both methods `m1` and `m2` of object `a` are `synchronized`.
- In such a case the thread gives up the lock only after exiting the last `synchronized` block or method.

Mutual exclusion

Example

Preventing the race condition in `BadExample`:

```
1  public class SynchronizedExample {
2      private int a, b;
3      synchronized void setValues(int a, int b) {
4          this.a = a;
5          this.b = b;
6      }
7      synchronized int getSum() {
8          return a + b;
9      }
10 }
```



Mutual exclusion

Piggybacking

- In the above example, the lock has two important effects:

  Mutual exclusion: The obvious consequence of a lock

  Ensuring happens-before ordering: Making sure data written before the lock was released can safely be read after it is acquired.

- Often, you only need the latter, e.g. when `synchronized` methods are used to hand off data (used only by one thread at a time) from one thread to another.
- In many cases you can **piggyback** the transfer of data between threads on a happens-before relationship induced by the internal locking of a shared data structure.
  - Many Java library classes (e.g. thread-safe collections) have this property; more on this next time.

Circular locking dependencies

Problem!

```
9    synchronized(object1) {
10       synchronized(object2) {
11           // Accessing object1 and object2
12       }
13   }

21   // And in another thread...
22   synchronized(object2) {
23       synchronized(object1) {
24           // Accessing object1 and object2
25       }
26   }
```

Circular locking dependencies

Problem

Two or more threads trying to acquire locks the other holds lead to deadlock.

Avoidance

- If possible, use `synchronized` only on `this`.
- While holding a lock, avoid calling methods with locking on another object or otherwise acquiring another lock.
- If necessary, merge several locks into one.
- When nesting locks, always use the same locking order.

Conclusion

Tips

- Remember the K.I.S.S. principle: “Keep It Short and Simple”:
  - The fewer variables you need to share between threads . . .
  - The fewer objects that are used by several threads . . .

  . . . the easier it is to implement proper synchronisation.
- Go through your solution step-by-step.
- Be completely sure that your implementation works correctly in all situations.
- You can’t assume anything about the thread scheduler. Always prepare for the worst situation.
- Test your code thoroughly.

Conclusion

Questions?

- Have you got any questions?

Next time

- Java monitors for waiting for conditions
- The `java.util.concurrent` library
- First programming assignment overview

Conclusion

References

- The Java Language Specification, Java SE 7 Edition
  - Especially Chapter 17.
  - `http://java.sun.com/docs/books/jls/`
- Java™ 2 Platform, Standard Edition 7 API Specification
  - Especially classes `java.lang.Object` and `java.lang.Thread`.
  - `http://docs.oracle.com/javase/7/docs/api/`
- Goetz et al.: Java Concurrency in Practice
  - Especially Chapters 2, 3, 14 and 16.