PV168

Memory Model

Retrospective

- Did you understand what was done in the last seminar?
- Was it intuitively understandable?
- Are you able to describe how it works?
- Did you find it difficult?

Monitor

- The goal is to prevent concurrent access using the critical section.
- Do not use multiple monitors on the same data.
- Each object instance and the class itself has a monitor.
    - Do not mix them.

Monitor

- Where to place `synchronized`?
  - in the method declaration
  - in the `static` method declaration
  - in its own code block

Monitor

```
class Counter {
    private int value = 0;
    public synchronized int getValue() {
        return value++;
    }

    public int getSameValue() {
        synchronized(this) {
            return value++;
        }
    }
}
```

Atomic numbers

- Useful for *just-a-counter* cases.
- Lower overhead
  - Monitors are quite expensive.
  - Can you say why?

Atomic numbers

```
class Counter {
    private AtomicInteger value = new AtomicInteger(0);
    public int getValue() {
        return value.getAndIncrement();
    }
}
```

But how monitors and atomics work?

Memory Model

> “ In computing, a memory model describes the interactions of threads through memory and their shared use of the data.[1] ”

- Essential for any programming environment using parallelism.
- Consists of:
    - Language specification
    - Compiler specification (including optimizations)
    - Hardware (JVM in Java)

[1]: https://en.wikipedia.org/wiki/Memory_model_(programming)

Memory Model

- Specifies: [2]
  - synchronization points and actions
  - what is affected by synchronization points
  - what applies to memory access
    - before the synchronization point
    - after the synchronization point
- Defines *Sequential Consistency* and *Happens-before* relation.

[2]: https://docs.oracle.com/javase/specs/jls/se17/html/jls-17.html#jls-17.4

Sequential Consistency

> “ Compilers are allowed to reorder the instructions, when this does not affect the execution of the thread in isolation. [2] ”

- Also, the JIT may reorder, delay, or cache accesses.
- Also, the JVM may reorder, delay, or cache accesses.
- Also, the CPU may reorder, delay, or cache accesses.

[2]: https://docs.oracle.com/javase/specs/jls/se17/html/jls-17.html#jls-17.4

Sequential Consistency

```
A = B = 0

// thread 1
r2 = A;
B = 1

// thread 2
r1 = B;
A = 2;
```

- What are the values of `r1` and `r2`?
  - It may happen that `r1 == 1` and `r2 == 2`.

Synchronization Points

- Java has 4 types:
  - `volatile`
  - Atomic classes
  - `synchronized`
  - starts and joins of threads

Actions

- Action types:
  - Load
  - Store
  - Synchronization
- Loads without any store are trouble-free.
- Any store requires a proper synchronization.
  - Including all load accesses!
- Synchronization locks and unlocks on the monitor.

Shared variables

- Shared variables (e.g. affected by the *Memory Model*):
  - instance fields
  - `static` fields
  - array elements
- Variables unaffected by the *Memory Model*:
  - local variables
  - methods arguments, catched exceptions

Happens-before

- Transitive relationship of actions.
  - “ If one action happens-before another, then the first is visible to and ordered before the second [3] ”
- Applies to all synchronization points.
- Partially applies to all `final` fields.
  - After construction, no further synchronization is needed.

[3]: https://docs.oracle.com/javase/specs/jls/se17/html/jls-17.html#jls-17.4.5

`volatile`

- Implements the memory barrier:
  - accesses to `volatile`s cannot be reordered
  - subsequent accesses cannot be reordered **before** `volatile` load
  - prior accesses cannot be reordered **after** `volatile` store
- Numeric operations are not atomic (`++`, `--`)
  - Store of 64 bit values is done in two steps.
- **DO NOT USE IT UNLESS YOU KNOW WHAT YOU ARE DOING!**

volatile

```
class VolatileExample {
    int x = 0;
    volatile boolean v = false;
    public void writer() {
        x = 42;
        v = true;
    }

    public void reader() {
        if (v == true) {
            //uses x - guaranteed to see 42.
        }
    }
}
```

Atomic classes

- Useful for numeric counters, gauges, and indicators.
- Thread safe for all basic numeric operations.
  - Increments, decrements, assignment
- "Synchronizes properly as you would expect."
  - Almost the same rules applies as for `volatile`s.
  - Allows finer specification of memory ordering.
- **DO NOT USE FOR OTHER THINGS THAN COUNTERS UNLESS YOU KNOW WHAT YOU ARE DOING!**

Atomic classes

```
class AtomicExample {
    private int value = 0;
    private AtomicBoolean flag = new AtomicBoolean(false);
    public void writer() {
        value = 42;
        flag.set(true);
    }

    public void reader() {
        while (!flag.get()) {} // busy wait here
        //uses value - guaranteed to see 42.
    }
}
```

`synchronized`

- When used in a method signature, it uses the monitor of the instance.
  - Or monitor of the class in case the method is `static`.
- When used in the body of a method, it uses the monitor of the specified object.

```
public synchronized void doSomething() { /* ... */ }
public void doSomethingElse() {
    synchronized(this.attribute) {/* ... */}
}
```

Threads

- Thread start and join have similar memory semantics as `synchronized`:
  - Any store which *happened-before* the start of the thread is visible by the thread.
  - Any store which *happened-before* the join of the *about-to-join* thread is visible in the joinee thread.

Threads

```java
class ThreadExample {
    public void run() {
        Integer value = 42;
        var t = new Thread(() -> value *= 2 );
        t.start();
        t.join();
        // value is guaranteed to be 84;
    }
}
```

`final` Fields

- Special rules apply to `final` fields: [4]
  - Set the final fields for an object in that object's constructor.
  - Do not write a reference to the object being constructed in a place where another thread can see it before the object's constructor is finished.
  - If this is followed, then when the object is seen by another thread, that thread will always see the correctly constructed version of that object's final fields.

[4]: https://docs.oracle.com/javase/specs/jls/se17/html/jls-17.html#jls-17.5

Double-Checked Locking

Is this correct?

```
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
}
```

Double-Checked Locking Is Broken

And there is no way how it can be fixed.

How To Test The Correct Usage?

- Unit tests are insufficient.
  - In fact, no commonly used automatization approaches can validate the correctness.
- To prove the correctness, one must provide a formal proof.
  - This is the reason airplanes work in strictly single-threaded environments.

Without correct synchronization, very strange, confusing and counterintuitive behaviors are possible.