

Thread Safety (Part Two)

Concurrent and Parallel Programming

Locks and context-switches

- ▶ **For clarification:** locks do not prevent threads from being interrupted by context-switches. Indeed, threads might be interrupted even when executing in the protected portion of the code.
- ▶ Instead, locks **ensure that only one thread at a time is allowed to execute the protected portion of code.**
- ▶ At the practical level, for the time of execution of the protected portion of code, **the program runs as if it were mono-threaded.**



But how to find out which portions of code need to be protected with mutexes?

- ▶ In a multi-threaded program, multiple streams of instruction are allowed to execute simultaneously and be interrupted by **context-switches**.
- ▶ There are situations in which the sequence of operations executed by a thread needs to run **from the first operation to completion, without any external modification of the used value** (e.g. by other threads).

- ▶ The main reason are dependencies between the values in variables/fields or between the executed operations.

If dependencies are present,
the sequence of operations must be:

ATOMIC!

Meaning indivisible.



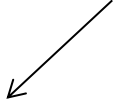
Compound actions

- ▶ **Compound actions:** are sequences of operations that need to execute atomically to be thread-safe.
- ▶ Compound actions A and B are **atomic** if a thread can only execute A when B has not yet executed or when B has already completely executed (and vice versa).
- ▶ **An action is atomic,** if it is atomic in relation to all other actions that share the same program state (variables/fields), including itself.

Example

```
class FibonacciCalculator implements Runnable {  
    private static int count = 2;  
    private FibonacciNumber number = null;  
  
    public FibonacciCalculator(FibonacciNumber number) {  
        this.number = number;  
    }  
  
    public void run() {  
        count++;  
        number.setNewValue(number.getPreviousValue() +  
                             number.getCurrentValue());  
        System.out.println("The " + count + " Fibonacci number is " +  
                             number.getCurrentValue());  
    }  
}
```

Compound
Action!



Locks and shared variables

- ▶ *Each shared and mutable variable has to be protected with a lock. The same lock must be used both for write and read operations.*
- ▶ If there are dependencies between variables/fields, all the involved variables/fields **must be protected with the same lock.**
- ▶ The rule: **each group of shared and mutable variables/fields needs its own lock!**

In general:

- ▶ If a compound action (e.g. a modification of two dependent variables) is not performed atomically, ... a **vulnerability window** occurs between the moment when one of the two variables has been modified and the other one is not yet modified.
- ▶ If, due to an unlucky overlap of operations, an external modification of one of the two variables occurs right in the middle of the vulnerability window, **this modification might get lost, overwritten earlier than required. As a consequence, an inconsistent execution might occur.**

Race-condition details

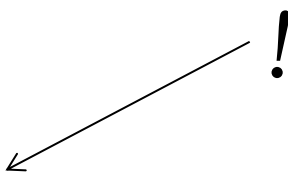
- ▶ The most common type of race condition is called **"check-then-act"**: a (potential wrong) read is used to take a decision on how the program executes (example: lazy initialization).
- ▶ Another common type of race condition is called **"read-modify-write"**: the state of an object is modified based on its (potentially wrong) previous value (example: ++var).

Example: lazy initialization

```
public class Fruit {  
    private static Map<String, Fruit> types = new HashMap<>();  
    private String type;  
  
    private Fruit(String type) {  
        this.type = type;  
    }  
  
    public static Fruit getFruit(String type) {  
        if (!types.containsKey(type)) {  
            types.put(type, new Fruit(type)); // Lazy initialization  
        }  
        return types.get(type);  
    }  
}
```

Example: lazy initialization

```
public class Fruit {  
    private static Map<String, Fruit> types = new HashMap<>();  
    private String type;  
  
    private Fruit(String type) {  
        this.type = type;  
    }  
  
    public static synchronized Fruit getFruit(String type) {  
        if (!types.containsKey(type)) {  
            types.put(type, new Fruit(type)); // Lazy initialization  
        }  
        return types.get(type);  
    }  
}
```



A diagram consisting of a black arrow pointing from the `getFruit` method call in the `main` method to the `types` static map in the `getFruit` method. A large black exclamation mark is placed next to the arrow, indicating a significant event or warning, such as the lazy initialization of the static map.

Example: read-modify-write

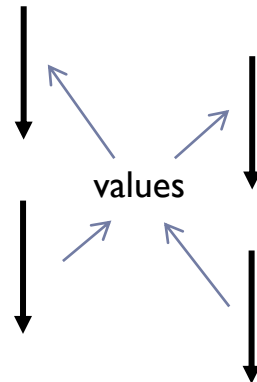
```
public class RandomGenerator {  
    private static final int BASE_RND_SEED = 1;  
    private static final int BASE_RND_CONST = 32767;  
    private static final int BASE_RND_BASE = 1664525;  
  
    private int uiRndSeed = BASE_RND_SEED;  
  
    public int generate() {  
        int tmp = uiRndSeed;  
        tmp = tmp * BASE_RND_BASE;  
        tmp = tmp + BASE_RND_CONST;  
        uiRndSeed = tmp;  
        return uiRndSeed;  
    }  
}
```

Example: read-modify-write

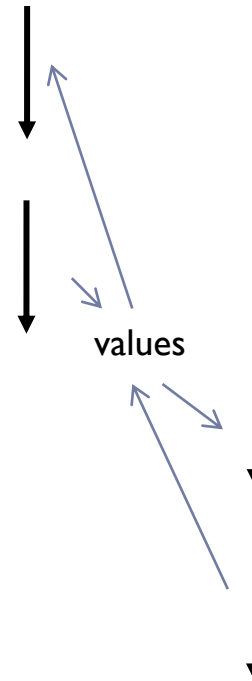
```
public class RandomGenerator {  
    private static final int BASE_RND_SEED = 1;  
    private static final int BASE_RND_CONST = 32767;  
    private static final int BASE_RND_BASE = 1664525;  
  
    private int uiRndSeed = BASE_RND_SEED;  
  
    ! →  
    public synchronized int generate() {  
        int tmp = uiRndSeed;  
        tmp = tmp * BASE_RND_BASE;  
        tmp = tmp + BASE_RND_CONST;  
        uiRndSeed = tmp;  
        return uiRndSeed;  
    }  
}
```

Race condition vs. synchronization

Race
condition



Synchronization



Concurrency problems

Let's now focus on the second concurrency problem:
correct visibility of the memory by multiple threads.

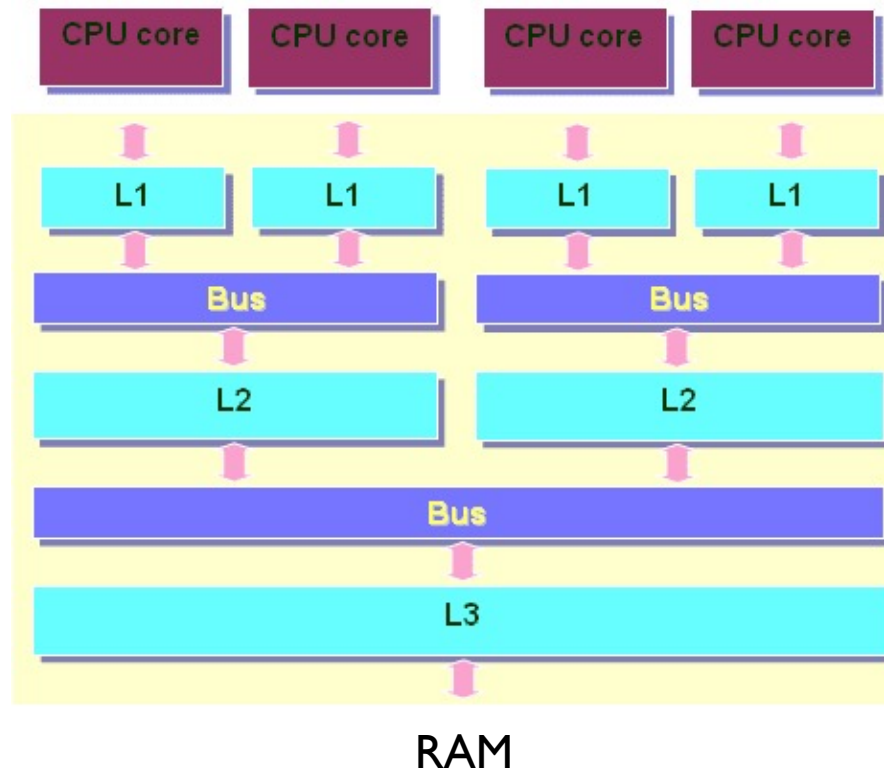
Registers and caches

- ▶ **Registers and caches** are small and fast memories available in the processor, that contain copies of the information (code and data) present in main memory.
- ▶ **Registers and caches** are used in CPUs to reduce the average memory access time.
(L1 miss = 10 cycles, L2 miss = 200 cycles)
- ▶ The information is loaded in the caches from the main memory in portions called **cache-lines**.
- ▶ There are caches **both for the data that has to be processed as well as for the instructions that have to be executed**.

Registers and caches

- ▶ Registers and caches take advantage of the **principle of locality of processing** at data or code level.
- ▶ By populating the caches with values and instruction **that are frequently used in a some period of time**, the program execution benefits from performance improvements.
- ▶ Registers and caches are essential ... **the main memory is too far away** (and therefore too slow) from the processor to be directly used for every single access.
 - ▶ *The current trend is that the main memory becomes larger and more distant, while the processors becomes faster.*
 - ▶ *As a result, the caches need also to be increasingly larger, or an alternative solution has to be used (e.g. distributed memories).*

Multi-core processors



Example

```
public class TestVolatile extends Thread {
    private boolean keepRunning = true;

    @Override
    public void run() {
        long count = 0;
        while (keepRunning) {
            count++;
        }
        System.out.println("Thread terminated." + count);
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("Program started.");
        TestVolatile t = new TestVolatile();
        t.start();
        Thread.sleep(1000);
        t.keepRunning = false;
        System.out.println("keepRunning set to false.");
    }
}
```

Output: Program started.
keepRunning set to false.

Consistency of caches

- ▶ A **mono-threaded** program accesses the information in registers and caches in a transparent way.
- ▶ However, in a **multi-threaded** program, it is **not granted that a thread reading information from the caches is able to read the most recent values written** by other threads.
- ▶ The leading cause of the problem are registers and caches, but other optimizations performed by the processor (e.g. out-of-order execution) might have an influence.
- ▶ If these problems are not correctly managed, consequences might appear in terms of **threads reading inconsistent information**.

Consistency of caches

- ▶ To solve the cache consistency problem, processors provide specific hardware features, such as the **directory-based cache coherence** mechanism.

Consistency of caches

- ▶ However, **it is not beneficial** to keep the caches constantly consistent.
- ▶ Every time a cache is made consistent, **the data from the modified cache has to be copied to the main memory; then, reloaded in all the other caches from main memory.**
- ▶ This represents a big overhead! At the practical level, **the positive effects** introduced by the caches are **inhibited**.
- ▶ The operation should therefore only be carried out if mandatorily needed.

Memory barriers

- ▶ As a consequence, in order to program with flexibility, processors provides specific instructions:

memory barriers (also called memory fences): low-level machine code that do not modify any data, but **grants a defined order of operations on the memory.**

- ▶ Practically speaking, memory barriers temporarily activate the hardware-provided features for **cache consistency.**

Memory barriers

- ▶ In addition, memory barriers **disable all processor optimizations** (e.g. out-of-order execution), that **might cause additional problems.**

- ▶ But when do memory barriers have to be activated?
Only for **shared and mutable data!**

Memory barriers


- ▶ Therefore, in languages such as Java, it is normally not required to call the memory barrier instructions directly.
 - ▶ To **protect the shared and mutable data** from race conditions, we already use synchronization primitives, such as the **"synchronized" keywords or explicit locks**.
 - ▶ For sake of simplicity, these synchronization primitives also automatically activate the memory barriers!
- ▶ The synchronization primitives have therefore a double role: **thread-safety with respect to race conditions, as well as with respect to visibility problems!**

```
public class TestReentrantLock extends Thread {
    private Lock lock = new ReentrantLock();
    private boolean keepRunning = true;

    @Override
    public void run() {
        long count = 0;
        while (checkKeepRunning()) {
            count++;
        }
        System.out.println("Thread terminated." + count);
    }

    private boolean checkKeepRunning() {
        lock.lock();
        try {
            return keepRunning;
        } finally {
            lock.unlock();
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    System.out.println("Program started.");
    TestReentrantLock t = new TestReentrantLock();
    t.start();
    Thread.sleep(1000);
    t.lock.lock();
    try {
        t.keepRunning = false;
    } finally {
        t.lock.unlock();
    }
    System.out.println("keepRunning set to false.");
}
```



Output: Program started.
keepRunning set to false.
Thread terminated. 62691308

Memory barriers

- ▶ In addition, memory barriers are also activated when the **start() and join() methods of threads are called.**
- ▶ Usually, when threads are started, initial values are provided to threads by using constructors.
- ▶ As a consequence, there's normally **no risk of memory consistency problems during the start and termination** of threads.
- ▶ However, **beware of race conditions at initialization and termination!** Memory barriers do not automatically solve that type of problem too.

Volatile variables

- ▶ The Java language provides a light form of synchronization tool: **volatile** variables.
- ▶ When a variable is declared 'volatile', the compiler and the JVM know that **memory barriers have to be activated for every access to the variable.**
- ▶ **At the practical level,** the situation is similar as if the value of the variable would be always accessed from main memory.
- ▶ Consequently, reading a volatile variable **always** returns **the most recent value**, even if the variable is accessed (read and write) by multiple threads simultaneously.

Example

```
public class TestVolatile extends Thread {
    private volatile boolean keepRunning = true;

    @Override
    public void run() {
        long count = 0;
        while (keepRunning) {
            count++;
        }
        System.out.println("Thread terminated." + count);
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("Program started.");
        TestVolatile t = new TestVolatile();
        t.start();
        Thread.sleep(1000);
        t.keepRunning = false;
        System.out.println("keepRunning set to false.");
    }
}
```

Output: Program started.
keepRunning set to false.
Thread terminated.1656361131

Volatile variables

- ▶ Volatile variables might prove useful, but have their **limits**. Common exploitations are for **completion, interrupt or status flags**.
- ▶ The semantics of volatile variables is **NOT** sufficiently strong to grant the atomicity of compound actions, even for simple operations such as `++var` (*unless the variable is always increased by just a single thread*).
- ▶ Volatile variables **only** grant **correct visibility**.

Volatile variables

- ▶ **Volatile variables should only be used when:**
 - ▶ Writing a variable does not depend on its previous value (there is no risk of a race condition of type read-modify-write), or only a single thread is allowed to update the variable's value.
 - ▶ The variable has no dependencies with other variables in compound actions.
 - ▶ Locking the variable (e.g. with “synchronized”) is not required for other reasons.

Volatile variables: advantages

- ▶ Synchronization **with low overhead compared to locks** (works at single variable level).
- ▶ Reduce the scheduling overhead because are not blocking (differently than locks).
- ▶ Provide excellent **scalability** (when the number of threads is increased) and good **liveness** capabilities. For example are immune to deadlocks.

Atomic variables

- ▶ To overcome the weaknesses of volatile variables (no guarantee of atomicity), **atomic variables** have been introduced in Java 5 and extended in later versions of Java.
- ▶ Atomic variables **use volatile variables internally**, but add some additional functionality.
- ▶ Like volatile variables, **atomic variables** are a light form of synchronization tool, with additional support for **atomicity** for a defined family of **compound actions**.
- ▶ The classes for atomic variables are provided in the **java.util.concurrent.atomic** package.

Atomic variables

Class	Description
AtomicBoolean	A boolean value that may be updated atomically.
AtomicInteger	An int value that may be updated atomically.
AtomicIntegerArray	An int array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.
AtomicLong	A long value that may be updated atomically.
AtomicLongArray	A long array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.
AtomicMarkableReference<V>	An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.
AtomicStampedReference<V>	An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.
DoubleAccumulator	One or more variables that together maintain a running double value updated using a supplied function.
DoubleAdder	One or more variables that together maintain an initially zero double sum.
LongAccumulator	One or more variables that together maintain a running long value updated using a supplied function.
LongAdder	One or more variables that together maintain an initially zero long sum.

Volatile vs. atomic variables

- ▶ **Atomic variables** support **atomic read-modify-write** operations. Can therefore be used in substitution to volatile variables, in situations where atomic updates are needed.
- ▶ To implement this behavior, atomic variables use special-purpose low-level instructions provided by processors.

Example: AtomicInteger

Modifier and Type	Method and Description
int	accumulateAndGet (int x, IntBinaryOperator accumulatorFunction) Atomically updates the current value with the results of applying the given function to the current and given values, returning the updated value.
int	addAndGet (int delta) Atomically adds the given value to the current value.
boolean	compareAndSet (int expect, int update) Atomically sets the value to the given updated value if the current value == the expected value.
int	decrementAndGet () Atomically decrements by one the current value.
double	doubleValue () Returns the value of this AtomicInteger as a double after a widening primitive conversion.
float	floatValue () Returns the value of this AtomicInteger as a float after a widening primitive conversion.
int	get () Gets the current value.
int	getAndAccumulate (int x, IntBinaryOperator accumulatorFunction) Atomically updates the current value with the results of applying the given function to the current and given values, returning the previous value.
int	getAndAdd (int delta) Atomically adds the given value to the current value.
int	getAndDecrement () Atomically decrements by one the current value.
int	getAndIncrement () Atomically increments by one the current value.

Example: AtomicInteger

int	getAndSet (int newValue) Atomically sets to the given value and returns the old value.
int	getAndUpdate (IntUnaryOperator updateFunction) Atomically updates the current value with the results of applying the given function, returning the previous value.
int	incrementAndGet () Atomically increments by one the current value.
int	intValue () Returns the value of this AtomicInteger as an int.
void	lazySet (int newValue) Eventually sets to the given value.
long	longValue () Returns the value of this AtomicInteger as a long after a widening primitive conversion.
void	set (int newValue) Sets to the given value.
String	toString () Returns the String representation of the current value.
int	updateAndGet (IntUnaryOperator updateFunction) Atomically updates the current value with the results of applying the given function, returning the updated value.
boolean	weakCompareAndSet (int expect, int update) Atomically sets the value to the given updated value if the current value == the expected value.

Example

```
class AtomicRunner implements Runnable {  
    private static AtomicInteger count = new AtomicInteger();  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 5; i++) {  
            performOperation(i);  
            count.incrementAndGet();  
        }  
    }  
  
    public static int getCount() {  
        return count.get();  
    }  
  
    private void performOperation(int i) {  
        // simulates an operation  
        try {  
            Thread.sleep(i * 1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Example

```
public class AtomicIntegerExample {  
    public static void main(String[] args) throws InterruptedException {  
        Thread t1 = new Thread(new AtomicRunner(), "t1");  
        Thread t2 = new Thread(new AtomicRunner(), "t2");  
        t1.start();  
        t2.start();  
        Thread.sleep(10000);  
        System.out.println("Processing count = " + AtomicRunner.getCount());  
        t1.join();  
        t2.join();  
    }  
}
```

Example of output:
Processing count = 10

Example: AtomicReference

Modifier and Type	Method and Description
V	accumulateAndGet(V x, BinaryOperator<V> accumulatorFunction) Atomically updates the current value with the results of applying the given function to the current and given values, returning the updated value.
boolean	compareAndSet(V expect, V update) Atomically sets the value to the given updated value if the current value == the expected value.
V	get() Gets the current value.
V	getAndAccumulate(V x, BinaryOperator<V> accumulatorFunction) Atomically updates the current value with the results of applying the given function to the current and given values, returning the previous value.
V	getAndSet(V newValue) Atomically sets to the given value and returns the old value.
V	getAndUpdate(UnaryOperator<V> updateFunction) Atomically updates the current value with the results of applying the given function, returning the previous value.
void	lazySet(V newValue) Eventually sets to the given value.
void	set(V newValue) Sets to the given value.
String	toString() Returns the String representation of the current value.
V	updateAndGet(UnaryOperator<V> updateFunction) Atomically updates the current value with the results of applying the given function, returning the updated value.
boolean	weakCompareAndSet(V expect, V update) Atomically sets the value to the given updated value if the current value == the expected value.

Atomic variables: advantages

- ▶ Synchronization **with low overhead compared to locks** (works at single variable level).
- ▶ Reduce the scheduling overhead because are not blocking (differently than locks).
- ▶ Provide excellent **scalability** (when the number of threads is increased) and good **liveness** capabilities. For example are immune to deadlocks.
- ▶ Are an improved version of volatile variables (atomic updates are also supported).
- ▶ Can be used to develop **non-blocking** algorithms (will be introduced later).

Visibility problems

To sum up:

- ▶ if synchronization tools are not used correctly, **memory visibility** (as well as race condition) problems may appear:
 - ▶ in the absence of cache coherence, **old values might be read from the caches**. These **dirty readings** can generate inconsistent application executions (throw of exception, corrupted data structures, infinite loops, incorrect calculations, ...).
- ▶ **Synchronization tools must be** used to grant correct memory visibility. These are, for example: **“synchronized” keyword, explicit locks, volatile variables, and atomic variables.**

Summary of topics

- ▶ Atomicity and compound actions
- ▶ Check-then-act and read-modify-write
- ▶ Registers and caches in multi-core processors
- ▶ Consistency of caches: memory barriers
- ▶ Volatile variables
- ▶ Atomic variables