Thread-Safety (Part Two)

Concurrent and Parallel Programming

Advantages and disadvantages of threads

- Advantages:
  - Profit from multi-core processors
  - Improve the performances (even on single-core processor, by taking advantage of multi-tasking)
  - Hide complex scheduling and synchronization logic (performed by the operating system)
- Disadvantages:
  - Potential thread-safety risks
  - Switching between threads and synchronization operations introduce performance overheads

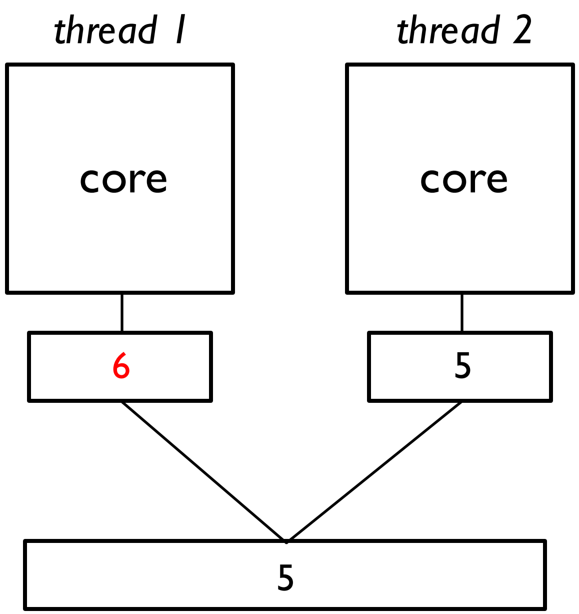
Review

- In a program that executes with multiple threads:
  - Problem 1 (visibility): data in registers or caches of a core can be modified by one of the threads, but the introduced modifications could remain invisible to the other threads. The consequence could be dirty reads (reads of old data).
  - Problem 2 (race condition): threads can run in parallel and be interrupted (context-switches) in the middle of what they are doing. Meanwhile, other threads could modify data in shared memory regions.

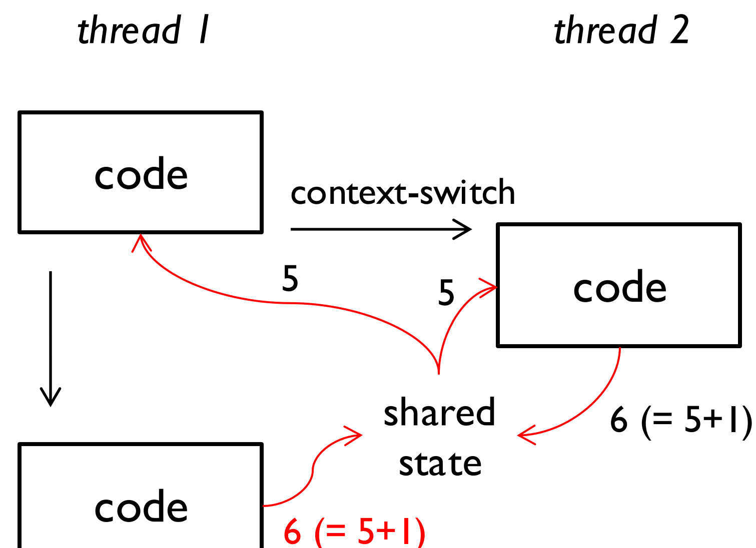
These two types of problems can generate

inconsistent executions of the application.

Review



Visibility problem



Race condition problem

Shared and mutable state

- Writing multi-thread programs implies correctly managing how threads access the program state (variables and objects).
- Care must be taken for variables and objects that are shared and mutable!
  - A variable/object is shared if it is possible to access the variable/object from multiple threads simultaneously.
  - A variable/object is mutable (modifiable) if its value can be modified during its life cycle.

Shared and mutable state

The following reasoning approach can be used:

- each thread has its own execution stack. Therefore, the information in the stack is never shared (values in local variables and parameters).
- In contrast, all the information in the heap and static variables can potentially be shared and modified by the running threads.

- To understand if the shared state is mutable, it is required to verify if there are any value assignments (writes) to the involved memory regions.

Atomicity

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., performed by other threads).

Atomicity

- 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.

SUPSI

Dipartimento tecnologie innovative

Example

```
class Performer implements Runnable {
  private int id;
  private SharedState state = null;

  public Performer(int id, SharedState state) {
    this.id = id;
    this.state = state;
  }

  public void run() {
    state.lock.lock();
    try {
      while (!state.stop) {
        state.value++;
        state.lock.unlock();

        // ... do other operations

        state.lock.lock();
        if (state.value == 500000) {
          System.out.println("ID: " + id + " - value " +
              "reached 500000. Reset to 0.");
          state.value = 0;
        } else if (state.value >= 1000000) {
          System.out.println("ID: " + id + " - value " +
              "reached 1000000. The program will stop!");
          state.stop = true;
        }
      }
    } finally {
      state.lock.unlock();
    }
  }
}
```

```
class SharedState {
  Lock lock = new ReentrantLock();
  int value = 0;
  boolean stop = false;
}
```

Must be atomic!

Types of race condition

- 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 ReentrantLock lock = new ReentrantLock();
    private static Map<String, Fruit> types = new HashMap<>();
    private String type;

    private Fruit(String type) {
        this.type = type;
    }

    public static Fruit getFruit(String type) {
        lock.lock();
        try {
            if (!types.containsKey(type)) {
                types.put(type, new Fruit(type)); // Lazy initialization
            }
            return types.get(type);
        } finally {
            lock.unlock();
        }
    }
}
```

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 ReentrantLock lock = new ReentrantLock();
    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() {
        lock.lock();
        try {
            int tmp = uiRndSeed;
            tmp = tmp * BASE_RND_BASE;
            tmp = tmp + BASE_RND_CONST;
            uiRndSeed = tmp;
            return uiRndSeed;
        } finally {
            lock.unlock();
        }
    }
}
```

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.

Compound actions

- *Each shared and mutable variable/field need to be protected with a lock (or at least the volatile keyword, if there's no risk of race condition). The same lock must be used both for all 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 is: each group of shared and mutable variables/fields needs its own lock!

General rule!

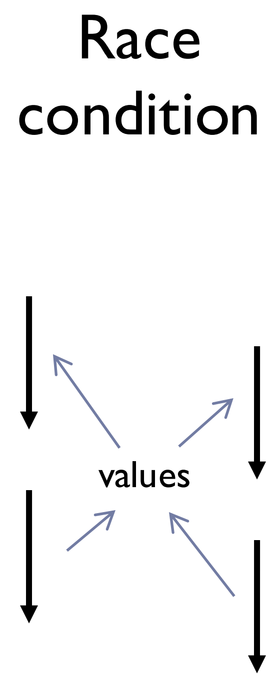
- When is it required to use synchronization tools?

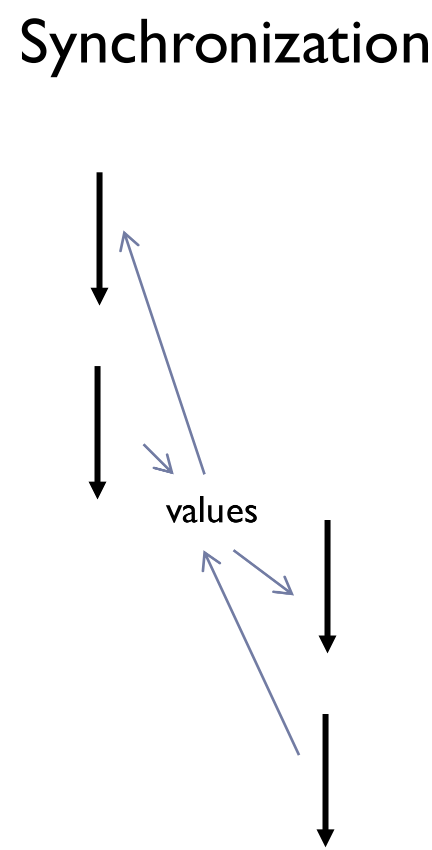
  If there is a shared and mutable state, ALWAYS!

  For EVERY READ and EVERY WRITE operation!

- To protect from visibility problems, at least volatile must be used.
- Depending on the type of compound action and the related potential race-condition, locks or other types of synchronization tools need to be used.

Race condition vs. synchronization





General rule!

If a variable/field is not shared, or is not mutable, you do not need to use locks or other synchronization solutions.

On the other hand, if a variable/field is shared and mutable, synchronization is required every time the variable/field is accessed, both for writing and reading operations!

A common mistake made by beginners is to add synchronization only when writing variables/fields, but not when reading them. THIS APPROACH IS USELESS!

Atomic variables

- Volatile variables might proof 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.

Atomic variables

- To overcome the weaknesses of volatile variables, 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.

SUPSI

Dipartimento tecnologie innovative

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.

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 the processors.

SUPSI

Dipartimento tecnologie innovative

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.

SUPSI

Dipartimento tecnologie innovative

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

Volatile vs. atomic variables

Advantages of volatile variables:

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

Volatile vs. atomic 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.

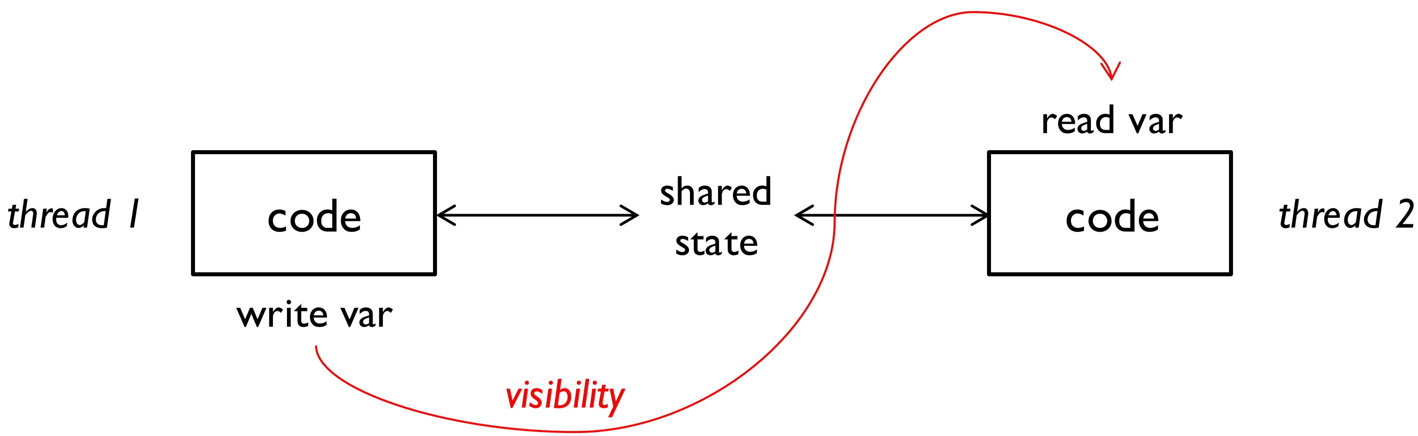
Volatile vs. atomic variables

Advantages of atomic variables:

- Synchronization with low overhead compared to locks (works at single variable level).
- Reduce the scheduling overhead because are not blocking (instead locks are blocking).
- 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).

Volatile/atomic vars and visibility

- The visibility effect of volatile and atomic variables extends beyond the volatile/atomic variable itself. If a thread A writes a volatile/atomic variable, which is then read from thread B, the value of all variables visible to A before writing the volatile/atomic variable, becomes correctly visible to B after reading the volatile/atomic variable.



Example

```
public class TestVolatileVisibility extends Thread {
    private int a;
    private int b;
    private volatile int c;

    @Override
    public void run() {
        while (c == 0); // Do nothing
        System.out.println("Thread ended. " + a + ", " + b + ", " + c);
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("Program started.");
        TestVolatileVisibility t = new TestVolatileVisibility();
        t.start();
        Thread.sleep(1000);
        t.a = 10;
        t.b = 20;
        t.c = 30;
        System.out.println("Program ended.");
    }
}
```

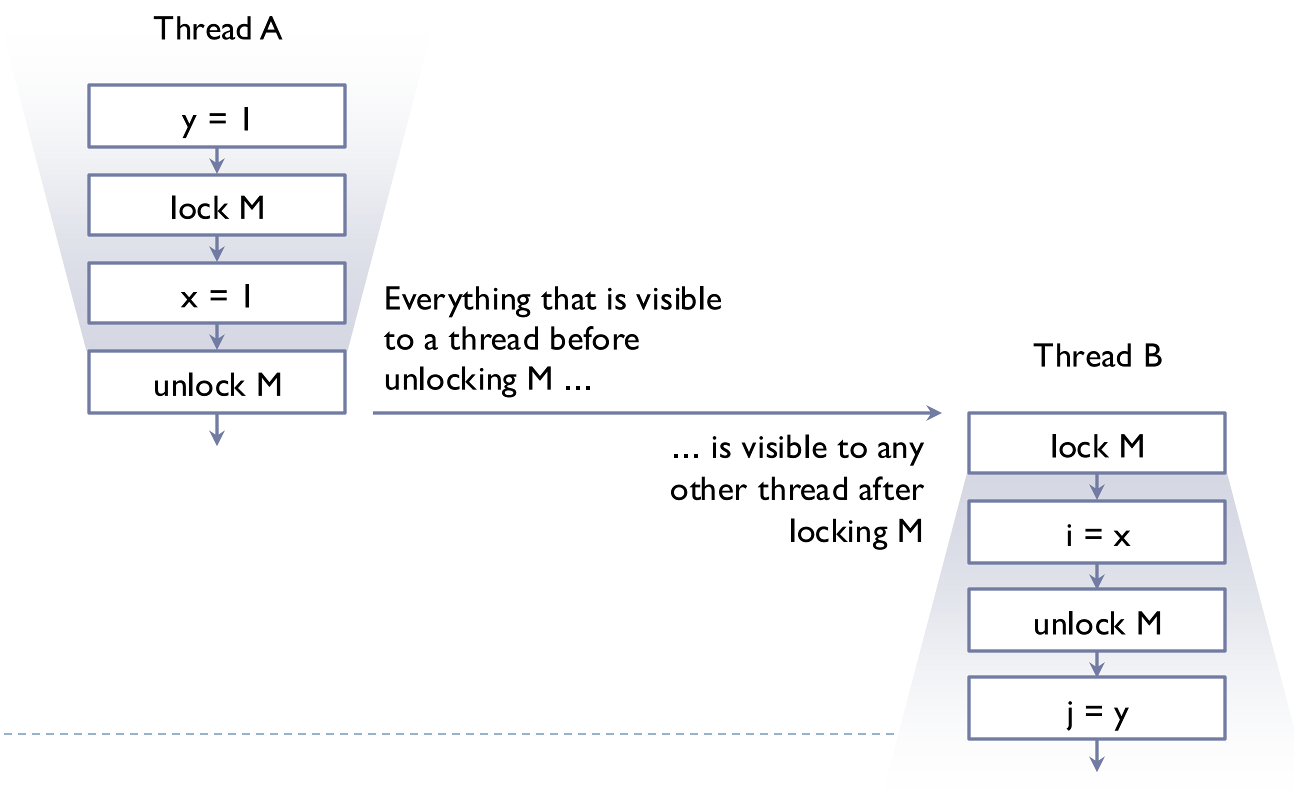
Output:Program started.
Program ended.
Thread ended. 10, 20, 30

Mutexes and visibility

- As previously introduced, mutexes also ensure correct memory visibility.
- For all threads to see the latest modifications of all other threads on shared and mutable variables, all threads (either performing read or write operations) must synchronize on a common lock.

Mutexes and visibility

- The visibility effect of locks works in the following way: anything visible to a thread before the lock is released, will be correctly visible to any thread using the same lock, from the moment the lock is acquired.



Example

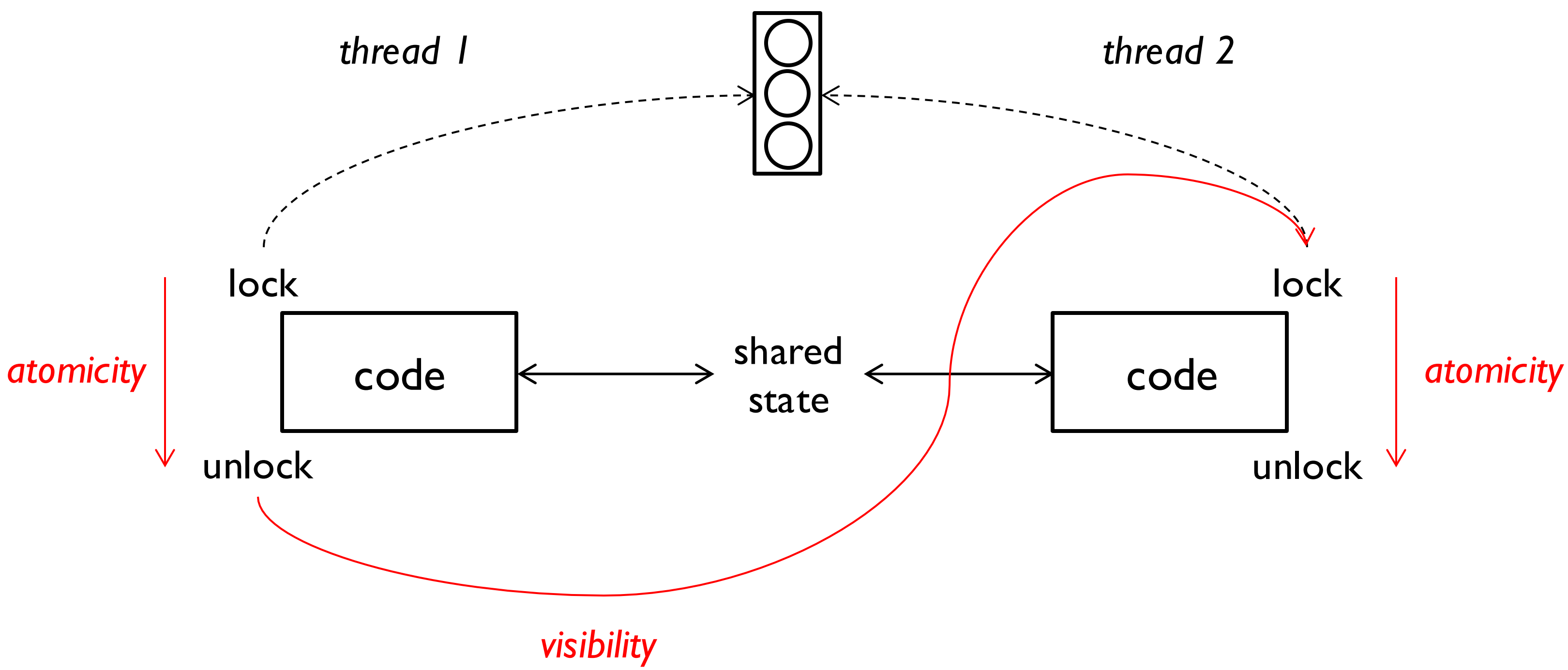
```
public class TestMutexVisibility extends Thread {
    private int a = 0;
    private int b = 0;
    private int c = 0;

    ReentrantLock lock = new ReentrantLock();

    @Override
    public void run() {
        int temp = 0;
        lock.lock();
        try {
            temp = c;
        } finally {
            lock.unlock();
        }

        while (temp == 0) {
            // Do nothing
            lock.lock();
            try {
                temp = c;
            } finally {
                lock.unlock();
            }
        }
        System.out.println("Thread ended. " + a + "," + b + "," + c);
    }
```

Example

```java
public static void main(String[] args) throws InterruptedException {
    System.out.println("Program started.");
    TestMutexVisibility t = new TestMutexVisibility();
    t.start();
    Thread.sleep(1000);
    t.a = 10;
    t.b = 20;
    t.lock.lock();
    try {
        t.c = 30;
    } finally {
        t.lock.unlock();
    }
    System.out.println("Program ended.");
}
```

Visibility vs. atomicity

- *Important: the visibility effect of the lock applies differently that the atomicity effect!*



SUPSI

Dipartimento tecnologie innovative

Summary of topics

- Atomicity and compound actions
- Details of race conditions
- Check-then-act and read-modify-write
- Atomic variables
- Volatile vs. atomic variables
- Visibility effect of volatile/atomic variables and mutexes

Asynchronous deepening

- ReadWriteLocks: the ReentrantReadWriteLock class.
- Details on race conditions: understanding, detecting and preventing concurrency issues.
- Atomic variables: the java.util.concurrent.atomic package.