Multi-threading and Thread Synchronization

Michele Fiori

EWLab – Università degli studi di Milano

Professor: Claudio Bettini

These slides are based on previous versions created by Letizia Bertolaja, Sergio Mascetti, Dario Freni, Claudio Bettini, Gabriele Civitarese, Riccardo Presotto and Luca Arrotta

Copyright

Every
Ware
Lab

Lesson Outline

- Client-Server Paradigm
- Iterative Servers
- Multi-threaded Servers
- Thread Synchronization - Synchronized

Every
Ware
Lab

Client-Server Paradigm

- **Clients** request specific services or resources, and **servers** provide them by processing the client's request.

- **Socket**: endpoint of a bidirectional communication between two processes across the network.
  Defined by an **IP address** and a **port number** to identify the source and destination of the communication.

Every Ware Lab

Client-Server Paradigm

Client

- creates a socket specifying the server address and the service port number
- Communication through the *established socket*

Server

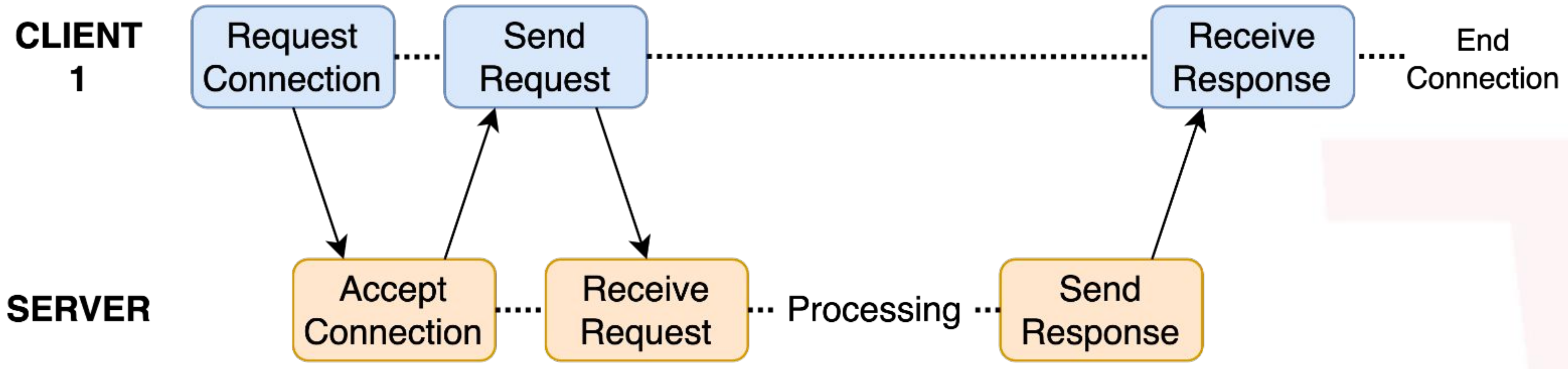
- creates a *listening socket* specifying the service port number
- Once an input connection is received, it creates an *established socket*
- Communication through the *established sockets* of the server and the client

Every Ware Lab

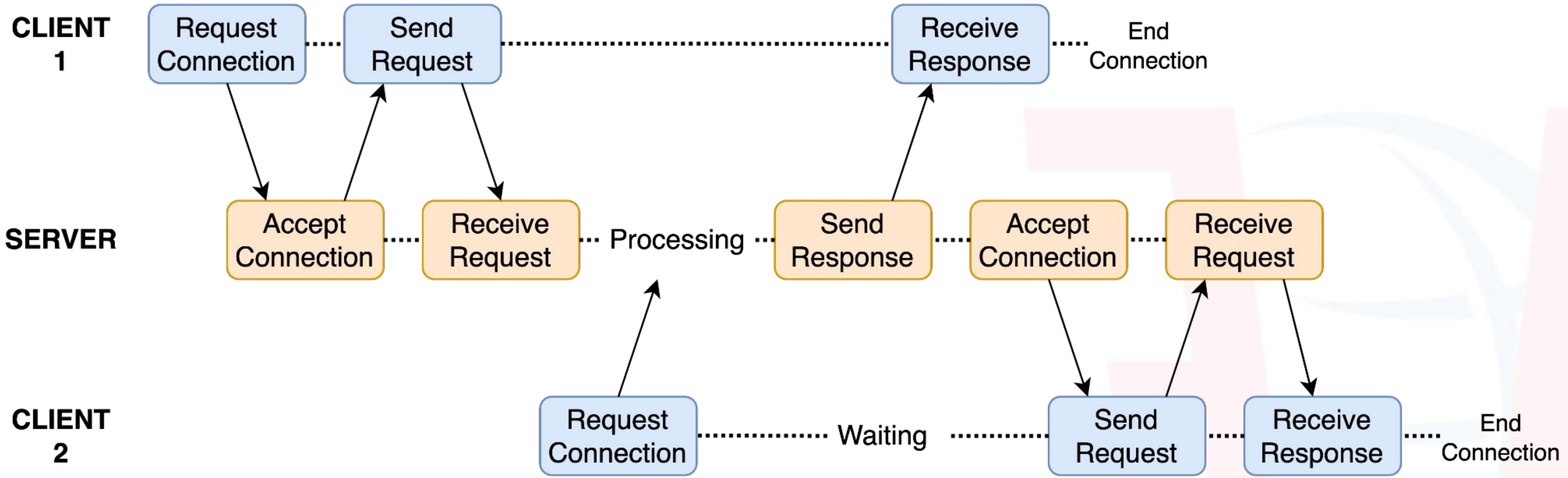
Iterative Servers

- **Iterative Servers** can handle one request at a time

Iterative Servers



Iterative Servers



Every Ware Lab

An Iterative Server in Java

UpperCaseServer

- The client reads a line from the user (*stream inFromUser*) and sends it to the server via socket (*stream outToServer*)
- The server reads the line from the socket
- The server makes the line upper case and sends it back to the client
- The client reads the converted line from the server (*stream inFromServer*) and prints it

Every
Ware
Lab

An Iterative Server in Java

Demo

An Iterative Server in Java – Issue

```java
clientSentence = inFromClient.readLine();

/* simulate a processing time of 10 seconds*/
Thread.sleep( millis: 10000);

/* Build the response */
capitalizedSentence = clientSentence.toUpperCase() + '\n';

/* Send the response to the client */
outToClient.writeBytes(capitalizedSentence);
```

What’s the problem here?

How to solve it?

Every
Ware
Lab

Threads

- Threads are independent execution units (or sequences) within the same process
- Threads within the same process share the same memory space
- Threads are scheduled as processes
- Useful for concurrent applications:
  - Asynchronous events
  - Overlapping between I/O and computation
  - In general, improved performances and use of resources

Every
Ware
Lab

Threads in Java – First Solution

Thread Inheriting

- Thread creation:
  - We extend the *Thread* class defining a constructor that takes as arguments the references to the data structure the thread will access to
  - We redefine the *run()* method so that the thread can executes its task

- Thread invocation:
  - We create an instance of the thread
  - We call the *start()* method

Every Ware Lab

Threads in Java – Second Solution

Implementing Runnable

- Thread creation:
  - We implement the *Runnable* interface by defining a constructor that takes as arguments the references to the data structure the thread will access to
  - We implement the *run()* method so that the thread can executes its task
- Thread invocation:
  - We create an instance of a *Thread* object, passing to its constructor an instance of the class that implements *Runnable*
  - We call the *start()* method

Every
Ware
Lab

Threads in Java – Solutions Comparison

Thread Inheriting

Thread creation

```
public class MyThread extends Thread{
...
public void run(){
...
```

Thread invocation

```
MyThread thread = new MyThread();
thread.start();
```

Implementing Runnable

Thread creation

```
public class RunnableThread implements Runnable{
...
public void run(){
...
```

Thread invocation

```
Thread thread = new Thread(new RunnableThred());
thread.start();
```

Every
Ware
Lab

Threads in Java

- Once the execution of a thread started, we have two executions in two different points of the code
  1. run() within the thread
  2. the point after the *start()* call
- The *father* thread can have a reference to the object related to the *son* thread
- We can use this reference to handle the threads in different ways
- After that the *run()* method ends, also the thread ends its execution

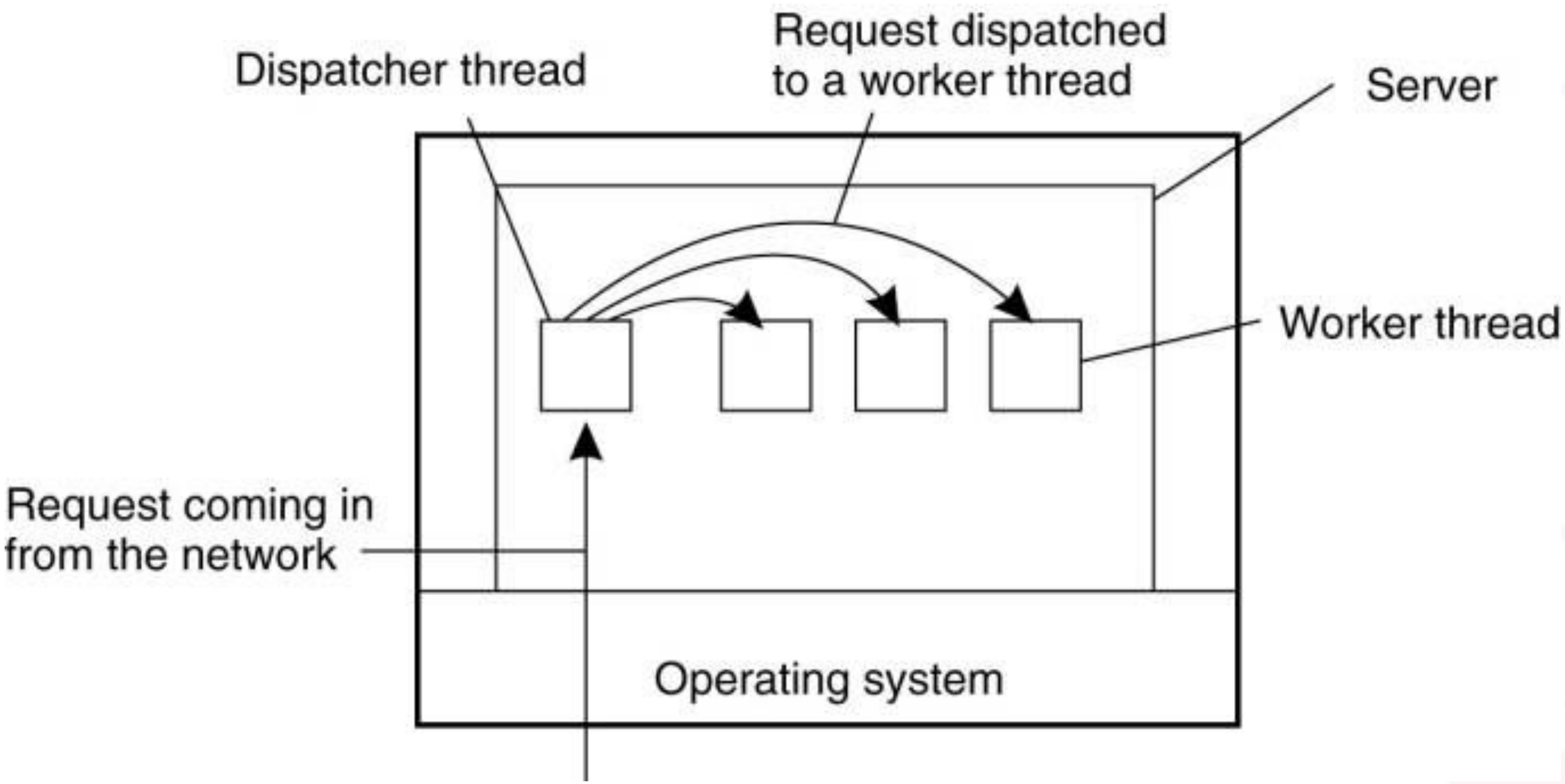
Every
Ware
Lab

Concurrent Servers

- Different clients can *concurrently* send different requests to the port the server is listening on
- In iterative servers, the requests are queued and the server will *sequentially* accept and handle each request
- Solution: the server can simultaneously handle multiple connections through threads. **It runs a different thread for each connection with a client**

Multi-threaded Server

Dispatcher-Worker Model



Multi-threaded Server in Java - Dispatcher

Demo

Exercise – A Service for Sums

- Client:
  - Reads address and port number of the server service from command line
  - Reads two numbers from standard input and sends them to the server
  - Receives and prints the response from the server
- Server:
  - Reads the port number of the service from command line
  - Prints address and port number of the connecting clients
  - Receives two integers from each client, computes the sum and sends back the response with the result
- You have to handle the possible exceptions
- Develop both an iterative and a multi-threaded version of the server

Every Ware Lab

Concurrent Programming

- A *concurrent program* is a set of instructions that could be executed simultaneously
- It is different from a *distributed system* in which several processes work in parallel, by communicating through a specific protocol
  - A concurrent program could run on a single processor (pseudo-parallelism)
- We will develop Java programs with different threads that work simultaneously
- The thread execution is **non-deterministic** and depends on the threads scheduling

Every
Ware
Lab

Concurrent Programming - Issues

- The threads share the memory space of the process they belong to
- The data exchange is very efficient but subject to some issues
  - **Thread Interference**
  - **Memory Inconsistency**
- The Thread Synchronization is necessary to overcome these issues!

Thread Interference

```java
class Counter {
    private int c = 0;

    public void increment() {
        int newValue = c + 1;
        c = newValue;
    }

    public void decrement() {
        int newValue = c - 1;
        c = newValue;
    }

```

- Threads A and B execute simultaneously operations on the *Counter* object
- A invokes *increment()* and B invokes *decrement()*
- Remember that the execution order is non-deterministic!
- What could happen?

Every Ware Lab

Thread Interference

```java
class Counter {
    private int c = 0;

    public void increment() {
        int newValue = c + 1;
        c = newValue;
    }

    public void decrement() {
        int newValue = c - 1;
        c = newValue;
    }

```

- *c* is equal to 0
- Possible execution order
  - A: *newValue = c+1 = 1;*
  - B: *newValue = c-1 = -1;*
  - A: *c = newValue = 1;*
  - B: *c = newValue = -1;*
- The result of the execution of A is being lost!

Every Ware Lab

Memory Inconsistency

```java
class Store {
    private int c = 10;

    // return true if can sell
    public boolean sell() {
        if(c>0) {
            c--;
            return true;
        }
        return false;
    }
}
```

- After some invocations, *c* is equal to 1
- Execution order:
  - A: *if (c > 0)*
  - B: *if (c > 0)*
  - A: *c--; return true;*
  - B: *c--; return true;*
- We’ve sold twice the last object!

Every Ware Lab

Synchronized Access

- Every object instance has associated an *instrinsic lock*, that is also called *monitor*
- If a method is declared *synchronized*, before the execution of the method, it must acquire the intrinsic lock
- The *synchronized* methods grant that a single thread at a time can access to the object
- Example of declaration:

  public synchronized int methodName(int param) {…}

Call of a Synchronized Method

- When a thread executes a synchronized method on an object *obj*, this is what atomically happens:
  - It is checked the value of the intrinsic lock associated with *obj*
  - If it is ''available'':
    - The value of the intrinsic lock is changed to ''not available''
    - The method is executed
    - Once the method ends, the value of the intrinsic lock is changed to ''available''
  - If ''not available'':
    - Wait until the intrinsic lock is ''available''

Every Ware Lab

Synchronization Example

- Let's consider a class *MyCollection* that stores a collection of objects, with two methods:
  - *sortItems()* that orders the collection
  - *getSmallest()* that returns the smallest element
- A thread wants to call *sortItems()*
- Another thread wants to call *getSmallest()*
- Why synchronization is required?
- How to synchronize?

Every
Ware
Lab

Synchronization Example

```java
class MyCollection {
    synchronized void sortItems() {
        // Sorting implementation
    }

    synchronized Object getSmallest() {
        // Complicated code to get the minimum
    }
}
```

- Note that the attributes can't be declared *synchronized*
- The access to the data structures that must be synchronized can occur only through methods

Every Ware Lab

Counter Class Correction

```java
class Counter {
    private int c = 0;

    public synchronized void increment(){
        int newValue = c + 1;
        c = newValue;
    }

    public synchronized void decrement(){
        int newValue = c - 1;
        c = newValue;
    }

```

- A invokes *increment()* and B invokes *decrement()*

- The only possible execution order:
    - A: *newValue = c+1 = 1;*
    - A: *c = newValue = 1;*
    - B: *newValue = c-1 = 0;*
    - B: *c = newValue = 0;*

Every
Ware
Lab

Synchronized Statement

- Another way to write synchronized code concerns the use of *synchronized* statements

```
synchronized(objVar) {
    // do stuff
}
```

- The *objVar* to specify is the object that contains the intrinsic lock we want to use
- *objVar* typically is also the object on which we want to grant atomicity
- To synchronize primitive types (*int*, *float*, …) we can create "dummy" objects which are used only for their lock

Every
Ware
Lab

Synchronized Statement

Why these two codes have the same behavior?

```
public synchronized void methodA(){
    // do stuff
}
```

```
public void methodB() {
    synchronized(this) {
        // do stuff
    }
}
```

Every Ware Lab

Synchronized Statement

- Why should we use a synchronized statement?
  - To avoid excessive synchronization
  - To achieve a finer synchronization

- In the following example, we consider two private fields *c1* and *c2* of the same class. c*1* and *c2* are **not** correlated

Every Ware Lab

Synchronized Statement

```java
public class FineGrainedSynchronization {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Every
Ware
Lab

Synchronized and Static Methods

- Also a static method can be defined *synchronized*

```
class Foo{
    synchronized static void foo(){
        // do stuff
    }
}
```

- The acquired lock is not related to the object instance, but to the class. This is equivalent to:

```
class Foo{
    static void foo(){
    synchronized(Foo.class){
        // do stuff
    }
}
```

Deadlock

- It is another problem we can have while using intrinsic locks
- A **deadlock** is a situation in which two or more threads are unable to proceed because each one is waiting for the other to release a resource.

Deadlock

```java
public class Model {
    private View myView;

    public synchronized void updateModel(Object someArg) {
        doSomething(someArg);
        myView.somethingChanged();
    }

    public synchronized Object getSomething() {
        return someMethod();
    }
}


public class View {
    private Model underlyingModel;

    public synchronized void somethingChanged() {
        doSomething();
    }

    public synchronized void updateView() {
        Object o = myModel.getSomething();
    }
}
```

The problem occurs if thread A enters updateModel and, before it can alert the view, thread B starts and enters updateView. UpdateView cannot call getSomething from myModel because thread A has the lock. Thread A at this point tries to go on but cannot call the somethingChanged() method of myView because thread B has the lock on View.

Deadlock

```java
public class Model {
    private View myView;

    public synchronized void updateModel(Object someArg) {
        doSomething(someArg);
        myView.somethingChanged();
    }

    public synchronized Object getSomething() {
        return someMethod();
    }
}


public class View {
    private Model underlyingModel;

    public synchronized void somethingChanged() {
        doSomething();
    }

    public synchronized void updateView() {
        Object o = myModel.getSomething();
    }
}
```

How to solve?

The order in which locks are accessed must always be the same.

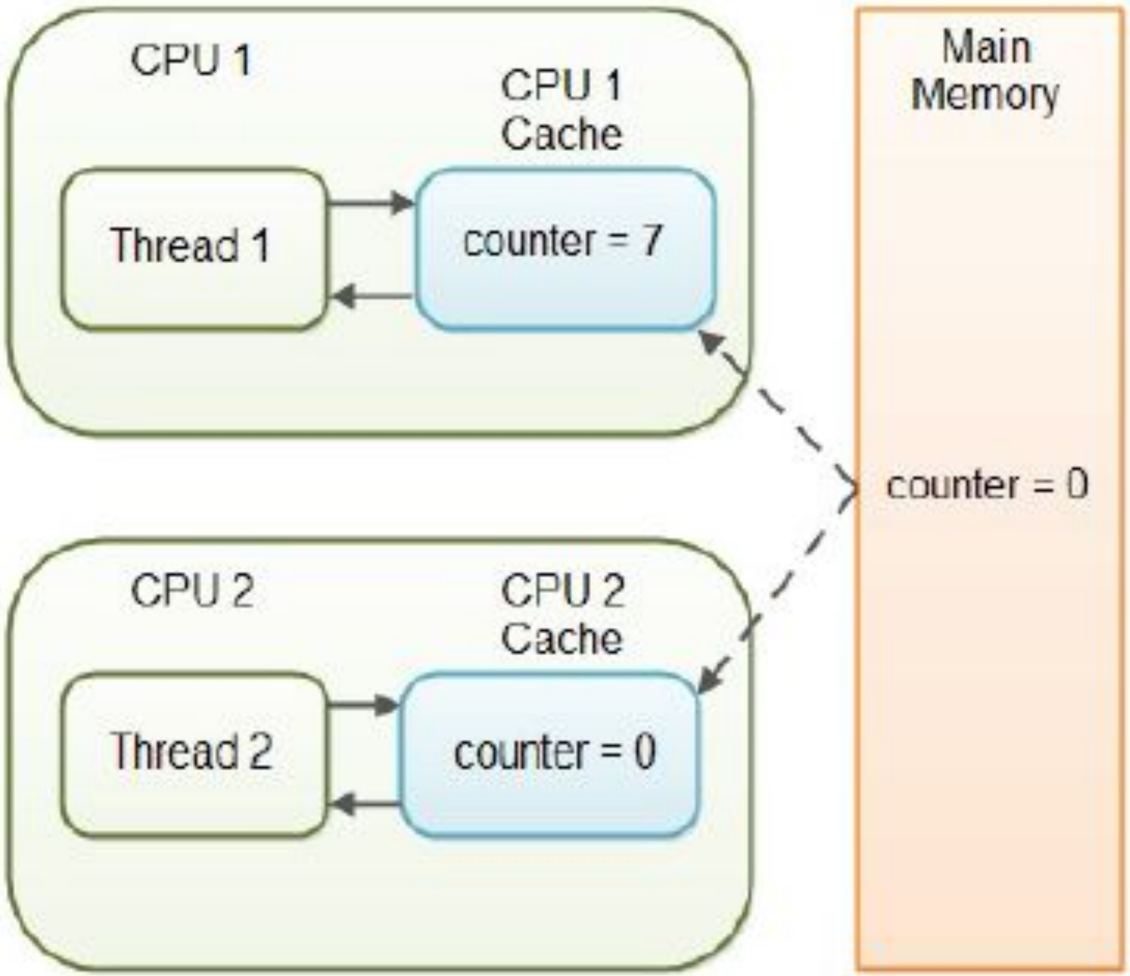
That is, one thread cannot ask for the lock on the model and then the lock on the view while the other thread can ask for lock on the view and then the lock of the model.

Some Observations

- The intrinsic lock associated with an object *obj* is used by all the *synchronized* methods and by the *synchronized* statements that specify *obj* as parameter
- It is **not** granted that the execution order of the waiting threads on a lock is equal to the order in which the threads have requested the lock
- The use of *synchronized* grants two properties:
  - **Mutual Exclusion**: only one thread at a time can obtain a specific lock
  - **Visibility**: the changes applied to the shared data before the lock is released must be visible to the threads that will acquire the lock later
- **Remember** to synchronize every time you have more threads that read and write the same data!

Every
Ware
Lab

Visibility



- For optimization purposes, a thread can copy variables from the main memory to a CPU cache
- In a multi-core setting, every thread can copy the variables in a different cache
- Without synchronization, we don't have guarantees about the update of a variable value

Every Ware Lab

Volatile

- *volatile* is a keyword we can assign to *variables*

```
volatile int x;
```

- It is a "light" version of *synchronized*
- It grants the **visibility**, but not the **mutual exclusion**
- The threads will automatically see the updated value of the *volatile* variables
  - The atomicity is granted only for direct reading and writing
- It must be used carefully: in a *volatile* variable we can write only values which are independent from any other state of the program (the variable itself included)
  - *x++* on a *volatile* variable is not thread-safe! Why?

Every Ware Lab

Volatile

```
class Worker {
    volatile boolean shutdownRequested;

    public void shutdown() { shutdownRequested = true; }
    public void doWork() {
        while (!shutdownRequested) {
            // do stuff
        }
    }
}
```

- A thread executes in loop *doWork()*, another thread ends the work with *shutdown()*
- *volatile* ensures that the two threads have the same view on the data, and it is simpler than the use of *synchronized*

Exercise – The Theatre

- Service to book theatre tickets
- Assumptions: a single show, a single type of ticket
- It must be developed as a concurrent server
- Issue: you mustn’t sell more than the available tickets
- You have to create the classes necessary for the multi-thread communication, and a class “Reservations” with a method without parameters that check if there are free seats:
  - If there are, it returns the number of the reserved seat
  - If there are not, it returns zero
- Check if the synchronization problems are solved by using Thread.sleep()

Every
Ware
Lab

References

- Code Examples:
  https://ewserver.di.unimi.it/gitlab/michelefiori/lab1-examples.git

- Exercises Setup:
  https://ewserver.di.unimi.it/gitlab/michelefiori/setup_test_sdp.git

Every
Ware
Lab

Contact

- You can contact me via email for any clarification or meeting:

michele.fiori@unimi.it

Every
Ware
Lab