Multi-threading and Thread Synchronization

Luca Arrotta and Riccardo Presotto

EWLab – Università degli studi di Milano

Professor: Claudio Bettini

These slides are based on previous versions created by Letizia Berolaja, Sergio Mascetti, Dario Freni, Claudio Bettini and Gabriele Civitarese

Copyright

Every Ware Lab

Lesson Outline

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

Client-Server Paradigm

- **Client-Server paradigm**: a Client asks a specific service to a Server
- **Socket**: endpoint of a bidirectional communication between two processes across the network

Every Ware Lab

Client-Server Paradigm

Client

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

Server

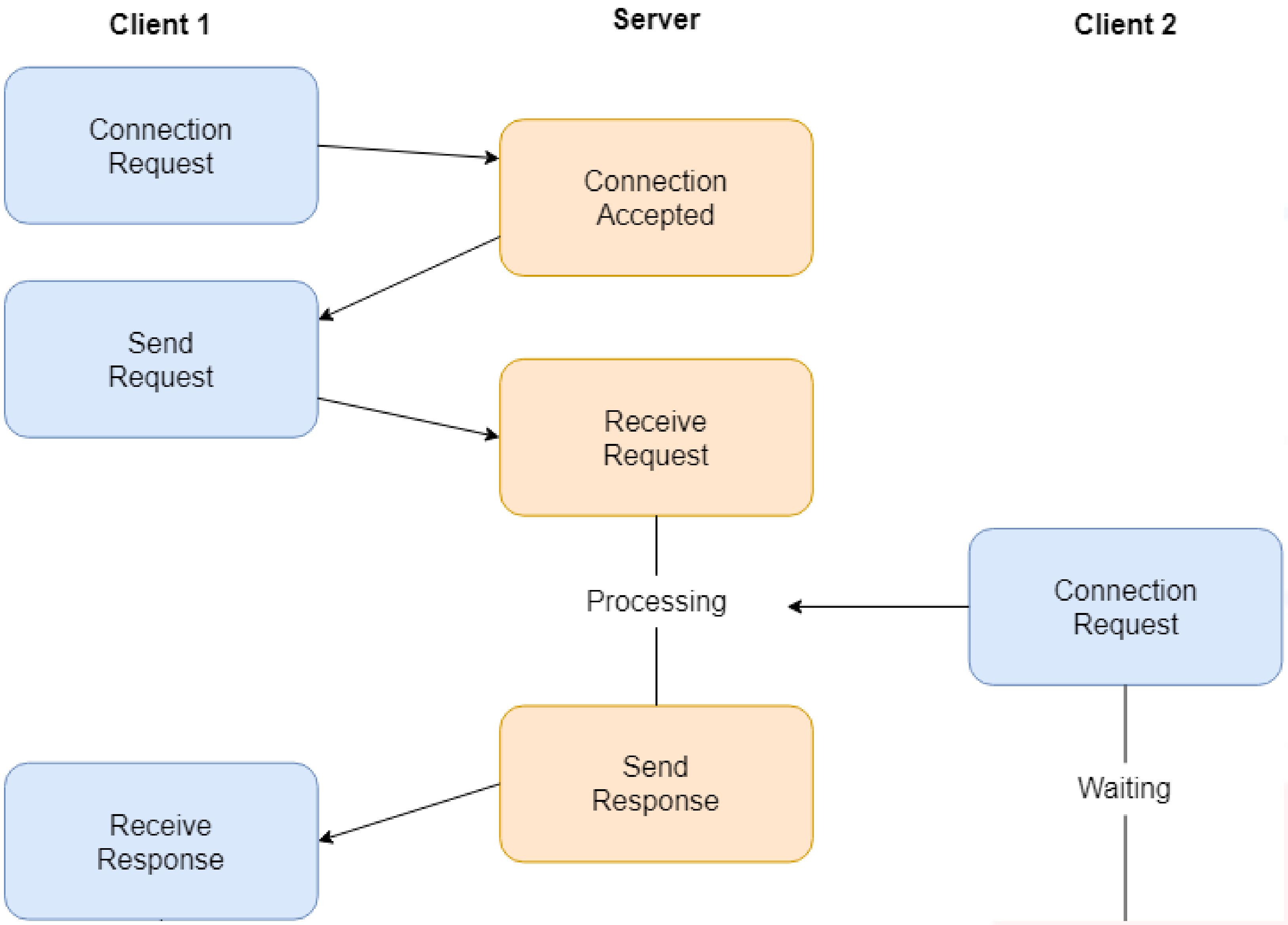
- creates a *listening socket* specifing the service port number
- Once an input connection is received, it creates an *established socket*
- Communication through the *established sockets* of the server and the client

Every Ware Lab

Iterative Servers

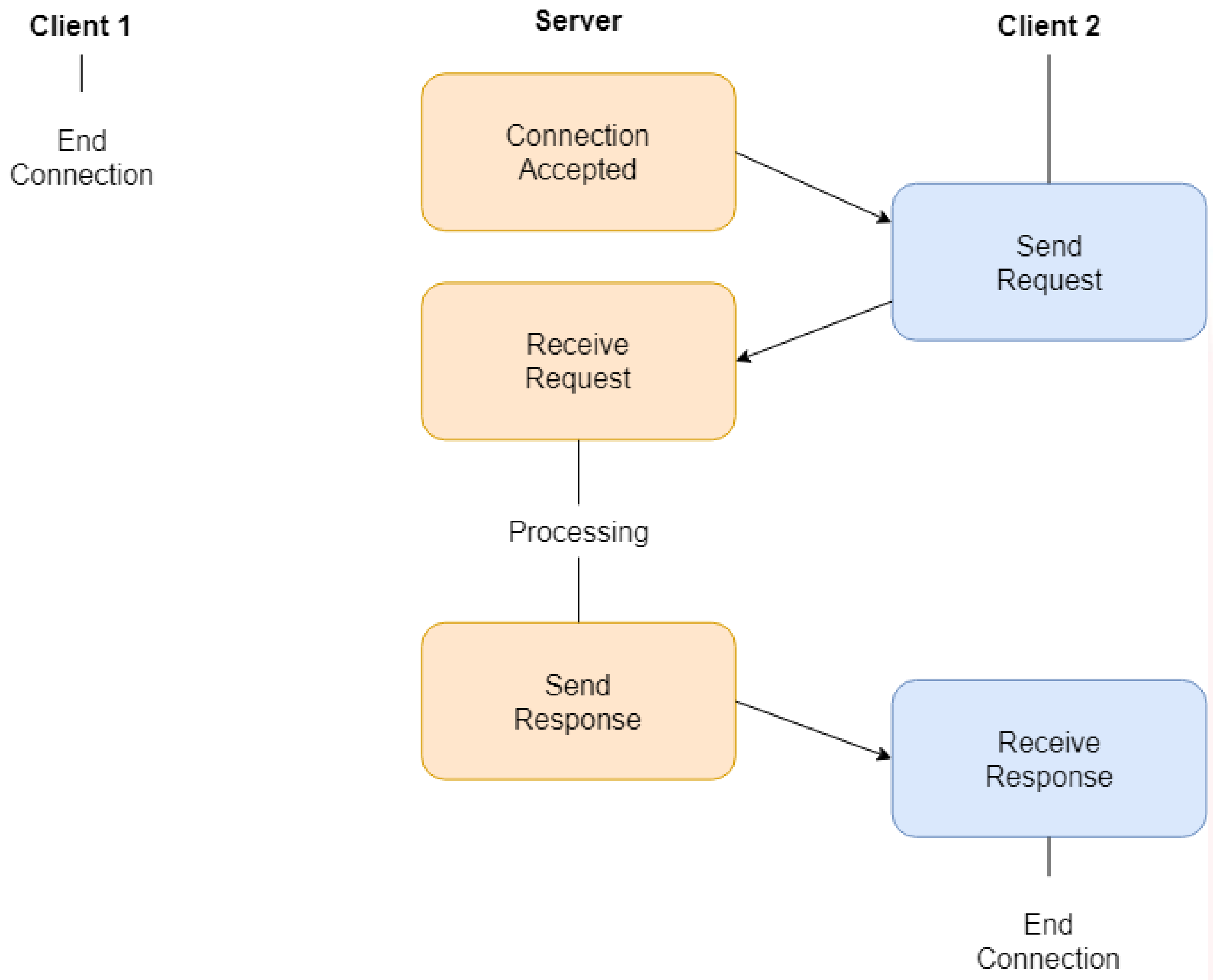
- **Iterative Servers** can handle one request at time

Iterative Servers



Every Ware Lab

Iterative Servers



Every
Ware
Lab

An Iterative Server in Java

UpperCaseServer

- The client read 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 read the converted line from the server (*stream inFromServer*) e prints it

Every Ware Lab

An Iterative Server in Java – Client

```java
import java.io.*;
import java.net.*;

class TCPClient {
    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;

        // input stream initialization (from user keyboard)
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        /* client socket initialization
            localhost: server address
            6789: server service port number */
        Socket clientSocket = new Socket("localhost", 6789);

        // output stream towards socket initialization
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
```

An Iterative Server in Java – Client

```java
            // input stream from socket initialization
            BufferedReader inFromServer =
                new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));

            // read a line from the user
            sentence = inFromUser.readLine();

            // send the line to the server
            outToServer.writeBytes(sentence + '\n');

            // read the response from the server
            modifiedSentence = inFromServer.readLine();
            System.out.println("FROM SERVER: " + modifiedSentence);
            clientSocket.close();
        }
}
```

Every Ware Lab

An Iterative Server in Java – Server

```java
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;

        // create a "listening socket" on the specified port
        ServerSocket welcomeSocket = new ServerSocket(6789);

        while(true) {
            /*  accept is a blocking call
                once a new connection arrived, it creates
                a new "established socket"  */
            Socket connectionSocket = welcomeSocket.accept();
```

Every Ware Lab

An Iterative Server in Java – Server

```java
            // input stream from the socket initialization
            BufferedReader inFromClient =
                new BufferedReader(
                    new InputStreamReader(connectionSocket.getInputStream()));

            // output stream to the socket initialization
            DataOutputStream outToClient =
                new DataOutputStream(connectionSocket.getOutputStream());

            // read a line (that terminates with \n) from the client
            clientSentence = inFromClient.readLine();
            capitalizedSentence = clientSentence.toUpperCase() + '\n';

            // send the response to the client
            outToClient.writeBytes(capitalizedSentence);
        }
    }
}
```

An Iterative Server in Java – Issue

```java
// read a line (that terminates with \n) from the client
clientSentence = inFromClient.readLine();

// wait for 10 seconds
Thread.sleep(10000);

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 different execution sequences within the same process
- Threads share the same memory space
- Threads are scheduled as processes
- Useful for concurrent applications:
  - asynchronous events
  - Overlapping between I/O and computation
  - …

Every
Ware
Lab

Threads in Java – First Solution

Thread Inheriting

- Thread creation:
  - We extend the *Thread* class by 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

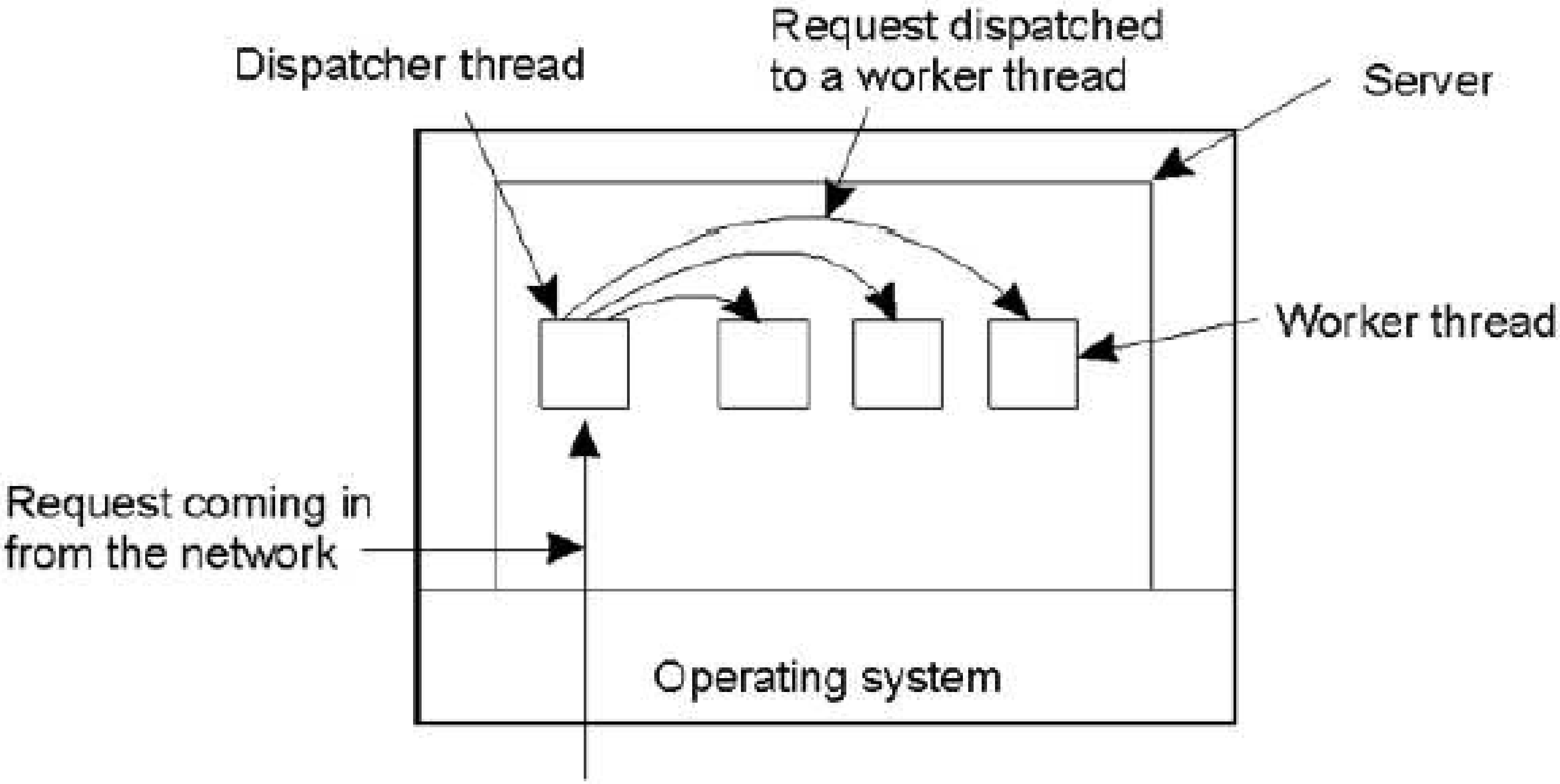
Concurrent Servers

- 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

Every
Ware
Lab

Multi-threaded Server

Dispatcher-Worker Model



Multi-threaded Server in Java - Dispatcher

```java
import java.io.*;
import java.net.*;

class TCPMultiServer {
    public static void main(String argv[]) throws Exception {
        ServerSocket welcomeSocket = new ServerSocket(6789);

        while(true) {
            Socket connectionSocket = welcomeSocket.accept();

            // thread creation passing the established socket as arg
            TCPServerThread theThread =
                new TCPServerThread(connectionSocket);

            // start of the thread
            theThread.start();
        }
    }
}
```

Ev
Wa
Lab

Multi-threaded Server in Java - Worker

```java
import java.io.*;
import java.net.*;

public class TCPServerThread extends Thread {
    private Socket connectionSocket = null;
    private BufferedReader inFromClient;
    private DataOutputStream outToClient;

    // the constructor argument is an established socket
    public TCPServerThread(Socket s) {
        connectionSocket = s;
        try {
            inFromClient =
                new BufferedReader(
                    new InputStreamReader(connectionSocket.getInputStream()));
            outToClient =
                new DataOutputStream(connectionSocket.getOutputStream());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Multi-threaded Server in Java - Worker

```java
    public void run() {
        String clientSentence;
        String capitalizedSentence;
        try {
            clientSentence = inFromClient.readLine();
            capitalizedSentence = clientSentence.toUpperCase() + '\n';
            outToClient.writeBytes(capitalizedSentence);
            connectionSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

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

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

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 simultaneosly 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, the intrinsic lock must be acquired
- 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’’

Synchronization Example

- Let’s consider a class *MyCollection* that stores a collection of objects
- A thread wants to order this collection
- Another thread wants to obtain the smallest element of the collection
- Why synchronization is required?
- How to synchronize?

Every
Ware
Lab

Synchronization Example

```
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;*

Synchronized Statement

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

```java
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

Synchronized Statement

Why these two codes have the same behavior?

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

```java
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
    }
}
```

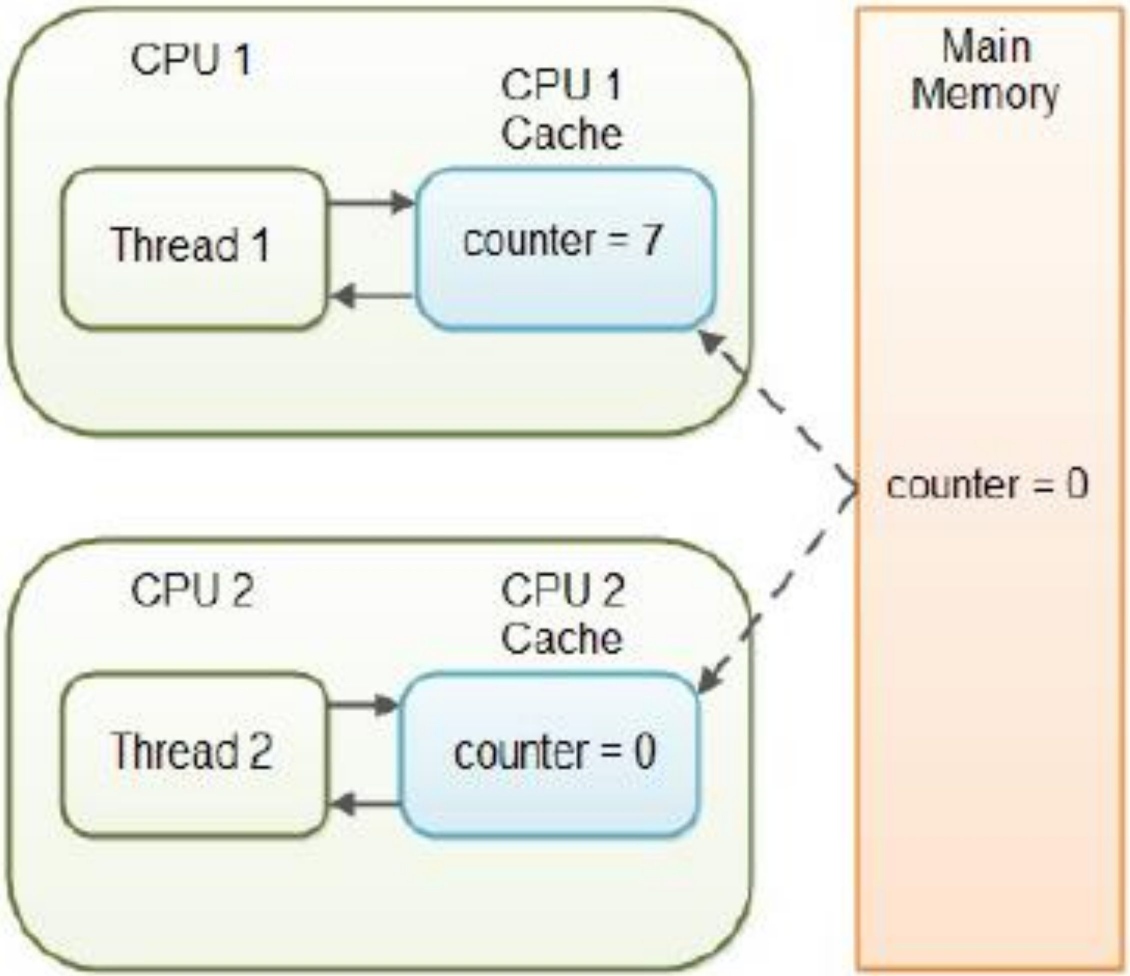
Deadlock

```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();
    }
}
```

Some Observations

- The intrinsic lock associated with an object *obj* is used by all the *synchronized* methods <u>and</u> 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!

Visibility



- For optmization purposes, a thread can copy variables from the main memory to a CPU cache
- In a multi-cores 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 progam (the variable itself included)
  - *x++* on a *volatile* variable is not thread-safe! Why?

Every Ware Lab

Volatile

```java
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*

Every Ware Lab

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/luca.arrotta/lab1-examples
- Exercises Setup: https://ewserver.di.unimi.it/gitlab/riccardopresotto/setup_test_sdp

Contact

- Contact the tutors via email for any clarification or meeting:

tutorsdp@di.unimi.it

Every
Ware
Lab