

Practical Concurrent And Parallel Programming

1. Intro to concurrency and the mutual exclusion problem: Define and motivate concurrency and mutual exclusion. Explain race conditions, critical sections and the happens-before relation. Show some examples of code from your solutions to the exercises in week 1.

Concurrency is the ability of a system to execute multiple independent tasks or processes simultaneously. In a concurrent system, multiple tasks may overlap in time, allowing for better resource utilization and improved system responsiveness. In the course we saw thread concurrency. A **thread** is a “lightweight process”, allow multiple streams of program control flow to coexist within a process, each thread has its own program counter, stack and local variables and shares with other thread from the same process resources like memory (the heap and process variables) and file handlers.

There are many reasons why we should use concurrency:

1. Resource utilization: Concurrency allows for better utilization of system resources, for example while one program waits for an input its more efficient to use the wait time to let another program run
2. Fairness It is preferable to let them share the computer via finer-grained time slicing than to let one program run to completion and then start another.
3. Convenience. It is often easier or more desirable to write several programs that each perform a single task and have them coordinate with each other as necessary than to write a single program that performs all the tasks

or from the concurrency note

1. Inherent: if my program interacts with the environment i may need concurrency (es UI)
2. Exploitation: hardware is nowadays capable of simultaneously executing multiple streams at the same time (multicore systems)
3. Hidden: use concurrency to enable multiple programs to share some resources in a manner where each one can act as if they had sole ownership over the resource.

however concurrency carries also some risks:

1. safety hazards: if i do not synchronize (protect the shared data) properly the outcome of my program can depend on the interleaving (**race condition** occurs when the behavior of a system depends on the relative timing of events, such as the order of execution of concurrent threads. Race conditions can lead to unpredictable and undesirable outcomes, such as data corruption or program crash.

occurs when the result of the computation depends on the interleaving of the operations

interleaving is a possible sequence of operations for a concurrent program

2. liveness hazard: risk of deadlocks
3. performance hazards: using thread carries some overhead (thread creation, context switches, synchronization,...)

So we want to avoid race conditions (we want our program to be stream safe), there are 2 definition of stream safe program:

1. a program is **stream safe** iff no concurrent execution of its statements result in a race condition.
2. a program is **stream safe wrt a specification** iff all concurrent execution of its statements satisfy the specification. (race condition can satisfy a specification).

to do so we introduced the concept of

Mutual exclusion is a mechanism that ensures that only one process or thread can access a shared resource at a time. It prevents multiple processes from interfering with each other's data and ensures the consistency of shared resources.

Motivation:

- Data integrity: Mutual exclusion prevents data corruption and ensures that shared resources are accessed in a controlled manner, avoiding conflicts.
- Correctness: Mutual exclusion helps maintain the correctness of concurrent programs by preventing race conditions and other synchronization issues.

A **critical section** is a part of a program that accesses shared resources and must be executed in a mutually exclusive manner. It is essential to properly synchronize critical sections to avoid race conditions and ensure data integrity and overall maintain program correctness.

A critical section is a part of the program that only one thread can execute at the same time.

```
public class TestLongCounterExperiments {

    LongCounter lc = new LongCounter();
    int counts = 1000000;

    ReentrantLock lock = new ReentrantLock();

    public TestLongCounterExperiments() {

        Thread t1 = new Thread(() -> {
            for (int i=0; i<counts; i++)
                lc.increment();
        });
        Thread t2 = new Thread(() -> {
            for (int i=0; i<counts; i++)
                lc.increment();
        });
        t1.start(); t2.start();
        try { t1.join(); t2.join(); }
        catch (InterruptedException exn) {
            System.out.println(x:"Some thread was interrupted");
        }
        System.out.println("Count is " + lc.get() + " and should be " + 2*counts);
    }
}
```

```

class LongCounter {
    private long count = 0;

    public void increment() {
        lock.lock();
        count += 1;
        lock.unlock();
    }

    public long get() {
        return count;
    }
}

```

```

public void increment() {
    lock.lock();
    try {
        count += 1;
    } finally {
        lock.unlock();
    }
}

```

to create critical sections we use Java locks. `lock()` acquires the lock if free otherwise it blocks. `unlock` releases the lock and signals one of the waiting threads if any. These are synchronization operations!!!! They established an execution order among the operations of different threads

The mutual exclusion property states that • Two or more threads cannot be executing their critical section at the same time.

An ideal solution to the mutual exclusion problem must ensure the following properties:

- Mutual exclusion: at most one thread executing the critical section at the same time
- Absence of deadlock: threads eventually exit the critical section allowing other threads to enter
- Absence of starvation: if a thread is ready to enter the critical section, it must eventually do so

The **happens-before relation** is a concept used in concurrent programming to define the order of events and ensure proper synchronization between threads. If event A happens before event B, it implies that the effects of event A are visible to event B.

Motivation is Establishing order: The happens-before relation helps define a consistent order of events in a concurrent system, allowing programmers to reason about the behavior of their programs and avoid unexpected outcomes.

1. Program order rule. Each action in a thread happens-before every action in that thread that comes later in the program order.
2. Monitor lock rule. An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock.

3. Volatile variable rule. A write to a volatile field happens-before every subsequent read of that same field.
4. Thread start rule. A call to Thread.start on a thread happens-before every action in the started thread.
5. Thread termination rule. Any action in a thread happens-before any other thread detects that thread has terminated, either by successfully return from Thread.join or by Thread.isAlive returning false.
6. Interruption rule. A thread calling interrupt on another thread happens-before the interrupted thread detects the interrupt (either by having InterruptedException thrown, or invoking isInterrupted or interrupted).
7. Finalizer rule. The end of a constructor for an object happens-before the start of the finalizer for that object.
8. Transitivity. If A happens-before B, and B happens-before C, then A happens-before C.

We say that an operation a happens-before an operation b , iff

- a and b belong to the same thread and a appears before b in the thread definition
- a is an unlock() and b is a lock() on the same lock

In the absence of happens-before relation between operations, the JVM is free to choose any execution order. In that case we say that operations are executed concurrently

Data races (occurs when a variable is read by more than one thread, and written by at least one thread, but the reads and writes are not ordered by happens-before. not all data race results in race condition)

2. Shared Memory I: Explain and motivate how locks, monitors and semaphores can be used to address the challenges caused by concurrent access to shared memory. Show some examples of code from your solutions to the exercises in week 2 and 3.

We know that because of possible interleaving between our threads, problems can be generated and become race conditions. To avoid this it is necessary to ensure a property of mutual exclusion.

An ideal solution to the mutual exclusion problem must ensure the following properties:

- **Mutual exclusion:** at most one thread executing the critical section at the same time
- **Absence of deadlock:** threads eventually exit the critical section allowing other threads to enter
- **Absence of starvation:** if a thread is ready to enter the critical section, it must eventually do

In Java, mutual exclusion can be achieved using the Lock interface in the `java.util.concurrent.locks` package

Locks

By establishing an happen before relation (unlock → lock) locks can:

- Ensure visibility
- Prevent reordering
- Ensure mutual exclusion

Java Lock is an interface, so we use an implementation of the Lock interface, namely `ReentrantLock`. In Java also, all objects have an intrinsic lock associated to it with a condition variable. Intrinsic locks are accessed via the **synchronized** keyword.

Synchronized can also be used on static methods **synchronized (C.class) { ... }**

Intrinsic locks in Java act as mutexes (or mutual exclusion locks), which means that at most one thread may own the lock.

`ReentrantLock` which acts like a regular lock, except that they allow locks to be locked more than once by the same thread. Reentrancy is implemented by associating with each lock an acquisition count and an owning thread. Without reentrant locks if a subclass overrides a synchronized method and then calls the superclass method, it would deadlock.

`ReentrantLock` has a fair flag that, when set to true, ensures that always the thread waiting longest in the entry queue is selected.

Locks can also be used to avoid visibility problems. In fact, by ensuring a happen before relationship the locks prevent different threads from reading old data from the cache.

Java provides a weak form of synchronization via the variable modifier **volatile**

Volatile variables are not stored in CPU registers or low levels of cache hidden from other CPUs and writes to volatile variables flush registers and low level cache to shared memory levels. Volatile cannot ensure mutual exclusion, only visibility and prevent reordering.

In the absence of data dependences or synchronization operations, the processor (CPU) or Just-In-Time compiler (JIT) in the Java Virtual Machine (JVM) can reorder Java bytecode operations.

Establishing a happen-before relation with a Lock prevents also reordering.

Monitors

If we also want to bind the execution of a critical section based on a condition here we use monitors. In fact, through the use of the condition they allow us to suspend the thread and release the lock before executing the operation of the critical section.

Starting from the Readers-Writers problem in which many threads can read from the structure as long as no thread is writing (this is the condition), we introduce the monitors. In Java, monitors are implemented as classes.

A monitor consists of:

- **Internal state** (data, for example numbers of readers)
- **Methods** (procedures, readLock, readUnlock, ...)
 - All methods in a monitor are mutually exclusive (ensured via locks)
 - Methods can only access internal state
- **Condition variables**: while a thread is waiting for something to happen, say, for another thread to place an item in a queue, it is a very good idea to release the lock on the queue, because otherwise the other thread will never be able to continue, release a lock temporarily is provided by a Condition object associated with a lock
 - Queues where the monitor can put threads to wait
 - await() – releases the lock, and blocks the thread (on the queue)
 - signal() – wakes up a thread blocked on the queue, if any
(When threads wake up the acquire the lock immediately)
 - signalAll() – wakes up all threads blocked on the queue, if any

Example for monitor:

```
int i = 0;
Lock l = new ReentrantLock();
Condition c = l.newCondition();
...

// method example
public void method(...) {
    l.lock();
    try{
        while(i>0) {
            condition.await();
        }
    }
    catch (InterruptedException e) {...}
    finally {l.unlock();}
}
```

```
public void readLock() {
    lock.lock();
    try {
        while(writer)
            condition.await();
        readers++;
    }
    catch (InterruptedException e) {...}
    finally {lock.unlock();}
}

public void readUnlock() {
    lock.lock();
    try {
        readers--;
        if(readers==0)
            condition.signalAll();
    }
    finally {lock.unlock();}
}
```

```
public void writeLock() {
    lock.lock();
    try {
        while(readers > 0 || writer)
            condition.await();
        writer=true;
    }
    catch (InterruptedException e) {...}
    finally {lock.unlock();}
}

public void writeUnlock() {
    lock.lock();
    try {
        writer=false;
        condition.signalAll();
    }
    finally {lock.unlock();}
}
```

Fairness in our course, refers to absence of starvation.

In our writers and readers example, writes may starve if readers keep coming (writers need to wait until there are 0 readers)

To fix this starvation problem writers can set the writer flag to true to indicate that they are waiting to enter

```
public void readLock() {
    lock.lock();
    try {
        while(writer)
            condition.await();
        readsAcquires++;
    }
    catch (InterruptedException e) {...}
    finally {lock.unlock();}
}
```

```
public void writeLock() {
    lock.lock();
    try {
        while(writer)
            condition.await();
        writer=true;
        while(readsAcquires != readsReleases)
            condition.await();
    }
    catch (InterruptedException e) {...}
    finally {lock.unlock();}
}
```

The condition variable in intrinsic locks is accessed via the methods **wait()**, **notify()**, **notifyAll()**. These are equivalent to **await()**, **signal()**, **signalAll()** in **ReentrantLock**.

```
Lock l = new Lock();
Condition c = l.addCondition()

l.lock()
try {
    // critical section code
    while(property)
        c.await();

    ...

    c.signalAll();
} finally {
    l.unlock()
}
```



```
Object o = new Object();

synchronized (o) {
    // critical section code
    while(property)
        o.wait();

    ...

    o.notifyAll();
}
```

Semaphores (less expressive than monitors)

Semaphores are synchronization primitives that allow at most *c* number of threads in the critical section where *c* is called the capacity. Consists of:

- An integer capacity – Initial number of threads allowed in the critical section
- A method **acquire()** – Checks if $c > 0$, if so, it decrements capacity by one ($c--$) and allows the calling thread to make progress, otherwise it blocks the thread – It is a blocking call
- A method **release()** – It checks whether there are waiting threads, if so, it wakes up one of them, otherwise it increases the capacity by one ($c++$) – It is non-blocking

If the capacity is 1 it behaves like a lock.

```
ReadWriteMonitor m = new ReadWriteMonitor();
Semaphore semReaders = new Semaphore(5,true);
Semaphore semWriters = new Semaphore(5,true);
for (int i = 0; i < 10; i++) {
    // start a reader
    new Thread() -> {
        m.readLock();
        semReaders.acquire();
        // read
        semReaders.release();
        m.readUnlock();
    }).start();

    // start a writer
    new Thread() -> {
        m.writeLock();
        semWriters.acquire();
        // write
        semWriters.release();
        m.writeUnlock();
    }).start();
}
```

3. Shared Memory II: Define and explain what makes a class thread-safe. Explain the issues that may make classes not thread-safe. Show some examples of code from your solutions to the exercises in week 3.

A **class** is said to be thread-safe if and only if **no concurrent execution of method calls or field accesses (read/write) result in data races** on the fields of the class.

A data race occurs when two concurrent threads:

- Access a shared memory location
- At least one access is a write
- There is no happens-before relation between the accesses

The above definition is for a **class**, instead a concurrent **program** is said to be thread-safe if and only if it is **race condition free**.

It is very important to note that for any program p , p only accesses thread-safe classes $\nRightarrow p$ is a thread-safe program!

If an object is not thread-safe, several techniques can still let it be used safely in a multithreaded program (like instance confinement or Java monitor pattern)

Thread-safe classes

The elements to identify/consider to ensure that there is a happens-before relation for any concurrent execution of field access and method calls where at least one of them results in a write access are:

- Class state

Identify the fields that may be shared by several threads (class variables)

Constraints placed on states create additional synchronization requirements, for example when multiple variables participate in an invariant, the lock that guards them must be held for the duration of any operation that accesses the related variables

Methods should only manipulate class state or parameters (avoid using object references as parameters)

- Escaping

Not expose shared state variables.

Defining all (shared) class state (primitive) variables as private ensures that these variables will only be accessed through public methods.

Remember that when a method returns an object, we get a reference to that object so return a copy of it.

- Safe publication

It is important to ensure that initialization happens-before publication

Before making accessible a reference to an object, all its fields must be correctly initialized.

We can use final and volatile declarations because they establish a happens-before relation between initialization and access to the object's reference (publication).

Volatile for primitive types, final for complex objects.

When we create a thread in the constructor we pass "this" reference. If the thread is started before the construction ends, it can read not yet initialize data.

- Immutability

An immutable object is one whose state cannot be changed after initialization

The final keyword in Java prevents modification of fields (but not what is inside, like array, so even if we have all fields immutable we can have problem)

Data races can still occur only during initialization of immutable variable if we don't use "final" variable.

- Mutual Exclusion

If class state must be mutable, ensure mutual exclusion (using Lock or Semaphore)

EXERCISE

Check if a provided class is thread safe, it was.

Implement a Person class with a unique id among all instance

```
public class Person {
    private static int lastId = 0;
    private final int id;
    private String name;
    private int zip;
    private String address;

    public Person(int initialId) {
        synchronized (Person.class) {
            if (lastId == 0) {
                this.id = initialId;
                lastId = initialId + 1;
            } else {
                this.id = lastId;
                lastId++;
            }
        }
    }

    public Person() {
        synchronized (Person.class) {
            this.id = lastId;
            lastId++;
        }
    }

    public synchronized void setAddressAndZip(String add, int zip) {
        this.address = add;
        this.zip = zip;
    }

    public synchronized void setName(String name) {
        this.name = name;
    }
}
```

Last one was to implement a Semaphore thread-safe class using Java Lock

```
public class OurSemaphore {
    private final int capacity;
    private int permits;
    private final Lock lock;
    private final Condition condition;

    public OurSemaphore(int capacity) {
        this.capacity = capacity;
        this.permits = 0;
        this.lock = new ReentrantLock();
        this.condition = lock.newCondition();
    }

    public void acquire() throws Exception {
        lock.lock();

        while (permits == capacity) {
            condition.await();
        }
        permits++;

        lock.unlock();
    }

    public void release() {
        lock.lock();
        try {
            if (permits > 0) {
                permits--;
                condition.signal();
            }
        } finally {
            lock.unlock();
        }
    }
}
```

4. Testing: Explain the challenges in ensuring the correctness of concurrent programs. Describe different testing strategies for concurrent programs, and their advantages and disadvantages. Show some examples of code from your solutions to the exercises in week 4.

A (concurrent) program is correct if and only if it satisfies its specifications which are often stated as a collection of program properties. There are two main types of properties:

- Safety: "Something bad never happens"
- Liveness: "Something good will eventually happen"

Testing concurrent programs is about writing tests to find undesired interleavings that violate a property. The major challenge in constructing tests for concurrent programs is that potential failures may be rare probabilistic occurrences rather than deterministic ones; tests that disclose such failures must be more extensive and run for longer than typical sequential tests. The test in which undesired interleavings appear is called a counterexample. Since for safety property the counterexample is a **finite** interleaving instead of an **infinite** for the liveness property we will explore only the first case.

Liveness tests include tests of progress (something will eventually happen) and non progress (tests that an algorithm does not deadlock)

We will use JUnit 5 which is a popular unit test framework for Java programs and makes it easy to implement and run functional correctness tests.

Testing safety

Tests of safety, which verify that a class's behavior conforms to its specification, usually take the form of testing invariants

Some strategies to take into account when developing a test:

1. Precisely define the property you want to test
2. If you are going to test multiple implementations, it is useful to define an interface for the class you are testing.

```
class CounterDR implements Counter {  
    private int count;  
  
    public CounterDR() {  
        count = 0;  
    }  
  
    public void inc() {  
        count++;  
    }  
  
    public int get() {  
        return count;  
    }  
}
```

```
class CounterSync implements Counter {  
    private int count;  
  
    public CounterSync() {  
        count = 0;  
    }  
  
    public synchronized void inc() {  
        count++;  
    }  
  
    public int get() {  
        return count;  
    }  
}
```

```
class CounterAto implements Counter {  
    private AtomicInteger count;  
  
    public CounterAto() {  
        count = new AtomicInteger(0);  
    }  
  
    public void inc() {  
        count.incrementAndGet();  
    }  
  
    public int get() {  
        return count.get();  
    }  
}
```

3. Concurrent tests require a setup for starting and running multiple threads
 - Maximize contention to avoid a sequential execution of the threads

```

public void testingCounterParallel(int nrThreads, int N) {
    ...

    // init barrier
    barrier = new CyclicBarrier(nrThreads + 1);

    for (int i = 0; i < nrThreads; i++) {
        new Thread() -> {
            barrier.await(); // wait until all threads are ready
            // thread execution
            barrier.await(); // wait until all threads are finished
        }.start();
    }

    try {
        barrier.await();
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException e) {
        e.printStackTrace();
    }
    ...
}

```

- You may need to define thread classes.

4. Run the tests multiple times and with different setups to try to maximize the number of interleavings tested

```

class TestCounter {
    ...
    @ParameterizedTest
    @MethodSource("argsGeneration")
    public void testingCounterParallel(int nrThreads,
                                     int N) {
        //body of the test
    }
    ...
}

```

```

// Loop to generate each parameter entry
// (2^j, i) for i \in {10_000, 20_000, ..., J}
// and j \in {1, ..., I}
for (int i = iInit; i <= I; i += iIncrement) {
    for (int j = jInit; j < J; j += jIncrement) {
        list.add(Arguments.of((int) Math.pow(2, j), i));
    }
}

// Return the list
return list;

```

We can use JUnit to test also other properties

Testing blocking operations

If a method is supposed to block under certain conditions, then a test for that behavior should succeed only if the thread does not proceed. Testing that a method blocks introduces an additional complication: once the method successfully blocks, you have to convince it somehow to unblock. The obvious way to do this is via interruption. Listing below shows an approach to testing blocking operations. It creates a “taker” thread that attempts to take an element from an empty buffer. If take succeeds, it registers failure. If the taker thread has correctly blocked in the take operation, it will throw `InterruptedException`, and the catch block for this exception treats this as success and lets the thread exit.

It is tempting to use `Thread.getState` to verify that the thread is actually blocked on a condition wait, but this approach is not reliable.

```

void testTakeBlocksWhenEmpty() {
    final BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) {}
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());
    } catch (Exception unexpected) {
        fail();
    }
}

```

Testing deadlock (liveness testing)

Testing for deadlocks is not really possible, we should define a maximum duration for an operation, after which, we deem the execution as deadlocked. If, when a running a test, we observe that the program does not terminate for a long time, it might be due to deadlock but we are not sure. Deadlock is a liveness property.

Testing resource management

Any object that holds or manages other objects should not continue to maintain references to those objects longer than necessary. Such storage leaks prevent garbage collectors from reclaiming memory and can lead to resource exhaustion and application failure.

Formal verification

Formal verification is a technology that tries to prove that a program satisfies a specification (properties) by mathematical techniques.

An automatic version of formal verification is model-checking.

Model-checking transforms programs into finite-state models that encapsulate all possible interleavings in the system like automata and properties are specified in some type of logic (First-Order Logic (FOL), propositional logic). The model of the program and the property are typically expressed in the same language, so it is possible to automatically check whether they are satisfied.

Static analysis

Static analysis tools are an effective complement to formal testing and code testing. Static code analysis is the process of analyzing code without executing it.

Static analysis tools contain bug-pattern detectors for many common coding errors, many of which can easily be missed by testing or code review.

Some concurrency related bug patterns are:

Inconsistent synchronization - Unreleased lock - Double-checked locking - Starting a thread from a constructor - Condition wait errors - Sleeping or waiting while holding a lock - Spin loops

EXERCISE

The main exercise was to test and fix a non thread-safe data structure.

```
/** Test threads */
public class AddNInteger extends Thread {
    // add number from 0 to N
    private final int N;

    public AddNInteger(int N) {
        this.N = N;
    }

    public void run() {
        try {
            barrier.await(); // waits until all threads all ready
            for (int i = 0; i < N; i++) {
                set.add(i);
            }
            barrier.await(); // waits until all threads are done
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

```
@BeforeEach
public void initialise() {
    //set = new ConcurrentIntegerSetBuggy();
    set = new ConcurrentIntegerSetSync();
    //set = new ConcurrentIntegerSetLibrary();
}

@ParameterizedTest
//@Disabled
@DisplayName("Test Add Integer Set Parallel")
@MethodSource("argsGeneration")
public void testingAddIntegerSetParallel(int nrThreads, int N) {
    System.out.printf("Parallel integer set tests with %d threads and %d iterations", nrThreads, N);

    // init barrier
    barrier = new CyclicBarrier(nrThreads + 1);

    // start threads
    for (int i = 0; i < nrThreads; i++) {
        new AddNInteger(N).start();
    }

    try {
        barrier.await(); // wait until threads are ready for execution (maximize contention)
        barrier.await(); // wait for threads to finish
    } catch (InterruptedException | BrokenBarrierException e) {
        e.printStackTrace();
    }

    // add assert true
    assertTrue(N == set.size(), "set.size() == " + set.size() + ", but we expected " + N);
}
```

5. Performance measurements: Motivate and explain how to measure the performance of Java code. Illustrate some of the pitfalls there are in doing such measurements. Show some examples of code from your solutions to the exercises in week 5.

The principal motivations to measure performance are:

- Analyze time consuming computation like string search or counting prime number
 - Analyze code and whether a new way to solve a problem is faster than the old one
- Measuring the execution time of a piece of software is an experimental activity, involving the software and a computer system, itself consisting of much (systems) software and some hardware. In practice, software performance measurements are influenced by so many factors that can be very hard pitfalls to avoid affecting the code execution measurements in unpredictable ways. Some of them are:
- **Garbage collection:** The timing of garbage collection is unpredictable, so there is always the possibility that the garbage collector will run during a measured test run.
 - **Dynamic compilation:** Allowing the compiler to run during a measured test run can bias test results in two ways: compilation consumes CPU resources, and measuring the run time of a combination of interpreted and compiled code is not a meaningful performance metric
 - **Unrealistic sampling of code paths:** JVM is permitted to use information specific to the execution in order to produce better code, which means that compiling method M in one program may generate different code than compiling M in another
 - **Unrealistic degrees of contention:** the test may introduce a level of contention that is too high for the actual use of the application or otherwise not sufficient
 - **Dead code elimination:** optimizing compilers are adept at spotting and eliminating dead code.

How to measure the performance

The simplest way to measure one operation is by using a Timer class and count the time elapsed to do the desired operation. This way is useless for the reasons listed before.

A much better approach is to start the timer, execute the function many times and print the elapsed time divided by 1 million. This can be not enough because the JIT compiler can realize that the result of is never used, and so the loop has no effect at all and therefore the loop is removed completely. To avoid this, we need to change the benchmark loop to pretend that the result is used, by adding it to a dummy variable whose value is returned by the method.

```
public static double Mark2() {
    Timer t = new Timer();
    int count = 100_000_000;
    double dummy = 0.0;
    for (int i=0; i<count; i++)
        dummy += multiply(i);
    double time = t.check() * 1e9 / count;
    System.out.printf("%6.1f ns%n", time);
    return dummy;
}
```

Instead of printing all the measurements we could compute and print the empirical mean and standard deviation of the measurements.

The number of iterations (count equals 100 million) used above was rather arbitrarily chosen. While it probably is sensible for the small multiply function, **it may be far too large when measuring a more time-consuming function**. To “automate” the choice of count, and to see how the mean and standard deviation are affected by the number of iterations, we add an outer loop. In this outer loop we double the count until the execution time is at least 0.25 seconds.

```
system.out.printf("%0.1f ns +/- %0.2f %10d%n", mean, sdev, c);
} while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
return count / 4 + 1000000;
```

Now that we are happy with the basic benchmarking code, we generalize it so that we can measure many different functions more easily. A general benchmarking function Mark6 would simply take an IntToDoubleFunction as argument and hence work for any such function

```
public interface IntToDoubleFunction { double applyAsDouble(int i); }

Mark6("multiply", i -> multiply(i));
```

Mark7 will print only final values by moving the computation and printing of mean and sdev out of the do-while loop.

EXERCISE

The first exercise was to compare our results with those of the teacher.

We did not notice huge variations but the ones we saw are probably given by different things that we discussed at the lecture, such as:

- Disturbances of external factors for example from other processes running on the computer
- Cost of creating and starting threads.
- Limitations in the number of cores, that's the reason why we don't see an increase of performance when reaching a large number of cores.

The third exercise confronted a standard variable with a volatile one.

Incrementing volatile int takes significantly more time, since each increment of the volatile variable is not saved in a cache but on the main memory (and accessing the memory is expensive timewise).

The last exercise was to compare the time taken by searching for occurrences of a word sequentially and in a parallel manner via threads.

```
> Task :app:run
Search string serial      26532740,0 ns  6096302,36      8
Search string parallel    6864548,3 ns   348686,02     64
BUILD SUCCESSFUL in 15s
```

The parallel searching is faster since the threads do the search independently in different parts of the file, however even if we used 4 threads the time is not divided by 4 because initializing and starting threads takes time, and also does synchronization over the shared variable.

6. Performance and Scalability: Explain how to increase the performance of Java code using threading. Illustrate some of the pitfalls there are in doing this. Show some examples of code from your solutions to the exercises in week 6.

After discovering how to measure the performance of a java program, we looked at how it improved with the use of parallel programming.

In using concurrency to achieve better performance, we are trying to do two things: utilize the processing resources we have more effectively and enable our program to exploit additional processing resources if they become available (keep the CPUs as busy as possible).

But before using threads we must also ensure that the problem is friendly to parallel decomposition and that our program effectively exploits this potential for parallelization.

Amdahl's law describes how much a program can theoretically be sped up by additional computing resources, based on the proportion of parallelizable and serial components. All concurrent applications have some sources of serialization (for example the execution of a critical section)

But using thread has some costs: **(continue here chapter 11)**

- **Context switching:** if there are more runnable threads than CPUs, eventually the OS will preempt one thread so that another can use the CPU. This causes a context switch, which requires saving the execution context of the currently running thread and restoring the execution context of the newly scheduled thread
- **Memory synchronization:** synchronized and volatile may entail using special instructions called memory barriers that can flush or invalidate caches, flush hardware write buffers, and stall execution pipelines.
- **Blocking:** suspending and retrieving the blocked thread require a lot of time

To try to reduce these costs there are three ways to reduce lock contention:

- Reduce the duration for which locks are held (moving code that doesn't require the lock out of synchronized blocks)
- Reduce the frequency with which locks are requested
- Replace exclusive locks with coordination mechanisms that permit greater concurrency (lock splitting and lock striping)

Lock splitting

If a lock guards more than one independent state variable, you may be able to improve scalability by splitting it into multiple locks that each guard different variables

Lock striping

A way to try to reduce the contention between threads (saturation loss) is by using different locks on the same structure ensuring that two threads will work on different portions (stripes) of our data structure. An example can be multiple threads trying to update different indexes of an array. We can put on lock for some index or even a lock for each index.

Alternatives to locks can be: concurrent collections, semaphore, immutable object and atomic variable (CAS low-level concurrency primitives)

To work around "slow" thread lifecycles, many developers turned to object pooling,

where objects are recycled instead of being garbage collected and allocated anew when needed

Task

Tasks are units of work, and threads are a mechanism by which tasks can run asynchronously. We encounter two fundamental types of task: **Runnable** and **Callable**. A **Runnable** is a simple interface in Java that represents a task that can be executed independently, without returning any result. On the other hand, a **Callable** is similar to a **Runnable**, but it can return a result and throw exceptions, making it more versatile for tasks that require computation and result retrieval.

Executors

Executor allows the asynchronous tasks execution.

In concurrent programming, a **pool** refers to a group of **tasks** that are managed collectively by an **ExecutorService** to be runned concurrently. **ExecutorService** is an interface derived from **Executor** which adds some methods for lifecycle management. This approach optimizes resource utilization allowing developers to focus on the logical aspects of their program rather than low-level thread management.

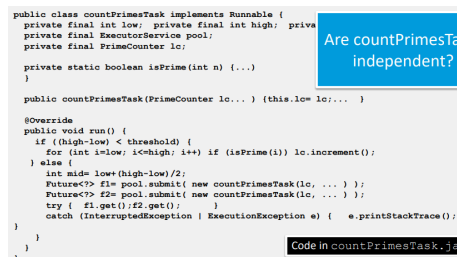
One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue.

Other types of pools are:

- *newSingleThreadExecutor*: creates a single worker thread to process tasks, replacing it if it dies unexpectedly.
- *newScheduledThreadPool*: A fixed-size thread pool that supports delayed and periodic task execution

To represent the result of an asynchronous computation, like a callable task, the concept of a **future** can be used. A **future** is a placeholder or a handle for the result of a computation that may not be available yet. It allows for non-blocking retrieval of the result once the computation is complete.

The real performance payoff of dividing a program's workload into tasks comes when there are a large number of independent, homogeneous tasks. If we divide heterogeneous tasks that require a lot of communication or unbalanced size the sequential version, but the code is a lot more complicated.



```
public class countPrimesTask implements Runnable {
    private final int low; private final int high; private
    private final ExecutorService pool;
    private final PrimeCounter lc;

    private static boolean isPrime(int n) {...}

    public countPrimesTask(PrimeCounter lc...) { this.lc= lc;... }

    @Override
    public void run() {
        if ((high-low) < threshold) {
            for (int i=low; i<=high; i++) if (isPrime(i)) lc.increment();
        } else {
            int mid= low+(high-low)/2;
            Future<?> f1= pool.submit( new countPrimesTask(lc, ... ) );
            Future<?> f2= pool.submit( new countPrimesTask(lc, ... ) );
            try { f1.get(); f2.get(); }
            catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }
        }
    }
}
```

Are countPrimesTask independent?

Code in countPrimesTask.java

In the example above future is used on a **Runnable** not to return a value but only to wait for the finish of the tasks. For returning the termination with the **Runnable** we must use the `.submit` procedure, the `.execute` one cannot return a result.

When we use parallel programming for increase the performance we must be sure not to fall into other pitfalls that would only worsen the total performance:

- **Starvation loss:** minimize the time that the task pool is empty
- **Separation loss:** find a good threshold to distribute workload evenly (example of prime number)
- **Saturation loss:** minimize high thread contention in the problem (locking common data structure will convert our parallel into serial execution)
- **Braking loss:** stop all tasks as soon as the problem is solved

EXERCISE

We compare the result of counting prime numbers with thread and with an Executors. As we expect we can see that when the number of threads exceeds the number of cores the execution time does not increase as in the previous case. The performances in this case are better compared to the version with threads; this is because if a pool of executors is used threads are reused to perform more tasks, reducing the overhead of creating and destroying threads. Also, the implementation with a local counter instead of a shared one (Atomic integer) was faster because there was no contention in accessing the shared counter.

```
final List<Future<>> list = new ArrayList<>();
final ExecutorService pool = new ForkJoinPool(8);

for (int t=0; t<threadCount; t++) {
    final int from = perThread * t,
            to = (t+1==threadCount) ? range : perThread * (t+1);
    final int threadNo = t;
    try {
        Future<> f1= pool.submit( () -> {
            long count = 0;
            for (int i=from; i<to; i++)
                if (isPrime(i))
                    count++;
            results[threadNo] = count;
        } );
        list.add(f1);
    } catch (Error ex) {
        System.out.println("At t = " + t + " I got error: " + ex);
        System.exit(0);
    }
}

try {
    for (Future<> f : list) {
        f.get(); // Wait for each future to be executed and add partial result
    }
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}

pool.shutdown(); // We are sure to be done, so we shut down the pool
long result = 0;
for (int t=0; t<threadCount; t++)
    result += results[t];
return result;
```

The last exercise was to compare lock striping on an array containing the results of the number of divisors of the prime numbers. The array with more individual locks was the

fastest.

Task .app.run			
HistogramPrimesThreads with 8 thread and 1 locks	4051550220,0 ns	255988309,52	2
HistogramPrimesThreads with 8 thread and 2 locks	3855917485,0 ns	174422401,58	2
HistogramPrimesThreads with 8 thread and 3 locks	3842269515,0 ns	197041132,56	2
HistogramPrimesThreads with 8 thread and 4 locks	3867901460,0 ns	195471330,44	2
HistogramPrimesThreads with 8 thread and 5 locks	3645858940,0 ns	172346001,04	2
HistogramPrimesThreads with 8 thread and 6 locks	3737202350,0 ns	308111952,50	2
HistogramPrimesThreads with 8 thread and 7 locks	3665925865,0 ns	116658981,59	2

7. Lock-free Data Structures: Define and motivate lock-free data structures. Explain how *compare-and-swap* (CAS) operations can be used to solve concurrency problems. Show some examples of code from your solutions to the exercises in week7

using too much locking can worsen the performances of our code, for this reason many solutions like striping and lock free data structures were introduced. Modern processors now provide the special instruction Compare And Swap, CAS, which is a store conditional operation: `value.compareAndSwap(a,b)` compares the values of `value` and `a`, if they are equal `val` is set to `b` otherwise it does nothing. Anyway it returns them current value of `value`. this operations are used for "optimistic concurrency" where you try several times until the operation succeeds. PROs of using them are:

- A CAS operation is faster than acquiring a lock
- An unsuccessful CAS operation does not cause thread blocking

BUT they result in high memory overhead, in general using CAS is more convenient performance-wise in scenarios with low contention, where multiple threads are not frequently trying to modify the same shared variable concurrently. In high-contention scenarios, CAS operations might lead to increased contention and reduced scalability.

This low-level JVM support is used by the atomic variable classes (`AtomicXxx` in `java.util.concurrent.atomic`) to provide an efficient CAS operation on numeric and reference types. Atomic variable classes provide a generalization of volatile variables to support atomic conditional read-modify-write operations

```
@Override
public void increment(int bin) {
    int currentValue;
    do {
        currentValue = count[bin].get();
    } while (!count[bin].compareAndSet(currentValue, currentValue + 1));
}
```

for example here we were implementing the histogram class, this method increments `count[bin]`. you try to modify the value of `count[bin]` until you are able to, so until no other thread modifies it between the time you save the current value in a variable and try to modify it with the compare and set instruction.

!!!!!!different from busy wait where the thread does not transition to the blocked state

An algorithm is called nonblocking if failure or suspension of any thread cannot cause failure or suspension of another thread; an algorithm is called lock-free if, at each step, some thread can make progress. Algorithms that use CAS exclusively for coordination between threads can, if constructed correctly, be both nonblocking and lock-free

lock free data structures (non blocking) divide in progress in 3 categories:

1. **wait free**: every call finishes its execution in a finite number of steps
2. **lock free**: e if executing the method guarantees that some method call (including concurrent) finishes in a finite number of steps
3. **obstruction free**: if, from any point after which it executes in isolation, it finishes in a finite number of steps; • My operations are guaranteed to complete in a bounded of steps (if I get to execute them)

$$\underset{\text{strongest}}{\text{wait}_{\text{free}}} \overset{\text{implies}}{\Rightarrow} \text{lock}_{\text{free}} \overset{\text{implies}}{\Rightarrow} \text{obstruction}_{\text{free}}$$

in Java the package `java.util.concurrent.atomic` is a lock free library from which you can access Atomic objects, objects whose methods are atomic (same semantic of volatile).

A CAS operation is faster than acquiring a lock, an unsuccessful CAS operation does not cause thread de-scheduling (blocking).

The ABA problem is an anomaly that can arise from the naive use of compare-and-swap in algorithms where **nodes can be recycled**. If you cannot avoid the ABA problem by letting the garbage collector manage link nodes for you, there is still a relatively simple solution: instead of updating the value of a reference, update a pair of values, a reference and a version number. An AtomicStampedReference object encapsulates both a reference to an object of Type T and an integer stamp. Java garbage collector ensures that a node cannot be reused by one thread, as long as that node is accessible to another thread.

EXERCISE we implemented as read-write lock:

```

public class ReadWriteCASLock implements SimpleRWTryLockInterface {

    private AtomicReference<Holders> holders = new AtomicReference<Holders>();

    // 7.2.1
    public boolean writerTryLock() {

        Writer writer = new Writer(Thread.currentThread());
        // Lock is currently unheld if holders is null
        if (holders.compareAndSet(expectedValue:null, writer)) {
            return true; // Success
        } else {
            return false; // fail
        }
    }

    // 7.2.2
    public void writerUnlock() {
        Writer current = (Writer) holders.get();

        if (current != null && current.thread == Thread.currentThread()) {
            holders.compareAndSet(current, newValue:null); // Release
        } else {
            throw new IllegalStateException(s:"Not the current write lock holder or lock is not held.");
        }
    }
}

```

```

// 7.2.3
public boolean readerTryLock() {
    Thread currentThread = Thread.currentThread();
    ReaderList newReader;

    while (true) {
        Holders currentHolders = holders.get();

        if (currentHolders == null || currentHolders instanceof ReaderList) { // no one or a Reader holds the lock
            newReader = currentHolders == null ? new ReaderList(currentThread, next:null)
                : new ReaderList(currentThread, (ReaderList) currentHolders);
            if (holders.compareAndSet(currentHolders, newReader))
                return true;
        } else {
            return false; // A writer has the lock
        }
    }
}

```

```

public void readerUnlock() {
    Thread currentThread = Thread.currentThread();

    while (true) {
        Holders currentHolders = holders.get();

        if (currentHolders == null) {
            throw new IllegalStateException(s:"Not holding a read lock");
        } else if (currentHolders instanceof ReaderList) {
            ReaderList currentReaderList = (ReaderList) currentHolders;

            // Check if the current thread is in the reader list
            if (currentReaderList.contains(currentThread)) {
                ReaderList updatedReaderList = currentReaderList.remove(currentThread);

                // Attempt to atomically update the holders field with the new reader list
                if (holders.compareAndSet(currentHolders, updatedReaderList)) {
                    return; // Success
                }
            } else {
                throw new IllegalStateException(s:"Not the current read lock holder");
            }
        } else {
            throw new IllegalStateException(s:"thread does not have read lock");
        }
    }
}

```

8. Linearizability: Explain and motivate linearizability. Explain for what type of concurrent objects linearizability is specially useful, and describe how it can be applied to reason about their correctness. Show some examples of code in your solutions to the exercises in week 8 where you used linearizability to reason about correctness.

We introduce linearizability to reason about the correctness (based on some notion of equivalence with sequential behavior) of lock free data structure and to write the concurrent specification of them. In fact it is easier to reason about concurrent objects if we can somehow map their concurrent executions to sequential ones, and limit our reasoning to these sequential executions. The GOAL of linearizability is to use specifications of sequential objects as a basis for the correctness of concurrent objects. In sequential programs, the compiler and CPU are allowed to re-order instructions as long as they produce a valid result (according to the specification). We examine three correctness conditions. **Quiescent consistency** is appropriate for applications that require high performance at the cost of placing relatively weak constraints on object behavior (doesn't require program order but

only order on operations not concurrent in real time). **Sequential consistency** is a stronger condition, often useful for describing low-level systems such as hardware memory interfaces. **Linearizability**, even stronger, is useful for describing higher level systems composed from linearizable components. For concurrent executions, we must define the conditions asserting that every thread is behaving consistently w.r.t. a sequential execution, to do so we introduce sequential consistency.

A concurrent execution is **sequentially consistent** iff there exists a reordering of operations producing a sequential execution where:

1. Operations happen one-at-a-time
2. Program order is preserved (for each thread)
3. The execution satisfies the specification of the object

linearizability extends sequential consistency requiring that the real time order of the execution is preserved, Each method call should appear to take effect instantaneously at some moment between its invocation and response. It can be used to reason about objects that do not use locks. To show that a concurrent execution is linearizable, we must define **linearization points (the instant in the execution when the method call takes effect)** for each method call, which map to sequential execution that satisfy the specification of the object.

now transfer it to object: A concurrent object is linearizable iff

1. All executions are linearizable, difficult to prove
2. All linearizations satisfy the sequential specification of the object (linearization points correspond to CAS operations).

PRO: If two objects are linearizable, any concurrent execution involving the two objects remains linearizable (linearizability is compositional; the result of composing linearizable objects is linearizable), Correctness proofs can be split into single objects that later can be safely composed, usable on objects that do not use locks.

EXERCISE is the implementation of a lock free stack

```

class LockFreeStack<T> {
    AtomicReference<Node<T>> top = new AtomicReference<Node<T>>(); // Initializes to null

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead)); // PUSH1
    }

    public T pop() {
        Node<T> newHead;
        Node<T> oldHead;
        do {
            oldHead = top.get();
            if (oldHead == null) { // POP1
                return null;
            }
            newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead, newHead)); // POP2

        return oldHead.value;
    }
}

```

PUSH1 is the linearization point for push if during the creation of the new node there were no other push/pop (the head did not change).

POP1 is the linearization point for POP if the queue is empty

POP2 is the linearization point of pop if the queue is not empty and no other push or pop occur during the creation of the new head.

correctness:

- if 2 threads execute push concurrently, only one succeeds in executing push1, while the other will repeat the push operation
- if queue is empty then POP1 succeeds
- if a thread executes pop while another thread just updated the head due to a push or pop then pop2 fails and the thread restarts the pop operation.
- if 2 threads execute pop concurrently before the head is updated and the queue is not empty then POP2 succeeds only for one of them and the other will restart the pop operation
- if a thread executes push while another thread updated the head due to a pop then PUSH1 fails and the thread restarts the push

9. Streams: Explain and motivate the use of streams to parallelize computation. Discuss issues that arise in operations executed by parallel streams. Show some examples of code from your solutions to the exercises in week 10.

Streams represent a different way to parallelize computation: they use a pipeline (a sequence of aggregate operations) approach where each thread has a different task. It is useful where data is communicated as a stream (for example when you read from a file, from the web,...). *Stream* is a sequence of data elements that can be processed concurrently; Java introduced them from Java 8, Java Stream is a functional programming approach (declarative and parallelizable). There are sequential streams and parallel streams, they both have lazy initialization : operations are not executed until a terminal operation is called, it can be said that data is pulled.

they are made of 3 different operation inside a pipeline:

1. sources (arrays, collections, IO, generators, BUFFERREADER(url/file), arrays.stream(array), stream(), Stream.of(1,2,3,4), IntStream.iterate(0, x->x+1))
Provides the data for the stream
2. intermediate operations (transforming one stream into another (e.g. filter, map, limit(n), skip(n), distinct, sorted) A computation on each element of the string
3. terminal operations (count, min, max, sum, average, reduce(identity, accumulator) , collect, forEach, ...)

in parallel stream (which can be created either with .parallel() on a stream or parallelStream() on a collection) disjointed streams are assigned to distinct threads from a thread pool. Their execution is parallel so the processing of the stream is not guaranteed to be in order.

Stream operations use internal iteration when processing elements of a stream so the total stream will not be in order.

Problem can arise from concurrent modifications, you may need to use special functions like groupingByConcurrent to avoid problems (grouping can give problem if parallelized and then modified).

Pro of parallel streams:

1. they enable the concurrent processing of elements in a stream and parallelization of many common operations on streams, resulting in better utilization of computational resources and often faster execution times.
2. Simplicity and Readability: easy syntax they make the code more readable and easier to understand. Developers can focus on the functional aspects of their code without explicitly managing threads or synchronization.
3. Automatic Load Balancing: framework automatically handles load balancing, distributing the workload evenly across available processors.

cons of parallel streams:

1. in sequential streams the elements are processed 1 by 1 in the order they appear in the array, while on parallel streams that order is disrupted.

2. java streams uses lazy evaluation, so operations are not executed until a terminal operation is called.
3. concurrent modifications in parallel streams can lead to unexpected behavior.
4. you do not necessarily see improved performances: depending on the hw, the dataset and the type of operations there is an overhead of parallelization.

EXE: here we had to count the number of prime integers in a range

```
// IntStream solution
private static long countIntStream(int range) {
    long count = 0;
    count = IntStream.range(startInclusive:0, range)
        .filter(x -> isPrime(x))
        .count();
    return count;
}
```

```
// parallel Stream solution
private static long countparallelStream(List<Integer> list)
{
    long count = 0;
    count = list.stream()
        .parallel()
        .filter(i -> isPrime(i))
        .count();
    return count;
}
```

another paradigm is RxJava, that is reactive programming approach. there are observer that receive data , observables propagates data from a source. the main difference to streams is that here data is pushed, which can lead to backpressure (An observable may emit items so fast that the consumer can not keep up).

10. Message Passing: Explain and motivate the actor model of concurrent computation. Discuss advantages and disadvantages of approaches to distribute computation in actor systems. Show some examples of code from your solutions to the exercises in week 11 and 12.

The actor model was introduced as an example of message passing paradigm, since sharing memory between threads can cause many problems like race conditions, data races, visibility issues,... we changed paradigm to message passing where threads only work on their own local memory, so they DO NOT share state and if synchronization is needed they send messages to each other. there are multiple ways to do so, for example using sockets, RPC,... or the actor model which is a concurrency model with message passing built in.

An actor can be seen as a sequential unit of computation, we think of actors as an abstraction of thread. each actor only has access to its local state and their mailbox, and is identified by an id. they can only:

1. receive messages from other actors, he takes messages from the mailbox and he blocks if the mail is empty
2. send asynchronous messages to other actors, the sender places the message in the mailbox of the receiver, NON BLOCKING
3. create new actors
4. change its behavior (state/messagehandlers)

NO GUARANTEES ON THE ORDER OF ARRIVAL OF MESSAGES, if a response is needed remember the mailboxes are FIFO.

Akka is the toolkit we studied to implement this model, each actor has its own class made of: the internal state, a private constructor that takes context as a parameter, an initial behaviour used to create the new actor, the messages he can handle (their classes need to be thread safe) and message handlers that return the behaviour of the actor after processing the message. Akka actors are in a hierarchy, The first actor to be created in the system is a top-level actor known as guardian, this actor is created with `ActorSystem.create` and then he creates the initial actors (`spawn()`) in the system when he receives a kickoff message.

The default supervision strategy is to stop the actor if an exception is thrown. In many cases you will want to further customize this behavior. To use supervision the actual Actor behavior is wrapped using `Behaviors.supervise`. Typically you would wrap the actor with supervision in the parent when spawning it as a child. This example restarts the actor

Vedi accenno cluster sulle slide

here an example of the guardian we created for the mobile payment system exercise:

```

public class Guardian extends AbstractBehavior<Guardian.GuardianCommand> {

    /* --- Messages ----- */
    public interface GuardianCommand {
    }

    public static final class KickOffGuardian implements GuardianCommand {
        private final String mobileAppName;

        public void KickOff(String mobileAppName) {
            this.mobileAppName = mobileAppName;
        }

        public String getMobileAppName() {
            return mobileAppName;
        }
    }
}

/* --- State ----- */
// empty

/* --- Constructor ----- */
private Guardian(ACTOR_CONTEXT<GuardianCommand> context) {
    super(context);
}

```

```

/* --- Actor initial state ----- */
public static Behavior<Guardian.GuardianCommand> create() {
    return Behaviors.setup(Guardian::new);
}

/* --- Message handling ----- */
@Override
public Receive<GuardianCommand> createReceive() {
    return newReceiveBuilder()
        .onMessage(KickOffGuardian.class, this::onKickOff)
        .build();
}

```

```

/* --- Handlers ----- */
private Behavior<GuardianCommand> onKickOff(KickOffGuardian msg) {

    // spawn the MobileApp actor 11.1.1
    ActorRef<MobileApp.MobileAppCommand> mobileApp = getContext().spawn(MobileApp.create(), "mobileApp_actor");
    getContext().getLog().info("Mobile app {} started!", "mobileApp_actor");

    // 11.1.5 (2 mobile apps, 2 banks, and 2 accounts)
    ActorRef<MobileApp.MobileAppCommand> mb1 = getContext().spawn(MobileApp.create(), "mb1_actor");
    ActorRef<MobileApp.MobileAppCommand> mb2 = getContext().spawn(MobileApp.create(), "mb2_actor");
    ActorRef<Bank.BankCommand> b1 = getContext().spawn(Bank.create(), "b1_actor");
    ActorRef<Bank.BankCommand> b2 = getContext().spawn(Bank.create(), "b1_actor");
    ActorRef<Account.AccountCommand> a1 = getContext().spawn(Account.create(), "a1_actor");
    ActorRef<Account.AccountCommand> a2 = getContext().spawn(Account.create(), "a2_actor");
    b1.tell(new Bank.Transaction(a1, a2, 100));
    b2.tell(new Bank.Transaction(a2, a1, 100));
    return this;
}

```

Actors Model	Akka
Actor	Actor class (AbstractBehaviour)
Mailbox Address	Reference to Actor class
Message	Message static final class
State	Actor class local attributes
Behaviour	Handler functions in the Actor class
Create actor	API function
Send message	API function
Receive message	Message handler builder (from API)

.tell(object message) is used to send messages, **spawn()** used by the guardian to create new actors.

the actor model has a natural mapping in distributed systems, but there are advantages and cons of using it:

1. load balancing, is done better if the topology is dynamic: The Actors model encourages creating many actors that perform small tasks and communicate with each other. however performance depends on the hardware: Akka implements actors systems using ForkJoinPools but , actor systems can be distributed among many JVMs and computer

PRO: distributing computation among actors makes it easy to implement fault-tolerant systems and adaptive load-balancing (using dynamic topologies and ForkJoinPools on many JVMs and computers)