IN5170 Models of Concurrency

Lecture 2: Java

Einar Broch Johnsen, S. Lizeth Tapia Tarifa, Eduard Kamburjan, Juliane Päßler

August 28, 2023

University of Oslo

Summary of Last Lecture

- Shared memory systems
- Parallel execution: many interleavings
- Atomic operations
- Program order, At-Most-Once, Interference
- Await-language and Critical sections
- Synchronization

Purpose

- Lecture focuses on general concepts in a simple language
- Mainstream language embed concurrency in further structures
- How to map concepts to languages?

- **Part I: Basic Java Concurrency**
- Part II: Concurrency in Go
- Part III: Concurrency in Rust

Threads Basics

Threads in Java

- Map to native threads to enable multi-core execution

Processes vs. Threads (in Java)

- A process is an independent instance running in its own memory space.
- A thread runs inside a process and shares its resources with other threads

- We focus on threads, multi-process applications in Java are possible but come with OS/JVM specific issues
- Both are handled by OS scheduler
- Both have costly context switches but threads are more light-weight
  - Only processes require full cache flushing as they change virtual memory
  - Thread-switch can retain caches, only changes processor state (registers etc.)

Threads

- The `Thread` class encapsulates a system thread
- The `Runnable` interface is used to define thread behavior

Threads

- The `Thread` class encapsulates a system thread
- The `Runnable` interface is used to define thread behavior

Java

```java
class Printer implements Runnable {
  private String text;
  public Printer(String text) { this.text = text; }
  public void run() { System.out.println(text); }
  public static void main(String args[]) {
    Thread t1 = new Thread(new Printer("Hello"));
    Thread t2 = new Thread(new Printer("Concurrency"));
    t1.start(); t2.start();
  }}
```

Threads

- The `Thread` class encapsulates a system thread
- The `Runnable` interface is used to define thread behavior

`Start` vs `run`

- `Thread.start()` starts a new *concurrent* thread
- `Runnable.run()` just executes the code *sequentially*
- `Thread.start()` calls `Runnable.run()` internally
- Calling `Runnable.run()` directly rarely makes sense

Threads

- The `Thread` class encapsulates a system thread
- The `Runnable` interface is used to define thread behavior

Common pattern: anonymous runnables

Java

```java
// with lambdas
new Thread ( () -> { /* do things */ } ).start();
//pre Java 8
new Thread(new Runnable(){
    public void run() { /* do things */ }}).start();
```

Threads

Shared State

Shared state between threads is introduced through multiple means

Threads

Shared State

Shared state between threads is introduced through multiple means

- Static state

Java

```java
class C { public static int i = 0 }
...
new Thread ( () -> { i++ } ).start();
new Thread ( () -> { i++ } ).start();
```

Threads

Shared State

Shared state between threads is introduced through multiple means

- Static state
- Shared references to objects

Java

```
class C { public int i = 0 }
...
final C c = new C(); //only final variable can be captured
new Thread ( () -> { c.i++ } ).start();
new Thread ( () -> { c.i++ } ).start();
```

Threads

Shared State

Shared state between threads is introduced through multiple means

- Static state
- Shared references to objects
- Resources, e.g., files

Java

```java
//No internal sharing, path may even be different (links)
new Thread( () -> { new File("/path/").delete();} ).start();
new Thread( () -> { new File("/path/").delete();} ).start();
```

Threads

- Java offers some additional operations that are abstracted away in Await
- Static methods of `Thread` refer to current thread

Threads

- Java offers some additional operations that are abstracted away in Await
- Static methods of `Thread` refer to current thread
- `join` waits for the thread to finish

Java

```java
public static void main(String args[]) {
    Thread t1 = new Thread(new Printer("Hello"));
    Thread t2 = new Thread(new Printer("Concurrency"));
    t1.start(); t1.join(); // waits for t1 to finish
    t2.start();
  }
```

Threads

- Java offers some additional operations that are abstracted away in Await
- Static methods of `Thread` refer to current thread
- `join` waits for the thread to finish
- `sleep` suspends the thread for at least n milliseconds

Java

```
int i = 0; //shared
  ...
  return m(){
    while(i == 0) Thread.sleep(10); //some constant chosen
    return 10/i;
  }
```

Threads

- Java offers some additional operations that are abstracted away in Await
- Static methods of `Thread` refer to current thread
- `join` waits for the thread to finish
- `sleep` suspends the thread for at least *n* milliseconds
- `yield` gives the scheduler the signal to schedule someone else first

Await

```
int i = 0; //shared
  ...
  return m(){
    while(i == 0) Thread.yield();
    return 10/i;
  }
```

Quiz: Java and Async

Await

co s1 || s2 **oc**; s3

Java

```
Thread t1 = new Thread(() -> {s1});
Thread t2 = new Thread(() -> {s2});
Thread t3 = new Thread(() -> {s3});
??
```

Quiz: Java and Async

Await

```
co s1 || s2 oc; s3
```

Java

```
Thread t1 = new Thread(() -> {s1});
Thread t2 = new Thread(() -> {s2});
Thread t3 = new Thread(() -> {s3});
t1.start(); t2.start();
t1.join(); t2.join();
t3.start();
```

Quiz: Java and Async

Await

co <s1> || <s2> **oc**

Java

```
public class C() {
??
s1
??
s2
??
}
```

Quiz: Java and Async

Await

co <s1> || <s2> **oc**

Java

```
public class C() {
public static synchronized void m1(){s1}
public static synchronized void m2(){s2}
...

new Thread(() -> {C.m1();}).start();
new Thread(() -> {C.m2();}).start();
}
```

Quiz: Java and Async

In the following, we mostly show the code inside the threads and omit the `Thread`/`Runnable`s

Atomic Blocks and `synchronized`

Synchronization

- Java does not have atomic blocks in the same form
- Instead: synchronized methods and blocks

Java

```java
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {c++;}
    public synchronized void decrement() {c--;}
    public synchronized int value() {return c;}
}
```

- Synchronized methods are atomic *per object*
- Only one thread can execute such a method at any time

Synchronization

- All synchronized methods are synchronized with each other
- I.e., no two synchronized methods can be executed at the same time on one instance

Synchronization

- All synchronized methods are synchronized with each other
- I.e., no two synchronized methods can be executed at the same time on one instance

Java

```java
SynchronizedCounter c1 = new SynchronizedCounter();
SynchronizedCounter c2 = new SynchronizedCounter();
Runnable r1 = new Runnable(){
    public void run() { c1.increment(); } }
Runnable r2 = new Runnable(){
    public void run() { c1.increment(); } }
Runnable r3 = new Runnable(){
    public void run() { c1.decrement(); } }
Runnable r4 = new Runnable(){
    public void run() { c2.increment(); } }
```

Synchronization

- All synchronized methods are synchronized with each other
- I.e., no two synchronized methods can be executed at the same time on one instance

The following will not interleave on `c1`:

Java

```java
new Thread(r1).start();
new Thread(r2).start();
```

Synchronization

- All synchronized methods are synchronized with each other
- I.e., no two synchronized methods can be executed at the same time on one instance

The following will also not interleave on c1:

Java

```
new Thread(r1).start();
new Thread(r3).start();
```

Synchronization

- All synchronized methods are synchronized with each other
- I.e., no two synchronized methods can be executed at the same time on one instance

The following will interleave – the call is to two different objects

Java

```
new Thread(r1).start();
new Thread(r4).start();
```

Synchronization

- Synchronized static methods are *per class*
- This is almost a global atomic block

Synchronization

- Synchronized static methods are *per class*
- This is almost a global atomic block

Java

```java
public class StaticSyncCounter {
    private static int c = 0;
    public synchronized static void increment() {c++;}
    public synchronized static void decrement() {c--;}
    public synchronized static int value() {return c;}
}
```

Synchronization

- Synchronized static methods are *per class*
- This is almost a global atomic block

Java

```java
Runnable s1 = new Runnable(){
    public void run() { StaticSyncCounter.increment(); } }
Runnable s2 = new Runnable(){
    public void run() { StaticSyncCounter.decrement(); } }
```

Synchronization

- Synchronized static methods are *per class*
- This is almost a global atomic block

The following will not interleave

Java

```java
new Thread(s1).start();
new Thread(s2).start();
```

Synchronization

- Synchronized blocks can be used outside methods with explicit lock

Synchronization

- Synchronized blocks can be used outside methods with explicit lock
- Any object can be a lock, just need some identity

Java

```java
public class C(){
  int l = 0;
  int r = 0;
  void method(Object lock, boolean left){
    synchronized(lock){
      if(left) l++ else r++;
    }
  }
}
```

Synchronization

- Synchronized blocks can be used outside methods with explicit lock
- Any object can be a lock, just need some identity
- Synchronized methods have **this** as the lock

Java

```java
public class C(){
  synchronized void method(){ ... }
  void method(){ synchronized(this) {...} }
}
```

Atomic Expressions

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```java
x++; // x is only local variable, declared as long
```

How many machine instructions will this be?

Atomic Expressions

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```java
x++; // x is only local variable, declared as long
```

How many machine instructions will this be? **4-6**

Atomic Expressions

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```java
x++; // x is only local variable, declared as long
```

JVM

The JVM has no registers, but loads from variables/heap onto a stack. All computations target the top values on the stack.

Atomic Expressions

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```java
x++; // x is only local variable, declared as long
```

```
LLOAD_1      // push value from local variable #1
LCONST_1     // push value 1
LADD         // add 2 top-most values
LSTORE_1     // store value into local variable #1
```

Atomic Expressions

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```
x++; // x is only local variable, declared as long
```

- Reference reads and writes are atomic
- Basic type reads and writes except `long` and `double` are guaranteed to be atomic
- On 64bit machines, `long` and `double` reads and writes *might* be atomic, *might* be two instructions
- Accessing variables modified by `volatile` is always atomic

Weak Memory and `volatile`

Weak Memory

Java Memory Model

The JVM defines a *weak* memory model.

A weak memory model allows certain reorderings in read and write operations.

Weak Memory

Java Memory Model

The JVM defines a *weak* memory model.

A weak memory model allows certain reorderings in read and write operations.

- Mainly targeting performance, e.g., for cache optimization

```
x := a; y := 1; z := x;          //y := 1 might flush cache
x := a; r := x; y := 1; z := r; //r is a register, not memory
```

- Can be done statically (by compiler) or dynamically (by processor)
- Which reorderings are allowed exactly, is defined by the memory model
- Strong memory model = no reorderings are allowed

Weak Memory

Independence

Reordering must take into account whether the operations are independent

- `x := 1; r := x;` cannot be reordered
- `r := x; x := 1;` cannot be reordered

Weak Memory

Independence

Reordering must take into account whether the operations are independent

- `x := 1; r := x;` cannot be reordered
- `r := x; x := 1;` cannot be reordered

- Read-read reordering can reorder reads
- Write-read reordering can move a read before a write
- Read-write reorderings can move a write before a read
- Write-write reorderings change order of stores
- Some architectures also reorder other atomic operations

Weak Memory

Weak Memory Models can lead to very unintuitive results in concurrent settings

Weak Memory

Weak Memory Models can lead to very unintuitive results in concurrent settings

```
int x,y; //default 0
```

```
x  := 1;  //shared variable
r1 := y;  //register
print r1;
```

```
y  := 1;  //shared variable
r2 := x;
print r2;
```

What are possible outputs?

Weak Memory Models can lead to very unintuitive results in concurrent settings

```
int x,y; //default 0
```

```
x  := 1;  //shared variable
r1 := y;  //register
print r1;
```

```
y  := 1;  //shared variable
r2 := x;
print r2;
```

Is `0,0` possible?

Weak Memory

Weak Memory Models can lead to very unintuitive results in concurrent settings

```
int x,y; //default 0
```

```
x  := 1;  //shared variable
r1 := y;  //register
print r1;
```

```
y  := 1;  //shared variable
r2 := x;
print r2;
```

- If the read of `x` in the second thread is reordered, then `0,0` is possible
- This output cannot be explained by reasoning about interleavings
- If the language does not require variables to be initialized, we get *out-of-thin-air* values. Then, even `12,13` is a possible output.

Weak Memory

Sequential Consistency

Most weak memory models guarantee *sequential consistency*: *If there is no data race, then the observable behavior of the program is as if under a strong memory model*.

Weak Memory

Sequential Consistency

Most weak memory models guarantee *sequential consistency*: *If there is no data race, then the observable behavior of the program is as if under a strong memory model*.

- "No data race" may be a very strong restriction and lead to unnecessary synchronization
- The term *observable behavior* depends on the programming language
- We need more fine-grained control – `volatile`

Weak Memory

Java

```
public class C {
  private volatile long l = 5;
  long incRet() { return l++; } //called from two threads
}
```

- All read and write accesses to `l` are atomic
- All write accesses to `l` are *immediately* visible to all threads
- In terms of memory model: no reads and writes to `l` are reordered before any write
- In terms of memory: `l` is read and written from global memory, not thread caches
- Does *not* introduce synchronization, but removes opportunities for optimization and makes access more expensive

Weak Memory

Weak memory is a complex topic, mostly relevant to low level architectures and compilers

- Further operations: read-own-write-early, read-others-write-early
- Leaks to programmer in concurrent settings
- Hard to debug, most languages have no clearly defined memory model
- Often hardware-dependent solutions

Weak Memory

Weak memory is a complex topic, mostly relevant to low level architectures and compilers

- Further operations: read-own-write-early, read-others-write-early
- Leaks to programmer in concurrent settings
- Hard to debug, most languages have no clearly defined memory model
- Often hardware-dependent solutions

Rough guideline on when to use `volatile`

- If a field is not supposed to have data races, do not use volatile
- If a field will have data races, and you do not want to remove them, consider using volatile to avoid unintuitive behavior

Further Concepts

Java's Standard Library

- Java's standard library offers further data structures for common patterns or to encapsulate complex but efficient solutions
- Thread-safe collections are less efficient, but internally race-free versions of collections
- Atomic classes encapsulate data with efficient, atomic access

Standard Library

- Atomic classes are available for data and references
- Fixed operations that are atomic without `synchronized` blocks
- Are more efficient (less blocking), but less clear control flow

Java

```java
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {c++;}
    public synchronized void decrement() {c--;}
    public synchronized int value() {return c;}
}
```

Standard Library

- Atomic classes are available for data and references
- Fixed operations that are atomic without `synchronized` blocks
- Are more efficient (less blocking), but less clear control flow

Java

```java
public class SynchronizedCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {c.incrementAndGet();}
    public void decrement() {c.decrementAndGet();}
    public int value() {return c.get();}
}
```

Standard Library

- Atomic classes are available for data and references
- Fixed operations that are atomic without `synchronized` blocks
- Are more efficient (less blocking), but less clear control flow

`AtomicReference` does *not* make the called methods of its content atomic.

Java

```java
AtomicReference<C> cache = new AtomicReference<C>();
cache.get();       // atomic
cache.get().m();  // m is not atomic
```

Standard Library

Standard Library

- Thread-safe collections provide atomic methods to access lists etc.

Concurrent Collections

Provide concurrent implementations that enable concurrent access

- `ConcurrentHashMap` vs `ConcurrentMap`
- `ConcurrentLinkedQueue` and variants for lists without random access
- `CopyOnWriteArrayList` makes an arraylist concurrent by making a copy on every write, is not efficient

Standard Library

Synchronized Collections

Normal implementations, but add **synchronized** at the right places

Java

```java
<T> Collection<T> synchronizedCollection(Collection<T> c)
```

Usage:

Java

```java
ArrayList<Object> a = new ArrayList<>();
Collection<Object> b = Collections.synchronizedCollection(a);
// access through a is still unsafe
```

Thread Management

- In bigger applications, you may need to manage sets of threads
- We consider three concepts
  - Lifecycle of a thread object
  - Interrupts
  - Thread pools

Lifecycle

- A created thread object is **New**
- After calling `start` the thread is either
  - **Running**, i.e., executes right now
  - **Runnable**, i.e., waits to be scheduled (`yields` gets you here)
  - **Waiting**/**Sleeping**/**Blocked**, i.e., waits for time to pass or some notification or lock
- Once the internal `run` method terminates, the object is **Dead**
- Once a thread is dead it cannot be restarted

Interrupts

An interrupt is an *indication* to a thread that it should stop what it is doing and reconsider.

- Can be invoked using `t.interrupt();`
- This sets the `Thread.interrupted` flag,
- Some methods, like `Thread.sleep()` will throw a `InterruptedException` if active
- The `run` method does not – programmer must take care of reacting to this flag

Java

```java
() -> {
    //long computation 1
    if(Thread.interrupted){ /* handler */ }
    //long computation 2
}
```

Thread Pools

- Creating and starting threads is costly
- Dead threads cannot be reused
- Solution: create a set of threads that do not terminate, but wait for new runnables to execute
- Automatic scaling

Thread Pools

An `ExecutorService` manages a set of threads, and accepts `Runnable` instance submissions

Thread Pools

An `ExecutorService` manages a set of threads, and accepts `Runnable` instance submissions

Java

```java
//has exactly 2 threads
ExecutorService service = Executors.newFixedThreadPool(2);
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
//last runnable put in query, will be executed later
```

Thread Pools

An `ExecutorService` manages a set of threads, and accepts `Runnable` instance submissions

Java

```java
ExecutorService service = Executors.newCachedThreadPool(0,3);
//starts with 0 threads
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
//up to 3 threads running
```

Thread Pools

- To join on such a task, we get a *Future*
- We will investigate futures in more detail in Part 2 of the course

Java

```
//has exactly 2 threads
ExecutorService service = Executors.newFixedThreadPool(2);
Future<Int> f = service.submit(() -> { /* do */ return 1;});
...
Int = f.get(); //essentially a join
```

- Thread pools have further capabilities (shutdown)
- Details very java-specific, omitted here

Outlook

- We use the material taught so far as the basis for obligs and exercises
- Next lectures connect concepts with corresponding Java concept
- Bigger projects use other concurrency libraries that build on `java.util.concurrent`, e.g., Google Guava