Zsm Prog 2

30. Mai, 2023; rev. 12. Juni 2023

Linda Riesen

Remember isch riese stress & You got this: Deadlock überspringen ⇒ braucht zuviel zeit. (Mutex ist ok), Lambda zuerst machen

Serilizable nachschauen

1 Build Automation

Software Automation: Helps with Build automation (Building Components, Resolving Depondencies, Running Tests, etc),

Software Automation: Helps with

- Build automation: (Building Components, Resolving Dependencies, Running Tests, etc)
- Continuous Integration (Automatically Build, Test, Integrate Components and Run Integration Tests) (Server)
- Continuous Delivery (Create Releases, Deploy, Run Acceptance Tests)
- Continuous Deployment (Automatically Deploy to Production)
- DevOps (Automatically run Operation of Production System (Config Management, Backup, etc.))

Build Automation (Gradle) has to be:

- Automated
- Repeatable, Consistent
- Incremental (if Build is Interrupted, should continue there, and only build new stuff)
- Platform independent
- Seamlessly Integrated (server / locally)

2 Concurrency

2.1 True Concurrency / Interleaving Concurrency

Computers with multiple CPU cores, each core can run a flow independently in parallel (true concurrency) if more flows than core: interleaving concurrency (= quasi/pseudo concurrency) simulated by **time slicing** (Cores switch in rapid succession)

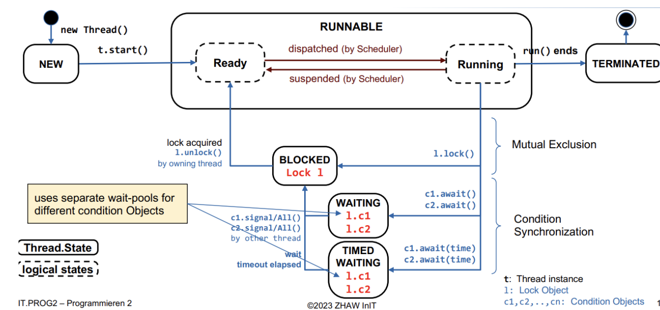

Abbildung 1: Thread Livecycle Overview

2.2 Scheduling Strategies

Non-Preemptive (cooperative) Process releases core voluntarily

Preemptive ("verdrängend") Scheduler can interrupt a process (time-slice based), priority based

Real Time Very tight timing requirements

2.3 Multi processing vs. Multi Threading

Multi Processing:

- Running in separate memory area → access to a memory area of another process is not possible

- Inter Process Communication (IPC) using special shared memory area or mechanisms like pipes or sockets
- Process switching is expensive; whole process state has to be saved and reloaded

Multi Threading

- Running in shared process memory → data is accessible by all threads
- Thread switching is cheaper; process state has not to be changed
- Basically it only moves the program Counter to the new position in the code, and saves / restores the registers

2.4 Mutual Exclusion (Mutex = "Wechselseitiger Ausschluss")

(Ansonsten: Lost Update, Race Condition)
Avoid shared resources (often not possible)
– For simple cases use Atomic Types
– Only give one thread access to the shared resource at a time → Mutual Exclusion

2.4.1 Mutex

- Lock with Atomic Boolean
- synchronized Block / Method: (Waiting Room Concept) also Possible Deadlocks

2.5 Deadlock

Mutual Exclusion (each resource only available once), Hold & Wait Condition (Processes which are already blocking resources claim additional), No Preemption (Cannot be Taken away by OS), Cyclic Waiting Condition (Processor / Successor Blocking)

2.6 Java

Thread ermöglichen:

- Extend: Thread Class and Override run()
  Thread myThread = new MyThreadClass();
  myThread.start() (needs to be ended with e.g. join)

- Implement: Runnable
  new Thread(new MyRunnable()).start() (needs to be ended with e.g. join)

Thread endet wenn:
run() Methode terminiert: (immer unklar wenn)
mit AtomicBoolean
mit join

2.7 Executor Framework

Executor: Simple functional interface that contains the method: public void execute(Runnable task): Promise to execute the given runnable at some time in the future
Executor Service provides methods to manage termination of tasks (e.g. shutdown), additional submit methods allow tracking progress of tasks by returning a Future object (required for Callable objects, which we will cover later)
Scheduled Executor Service ExecutorService, that adds functionality to schedule the execution of a task for a specific time and interval.
ExecutorService executorService = Executors.newSingleThreadExecutor();
Runnable runnable = new SimpleTask();
executorService.execute(runnable);

2.8 Thread Pool

- SingleThreadExecutor: ThreadPool"with only one thread which will execute the submitted tasks sequentially
- FixedThreadPool: Creates and reuses a fixed number of threads
- CachedThreadPool: creates new threads as needed, but reuses previously created threads when they are available. idle threads will be terminated and removed after 60s (shrinking the pool), ideal for programs using many short-lived asynchronous tasks

2.9 Nested Classes

Used to create simple one time tasks

3 Java FX

launch(Class<?extends Application> appClass, String... args) or launch(String... args) // uses current class
Static method, generally called from main()

```
//Creating the mouse event handler
EventHandler<MouseEvent> eventHandler = new EventHandler<MouseEvent>
   @Override
   public void handle(MouseEvent e) {
      System.out.println("Hello World");
      circle.setFill(Color.DARKSLATEBLUE);
   }
};
//Adding event Filter
Circle.addEventFilter(MouseEvent.MOUSE_CLICKED, eventHandler);
```

n the same way, you can remove a filter using the method removeEventFilter(
s shown below –

```
circle.removeEventFilter(MouseEvent.MOUSE_CLICKED, eventHandler);
```

Abbildung 2: Eventhandler Usage

```
@FXML
private Text label;

private SimpleStringProperty text = new SimpleStringProperty("A");
@FXML
private TextField tf1;
@FXML
private TextField tf2;

private ObservableList<String> list = FXCollections.observableArrayList();

@Override
public void initialize(URL url, ResourceBundle rb) {
    //sumNonObservable();
    sumObservable();

    MyTask task = new MyTask();
    Timer timer = new Timer(true);
    timer.scheduleAtFixedRate(task, 1000, 1000);

    label.textProperty().bind(text);

    tf1.textProperty().bindBidirectional(tf2.textProperty());

    list.addListener((javafx.beans.Observable observable) -> {
        System.out.println("List invalidated");
    });
}

private void sumNonObservable() {
    int a = 10;
    int b = 10;
    int sum = a + b;
    System.out.println(sum); //20
    a = 20;
    System.out.println(sum); //20
}
```

Abbildung 3: Observable Usage

```
class MyTask extends TimerTask {

    @Override
    public void run() {
        final int ch = text.get().charAt(0);
        Platform.runLater(() -> {
            String str = String.valueOf((char) (ch + 1));
            list.add(str);
            text.set(String.valueOf((char) ch));
        });
    }
}
```

Abbildung 4: Fxml Tasks

1. Constructs an instance of the given javafx.application.Application class
2. Creates an initial window (stage) and starts a thread to handle events

3. init() method is called (empty method, can be overridden to initialize environment (e.g., external connections), no GUI elements (stage) available)
4. start(javafx.stage.Stage) method is called (Mandatory abstract entry point method to set up JavaFX UI)
5. Waits for the application to finish, which happens when either of the following occur: the application calls Platform.exit() (if System.exit() is called, stop() is not executed), the last window has been closed
6. stop() method is called (Empty method, can be overridden to cleanup environment)

3.1 Model / View / Controller

User Interface Contains the GUI components& logic: Responsible for the interaction between a user and the application
Model Contains the data and implementation of (domain) logic: Usually not visible to the user, should be independent of the UI
View Contains GUI components (Buttons, ...): Knows the Model to query data to display, Listens to changes in the data (model)
Controller Listens to events from the View (user input): Knows Model to control/invoke domain logic, Knows the View to refresh its content (e.g., deactivate Button)

4 Testing

Testing is the process of executing a program with the intent of finding errors. Impossible to find all errors through testing.

- Input / Output must be specified: exactly defined
- Creation and Testing must be separated: Creator should not test their own programs (can contain errors due to misunderstanding of problem statement)
- Tests must be written for invalid/unexpected conditions and for valid/expected conditions
- Testing is investment, automatic testing is better then on-the-fly testing/ throwaway testing
- Error Clusters: errors tend to come in clusters (more errors in one same place)

4.1 Test Double (Mock Testing in Java)

https://www.tutorialspoint.com/mockito/mockito_quick_guide.htm (A to be tested, depends on B):
A test double is used for B
– Test doubles provide the minimum necessary function for A to be tested
– One can also say that the test doubles simulate the other classes

5 Input-Output Streams

Byte Streams Byte oriented (8-Bit), Generic, i.e. binary data **Character Streams** Character oriented, Unicode based

5.1 Decorator Streams

Decorator Streams add additional features or behavior to an existing Stream (e.g. BufferdIOStream, DataIOStream, etc.)

5.2 Positioning

For Reading only (Navigation) For Writing No Positioning available (except for append)

5.3 Random Access File

RAF provides its own read and write operations

5.4 Serialization / Deserialization

Serialization*): Converting/saving objects into a Byte-Stream https://www.baeldung.com/java-serialization Deserialization*): Recovering/reading objects from a Byte-Stream
Types may be declared or inferred Must implement the Serializeable interface, not operable between other languages.

6 Resource Files

Classloader, Class Object, Java Properties (key, value)

7 Logging

import java.util.logging.*
Log Handler for Console / File / etc.
Log Level Config

8 Functional Interface

interface that has just one abstract method + methods of object (default, static methods are allowed)
For each Functional Interface Lambda Expressions can be written (L-E can't stand without passendes Functional Interface)

9 Functional Composition

Combine Predicate Functions: default Values: default IntPredicate negate() (Returns predicate negated)
Combine Functions: DoubleUnaryOperator addOne = x->x+1;
Optional: instead of null, better (final, immutable), can be null!

10 Procedural / Object Oriented / Functional

Procedural: y=f(x)
Object Oriented: y = object.f(x)
Functional: y = O(f,x) resp. y = x.O(f) → read as Äpply function f to values of x"

10.1 Functional Programming

For Functional Programming to Work (incl. nesting, chaining (= pipeling)): No global variables, no Mutex Objects modifying, Functions return no Input Values, Using Immutable Objects

11 Stream

Allows code to be written in declarative (not impearative) style

- A stream represents a sequence of data elements
- Streams can represent an infinite number of data elements
- Does not permanently store data elements (consumes data elements from a data source (stream, collection, array, file, . . .), deploys results to a data target (stream, variable, collection, array, file, . . .))
- Operations can be chained
- Stream has to be recreated for each run
- Stream Operations are processing each element of the stream (using internal iteration) and generate a new stream containing the sequence of resulting data elements resp. the final result (value, collection, array,...)
- The processing of the pipeline is optimized (lazy-evaluation, short-circuiting and merging of operations)