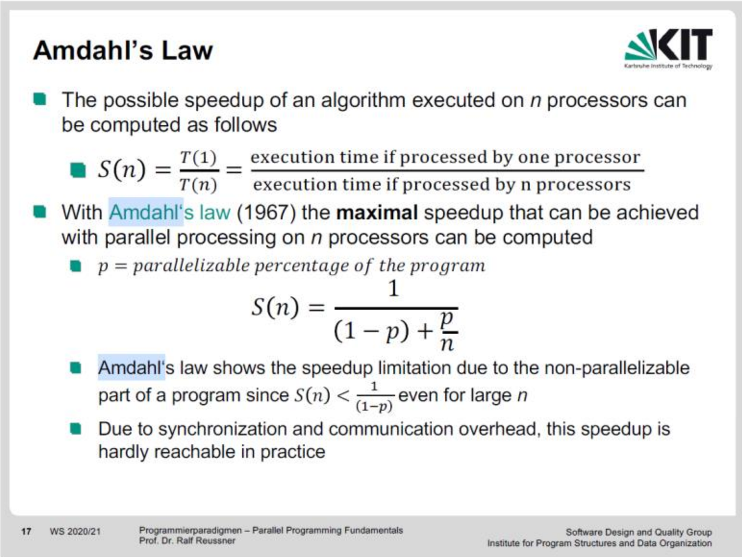
Java Parallelprogrammierung

Contents

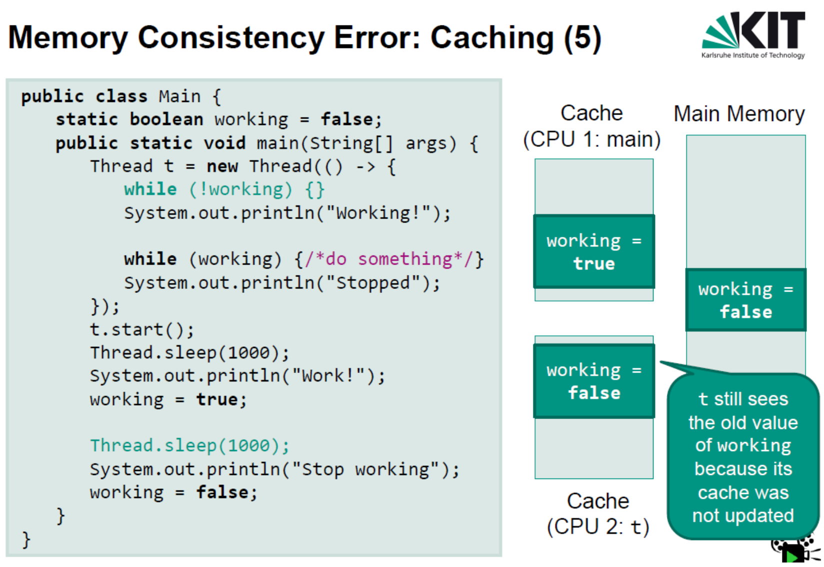
Parallelprogrammierung

Amdahlsches Gesetz



Memory Consistency Error

Caching



Reordering

Memory Consistency Error: Reordering (2)

```java
public class Main {
public static void main(String[] args) {
    State state = new State();
    new Thread(() -> {
       state.a = 1;
       state.b = 1;
       state.c = state.a + 1;
    }).start();
    new Thread(() -> {
       if (state.c == 2
             && state.b == 0) {
          System.out.println("Wrong");
       }
    }).start();
}

public class State {
    int a = 0; int b = 0; int c = 0;
}
```

- Possible reordering (no change in local execution result):

```java
state.a = 1;
state.c = state.a + 1;
state.b = 1;
```

Consequence:
Second thread can print "Wrong" if it is executed before `state.b = 1;`

Happens-before-Beziehung

Hilft, die Memory Consistency Errors zu vermeiden

Happens-before Relationship (1)

```java
public class Main {
    static int flag = 0;
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> flag = 1);
        Thread t2 = new Thread(() -> System.out.println(flag));
        t1.start();
        t2.start();
    }
}
```

- Java defines the happens-before relationship to avoid such errors
  - If two statements *s1* and *s2* have a happens-before relationship, Java guarantees that a potential write in *s1* is visible to *s2*
  - The happens-before relation is transitive, so if *s1* happens before *s2* and *s2* before *s3*, there is also a happens-before relation between *s1* and *s3*
  - In the above example, there is no happens-before relationship between the statements in `t1` and `t2`

Happens-before Relationship (2)

- Each statement has a happens-before relationship to every statement in the thread that comes later in the program's order
- There are several statements that introduce a happens-before relationship:
  - `Thread.start`: all statements executed before starting the thread (within the starting thread) have a happens-before relationship to statements in the new thread
  - `Thread.join`: all statements in the terminated thread have a happens-before relationship to the statements following the join
  - `synchronized`: all statements in a synchronized block have a happens-before relationship to all statements in a subsequent execution of a synchronized block using the same monitor
  - We will see further statements in the following
- **Again**: If there is no happens-before relationship between two statements, the second one may not see the result of the first one!

Schlüsselwort volatile

Volatile Variable wird immer in RAM gespeichert und nicht in Cache
Keine Reordering für volatile
Hilft, die Memory Consistency Error wegen Caching (Variable in RAM) und Reordering (happens-before-Beziehung) zu vermeiden

The volatile-Keyword

- `volatile` ensures that changes to variables are immediately visible to all threads / processors
  - `volatile` establishes a happens-before relationship: a write to a volatile variable happens-before every subsequent read to that variable
  - This means that all writes to (potentially different) variables before writing a volatile variable are visible to all reads of that variables after reading the volatile variable, because statements within a thread have a happens-before relationship in their program order
  - Values are not locally cached in a CPU cache (every read/write directly back to main memory)
  - Compiler/Processor optimizations are disabled: instruction reordering is not possible for the volatile variable

Happens-before ⇏ happens before

- A happens-before relation does *not* mean that one statement is actually executed before another
- No reordering for statements with inter-thread happens-before relations (e.g. writes to `volatile` variables), but possible reordering for other statements if behavior is not changed
- E.g.: Writes to `state.a` and `state.b` can still be reordered

```
public class Main {
    public static void main(String[] args) {
        State state = new State();
        new Thread(() -> {
            state.a = 1;
            state.b = 1;
            state.c = state.a + 1;
        }).start;
        new Thread(() -> {
            if(state.c == 2 && state.b == 0) {
                System.out.println("Wrong");
            }
        }).start();
    }
}

public class state {
    int a = 0; int b = 0;
    volatile int c = 0;
}
```

happens-before

can be reordered

Executor

Executor ist ein Woker von ExecutorService

Executor

- Executors abstract from thread creation
  - Simple implementations only start a thread
  - Other implementations, for example, reuse already created threads
- Java defines three executor interfaces in `java.util.concurrent`
- The most generic interface is `Executor`
  - A simple interface supporting the execution of tasks
  - Provides an `execute` method that accepts a `Runnable`

```
void execute(Runnable runnable)
```

ExecutorService

ExecutorService

- The most important interface is `ExecutorService`:
  - A subinterface of `Executor`
  - Provides further lifecycle management logic
- The class `Executors` (not to confuse with the `Executor` interface) provides convenient factory methods for creating an `ExecutorService`
  - `newSingleThreadExecutor()` creates an `Executor` using a single thread
  - `newFixedThreadPool(int)` creates a thread pool with reused threads of fixed size
  - `newCachedThreadPool()` creates a thread pool with reused threads of dynamic size

submit

Future <T> submit(task):

submit

`<T> Future<T> submit(Callable<T> task)`

Submits a value-returning task for execution and returns a Future representing the pending results of the task. The Future's `get` method will return the task's result upon successful completion.

If you would like to immediately block waiting for a task, you can use constructions of the form `result = exec.submit(aCallable).get();`

Note: The `Executors` class includes a set of methods that can convert some other common closure-like objects, for example, `PrivilegedAction` to `Callable` form so they can be submitted.

Parameters:

`task` - the task to submit

Returns:

a Future representing pending completion of the task

Throws:

`RejectedExecutionException` - if the task cannot be scheduled for execution

`NullPointerException` - if the task is null

task == lambda Funktion

Verwendung mit Lambda:

```
service.submit(() -> calculateX(elementIdx));
```

gibt Future<Integer> zurück, der später mit future.get() in einer Integer Variable geschrieben sein kann.
calculateX(elementIdx) ist eine Methode, die ein Integer zurückgibt.

```
Fututre<Integer> future = service.submit(() -> calculateX(elementIdx));
...
Integer result = future.get();
```

Futures

Futures: Representation of Results

- A Future<V> represents the (future) result of an asynchronous computation (i.e. Callable)
- The computation can either be (not yet) finished or cancelled
- Results can only be acquired when the computation is finished
- ExecutorService provides an additional submit() method, which expects a Runnable, but also a Callable in an overloaded version

```
<T> Future<T> submit(Callable<T> callable)
```

- submit() returns a Future, which represents the (future) result of the provided Callable

```
ExecutorService executorService = Executors.newCachedThreadPool();
for (int i = 0; i < 10; i++) {
    final int currentValue = i;
    Callable<Integer> myCallable = () -> {return currentValue;};
    Future<Integer> myFuture = executorService.submit(myCallable);
}
```

get() blockiert Thread

Futures: Retrieving Results

- The result of the Callable can be retrieved from the Future using its get() method

```
ExecutorService executorService = Executors.newCachedThreadPool();
List<Future<Integer>> futures = new ArrayList<Future<Integer>>();
for (int i = 0; i < 10; i++) {
    final int currentValue = i;
    Callable<Integer> myCallable = () -> {return currentValue;};
    futures.add(executorService.submit(myCallable));
}
for (Future<Integer> future : futures) {
    try {
        Integer result = future.get();
        System.out.println(result);
    } catch (ExecutionException ex) {}
}
executorService.shutdown();
```

Blocks until the Thread finished and the Future contains the value get(30, TimeUnit.SECONDS) -> warte max 30 Sekunden auf eine Antwort. Falls keine -> TimeoutException

Futures: Waiting for Results

- The `get()` method of `Future` blocks until the result is available
- A `Future` provides further methods for waiting for the completion of a submitted `Callable`:
  - `isDone()`: Returns whether the task finished
  - `get(int timeout, TimeUnit unit)`: Allows to wait for a specified amount of time

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        return 123;
    }
    catch (InterruptedException e) { }
});
future.get(1, TimeUnit.SECONDS);
```

- The above example will result in a `TimeoutException`

Berechnung mit ExecutorService und Futures

Vorgehensweise

1. Zuerst mus man ein ExecutorService erstellen und Anzahl von Executors einstellen.

```
ExecutorService service = Executors.newFixedThreadPool(amountThreads);
```

oder

```
ExecutorService executor = Executors.newCachedThreadPool();
(stellt amountThreads automatisch an)
```

2. Dann wird eine Liste von Futures erstellt

```
List<Future<Integer>> futures = new ArrayList<Future<Integer>>();
```

3. Iterire über Eingabedaten und fülle die Liste von Futures.
Wichtig: Anzahl der Iterationen (range von i) == Anzahl der Threads

```
for (int i = 0; i < input.size(); i += i) {
  final int elemeintIdx = i;
  futures.add(service.submit(() -> calculateX(elementIdx)));
}
```

4. Dann iteriere über liste von Futures und berechne das Programmergebnis:

```
int count = 0;
for(Future<Integer> future: futures) {
  count += future.get();
}
```

5. Schalte service ab

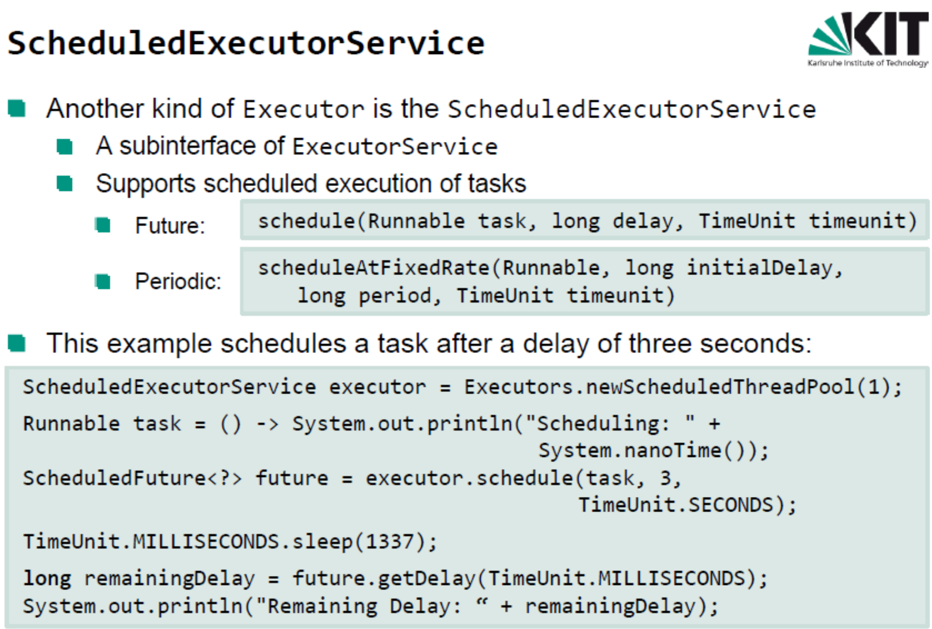
```
service.shutdown()
```

Complex Case

```
final int target = Integer.parseInt(args[0]);
final int amountThreads = Integer.parseInt(args[1]);
ExecutorService service = Executors.newFixedThreadPool(amountThreads);
for (int i = SEARCH_BEGIN; i < target; i += BLOCK_SIZE) {
  final int from = i;
  final int until = i + BLOCK_SIZE;
  futures.add(service.submit(() -> countPrimes(from, until)));
}
int count = 0;
  for(Future<Integer> future: futures) {
  count += future.get();
}
service.shutdown()
```

ScheduledExecutorService

ExecutorService mit Scheduling-Mechanismus



CompletableFutures

futureCount.get() führt zusätzlich die Funktion aus supplyAsync() aus (? - nicht ganz klar, abe aber eher unwichtig)

CompletableFutures (1)

- Drawback of `Futures`: The caller can query a result, but not register a callback
- Java 5 provided the `ExecutorCompletionService` for that purpose
- In Java 8, the `CompletableFuture` was introduced, which provides the `supplyAsync` method, to which the asynchronous task can be passed:

```
CompletableFuture<Integer> futureCount = CompletableFuture.supplyAsync(
    () -> {
        try {
            // simulate long running task
            Thread.sleep(5000);
        } catch (InterruptedException ex) { }
        return 20;
    }
);
int count = futureCount.get();
```

Aufgaben

Amdahlsches Gesetz aus Ü9

Aufgabe: geg. Thread Pool. Jeder Leser und Schreiber werden repräsentiert durch einen Thread. 90% Threads sind Leser, 10% sind Schreiber.
Leser sind nicht blockierend, benötigt 2 Sekunden
Schreiber ist blockierend für alle Schreiber und Lese, benötigt 3 Sekunden.
Lösung:

P: Anteil eines Programms, der parallelisiert werden kann
N: Anzahl der Prozessoren

$$S(P) = \frac{1}{(1-P)+\frac{P}{N}}$$

$$P = \frac{(2*0.9)}{2*0.9+3*0.1} \approx 0.86$$

Mit $N = 4$ resultiert dies in: $S(P) = \frac{1}{(1-0.86)+\frac{0.86}{4}} \approx 2.82$

P = P(Leser)/P(Gesamt)

Happens-Before SS20

(a) Erklären Sie den Begriff der *Happens-before-Beziehung*. Nennen Sie außerdem einen Ausdruck oder ein Keyword aus der Vorlesung, welches eine *Happens-before-Beziehung* herstellt. [2 Punkte]

Beispiellösung: Eine *Happens-before-Beziehung* zwischen zwei Statements `s1` und `s2` besagt, dass ein mögliches Schreiben in `s1` immer für `s2` sichtbar ist. Statements die eine *Happens-before-Beziehung* herstellen sind z.B.

- `Thread.start`
- `Thread.join`
- `synchronized`
- `volatile`

Es gibt jedoch noch weitere Ausdrücke. Diese sind u.a. auf der Oracle Dokumentationswebseite des Pakets java.util.concurrent zu finden.

(b) In der gegebenen Implementierung kann es zu Speicherinkonsistenzen kommen, wodurch das beschriebene fehlerhafte Verhalten auftritt. Beschreiben Sie, wie Sie die gegebene Implementierung verändern müssen, damit das Problem der Speicherinkonsistenzen nicht mehr auftritt und sich hierdurch das Programm korrekt verhält. Begründen Sie unter Verwendung der *Happens-before-Beziehung* und mit Bezug auf die gegebene Implementierung, warum Ihre Änderung das Problem behebt. [3 Punkte]

Beispiellösung: Das Problem kann durch die Deklaration von `numberAvailable` als `volatile` gelöst werden. `Volatile` erzeugt eine *Happens-before-Beziehung* zwischen jedem Schreiben der deklarierten Variable, hier `numberAvailable`, und jedem folgenden Lesen über alle Threads. Hierdurch wird garantiert, dass bei jedem Lesen von `numberAvailable` der aktuellste Wert (die letzte Modifikation bzw. das letzte Schreiben) sichtbar ist. Dies kann z.B. durch das garantierte Schreiben des Inhalts der modifizierten Variable in den Hauptspeicher und dem garantierten Lesen der Variable aus dem Hauptspeicher realisiert werden. Hierdurch wird von `consumer` und `producer` immer die Änderung von `numberAvailable` gesehen wodurch der gegenseitige Fortschritt sichergestellt ist. *Hinweis:* Die Variable `number` verursacht das Problem nicht, da der Programmfortschritt durch deren Lesen und Schreiben nicht beeinflusst wird.

Data- vs Task- Parallelismus

Two approaches:

1. Task parallelism: functional decomposition
   - Define tasks that can be executed in parallel
   - Tasks should be as independent as possible
2. Data parallelism: data decomposition
   - Partition the data on which the same operation is executed in parallel
   - Tasks should be able to work on the partitions as indepdently as possible

Task: WebSerber

Data: Searching for max number in Array

Wird für obige parallele Implementierung (Teilaufgabe (a)) für `calculateMax` Task- oder Daten-Parallelismus verwendet? Begründen Sie Ihre Antwort kurz. [2 Punkte]

Beispiellösung: Es wird Datenparallelismus verwendet. Die Daten werden in der Vorverarbeitung in unabhängige Blöcke aufgeteilt. Jedem Task der in den `ExecutorService` per `submit` hinzugefügt wird, erhält einen Block und führt darauf die gleiche Methode (`findMax`) parallel aus.