Karlsruhe Institute of Technology

Programmierparadigmen

Übung 3 (Reussner) – 16.01.2026

Lars König, M.Sc.

Parallelprogrammierung in Java

Mögliche Probleme

- Race Conditions
- Deadlocks
- Inkonsistente Daten (z.B. durch Reorderings oder Memory Inconsistencies)

Lösungsstrategien

- Atomare Operationen
- Synchronisation von kritischen Abschnitten
- Vorbeugen von Deadlocks
- Herstellen von happens-before-Beziehungen

Parallelprogrammierung in Java

Fortgeschrittene Strategien

- Synchronisationskonstrukte (Barrieren, Semaphoren, …)
- Futures
- Streams
- Aktoren

Parallelprogrammierung in Java

Mögliche Probleme

- Race Conditions
- Deadlocks
- Inkonsistente Daten (z.B. durch Reorderings oder Memory Inconsistencies)

Lösungsstrategien

- Atomare Operationen
- Synchronisation von kritischen Abschnitten
- Vorbeugen von Deadlocks
- Herstellen von happens-before-Beziehungen

Reordering

- in Java kann die Ausführungsreihenfolge verändert werden

Thread 1:

```
result.value = 42;
result.isPresent = true;
```

Thread 2:

```
if (result.isPresent) {
     doSomething(result.value);
}
```

Memory Consistency

- die Variablen eines Threads liegen im Cache der CPU, auf der er ausgeführt wird
- Änderungen an diesen Variablen werden verzögert in anderen Caches aktualisiert

Thread 1:

```
ping = true;
while (ping == true) {}
```

Thread 2:

```
while (ping == false) {}
ping = false;
```

Happens-Before

- Eine happens-before-Beziehung garantiert, dass das Ergebnis einer Operation sichtbar ist
- Happens-before-Beziehungen sind transitiv

$\Rightarrow$ Einschränkungen für reordering und Garantien für Speicherkonsistenz

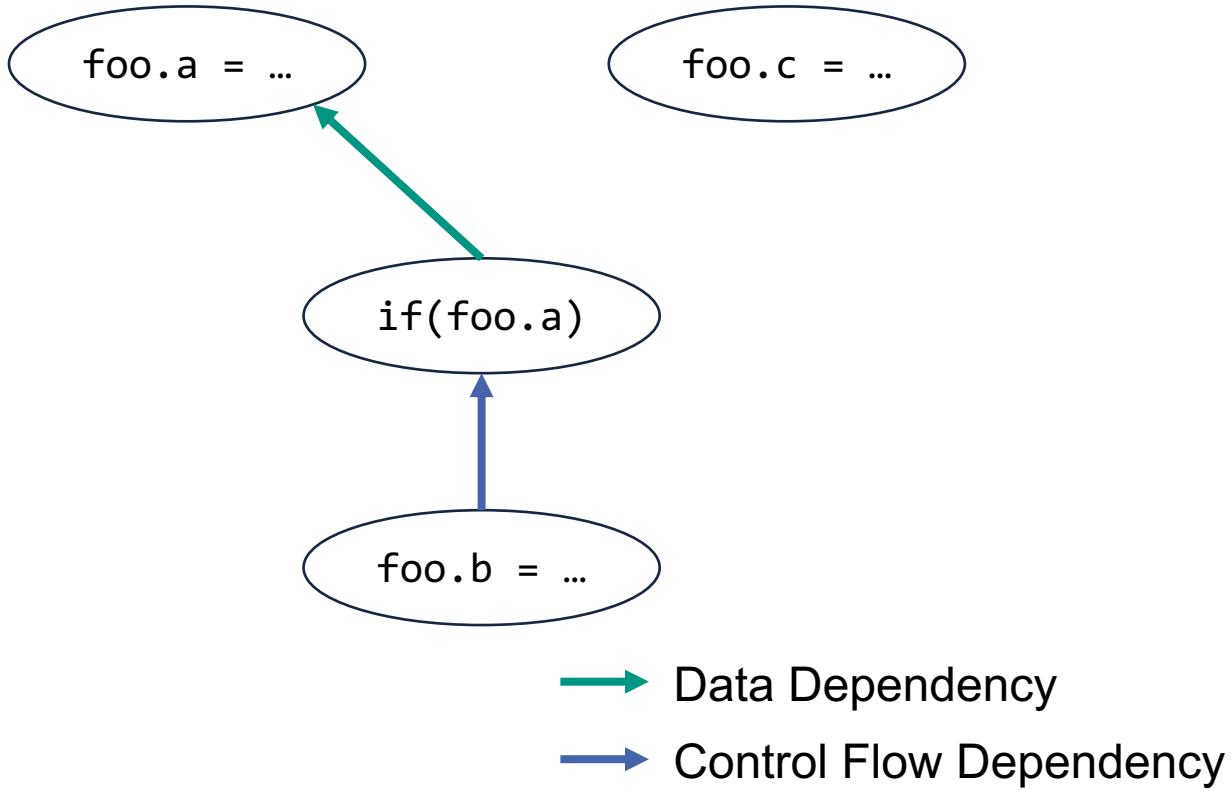
Zwischen welchen Anweisungen bestehen in Java happens-before-Beziehungen?

Hintergrund

Program Order:

```
foo.a = …;
foo.c = 5.2;
if (foo.a) {
      foo.b = 42;
}
```

Execution Order:



Hintergrund

Program Order:

```
foo.a = …;
foo.c = 5.2;
if (foo.a) {
     foo.b = 42;
}
```

Execution Order (Beispiel):

```
foo.c = 5.2;
foo.a = …;
if (foo.a) {
     foo.b = 42;
}
```

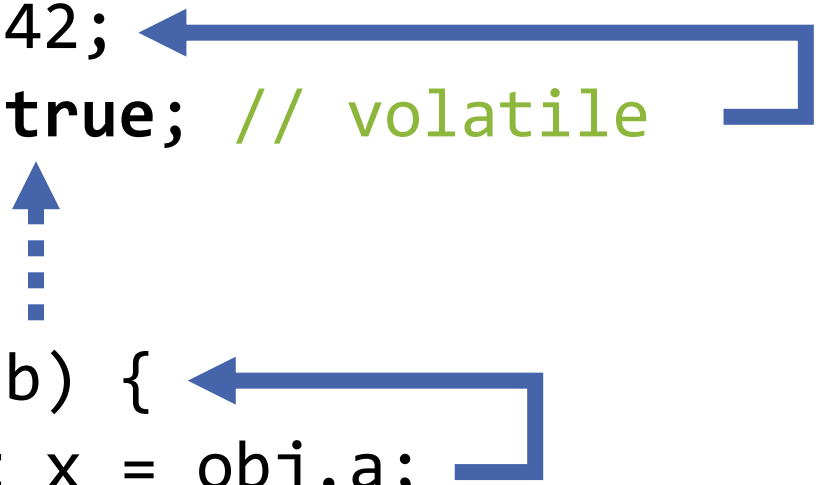
Happens-before

Thread 1:

```
obj.a = 42;
obj.b = true; // volatile
```

Thread 2:

```
if (obj.b) {
    int x = obj.a;
}
```



Annahme: Der Code in Thread 1 wird zuerst ausgeführt.

Aufgabe: Happens-before

```
1  prepareWebpage();
2  Future<Document> futureTicket =
3      executor.submit(() -> {
4          PersonalData data = getPersonalData();
5          synchronized (reservation) {
6              return generateTicketConfirmation(
                   data, reservation);
7          } });
8  Future<Document> futureSeats =
9      executor.submit(() -> {
10         PersonalData data = getPersonalData();
11         synchronized (reservation) {
12             return generateSeatsConfirmation(
                   data, reservation);
13         } });
14 updateWebpage();
15 Document ticket = futureTicket.get();
16 startDownload(ticket);
```

- Ausgehend von dem Statement mit dem weißen Punkt, zu welchen anderen Statements besteht eine happens-before-Beziehung?

Annahme: Der Thread der zuerst übergeben wird, wird vollständig ausgeführt, bevor mit der Ausführung des anderen Threads begonnen wird.

Aufgabe: Happens-before

```
1  prepareWebpage();
2  Future<Document> futureTicket =
3      executor.submit(() -> {
4          PersonalData data = getPersonalData();
5          synchronized (reservation) {
6              return generateTicketConfirmation(
                   data, reservation);
7          } });
8  Future<Document> futureSeats =
9      executor.submit(() -> {
10         PersonalData data = getPersonalData();
11         synchronized (reservation) {
12             return generateSeatsConfirmation(
                   data, reservation);
13         } });
14 updateWebpage();
15 Document ticket = futureTicket.get();
16 startDownload(ticket);
```

- ● ja, vor `Thread.start`
- ● ja, im gleichen Thread
- ○
- ● nein
- ● ja, `synchronized`
- ● nein
- ● ja, nach `Thread.join`

Future

- repräsentiert das (ggfs. noch nicht vorhandene) Ergebnis einer asynchronen Berechnung
- das Ergebnis steht erst zur Verfügung, wenn die Berechnung abgeschlossen ist

```
Future f = …;
doSomething();
f.get();
doSomethingElse();
```

Executors

- verwalten die Erstellung von Threads für die Ausführung von Tasks
- `execute(..)` führt einen Task mit dem nächsten verfügbaren Thread aus
- auch wiederholtes Scheduling von Tasks möglich

Beispiele:

- `SingleThreadExecutor()`
  Executor mit einem einzigen Thread
- `FixedThreadPool(int)`
  Executor mit einer festen Anzahl an Threads
- `CachedThreadPool()`
  Executor mit einer dynamischen Anzahl an Threads

Future

- Executor geben Futures zurück, die das Ergebnis der Ausführung der übergebenen Tasks repräsentieren

```
Future<Integer> future = executor.submit(longTask);
Integer a = otherLongTask();
Integer b = future.get();
return a + b;
```

CompletableFuture

- `CompletableFuture` erlaubt die verkettete, asynchrone Ausführung von Berechnung

```
CompletableFuture<Integer> result = CompletableFuture
    .supplyAsync(() -> retrieveData())
    .thenApplyAsync(data -> aggregate(data));

int result = result.get();
```

Übungsblatt 2 – Aufgabe 5.1

Streams

- stellen Listenoperationen zur Verfügung
  - `filter`
  - `map`
  - `collect`
  - `findAny`, `findFirst`
  - `min`, `max`
  - …
- benötigen keinen zusätzlichen Speicherplatz sondern berechnen das Ergebnis on-demand mit lazy evaluation
- Parallelausführung für manche Operationen möglich

Übungsblatt 2 – Aufgabe 5.2

Korrektheit von Software

- Korrektheit ist relativ
- gilt bezüglich einer Spezifikation
- oft implizit

- Validierung: mit Beispieleingaben testen
- Verifikation: Korrektheit für alle Eingaben *beweisen*

Software Contracts

- Vertrag zwischen Aufrufer und Aufgerufenem
- legt keine konkrete Implementierung fest

- Vorbedingung: muss vom Aufrufer erfüllt werden
- Methode: ausgeführter Code
- Nachbedingung: muss von der Methode erfüllt werden, *falls die Vorbedingung gilt*

- Klasseninvariante: muss in jedem *sichtbaren* Zustand erfüllt sein

Aufgabe 1.1

Java Modelling Language (JML)

- Definition von Verträgen für Java Klassen/Methoden
- als Dokumentation oder zur Überprüfung

Anwendungsfälle:

- statische Überprüfung des Quelltexts
- Überprüfung während der Ausführung
- Generierung von Testfällen

Syntax

`requires`	definiert Vorbedingung
`ensures`	definiert Nachbedingung
`\old`	gibt den Wert eines Ausdrucks *vor* Ausführung der Methode an
`\result`	steht für den Rückgabewert der Methode

```
/*@ requires counter > 0 ;
  @ ensures counter == \old{counter} - 1;
  @ ensures \result == counter;
  @*/
int dec() { ... }
```

Syntax

`invariant` definiert Klasseninvariante
`pure` gibt an, dass eine Methode keine Seiteneffekte hat

```
class Counter {
      //@ invariant counter >= 0 ;
      private /*@ spec_public @*/ int counter = 0;

      /*@ ensures \result == counter; @*/
      /*@ pure @*/ int getCounter() { ... }
}
```

Java Assertions

- Überprüfen eine Bedingung zur Laufzeit
- sollten nur zum Debuggen benutzt werden
- nur Java-Code erlaubt

```
void compute() {
     assert this.data != null;

     ...
}
```

Java Assertions

1. Sammeln Sie Vor- und Nachteile von Assertions gegenüber JML.

2. Eignen sich Assertions zur Umsetzung von Tolerant Precondition Design?

Zusammenfassung

- Grundlagen der Parallelprogrammierung
- Message Passing Interface (MPI)
  - Direkte und kollektive Operationen
  - Unterschiedliche Kommunikations- und Aufrufmodi
- Parallelprogrammierung in Java
  - Probleme und Lösungsstrategien
  - Fortgeschrittene Konzepte
- Design by Contract