UNIVERSITÄT
LEIPZIG

Einführung in die Objekt-Orientierte Modellierung und Programmierung

Wintersemester 2025/2026

Dirk Zeckzer

Institut für Informatik

Teil XXII

Parallelität

Threads

Ablauf einer Berechnung
bei einer rechnenden Einheit:

- 3 Programme
  - rot
  - grün
  - blau
- Jedes Programm bekommt
  - **fair**
  - kurze

  Rechenzeit
- Für Anwender sieht es so aus, als würden alle drei Programme **gleichzeitig** ablaufen.

Threads

- Zunächst: CPU's mit einer rechnenden Einheit
- Beschleunigung einer einzelnen rechnenden Einheit nicht mehr möglich (physikalische Grenzen)
- $\rightarrow$ CPU's mit **mehreren** rechnenden Einheiten

Threads

- Programme
  - Betriebssystemebene:
    1 Programm $\leftrightarrow$ 1 **Prozess**
  - Prinzipiell:
    n rechnende Einheiten $\rightarrow$
    n Prozesse (Programme)
    gleichzeitig (**parallel**)
- Threads
  - **Thread**: Teil eines Programmes
  - Prinzipiell:
    n rechnende Einheiten $\rightarrow$
    n Threads **gleichzeitig** (**parallel**)

- (Tatsächlich ist die Situation wesentlich komplexer)

Beispiel: Summe von Listenelementen

- Aufgabe
  - Eingabe: Zwei Listen mit ganzen Zahlen
  - Ausgabe: Eine Liste mit den Summen der Zahlen
- Beispiel
  - Liste 1: $\{1, 4, 9, 16, 25, \ldots, 100^2\}$
  - Liste 2: $\{1, 8, 27, 64, 125, \ldots, 100^3\}$
  - Ergebnis: $\{2, 12, 36, 80, 150, \ldots, 1010000\}$

Beispiel: Summe von Listenelementen

```
1  package summierer;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.function.Function;
6  import summierer.parallelThread.SumOfListsParallelThread;
7  import summierer.parallelThreadPool.SumOfListsParallelThreadPool;
8  import summierer.sequential.SumOfListsSequential;
```

Beispiel: Summe von Listenelementen

```
1   public class SumOfLists {
2
3     private static final int MAXIMAL_NUMBER = 100;
4
5     public static void main(String[] args) {
6       // Erzeuge die beiden Listen
7       final List<Integer> listOfSquares
8         = createList(MAXIMAL_NUMBER, i -> (i * i));
9       final List<Integer> listOfPowerToThree
10        = createList(MAXIMAL_NUMBER, i -> (i * i * i));
11
12      // Berechne die Summe der beiden Listen sequentiell: Methode
13      final List<Integer> summenSequentialMethod
14        = computeSumSequentialMethod(listOfSquares, listOfPowerToThree);
15      printElements(summenSequentialMethod,
16                    "sequentiell Methode");
17    }
```

Beispiel: Summe von Listenelementen

```
19      private static List<Integer> createList(
20        final int n,
21        final Function<Integer,Integer> funktion
22      ) {
23        final List<Integer> result = new ArrayList<>();
24
25        for (int i = 1;
26             i <=  n;
27             ++i) {
28          Integer number = funktion.apply(i);
29          result.add(number);
30        }
31
32        return result;
33      }
```

```
35      private static void printElements(
36        final List<Integer> listOfSums,
37        final String method
38      ) {
39        System.out.println("Liste der Summen (" + method + ")");
40
41        for (Integer sum
42             : listOfSums) {
43          System.out.print(sum + " , ");
44        }
45        System.out.println();
46      }
```

Beispiel: Summe von Listenelementen

```
48      private static List<Integer> computeSumSequentialMethod(
49        final List<Integer> summand1,
50        final List<Integer> summand2
51      ) {
52        final List<Integer> summen = new ArrayList<>();
53
54        if (summand1 == null) {
55          return summen;
56        }
57
58        if (summand2 == null) {
59          return summen;
60        }
61
62        for (int i = 0;
63             i < summand1.size() && i < summand2.size();
64             ++i) {
65          summen.add(summand1.get(i) + summand2.get(i));
66        }
67
68        return summen;
69      }
70    }
```

Beispiel: Summe von Listenelementen

```java
public class SumOfListsSequential {

  // Eingaben
  private final List<Integer> summand1;
  private final List<Integer> summand2;

  // Ausgabe
  private final List<Integer> summen;

  public SumOfListsSequential(
    final List<Integer> summand1,
    final List<Integer> summand2
  ) {
    // Initialize input
    this.summand1 = summand1;
    this.summand2 = summand2;

    // Initialize result
    summen = new ArrayList<>();
  }

  public void computeSum() {
    summen.clear();

    if (summand1 == null) {
      return;
    }

    if (summand2 == null) {
      return;
    }

    for (int i = 0;
         i < summand1.size()
         && i < summand2.size();
         ++i) {
      summen.add(summand1.get(i)
                 + summand2.get(i));
    }
  }

  public List<Integer> getSummen() {
    return summen;
  }
}
```

Threads

- Thread starten (Methode: start)
- auf das Ende des Threads warten (Methode: join)

main thread	start threads	...	wait for join	...
	`t1.start()`	...	`t1.join()`	
	`t2.start()`	...	`t2.join()`	
	`t3.start()`	...	`t3.join()`	
	`t4.start()`	...	`t4.join()`	
	`t5.start()`	...	`t5.join()`	

Beispiel

- Aufgabe
  - Eingabe: Zwei Listen mit ganzen Zahlen
  - Ausgabe: Eine Liste mit den Summen der Zahlen
- Beispiel
  - Liste 1: $\{1, 4, 9, 16, 25, \ldots, 100^2\}$
  - Liste 2: $\{1, 8, 27, 64, 125, \ldots, 100^3\}$
  - Ergebnis: $\{2, 12, 36, 80, 150, \ldots, 1010000\}$

Beispiel

Lösung mit 2 Threads

- Idee: Verteile die Arbeit gleichmäßig auf beide Threads
- Jeder Thread übernimmt die Hälfte der Additionen
- Unterteile beide Listen in vordere und hintere Hälfte:
  - $[0; \mathit{liste.size}()/2 - 1]$
  - $[\mathit{liste.size}()/2; \mathit{liste.size}() - 1]$
- Erzeuge je einen Thread mit jeweils
  - der vorderen Hälfte
  - der hinteren Hälfte

  der Listen 1 und 2
- Starte beide Threads
- Warte auf das Ende beider Threads
- Hole die Ergebnisse aus beiden Threads
- Hänge die Ergebnisse aneinander (Combine)

Beispiel: Summe von Listenelementen

```java
public class SumOfListsParallelThread {

  private SumOfListsParallelThread() {
  }

  public static List<Integer> computeSum(
    final List<Integer> list1,
    final List<Integer> list2
  ) {
    // Erzeuge Runnables (Split)
    final SummingRunnable summingRunnable1
      = new SummingRunnable(
          list1.subList(0,
                        list1.size() / 2),
          list2.subList(0,
                        list2.size() / 2));

    final SummingRunnable summingRunnable2
      = new SummingRunnable(
          list1.subList(list1.size() / 2,
                        list1.size()),
          list2.subList(list2.size() / 2,
                        list2.size()));

    // Erzeuge Threads
    final Thread summingThread1
      = new Thread(summingRunnable1);
    final Thread summingThread2
      = new Thread(summingRunnable2);

    // Starte Threads
    summingThread1.start();
    summingThread2.start();

    // Warte auf das Ende der Threads
    try {
      summingThread1.join();
      summingThread2.join();
    } catch (InterruptedException iEx) {
    }

    // Bilde eine Liste mit den Summen
    // (Combine)
    final List<Integer> listOfSums
      = summingRunnable1.getSummen();
    listOfSums.addAll(
      summingRunnable2.getSummen());

    // return result
    return listOfSums;
  }
}
```

Beispiel: Summe von Listenelementen

```java
public class SummingRunnable
  implements Runnable {

  // Eingaben
  private final List<Integer> summand1;
  private final List<Integer> summand2;

  // Ausgabe
  private final List<Integer> summen;

  public SummingRunnable(
    final List<Integer> summand1,
    final List<Integer> summand2
  ) {
    this.summand1 = summand1;
    this.summand2 = summand2;
    summen = new ArrayList<>();
  }
}
```

```java

  @Override
  public void run() {
    summen.clear();

    if (summand1 == null) {
      return;
    }

    if (summand2 == null) {
      return;
    }

    for (int i = 0;
         i < summand1.size()
         && i < summand2.size();
         ++i) {
      summen.add(summand1.get(i)
                 + summand2.get(i));
    }
  }

  public List<Integer> getSummen() {
    return summen;
  }
}
```

Beispiel: Summe von Listenelementen

```
1   // Erzeuge Threads
2
3
4
5   SummingRunnable t1
6     = new SummingRunnable(
7         list1.subList(0,
8                       list1.size() / 2),
9         list2.subList(0,
10                      list2.size() / 2)
11      );
12
13  SummingRunnable t2
14    = new SummingRunnable(
15        list1.subList(list1.size() / 2,
16                      list1.size()),
17        list2.subList(list2.size() / 2,
18                      list2.size())
19      );
```

```
1   // Erzeuge die beiden Threads
2   int size = Integer.min(list1.size(),
3                          list2.size());
4
5   SummingRunnable t1
6     = new SummingRunnable(
7         list1,
8         list2,
9         0,
10        size / 2
11      );
12
13  SummingRunnable t2
14    = new SummingRunnable(
15        list1,
16        list2,
17        size / 2,
18        size
19      );
```

Beispiel: Summe von Listenelementen

```java
public class SummingRunnable
  implements Runnable {

  // Eingaben
  private final List<Integer> summand1;
  private final List<Integer> summand2;
  private final int start;
  private final int end;

  // Ausgabe
  private final List<Integer> summen;

  public SummingRunnable(
    final List<Integer> summand1,
    final List<Integer> summand2,
    final int start,
    final int end
  ) {
    this.summand1 = summand1;
    this.summand2 = summand2;
    this.start = start;
    this.end = end;
    summen = new ArrayList<>();
  }

  @Override
  public void run() {
    if (summand1 == null) {
      return;
    }

    if (summand2 == null) {
      return;
    }

    for (int i = start;
         i < end;
         ++i) {
      summen.add(
        summand1.get(i)
        + summand2.get(i)
      );
    }
  }

  public List<Integer> getSum() {
    return summen;
  }
}
```

java.util.concurrent

```java
@FunctionalInterface
public interface Runnable {
  void run();
}
```

```java
public class Thread
  implements Runnable {
  public Thread(
    ThreadGroup group,
    Runnable target,
    String name
  )

  ...
}
```

- `java.lang`
  - `Runnable`, `Thread`
  - Seit Java 1.0
  - Grundlegende Funktionalität
  - Keine Eingaben (Konstruktor)
  - Keine Rückgabe (über Getter)
  - Wahrscheinlicher Grund: Keine Generics
- `java.util.concurrent`
  - Seit Java 1.5
  - Erweiterte Funktionalität

java.util.concurrent

- Wunschliste
  - Thread-Pool (Wiederverwendung von Threads)
  - Komfortableres Warten auf Ende
  - Direkte Verwendung des Ergebnisses bei Rückgabe

- Thread-Pool
  - Erzeuge mehrere Threads
  - Speichere Sie im Pool
  - Falls Pool nicht leer
    - nimm einen Thread aus dem Pool
  - Sonst
    - warte bis Thread zurückgegeben wird
  - Gib Thread nach Beendigung der Aufgabe in den Pool zurück

Beispiel: Summe von Listenelementen

```
1   package summingInParallel.threadPool;
2
3   import java.util.ArrayList;
4   import java.util.List;
5   import java.util.concurrent.CompletableFuture;
6   import java.util.concurrent.ExecutionException;
7   import java.util.concurrent.ExecutorService;
8   import java.util.concurrent.Executors;
9   import summingInParallel.SummingInParallel;
10
11  public class SummingThreadPoolBased {
12
13      private static final int NUMBER_OF_THREADS = 6;
14  //    private static final int NUMBER_OF_TASKS = 32;
15      private static final int NUMBER_OF_TASKS = 6;
16
17      private SummingThreadPoolBased() {
18      }
```

Beispiel: Summe von Listenelementen

```
20      public static List<Integer> computeSum(
21        final List<Integer> list1,
22        final List<Integer> list2
23      ) {
24        // Create Thread Pool with fixed number of threads
25        final ExecutorService executorService
26          = Executors.newFixedThreadPool(NUMBER_OF_THREADS);
27
28        try {
29          // Erzeuge Liste der Aufgaben
30          final List<SummingTask> aufgaben
31            = createTasks(list1, list2);
32
33          // Erledige Aufgaben
34          final List<CompletableFuture<List<Integer>>> aufgabenErgebnisse
35            = scheduleTasks(executorService,
36                            aufgaben);
37
38          // Warte auf das Ende aller Aufgaben
39          aufgabenErgebnisse.forEach(CompletableFuture::join);
40
41          // Kombiniere die Ergebnisse aller Aufgaben
42          final List<Integer> result = combineResults(aufgabenErgebnisse);
43
44          return result;
45        } finally {
46          // Shutdown ThreadPool
47          executorService.shutdown();
48        }
49      }
```

Beispiel: Summe von Listenelementen

```
51      private static List<SummingTask> createTasks(
52          final List<Integer> list1,
53          final List<Integer> list2
54      ) {
55          final List<SummingTask> aufgaben = new ArrayList<>();
56
57          // Berechne Groesse der kleineren Liste
58          final int size = Integer.min(list1.size(),
59                                       list2.size());
60
61          // Erzeuge und starte Teilaufgaben
62          for (int i = 0;
63               i < NUMBER_OF_TASKS;
64               ++i) {
65              // Erzeuge Aufgabe
66              final SummingTask aufgabe = new SummingTask(
67                  list1.subList((i * size) / NUMBER_OF_TASKS,
68                                ((i + 1) * size) / NUMBER_OF_TASKS),
69                  list2.subList((i * size) / NUMBER_OF_TASKS,
70                                ((i + 1) * size) / NUMBER_OF_TASKS)
71              );
72              aufgaben.add(aufgabe);
73          }
74
75          return aufgaben;
76      }
```

Beispiel: Summe von Listenelementen

```
78      private static List<CompletableFuture<List<Integer>>> scheduleTasks(
79        final ExecutorService executorService,
80        final List<SummingTask> aufgaben
81      ) {
82        final List<CompletableFuture<List<Integer>>> aufgabenErgebnisse
83          = new ArrayList<>();
84
85        for (SummingTask aufgabe
86             : aufgaben) {
87
88          // Fuehre Aufgabe aus
89          final CompletableFuture<List<Integer>> aufgabeCompletableFuture
90              = CompletableFuture.supplyAsync(
91                  aufgabe::execute,
92                  executorService
93              );
94
95          // Fuege Aufgaben zur Liste der Aufgaben hinzu
96          aufgabenErgebnisse.add(aufgabeCompletableFuture);
97        }
98
99        return aufgabenErgebnisse;
100     }
```

Beispiel: Summe von Listenelementen

```
102     private static List<Integer> combineResults(
103       final List<CompletableFuture<List<Integer>>> aufgabenErgebnisse
104     ) {
105       // Kombiniere Ergebnisse (Summen)
106       final List<Integer> ergebnisse = new ArrayList<>();
107
108       // Fuege Ergebnisse der Aufgabe zur Liste der Ergebnisse (Summen) hinzu
109       for (CompletableFuture<List<Integer>> aufgabe
110            : aufgabenErgebnisse) {
111         try {
112           ergebnisse.addAll(aufgabe.get());
113         } catch (InterruptedException interruptedException) {
114         } catch (ExecutionException executionException) {
115         }
116       }
117
118       return ergebnisse;
119     }
120   }
```

Beispiel: Summe von Listenelementen

```java
public class SummingTask {

  // Eingaben
  private final List<Integer> summand1;
  private final List<Integer> summand2;

  // Ausgabe
  private final List<Integer> summen;

  public SummingTask(
    final List<Integer> summand1,
    final List<Integer> summand2
  ) {
    this.summand1 = summand1;
    this.summand2 = summand2;
    summen = new ArrayList<>();
  }

  public List<Integer> execute() {
    if (summand1 == null) {
      return summen;
    }

    if (summand2 == null) {
      return summen;
    }

    for (int i = 0;
         i < summand1.size()
         && i < summand2.size();
         ++i) {
      summen.add(summand1.get(i)
                 + summand2.get(i));
    }

    return summen;
  }
}
```

Sequentiell vs. Parallel

Wann lohnt sich Parallelisierung?

- Zusätzliche Planung
  - Wie viele Threads?
  - Wie können die Daten
    - gleichmäßig aufgeteilt
    - kombiniert

    werden?
  - Was tun
    - `InterruptedException` beim Warten auf die Threads?
    - `ExecutionException` beim Ausführen von Threads?
- Zusätzlicher Zeitbedarf
  - Setup
    - Unterteilung der Daten
    - Erzeugung der Threads
  - Combine
    - Holen der Ergebnisse
    - Kombinieren der Ergebnisse
- Weniger Zeitbedarf
  - Mehrere Berechnungen gleichzeitig
- Insgesamt

  $t_{setup} + t_{parallel} + t_{combine} < t_{sequentiell}$

Parallelisierung: Mögliche Fehlerquellen

Ursache	Name	Lösung
Dasselbe Element: Ein oder mehrere Threads lesen	keine Probleme	
Dasselbe Element: Ein oder mehrere Threads lesen ein Thread schreibt	Race Condition	Semaphore (synchronized)
Dasselbe Element: Mehrere Threads schreiben	Race Condition	Semaphore (synchronized)
Threads blockieren sich gegenseitig	Deadlock	Analyse (Petri-Netze, . . .)

Race Condition & Synchronized: Counting

```
1   public class CountingRunnable
2    implements Runnable {
3
4    // Eingabe
5    private final List<Integer> values;
6
7    // Ausgabe
8    private final Map<Integer, Integer> counts;
9
10   public CountingRunnable(
11    final List<Integer> values,
12    final Map<Integer, Integer> counts
13   ) {
14    this.values = values;
15    this.counts = counts;
16   }
```

```
18   @Override
19   public void run() {
20    for (Integer value : values) {
21     synchronized (counts) {
22      if (counts.containsKey(value)) {
23       int count = counts.get(value);
24       ++count;
25       counts.put(value, count);
26      } else {
27       counts.put(value, 1);
28      }
29     }
30    }
31   }
32  }
```

Race Condition & Synchronized: Counting

```
1     private static void testThread(
2       final List<Integer> values,
3       final Map<Integer, Integer> counts
4     ) {
5       final Thread counterThreads[]
6         = new Thread[10];
7
8       // Erzeuge Threads
9       for (int i = 0;
10           i < counterThreads.length;
11           ++i) {
12        counterThreads[i]
13          = new Thread(
14              new CountingRunnable(values,
15                                   counts));
16      }
```

```
18      // Starte Threads
19      for (int i = 0;
20           i < counterThreads.length;
21           ++i) {
22        counterThreads[i].start();
23      }
24
25      // Warte auf Threads
26      for (int i = 0;
27           i < counterThreads.length;
28           ++i) {
29        try {
30          counterThreads[i].join();
31        } catch (Throwable thr) {
32        }
33      }
34    }
```

Race Condition & Synchronized: Counting

```java
public class ThreadExample {

  public static void main(String[] args) {
    // Erzeuge und fuelle Liste
    final List<Integer> values
      = new ArrayList<>();
    for (int i = 1; i <= 50; ++i) {
      values.add(i);
    }

    // Erzeuge Ergebnis Collection
    Map<Integer, Integer> counts;
    counts = new HashMap<>();

    // Name der Methode ausgeben
    System.out.println("testThread HashMap");

    // Zaehle in parallel
    testThread(values, counts);

    // gib das Ergebnis aus
    printContent(counts);
  }

  private static void printContent(
    final Map<Integer, Integer> counts
  ) {
    for (Map.Entry<Integer, Integer>
         pair : counts.entrySet()) {
      System.out.print(
        pair.getKey()
        + ":"
        + pair.getValue()
        + " "
      );
    }
    System.out.println("");
  }
}
```

Race Condition & Synchronized: Counting

Sequentiell oder `synchronized`

c: counts

v: value = 4

t1	t2	c (t1)	c (t2)	{(v,c)}
				$\{(4,1)\}$
`c = counts.get(v);`		1		$\{(4,1)\}$
`++c;`		2		$\{(4,1)\}$
`counts.put(v, c);`		2		$\{(4,2)\}$
	`c = counts.get(v);`		2	$\{(4,2)\}$
	`++c;`		3	$\{(4,2)\}$
	`counts.put(v, c);`		3	$\{(4,3)\}$

Race Condition & Synchronized: Counting

Nicht synchronisiert oder Hashtable

c: counts

v: value = 4

t1	t2	c (t1)	c (t2)	{(v,c)}
				{(4,1)}
`c = counts.get(v);`		1		{(4,1)}
`++c;`		2		{(4,1)}
	`c = counts.get(v);`		1	{(4,1)}
`counts.put(v, c);`		2		{(4,2)}
	`++c;`		2	{(4,2)}
	`counts.put(v, c);`		2	{(4,2)}

Race Condition & Synchronized: Counting

Nicht synchronisiert oder Hashtable

c: counts

v: value = 4

t1	t2	c (t1)	c (t2)	{(v,c)}
				$\{(4,1)\}$
`c = counts.get(v);`		1		$\{(4,1)\}$
`++c;`		2		$\{(4,1)\}$
	`c = counts.get(v);`		1	$\{(4,1)\}$
	`++c;`		2	$\{(4,2)\}$
`counts.put(v, c);`		2		$\{(4,2)\}$
	`counts.put(v, c);`		2	$\{(4,2)\}$

Race Condition & Synchronized: Counting

Nicht synchronisiert oder Hashtable

c: counts

v: value = 4

t1	t2	c (t1)	c (t2)	{(v,c)}
				$\{(4,1)\}$
`c = counts.get(v);`		1		$\{(4,1)\}$
`++c;`		2		$\{(4,1)\}$
	`c = counts.get(v);`		1	$\{(4,1)\}$
	`++c;`		2	$\{(4,2)\}$
	`counts.put(v, c);`		2	$\{(4,2)\}$
`counts.put(v, c);`		2		$\{(4,2)\}$

Race Condition & Synchronized: Counting

Nicht synchronisiert oder Hashtable

c: counts

v: value = 4

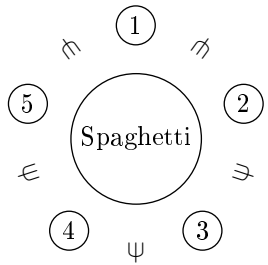
t1	t2	c (t1)	c (t2)	{(v,c)}
				$\{(4,1)\}$
	`c = counts.get(v);`		1	$\{(4,1)\}$
`c = counts.get(v);`		1		$\{(4,1)\}$
`++c;`		2		$\{(4,1)\}$
`counts.put(v, c);`		2		$\{(4,2)\}$
	`++c;`		2	$\{(4,2)\}$
	`counts.put(v, c);`		2	$\{(4,2)\}$

Parallelisierung: Semaphore

- Semaphore (synchronized)
  - Verhindert Race Conditions
  - Ein Teil des Codes wird sequentiell ausgeführt
    - längere Laufzeit
    - "Bottleneck"
  - Bereich sollte
    - so klein wie möglich
    - so groß wie nötig

    sein
- Neue Fehlerquelle: Deadlock
  - Alle threads bis auf einen
    - werden zu Beginn des synchronized-Blocks blockiert
    - warten bis der synchronized-Block freigegeben wird
  - Ein Thread kann sich nicht selbst blockieren
  - Blockieren zwei Threads zwei unterschiedliche synchronized-Blöcke, so können sie sich gegenseitig blockieren
    $\rightarrow$ Deadlock

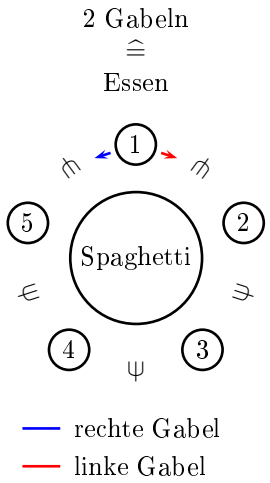
Deadlock: Dining Philosophers



- Gegeben:
  - n Philosophen
  - Teller mit Spaghetti
    (wird immer nachgefüllt)
  - n Gabeln zwischen den Philosophen
- Aufgabe der Philosophen
  - Nachdenken
  - Essen:
    Philosoph benötigt 2 Gabeln, die rechts und links neben ihm liegen
- Constraint:
  - Will ein Philosoph essen, so nimmt er immer zuerst die rechte und dann die linke Gabel

Deadlock: Dining Philosophers

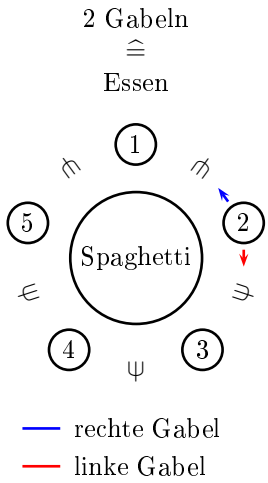
Abbildung: Sequentiell: Philosoph 1



- Gegeben:
  - n Philosophen
  - Teller mit Spaghetti (wird immer nachgefüllt)
  - n Gabeln zwischen den Philosophen
- Aufgabe der Philosophen
  - Nachdenken
  - Essen: Philosoph benötigt 2 Gabeln, die rechts und links neben ihm liegen
- Constraint:
  - Will ein Philosoph essen, so nimmt er immer zuerst die rechte und dann die linke Gabel

Deadlock: Dining Philosophers

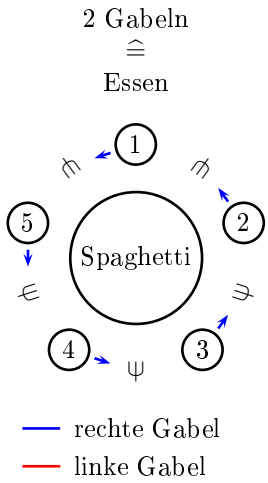
Abbildung: Sequentiell: Philosoph 2



- Gegeben:
  - n Philosophen
  - Teller mit Spaghetti (wird immer nachgefüllt)
  - n Gabeln zwischen den Philosophen
- Aufgabe der Philosophen
  - Nachdenken
  - Essen: Philosoph benötigt 2 Gabeln, die rechts und links neben ihm liegen
- Constraint:
  - Will ein Philosoph essen, so nimmt er immer zuerst die rechte und dann die linke Gabel

Deadlock: Dining Philosophers

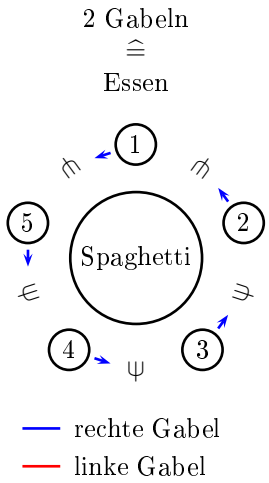
Abbildung: Parallel: alle gleichzeitig



- Gegeben:
  - n Philosophen
  - Teller mit Spaghetti (wird immer nachgefüllt)
  - n Gabeln zwischen den Philosophen
- Aufgabe der Philosophen
  - Nachdenken
  - Essen: Philosoph benötigt 2 Gabeln, die rechts und links neben ihm liegen
- Constraint:
  - Will ein Philosoph essen, so nimmt er immer zuerst die rechte und dann die linke Gabel

Deadlock: Dining Philosophers

Abbildung: Parallel: alle gleichzeitig



- Aufgabe der Philosophen
  - Essen:
    Philosoph benötigt 2 Gabeln, die rechts und links neben ihm liegen
- Constraint:
  - Will ein Philosoph essen, so nimmt er immer zuerst die rechte und dann die linke Gabel
- Keiner kann die linke Gabel nehmen, da sie der links sitzende Philosoph schon als seine rechte Gabel benutzt.
  $\rightarrow$ Deadlock