Karlsruher Institut für Technologie

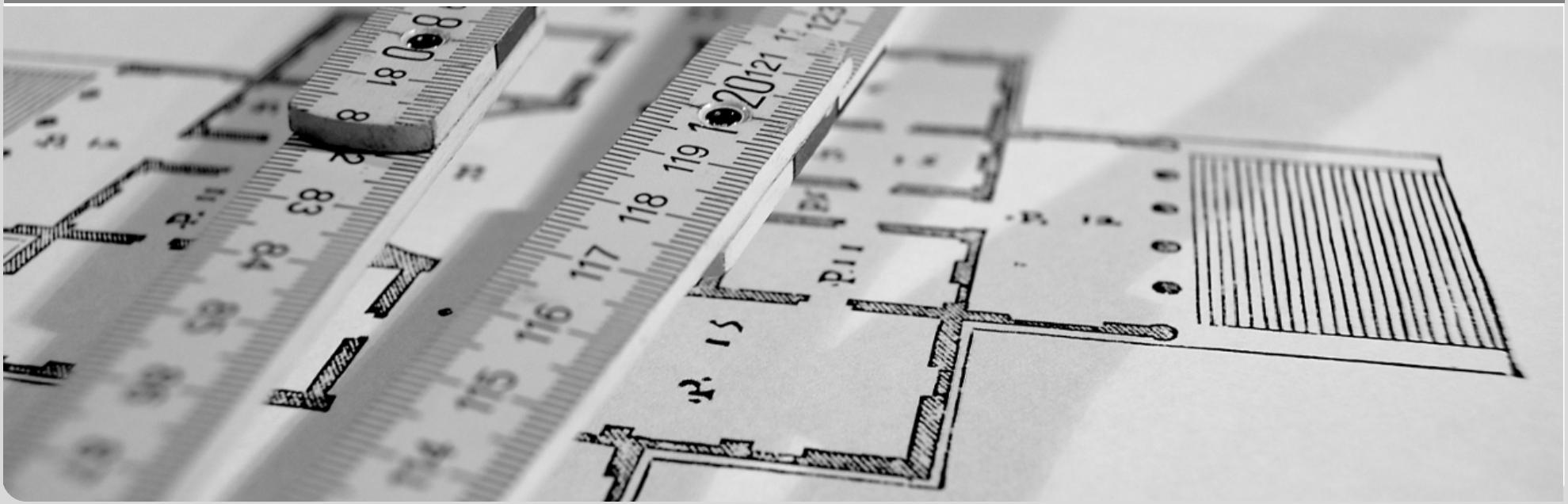
SDQ Software Design and Quality

Softwaretechnik - 6. Tutorium

Tutorium Nr. 17

Kay Schmitteckert | 02.07.2015

INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION (IPD)



Zu...

Übungsblatt 5

Zum Aufwärmen

Karlsruher Institut für Technologie

Quiz

Quiz

Aussage	wahr	falsch
Bei einer seichten Kopie eines Objekts werden alle Attribute kopiert, ausschlieSSlich der Referenzen auf andere Objekte		
UML-Anwendungsfalldiagramme werden während der Planungsphase verwendet, um das von aussen sichtbare Verhalten des Systems darzustellen		
Die Signatur einer Methode besteht aus dem Methodennamen und dem Rückgabetyp		
Ein Modul ist eine Menge von Programmelementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden		

Quiz

Aussage	wahr	falsch
Bei einer seichten Kopie eines Objekts werden alle Attribute kopiert, ausschlieSSlich der Referenzen auf andere Objekte		×
UML-Anwendungsfalldiagramme werden während der Planungsphase verwendet, um das von aussen sichtbare Verhalten des Systems darzustellen		
Die Signatur einer Methode besteht aus dem Methodennamen und dem Rückgabetyp		
Ein Modul ist eine Menge von Programmelementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden		

Quiz

Aussage	wahr	falsch
Bei einer seichten Kopie eines Objekts werden alle Attribute kopiert, ausschlieSSlich der Referenzen auf andere Objekte		×
UML-Anwendungsfalldiagramme werden während der Planungsphase verwendet, um das von aussen sichtbare Verhalten des Systems darzustellen	×	
Die Signatur einer Methode besteht aus dem Methodennamen und dem Rückgabetyp		
Ein Modul ist eine Menge von Programmelementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden		

Quiz

Aussage	wahr	falsch
Bei einer seichten Kopie eines Objekts werden alle Attribute kopiert, ausschlieSSlich der Referenzen auf andere Objekte		×
UML-Anwendungsfalldiagramme werden während der Planungsphase verwendet, um das von auSSen sichtbare Verhalten des Systems darzustellen	×	
Die Signatur einer Methode besteht aus dem Methodennamen und dem Rückgabetyp		×
Ein Modul ist eine Menge von Programmelementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden		

Quiz

Aussage	wahr	falsch
Bei einer seichten Kopie eines Objekts werden alle Attribute kopiert, ausschlieSSlich der Referenzen auf andere Objekte		×
UML-Anwendungsfalldiagramme werden während der Planungsphase verwendet, um das von auSSen sichtbare Verhalten des Systems darzustellen	×	
Die Signatur einer Methode besteht aus dem Methodennamen und dem Rückgabetyp		×
Ein Modul ist eine Menge von Programmelementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden	×	

Quiz

Aussage	wahr	falsch
Die Entwicklungskosten eines Software-Systems bestehen zum gröSSten Teil aus Personalkosten		
Für ein Softwareentwicklungsprojekt gilt die Faustregel: Der Aufwand für Wartung und Pflege ist typischerweise um einen Faktor von 2 bis 4 gröSSer als der Entwicklungsaufwand		
Die 4. Phase des Wasserfallmodells ist die Implementierungsphase		
Jeder Kontrollfaden eines Prozesses besitzt ein eigenes Code- und Datensegment im Hauptspeicher		

Quiz

Aussage	wahr	falsch
Die Entwicklungskosten eines Software-Systems bestehen zum gröSSten Teil aus Personalkosten	×	
Für ein Softwareentwicklungsprojekt gilt die Faustregel: Der Aufwand für Wartung und Pflege ist typischerweise um einen Faktor von 2 bis 4 gröSSer als der Entwicklungsaufwand		
Die 4. Phase des Wasserfallmodells ist die Implementierungsphase		
Jeder Kontrollfaden eines Prozesses besitzt ein eigenes Code- und Datensegment im Hauptspeicher		

Quiz

Aussage	wahr	falsch
Die Entwicklungskosten eines Software-Systems bestehen zum gröSSten Teil aus Personalkosten	×	
Für ein Softwareentwicklungsprojekt gilt die Faustregel: Der Aufwand für Wartung und Pflege ist typischerweise um einen Faktor von 2 bis 4 gröSSer als der Entwicklungsaufwand	×	
Die 4. Phase des Wasserfallmodells ist die Implementierungsphase		
Jeder Kontrollfaden eines Prozesses besitzt ein eigenes Code- und Datensegment im Hauptspeicher		

Quiz

Aussage	wahr	falsch
Die Entwicklungskosten eines Software-Systems bestehen zum gröSSten Teil aus Personalkosten	×	
Für ein Softwareentwicklungsprojekt gilt die Faustregel: Der Aufwand für Wartung und Pflege ist typischerweise um einen Faktor von 2 bis 4 gröSSer als der Entwicklungsaufwand	×	
Die 4. Phase des Wasserfallmodells ist die Implementierungsphase	×	
Jeder Kontrollfaden eines Prozesses besitzt ein eigenes Code- und Datensegment im Hauptspeicher		

Quiz

Aussage	wahr	falsch
Die Entwicklungskosten eines Software-Systems bestehen zum gröSSten Teil aus Personalkosten	×	
Für ein Softwareentwicklungsprojekt gilt die Faustregel: Der Aufwand für Wartung und Pflege ist typischerweise um einen Faktor von 2 bis 4 gröSSer als der Entwicklungsaufwand	×	
Die 4. Phase des Wasserfallmodells ist die Implementierungsphase	×	
Jeder Kontrollfaden eines Prozesses besitzt ein eigenes Code- und Datensegment im Hauptspeicher		×

Parallelität in Java

Parallelität in Java

Motivation

- Zwei Fäden führen parallel den gleichen Code aus
- Keine Garantie einer bestimmten Reihenfolge (Wettlauf)

Faden 1

```
                    // globalVar == 1
if (globalVar > 0) {


  globalVar--;
}
```

Faden 2

```
if (globalVar > 0) {
  globalVar--;
}
```

Parallelität in Java

Motivation

- Zwei Fäden führen parallel den gleichen Code aus
- Keine Garantie einer bestimmten Reihenfolge (Wettlauf)

Faden 1

```
                    // globalVar == 1
if (globalVar > 0) {


  globalVar--;
}
```

Faden 2

```
if (globalVar > 0) {
  globalVar--;
}
```

Parallelität in Java

Kritischer Abschnitt

- Bereich, in dem ein Zugriff auf gemeinsam genutzte Attribute bzw. auf gemeinsam genutzten Zustand stattfindet
- Wettlaufsituationen vermeiden
  - Nur eine Aktivität gleichzeitig in einen kritischen Abschnitt lassen

Parallelität in Java

Kritischer Abschnitt

- Bereich, in dem ein Zugriff auf gemeinsam genutzte Attribute bzw. auf gemeinsam genutzten Zustand stattfindet
- Wettlaufsituationen vermeiden
  - Nur eine Aktivität gleichzeitig in einen kritischen Abschnitt lassen

Parallelität in Java

Kritischer Abschnitt

- Bereich, in dem ein Zugriff auf gemeinsam genutzte Attribute bzw. auf gemeinsam genutzten Zustand stattfindet
- Wettlaufsituationen vermeiden
  - Nur eine Aktivität gleichzeitig in einen kritischen Abschnitt lassen

Parallelität in Java

Monitor

- Jedes Objekt kann ein Monitor sein
- dient dem Schutz kritischer Abschnitte (Vermeidung von Wettläufen)
- Versucht eine Aktivität, einen schon besetzten Monitor zu betreten, wird sie so lange blockiert, bis der Monitor wieder freigegeben wird
- Die selbe Aktivität kann einen Monitor beliebig oft betreten (sinnvoll z.B. bei Rekursion)

Parallelität in Java

Monitor

- Jedes Objekt kann ein Monitor sein
- dient dem Schutz kritischer Abschnitte (Vermeidung von Wettläufen)
- Versucht eine Aktivität, einen schon besetzten Monitor zu betreten, wird sie so lange blockiert, bis der Monitor wieder freigegeben wird
- Die selbe Aktivität kann einen Monitor beliebig oft betreten (sinnvoll z.B. bei Rekursion)

Parallelität in Java

Monitor

- Jedes Objekt kann ein Monitor sein
- dient dem Schutz kritischer Abschnitte (Vermeidung von Wettläufen)
- Versucht eine Aktivität, einen schon besetzten Monitor zu betreten, wird sie so lange blockiert, bis der Monitor wieder freigegeben wird
- Die selbe Aktivität kann einen Monitor beliebig oft betreten (sinnvoll z.B. bei Rekursion)

Parallelität in Java

Monitor

- Jedes Objekt kann ein Monitor sein
- dient dem Schutz kritischer Abschnitte (Vermeidung von Wettläufen)
- Versucht eine Aktivität, einen schon besetzten Monitor zu betreten, wird sie so lange blockiert, bis der Monitor wieder freigegeben wird
- Die selbe Aktivität kann einen Monitor beliebig oft betreten (sinnvoll z.B. bei Rekursion)

Parallelität in Java

Monitor

- Java erzwingt die paarweise Verwendung von Monitor-Anforderung und -Freigabe durch eine Blocksyntax

```
/*synchronisierter Block*/
synchronized (obj) {
  // kritischer
  // Abschnitt
}
```

```
/*synchronisierte Methode*/
synchronized void foo(){
  // ganze Methode
  // kritischer Abschnitt
}
```

- Wenn eine Aktivität versucht, einen besetzten Monitor zu betreten, wird sie ununterbrechbar blockiert

Parallelität in Java

Warten und Benachrichtigen

- Manchmal ist wechselseitiger Ausschluss nicht genug
- Kommunikation zwischen Ausführungsfäden kann nötig sein
- Methoden dazu in `java.lang.Object`
  - `wait()` - Aktivität wird in Monitor-Warteschlange schlafen gelegt
  - `notify()` - eine Aktivität in der Warteschlange wird aufgeweckt
  - `notifyAll()` - alle Aktivitäten in der Warteschlange werden aufgeweckt
- Um `wait` und `notify` aufrufen zu können, muss die aktuelle Aktivität mit `synchronized` den zugehörigen Monitor bereits betreten haben

Parallelität in Java

Warten und Benachrichtigen

- Manchmal ist wechselseitiger Ausschluss nicht genug
- Kommunikation zwischen Ausführungsfäden kann nötig sein
- Methoden dazu in `java.lang.Object`
  - `wait()` - Aktivität wird in Monitor-Warteschlange schlafen gelegt
  - `notify()` - eine Aktivität in der Warteschlange wird aufgeweckt
  - `notifyAll()` - alle Aktivitäten in der Warteschlange werden aufgeweckt
- Um `wait` und `notify` aufrufen zu können, muss die aktuelle Aktivität mit `synchronized` den zugehörigen Monitor bereits betreten haben

Parallelität in Java

Warten und Benachrichtigen

- Manchmal ist wechselseitiger Ausschluss nicht genug
- Kommunikation zwischen Ausführungsfäden kann nötig sein
- Methoden dazu in `java.lang.Object`
  - `wait()` - Aktivität wird in Monitor-Warteschlange schlafen gelegt
  - `notify()` - eine Aktivität in der Warteschlange wird aufgeweckt
  - `notifyAll()` - alle Aktivitäten in der Warteschlange werden aufgeweckt
- Um `wait` und `notify` aufrufen zu können, muss die aktuelle Aktivität mit `synchronized` den zugehörigen Monitor bereits betreten haben

Parallelität in Java

Warten und Benachrichtigen

- Manchmal ist wechselseitiger Ausschluss nicht genug
- Kommunikation zwischen Ausführungsfäden kann nötig sein
- Methoden dazu in `java.lang.Object`
  - `wait()` - Aktivität wird in Monitor-Warteschlange schlafen gelegt
  - `notify()` - eine Aktivität in der Warteschlange wird aufgeweckt
  - `notifyAll()` - alle Aktivitäten in der Warteschlange werden aufgeweckt
- Um `wait` und `notify` aufrufen zu können, muss die aktuelle Aktivität mit `synchronized` den zugehörigen Monitor bereits betreten haben

Parallelität in Java

Warten und Benachrichtigen

- Manchmal ist wechselseitiger Ausschluss nicht genug
- Kommunikation zwischen Ausführungsfäden kann nötig sein
- Methoden dazu in `java.lang.Object`
  - `wait()` - Aktivität wird in Monitor-Warteschlange schlafen gelegt
  - `notify()` - eine Aktivität in der Warteschlange wird aufgeweckt
  - `notifyAll()` - alle Aktivitäten in der Warteschlange werden aufgeweckt
- Um `wait` und `notify` aufrufen zu können, muss die aktuelle Aktivität mit `synchronized` den zugehörigen Monitor bereits betreten haben

Parallelität in Java

Warten und Benachrichtigen

- Manchmal ist wechselseitiger Ausschluss nicht genug
- Kommunikation zwischen Ausführungsfäden kann nötig sein
- Methoden dazu in `java.lang.Object`
  - `wait()` - Aktivität wird in Monitor-Warteschlange schlafen gelegt
  - `notify()` - eine Aktivität in der Warteschlange wird aufgeweckt
  - `notifyAll()` - alle Aktivitäten in der Warteschlange werden aufgeweckt
- Um `wait` und `notify` aufrufen zu können, muss die aktuelle Aktivität mit `synchronized` den zugehörigen Monitor bereits betreten haben

Parallelität in Java

Warten und Benachrichtigen

- Manchmal ist wechselseitiger Ausschluss nicht genug
- Kommunikation zwischen Ausführungsfäden kann nötig sein
- Methoden dazu in `java.lang.Object`
  - `wait()` - Aktivität wird in Monitor-Warteschlange schlafen gelegt
  - `notify()` - eine Aktivität in der Warteschlange wird aufgeweckt
  - `notifyAll()` - alle Aktivitäten in der Warteschlange werden aufgeweckt
- Um `wait` und `notify` aufrufen zu können, muss die aktuelle Aktivität mit `synchronized` den zugehörigen Monitor bereits betreten haben

Parallelität in Java

Warten und Benachrichtigen

- Beispiel: Hersteller-Verbraucher-Muster

```
// Hersteller
synchronized void put(Work w) {
  while (queue.isFull()) { this.wait(); }
  queue.add(w);
  this.notifyAll();
}

// Verbraucher
synchronized Work take() {
  while (queue.isEmpty()) { this.wait(); }
  this.notifyAll();
  return queue.remove();
}
```

Verklemmungen/Deadlocks

Verklemmungen/Deadlocks

```
Thread t1 = new Thread(new Runnable() {
  public void run(){
    synchronized (monitor1) {
      synchronized (monitor2) { rechne();}
    }
  }
});
Thread t2 = new Thread(new Runnable() {
  pubilc void run(){
    synchronized (monitor2) {
      synchronized (monitor1) { rechne();}
    }
  }
});
t1.start(); // sperrt monitor1
t2.start(); // sperrt monitor2; jetzt beide blockiert!
```

Parallelität in Java

Semaphor

- Anzahl Genehmigungen
- `acquire` blockiert, bis eine Genehmigung verfügbar ist und verringert anschlieSSend Anzahl der Genehmigungen um 1
- `release` erhöht Anzahl der Genehmigungen um 1 (oder Parameterwert)

Parallelität in Java

Semaphor

- Anzahl Genehmigungen
- `acquire` blockiert, bis eine Genehmigung verfügbar ist und verringert anschlieSSend Anzahl der Genehmigungen um 1
- `release` erhöht Anzahl der Genehmigungen um 1 (oder Parameterwert)

Parallelität in Java

Semaphor

- Anzahl Genehmigungen
- `acquire` blockiert, bis eine Genehmigung verfügbar ist und verringert anschlieSSend Anzahl der Genehmigungen um 1
- `release` erhöht Anzahl der Genehmigungen um 1 (oder Parameterwert)

Semaphor - Beispiel

```
public class Semaphore {
  private int tickets;
  public synchronized void aquire()
                              throws InterruptedException {
    while (count<= 0) { wait(); }
      --tickets;
    }

  public synchronized void release() {
    ++tickets;
    notifyAll();
  }

  public Semaphore(int capacity) {
  tickets = capacity; }
}
```

Parallelität in Java

Zyklische Barriere (CyclicBarrier)

- synchronisiert Gruppe von n Fäden
- Funktionsweise:
  - Fäden kommen an der Barriere an
  - ... und rufen `await()`-Methode der Barriere auf
  - Diese blockiert so lange, bis n Fäden warten
  - Fäden dürfen ihre Ausführung fortzusetzen (Barriere wird zurückgesetzt)
- kann wiederholt benutzt werden (daher der Name)

Parallelität in Java

Zyklische Barriere (CyclicBarrier)

- synchronisiert Gruppe von n Fäden
- Funktionsweise:
  - Fäden kommen an der Barriere an
  - ... und rufen `await()`-Methode der Barriere auf
  - Diese blockiert so lange, bis n Fäden warten
  - Fäden dürfen ihre Ausführung fortzusetzen (Barriere wird zurückgesetzt)
- kann wiederholt benutzt werden (daher der Name)

Parallelität in Java

Zyklische Barriere (CyclicBarrier)

- synchronisiert Gruppe von n Fäden
- Funktionsweise:
  - Fäden kommen an der Barriere an
  - ... und rufen `await()`-Methode der Barriere auf
  - Diese blockiert so lange, bis n Fäden warten
  - Fäden dürfen ihre Ausführung fortzusetzen (Barriere wird zurückgesetzt)
- kann wiederholt benutzt werden (daher der Name)

Parallelität in Java

Zyklische Barriere (CyclicBarrier)

- synchronisiert Gruppe von n Fäden
- Funktionsweise:
  - Fäden kommen an der Barriere an
  - ... und rufen `await()`-Methode der Barriere auf
  - Diese blockiert so lange, bis n Fäden warten
  - Fäden dürfen ihre Ausführung fortzusetzen (Barriere wird zurückgesetzt)
- kann wiederholt benutzt werden (daher der Name)

Parallelität in Java

Zyklische Barriere (CyclicBarrier)

- synchronisiert Gruppe von n Fäden
- Funktionsweise:
  - Fäden kommen an der Barriere an
  - ... und rufen `await()`-Methode der Barriere auf
  - Diese blockiert so lange, bis n Fäden warten
  - Fäden dürfen ihre Ausführung fortzusetzen (Barriere wird zurückgesetzt)
- kann wiederholt benutzt werden (daher der Name)

Parallelität in Java

Zyklische Barriere (CyclicBarrier)

- synchronisiert Gruppe von n Fäden
- Funktionsweise:
  - Fäden kommen an der Barriere an
  - ... und rufen `await()`-Methode der Barriere auf
  - Diese blockiert so lange, bis n Fäden warten
  - Fäden dürfen ihre Ausführung fortzusetzen (Barriere wird zurückgesetzt)
- kann wiederholt benutzt werden (daher der Name)

Parallelität in Java

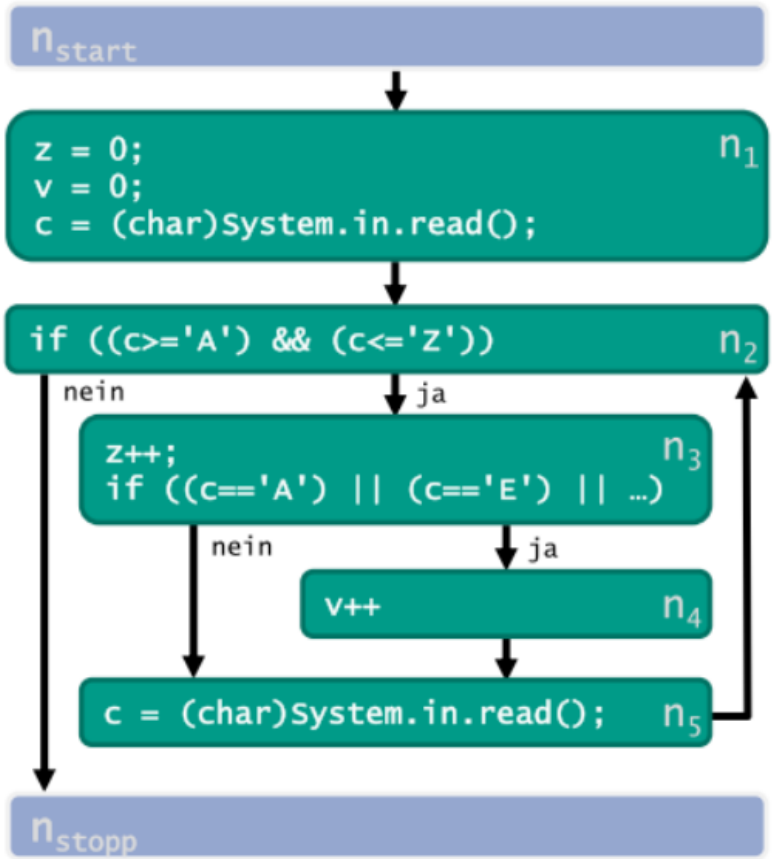
Zyklische Barriere (CyclicBarrier)

- synchronisiert Gruppe von n Fäden
- Funktionsweise:
  - Fäden kommen an der Barriere an
  - ... und rufen `await()`-Methode der Barriere auf
  - Diese blockiert so lange, bis n Fäden warten
  - Fäden dürfen ihre Ausführung fortzusetzen (Barriere wird zurückgesetzt)
- kann wiederholt benutzt werden (daher der Name)

Kontrollflussgraph

Kontrollflussgraph

- Kontrollflußgraph sind Automaten
- Trenne Zwischencode an jedem Label und an jeder Verzweigung in mehrere Blöcke
- Jeder Block wird zu einem Knoten im Automat
- Jede Verzweigung erzeugt 2 Kanten, für WAHR und FALSCH
- Jede Aufrufreihenfolge des Programmcodes wird nun auch von dem Automat modelliert!
- n_start und n_stopp enthalten Eingabeparameter und Rückgabewerte

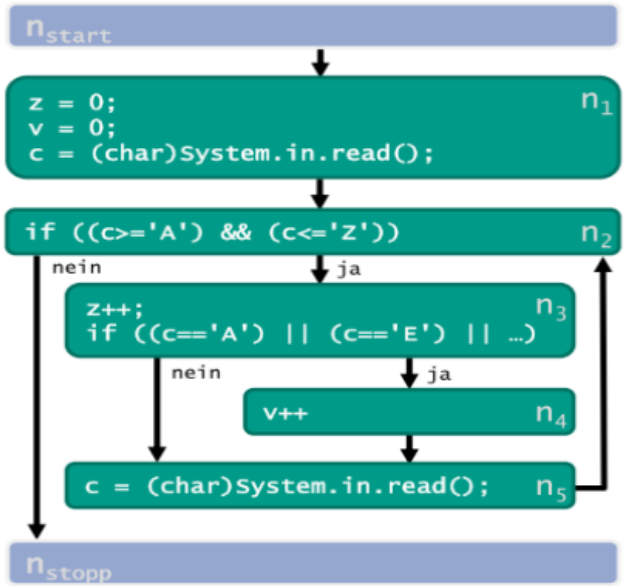


Aufgaben

Kontrollflussgraph - Aufgabe

```
public void sortiere(int[] feld) {
   if (feld != null) {
      if (feld.length == 1) {
         return;
      } else {
         int j, alterWert;
         for (int i = 1; i < feld.length; i++) {
            j = i;
            alterWert = feld[i];
            while(j > 0 && feld[j -1]> alterWert) {
               feld[j] = feld[j - 1];
               j--;
            }
            feld[j]=alterWert;
         }
      }
   }
}
```

- Erinnerung an Beispiel:



Parallelität - Klausuraufgabe SS 2009

Erklären Sie was passiert, wenn in Java ein Faden `f1` versucht, einen bereits von einem anderen Faden `f2` besetzten Monitor zu betreten. Wieso gibt es keine Methode `wouldBlock(object)`, die überprüft, ob der Faden bei der Monitoranforderung blockiert? Welche alternative Methode bietet die Klasse `java.lang.Thread` an? Was tut diese Methode?

Parallelität - Klausuraufgabe SS 2009

Musterlösung

- Der Faden `f1` wird ununterbrechbar blockiert, bis der Monitor durch den anderen Faden `f2` freigegeben wird
- Zwischen Test (`wouldBlock`) und Aktion danach kann sich die Situation wieder geändert haben: Monitor ist wieder freigegeben bzw. von einem anderen Faden belegt worden
- `java.lang.Thread.holdsLock(Object)`: Prüft, ob der aufrufende Faden den angegebenen Monitor hält

Parallelität - Klausuraufgabe SS 2010

b) Gegeben sei folgender Algorithmus zur Multiplikation zweier n×n-Matrizen a und b

```
01  private final int n = 42;
02  public int[][] matrixMult(int[][] a, int[][] b) {
03    int[][] c = new int[n][n];
04    for (int i = 0; i < n; i++) {
05      for (int j = 0; j < n; j++) {
06        for (int k = 0; k < n; k++) {
07          c[i][j] += a[i][k] * b[k][j];
08        }
09      }
10    }
11    return c;
12  }
```

Geben Sie seinen Namen und seine Funktionsweise an. Welchen Nachteil bezüglich der Cache-Nutzung hat der Algorithmus?

Parallelität - Klausuraufgabe SS 2010

Musterlösung

- Name: ijk-Algorithmus
- Funktionsweise: Klassische Matrixmultiplikation
- Performanz: Matrizen werden im Speicher zeilenweise abgelegt. Beim Zugriff auf ein Spaltenelement der Matrix b wird aber die gesamte Zeile der Matrix in den Cache eingelagert. Bei groSSen Matrizen würden es mit hoher Wahrscheinlichkeit häufig zu Cache-Misses kommen. Der Algorithmus ist somit deshalb nicht cachefreundlich, weil Spaltenelemente von b gelesen werden, die über die Cache-Zeilen verteilt sind

Parallelität - Klausuraufgabe SS 2010

c)

Erklären Sie, was eine Verklemmung ist und welche Auswirkung sie hat

Parallelität - Klausuraufgabe SS 2010

Musterlösung

- Definition: Eine Blockade, die durch eine zyklische Abhängigkeit von Fäden auf Ressourcen hervorgerufen wird
- Auswirkung: Eine Verklemmung führt dazu, dass alle beteiligten Fäden ewig im Wartezustand verharren

Parallelität - Klausuraufgabe SS 2010

d) Gegeben sei die Methode `ueberweise()`, die den Betrag `betrag` vom Konto `quelle` auf das Konto `ziel` überweist. Stellen Sie sich nun vor, dass diese Methode parallel von zwei Fäden aufgerufen wird. Zeigen Sie, bei welcher Wahl der Parameter eine Verklemmung in der Methode `ueberweise()` auftreten kann und begründen Sie warum

```
13  void ueberweise(Account quelle, Account ziel,
                                          int betrag) {
14    synchronized (quelle) {
15      synchronized (ziel) {
16        quelle.abheben(betrag);
17        ziel.einzahlen(betrag);
18      }
19    }
20  }
```

Parallelität - Klausuraufgabe SS 2010

Karlsruher Institut für Technologie

Musterlösung

- Auftreten:

```
Account k1 = new Account();
Account k2 = new Account();
```

- Parameterwahl:

Faden 1: `ueberweise(k1, k2, 100)` → `entermonitor k1;`
Faden 2: `ueberweise(k2, k1, 200)` → `entermonitor k2;`

- Ursache:

Wenn Faden 1 und 2 parallel laufen und zeitgleich die Methode `ueberweise()` aufrufen, sperren beide Fäden ihre Quellkonten `k1` und `k2`. Beim Versuch, das Zielkonto zu sperren, kann es dann zu einem Zyklus kommen. Jeder der beiden Fäden fordert eine Ressource an, die vom jeweils anderen Faden gehalten wird

Parallelität - Klausuraufgabe SS 2011

a)

Nennen Sie vier Synchronisationsmechanismen in Java und erklären Sie jeweils kurz deren Funktionsweise

Parallelität - Klausuraufgabe SS 2011

Karlsruher Institut für Technologie

Musterlösung

- Wechselseitiger Ausschluss:
  - Monitore: Markierung kritischer Abschnitte, die nur von einer Aktivität gleichzeitig betreten werden dürfen
  - Warten auf Ereignisse und Benachrichtigung, notify/wait: Aktivitäten können auf Zustandsänderungen warten, die durch andere Aktivitäten verursacht werden. Aktivitäten informieren andere, wartende Aktivitäten über Signale
  - Unterbrechungen: Eine Aktivität, die auf ein nicht (mehr) eintretendes Ereignis wartet, kann über eine Ausnahmebedingung abgebrochen werden (interrupt())

Parallelität - Klausuraufgabe SS 2011

Musterlösung

- CyclicBarrier: Fäden rufen `await()`-Methode der Barriere auf, die so lange blockiert, bis n Fäden warten. Danach wird den Fäden erlaubt, ihre Ausführung fortzusetzen (die Barriere wird zurückgesetzt)
- Semaphore: `acquire` blockiert, bis eine Genehmigung verfügbar ist und erniedrigt anschlieSSend Anzahl der Genehmigungen um 1; `release` erhöht Anzahl der Genehmigungen um 1

Parallelität - Klausuraufgabe SS 2011

b) Geg. sei folgende Implementierung der Methode `ParalleleBerechnung()`:

```
final Object logbuch = new Object();
final Object zaehler = new Object();
Thread faden1 = new Thread(new Runnable() {
  public void run() {
    synchronized (logbuch) {
      synchronized (zaehler) {/* Berechne etwas */}}}
});
Thread faden2 = new Thread(new Runnable() {
  public void run() {
    synchronized (zaehler) {
      synchronized (logbuch) {/* Berechne etwas */}}}
});
faden1.start(); faden2.start();
```

Welches Problem kann bei der Ausführung dieser Methode auftreten? (Fachwort) Welche Bedingungen lassen sich im Programm finden, sodass dieses Problem überhaupt auftritt? Wie kann das Problem behoben werden?

Parallelität - Klausuraufgabe SS 2011

Musterlösung

- Problem: Verklemmung
- Behebung: Anfordern der Monitore immer in der gleichen Reihenfolge

Parallelität - Klausuraufgabe SS 2011

d)

Erklären Sie die Begriffe Beschleunigung *S(p)* und Effizienz *E(p)*, welche zur Bewertung der Parallelität verwendet werden

Parallelität - Klausuraufgabe SS 2011

Musterlösung

- Beschleunigung *S(p)*: gibt an, um wie viel schneller der Algorithmus mit p Prozessoren im Vergleich zur besten sequenziellen Ausführung wird

$$S(p) = \frac{T(1)}{T(p)}$$

- Effizienz *E(p)*: gibt den Anteil an der Ausführungszeit an, die jeder Prozessor mit nützlicher Arbeit verbringt

$$E(p) = \frac{T(1)}{p \cdot T(p)} = \frac{S(p)}{p}$$

Übungsblatt 6