

# Skript: Thread Programmierung, HTWK

Gehalten von Prof. Geser im Sommersemester 2016

Lukas Werner

Ann Kathrin Hartmann

Toni Pohl

Stephan Kemper

25. April 2016

# Inhaltsverzeichnis

<b>1</b>	<b>Grundbegriffe</b>	<b>4</b>
1.1	Threads . . . . .	4
1.2	Nicht-Determinismus . . . . .	5
1.3	Kritische Bereiche . . . . .	7
1.4	Sperren . . . . .	8
<b>2</b>	<b>Verifikation</b>	<b>10</b>
2.1	Zeitliche Abläufe . . . . .	10
2.2	Serielle Abläufe . . . . .	10
2.3	Faire Mischung . . . . .	10
2.4	Sicherheits- und Liveness-Eigenschaften . . . . .	10
2.5	Modellierung . . . . .	10
<b>3</b>	<b>Synchronisation</b>	<b>11</b>
3.1	Signale . . . . .	11
3.2	Beispiel: Erzeuger/Verbraucher (1) . . . . .	11
3.3	Semaphore . . . . .	11
3.4	Beispiel: Erzeuger/Verbraucher (2) . . . . .	11
3.5	Bedingte kritische Bereiche . . . . .	11
3.6	Beispiel: Erzeuger/Verbraucher (3) . . . . .	11
3.7	Wiederbetretbare Sperren . . . . .	11
3.8	Leser/Schreiber-Problem . . . . .	11
<b>4</b>	<b>Feinkörnige Nebenläufigkeit</b>	<b>12</b>
4.1	Methoden . . . . .	12
4.2	Beispiel: Mengen, grobkörnig . . . . .	12
4.3	Beispiel: Mengen, feinkörnig . . . . .	12
4.4	Beispiel: Mengen, optimistisch . . . . .	12
4.5	Beispiel: Mengen, faul . . . . .	12
<b>5</b>	<b>Implementierung</b>	<b>13</b>
5.1	Atomare Befehle . . . . .	13
5.2	Konsenszahlen . . . . .	13
5.3	Zwischenspeicher . . . . .	13
5.4	Bäckerei-Algorithmus . . . . .	13
<b>6</b>	<b>Transactional Memory</b>	<b>14</b>
6.1	Probleme mit Sperren . . . . .	14

6.2	Transaktionen . . . . .	14
6.3	Software Transactional Memory (STM) . . . . .	14
6.3.1	Transaktionsstatus . . . . .	14
6.3.2	Transactional Thread . . . . .	14
6.3.3	Zwei Implementierungen . . . . .	14

# 1 Grundbegriffe

## 1.1 Threads

**Prozess** Sequentieller Rechenvorgang

**sequentiell** Alle Rechenschritte laufen nacheinander in einer vorgegebenen Reihenfolge ab.

**Thread** „leichte“ Variante eines Prozesses

Allgemeine Tendenz:

1. Systemkern möglichst „schlank“ halten
2. Systemkern möglichst selten betreten

Unterschied zu Prozess:

- Kein eigener Speicherbereich
- Üblicherweise nicht vom Systemkern verwaltet („light-weight process“), vom Systemkern verwaltet

Vorteile:

- Wechsel zwischen Threads weniger aufwändig als Wechsel zwischen Prozessen
- Threads benötigen weniger Speicher
- Man kann viel mehr Threads ( $\approx 10.000$ ) als Prozesse ( $\approx 100$ ) laufen lassen.

Nachteil:

Anwendungsprogrammierer muss sich um Verwaltung der Threads kümmern. Viele Programmiersprachen bieten heutzutage Programmbibliotheken für Threads an (Beispiel: *PThread* in C). Wir verwenden in dieser Veranstaltung *Java* als Programmiersprache.

**parallel** Mehrere Threads laufen gleichzeitig auf verschiedenen Rechnerkernen.

**verschränkt (engl. interleaved)** Threads laufen abwechselnd je ein Stück weit.

**nebeneinander laufend (auch: nebenläufig, engl. concurrent)** Mehrere Threads laufen parallel oder miteinander verschränkt.

Auch Mischformen sind möglich.

**Unterschied:**

**Rechenzeit (engl. cpu time)** Zeit, die der Prozessor mit Rechnen zubringt.

**Bearbeitungszeit (engl. wall clock time)** Umfasst auch Wartezeiten.

**Amdahlsches Gesetz (Gene Amdahl, 1967):**

Wenn eine Aufgabe die Bearbeitungszeit  $a$  benötigt und der Anteil  $0 \leq p \leq 1$  davon parallelisierbar ist, dann benötigt sie auf  $n$  Prozessoren die Bearbeitungszeit

$$a \left(1 - p + \frac{p}{n}\right). \quad (1.1)$$

Beispiel:

$$p = \frac{9}{10}, n = 100$$

Beschleunigung (speed up):

$$\frac{a}{a \left(1 - p + \frac{p}{n}\right)} = \frac{1}{1 - \frac{9}{10} + \frac{9}{1000}} \approx 9,17$$

$$\text{Sogar } \lim_{n \rightarrow \infty} \frac{1}{1 - p + \frac{p}{n}} = \frac{1}{1 - p} = 10$$

Fazit: Der nicht-parallelisierbare Anteil dominiert die Bearbeitungszeit.

## 1.2 Nicht-Determinismus

**Nicht-Determinismus (engl. nondeterminism)** Das Verhalten eines Systems hat Freiheitsgrade.

Nicht-Determinismus hat zwei Anwendungen:

1. Möglichkeiten des Verhaltens der Systemumgebung zusammenfassen (engl. don't know nondeterminism)
2. Spielraum für Implementierungen vorsehen (engl. don't care nondeterminism)

Hier: System von Threads

Man muss davon ausgehen, dass die Rechenschritte der Threads beliebig miteinander verschränkt sind. Die Reihenfolge der Schritte eines Threads ist durch sein Programm vorgegeben („Programm-Reihenfolge“).

Der Zeitplaner (engl. scheduler) legt zur Laufzeit fest, in welcher Reihenfolge die Schritte zweier Threads zueinander ablaufen. Man möchte den Zeitplaner in seiner Entscheidungsfreiheit nicht unnötig einschränken, sondern einen möglichst großen Spielraum lassen.

Man verlangt deshalb, dass das System von Threads korrekt zusammenarbeitet unabhängig davon, wie der Zeitplaner die Verschränkung bildet. Don't know nondeterminism

aus der Sicht des Anwendungsprogrammierers, don't care nondeterminism aus der Sicht des Zeitplaners.

Beispiel:

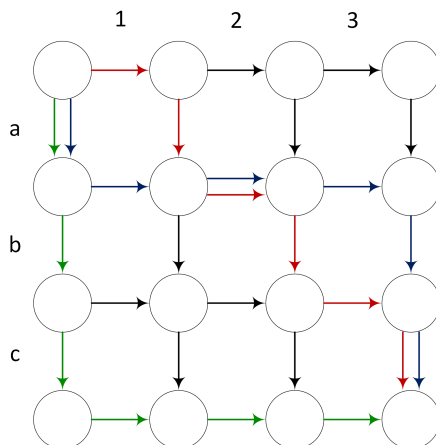
Thread 1 führt aus: (1) (2) (3)

Thread 2 führt aus: (a) (b) (c)

Beispiele für mögliche Abläufe:

- (1) (a) (2) (b) (3) (c)
- (1) (a) (2) (b) (3) (c)
- (1) (a) (2) (b) (3) (c)
- ...

Mögliche Abläufe visualisiert (Beispiele sind farblich markiert):



Da bei jedem Test der Zeitplaner eine andere Ausführungsreihenfolge (Umstände des Wettrennens, engl. race conditions) wählen kann, ist der Test praktisch nicht reproduzierbar. Wegen der großen Anzahl möglicher Abläufe ist ein systematisches Testen aussichtslos („Zustandsexplosion“). Der Entwickler muss deshalb die Korrektheit seines Programms mathematisch beweisen (verifizieren). Um Flüchtigkeitsfehler und übersehene Spezialfälle auszuschließen, lässt man die Beweise maschinell überprüfen (formale Verifikation).

Threads sind asynchron, d.h. sie laufen mit versch. Geschwindigkeiten. Es treten Wartezeiten auf, deren Zeitpunkt und Dauer nicht vorhersehbar ist, z.B. Laden einer neuen Speicherseite („page fault“), oder ein Zugriff auf den Zwischenspeicher (cache) scheitert.

## 1.3 Kritische Bereiche

Beispiel:

Zähler `z` wird initialisiert mit 0. Jeder Thread (von z.B. 10.000) addiert 1 zu `z`. `z++` wird vom Compiler sinngemäß so übersetzt:

```
1 int temp = z;
2 temp = temp + 1;
3 z = temp;
```

`z` ist eine gemeinsame Variable (gemeinsamer Speicher, shared memory), `temp` ist eine lokale Variable (temporäre Variable). Jeder Thread hat seine eigene Version von `temp`.

Beispielablauf für 3 Threads T1, T2, T3:

T1		T2		T3		z
Zeile	temp1	Zeile	temp2	Zeile	temp3	
1	0					0
		1	0			
		2	1			
		3				1
				1	1	
				2	2	
				3		2
2	1					
3						1

Der Wert von `z` sollte am Ende 3 sein. T1, T2, T3 kommen sich gegenseitig in die Quere: Einmischung (interference). Einmischung kann es nur über gemeinsame Variablen geben. Eine Methode, um Einmischung zu verhindern ist die Verwendung von *kritischen Bereichen*.

### Kritischer Bereich (auch kritischer Abschnitt, Emil. critical region, critical section)

Programmfragment in dem sich zu jedem Zeitpunkt höchstens ein Thread befindet.

### Gegenseitiger Ausschluss (mutual exclusion, exklusives Betriebsmittel)

Wenn sich ein Thread in einem kritischen Bereich befindet, dann werden alle anderen Threads davon abgehalten, den kritischen Bereich zu betreten.

Beispiele für exklusive Betriebsmittel:

- Stuhl
- Rechnerkern
- Schreibzugriff auf Speicherblock

- Schreibzugriff auf Bus
- Drucker
- Soundkarte (?)

Beispiel für einen kritischen Bereich:

```
gemeinsam int z = 0; // Deklaration der gem. Var. z
kritisch z {
z++ // kritischer Bereich, nur hier darf auf z zugegriffen werden
}
```

Wenn kritische Bereiche als Sprachmittel gegeben sind, kann der Compiler die korrekte Verwendung derselben überprüfen.

Der kritische Bereich kann von mehreren Threads nicht echt nebeneinander abgearbeitet werden, er wird also von den Threads in einer gewissen Reihenfolge nacheinander abgearbeitet (*Serialisierbarkeit*). Zwischenzustände sind von anderen Threads nicht beobachtbar, weil die gemeinsamen Variablen nur im kritischen Bereich zugreifbar sind. Der kritische Bereich wirkt wie eine einzelne Aktion: er ist *unteilbar* (atomar, engl. atomic).

## 1.4 Sperren

**Sperre (lock)** Datenstruktur mit Operationen belegen und freigeben.

**erzeugen(l)** legt Sperre l an und initialisiert l.frei mit false (l.frei: Sperre frei).

**belegen(l)** wartet solange, bis l.frei den Wert true hat und setzt es dann auf false.

**freigeben(l)** Setzt l.frei auf true.

Sperren können verwendet werden, um kritische Bereiche zu implementieren.

Beispiel: Sperre l bewacht die gemeinsame Variable z.

Programm (HP):

```
Sperre l anlegen mit l.frei = false
Threads anlegen Setze gem. Var. z = 0
freigeben(l)
```

In jedem Thread:

```
0 belegen(l);
1 int temp = z;
2 temp = temp + 1;
3 z = temp;
4 freigeben(l);
```



Paradox: Um kritische Bereiche nutzen zu können, braucht man schon kritische Bereiche.  
 Beispielablauf für 2 Threads:

T1		T2		z	l.frei	Bemerkung
<i>Zeile</i>	<i>temp1</i>	<i>Zeile</i>	<i>temp2</i>			
0		0		0	true false	T2 wartet in Zeile 0
1	0	0				
2	1	0				
3		0		1		
4		0			true false	Ende der Wartezeit
		0				
		1	1			
		2	2			
		3		2		
		4			true	

Sprechweise:

- Thread bewirbt sich für die Sperre (= ruft `belegen(1)` auf )
- Thread besitzt Sperre 1 (= ist im krit. Bereich/hat `belegen(1)` erfolgreich aufgerufen)
- Thread erwirbt die Sperre 1 (= betritt den krit. Bereich), Streit um die Sperre (engl. lock contention)

## 2 Verifikation

### 2.1 Zeitliche Abläufe

Vorgeben: Menge A von Aktionen

Ereignis (hier) Paar bestehend aus Aktion und Zeitpunkt  
aktion(e), zeit(e) für Ereignis e.

Beispiel: Schlacht bei Isis 333 v. Chr. → Aktion, Zeitpunkt

**Idealisierende Annahmen:**

1. Alles findet praktisch am selben Ort statt, keine Probleme mit der Lichtgeschwindigkeit (30cm in 1ns).

**Zeit (hier)** Newtonsche Zeit, Sie verläuft

- absolut d.h. unabhängig von Beobachter (sonst: spezielle Relativitätstheorie)
- stetig, d.h. ohne Sprünge (sonst Quantenmechanik)
- unbeeinflusst von der Umgebung (sonst: allg. Relativitätstheorie)
- Zeitpunkt = reale Zahl

2. Ein Ereignis hat die Dauer Null. Einen Zeitraum kann man darstellen durch die Ereignisse "Ende des Zeitraums".

3. Gleichzeitige Ereignisse sind ausgeschlossen, d.h. zwei Ereignisse, die die zur gleichen Zeit stattfinden, sind gleich  
 $zeit(e)_1 = zeit(e)_2 \leftrightarrow e_1 = e_2$

**diskreter zeitlicher Ablauf (auch Geschichte)** Menge E von Ereignissen, so dass:

1. die Menge der Zeitpunkte E keinen Häufungspunkt hat
2. die Menge der Zeitpunkte von E ein kleinstes Element hat

Interessant sind hier nicht die Zeitpunkte selber, sondern nur deren Lage zueinander, d.h. die Reihenfolge der Aktionen. (Wenn dies nicht der Fall ist und Termine eingehalten werden müssen → Echtzeitsystem)

**Def: (Leslie Lamport 1978)** Ereignis  $e_1$  kommt vor Ereignis  $e_2$  :

$$e_1 \rightarrow e_2 \leftrightarrow zeit(e_1) < zeit(e_2)$$

Beispiel: Hochmut  $\rightarrow$  Fall.

Es gilt:  $\rightarrow$  ist reflexiv, transitiv, total, fundiert (d.h. Wohlordnung)

Eine Relation  $R \in E \times E$  auf der Menge  $E$  heißt

- irreflexiv, falls  $\forall e \in E$  gilt  $(e, e) \notin R$
- transitiv, falls  $\forall e_1, e_2, e_3 \in E$  gilt: Falls  $(e_1, e_2) \in R$  und  $(e_2, e_3) \in R$ , dann  $(e_1, e_3) \in R$ .
- total, falls  $\forall e_1, e_2 \in E$  gilt: Falls  $e_1 \neq e_2$ , dann  $(e_1, e_2) \in R$  oder  $(e_2, e_1) \in R$
- fundiert, falls es keine unendliche Folge  $(e_i)_{i \in \mathbb{N}}$  gibt mit  $e_i \in E$  für alle  $i \in \mathbb{N}$  und  $(e_i, e_{i+1}) \in R$  für alle  $i \in \mathbb{N}$

Einschub:  $R$  azyklisch, falls es keine endliche Folge  $(e_1, \dots, e_n)$  gibt mit  $(e_1, e_2) \in R, (e_2, e_3) \in R, \dots, (e_{n-1}, e_n) \in R, (e_n, e_1) \in R$ . Falls  $R$  irreflexiv und transitiv ist, dann ist  $R$  auch azyklisch.

Für einen nicht leere Geschichte  $E$  sei  $\min E$  definiert als das kleinste Element von  $E$  bezüglich  $\rightarrow$ , d.h. dasjenige  $e \in E$  für das gilt:

$$\forall f \in E \setminus e : e \rightarrow f$$

Tipp: Relation als Graph vorstellen mit Wegen.

- Es existiert kein Weg der Länge 1 zu sich selber.
- Wenn es einen Weg von 1 zu 2 und 2 zu 3 gibt, dann existiert eine Abkürzung von 1 zu 3.
- Es gibt immer Weg von jedem zu jedem Knoten.
- Es existiert kein unendlicher Weg.

**Implizite Definition** Definition durch eine charakterisierende Eigenschaft

**Wohldefiniertheit der implizierten Definition** Es gibt genau ein Objekt, dass die charakterisierende Eigenschaft erfüllt. (Bsp.: "Wurzel von  $x$  ist das, was quadriert  $x$  ergibt" ist nicht eindeutig (gar keine Lsg. bzw. mehrere))

Wohldefiniertheit von  $\min E$  gilt, weil  $R$  total und  $E$  (mind.) ein kleinstes Element hat. (Ganze Zahlen sind z.B. total auf  $<$ , haben aber kein kleinstes Element)

Das  $i$ -te Element aus  $E$  ( $E^i$ ) ist dann für  $i \in \mathbb{N}, i \leq |E|$ :

$$E^i := \begin{cases} \min E & , \text{ falls } i = 1 \\ (E \setminus \min E)^{i-1} & , \text{ sonst} \end{cases}$$

Auch hier ist Wohldefiniertheit zu zeigen.

Projektion auf eine Menge  $B$  von Aktionen ("Sicht"):

$$\pi_B(E) := \{e \in E \mid \text{aktion}(e) \in B\}$$

Zustand zum Zeitpunkt  $t \in R$ :

$$z_t(E) := e \in E \mid \text{zeit}(e) \leq t$$

$\rightarrow$  für Zeiträume: Ende von Zeitraum A kommt vor Anfang von Zeitraum B:  $A \rightarrow B$  Es gilt: Für Zeiträume ist  $\rightarrow$  nicht total!  $A \rightarrow B \vee B \rightarrow A \leftrightarrow A$  und B überlappen nicht. (Wenn sich A und B überlappen gilt weder  $A \rightarrow B$  noch  $B \rightarrow A$ )

**Prozessalphabet** Menge der Aktionen, die der Thread p fließt"

**Gemeinsame Aktionen von  $p_1$  und  $p_2$**   $\alpha(p_1) \cap \alpha(p_2)$

**Einigkeit (engl. match)** Ereignisse mit gemeinsamen Aktionen finden gemeinsam statt:

- (1)  $\pi_{\alpha(p_1) \cap \alpha(p_2)}(E_1 \cup E_2 = E_1 \cap E_2$  Gleichwertig zu (1) sind:
- (2)  $\pi_{\alpha(p_1) \cap \alpha(p_2)}(E_1 \oplus E_2 = \emptyset$  // symmetr. Differenz: Vereinigung ohne Schnitt
- (3)  $\pi_{\alpha(p_1)} = \pi_{\alpha(p_2)}$

$E_i$  **Ereignis von Thread i** Es gilt:  $\forall e \in E_i : \text{aktion}(e) \in \alpha(p_i)$

**Faire Mischung**  $E_1 \cup E_2$

**Gemeinsame Mischung**  $\pi_{\alpha(p_i)}(E_1 \cap E_2) = E_i, \text{ für } i \in 1, 2$ , falls sich  $p_1$  und  $p_2$  einig sind.  
Es gilt:  $E_1 \cup E_2$ .

## 2.2 Serielle Abläufe

Wenn man nicht an den Zeitpunkten der Ereignisse interessiert ist, sondern nur an ihrer Lage zueinander, kann man statt einer Ereignismenge auch eine Aktionenfolge als Beschreibungsmittel für einen Ablauf nehmen.

Beispiele: Sei  $A = \{a, b\}$ . Endliche Folge (a, b, a) kann auch dargestellt werden als Funktion

$$f : 1, 2, 3 \rightarrow A \text{ mit } f(x) = \begin{cases} a & , \text{ falls } x = 1 \text{ oder } x=3 \\ b & \text{sonst} \end{cases}$$

Wertetabelle von f:

x 1 2 3

f(x) a b a

Unendliche Folge (a, b, b, a, b, b, ...) als Funktion  $f : \mathbb{N} \rightarrow A$  mit  $f(x) = \begin{cases} a & , \text{ falls } x \bmod 3 = 1, \\ b & \text{sonst} \end{cases}$

$A^k$  k-Tupel von Elementen aus A und

$i \in \mathbb{N} \mid i \leq k \rightarrow A$  Folgen der Länge k werden miteinander identifiziert.

**2.3 Faire Mischung**

**2.4 Sicherheits- und Liveness-Eigenschaften**

**2.5 Modellierung**

## 3 Synchronisation

### 3.1 Signale

### 3.2 Beispiel: Erzeuger/Verbraucher (1)

### 3.3 Semaphore

### 3.4 Beispiel: Erzeuger/Verbraucher (2)

### 3.5 Bedingte kritische Bereiche

### 3.6 Beispiel: Erzeuger/Verbraucher (3)

### 3.7 Wiederbetretbare Sperren

### 3.8 Leser/Schreiber-Problem

## 4 Feinkörnige Nebenläufigkeit

### 4.1 Methoden

### 4.2 Beispiel: Mengen, grobkörnig

### 4.3 Beispiel: Mengen, feinkörnig

### 4.4 Beispiel: Mengen, optimistisch

### 4.5 Beispiel: Mengen, faul

## 5 Implementierung

5.1 Atomare Befehle

5.2 Konsenszahlen

5.3 Zwischenspeicher

5.4 Bäckerei-Algorithmus



## 6 Transactional Memory

### 6.1 Probleme mit Sperren

### 6.2 Transaktionen

### 6.3 Software Transactional Memory (STM)

#### 6.3.1 Transaktionsstatus

#### 6.3.2 Transactional Thread

#### 6.3.3 Zwei Implementierungen

# Literaturverzeichnis

- [1] Maurice Herlihy, Nir-Shavit: „The Art of Multiprocessor Programming“, Morgan Kaufmann, 2008.
- [2] Kalvin Lin, Larry Snyder: „Principles of Parallel Programming“, Addison Wesley.
- [3] Greg Andrews: „Concurrent Programming“, Addison Wesley, 1991.
- [4] Brian Goetz, u.a.: „Java Concurrency in Practice“, Addison Wesley.