Skript: Thread Programmierung, HTWK

Gehalten von Prof. Geser im Sommersemester 2016

Lukas Werner Ann Kathrin Hartmann Toni Pohl

Stephan Kemper

17. Juli 2016

Anmerkungen

- Die natürlichen Zahlen $\mathbb{N}$ werden in dieser Veranstaltung ohne die 0 angenommen ($\mathbb{N}\backslash\{0\}$).

Inhaltsverzeichnis

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 („leight-weight process“), vom Systemkern verwaltet

Vorteile:

- Wechsel zwischen Threads weniger aufwändig als Wechsel zwischen Prozessen
- Threads benötigen weniger Speicher
- Man kann viel mehr Threads (≈ 10.000) als Prozesse (≈ 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

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

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

Beschleunigung (speed up):

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

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

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.

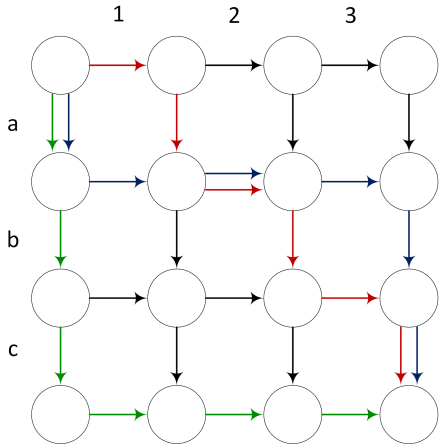
Beispiel:
Thread 1 führt aus: (1) (2) (3)
Thread 2 führt aus: (a) (b) (c)

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*	0
1	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**
Zeile	*temp1*	*Zeile*	*temp2*	0	true	
0					false	
		0				T2 wartet in Zeile 0
1	0	0				
2	1	0				
3		0		1		
4		0			true	
		0			false	Ende der Wartezeit
		1	1			
		2	2			
		3		2		
		4			true	

Sprechweise:

- Thread bewirbt sich für die Sperre (= ruft `belegen(l)` auf)
- Thread besitzt Sperre `l` (= ist im krit. Bereich/hat `belegen(l)` erfolgreich aufgerufen)
- Thread erwirbt die Sperre `l` (= 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
   $zeit(e)1 = zeit(e2) \Leftrightarrow e1 = e2$

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

Sonst: kontinuierliche Vorgänge (reaktive Systeme).
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 :
$e_1 \rightarrow e_2 \Leftrightarrow \text{zeit}(e_1) < \text{zeit}(e_2)$

Beispiel: Hochmut $\rightarrow$ Fall.
Es gilt: $\rightarrow$ ist irreflexiv, transitiv, total, fundiert (d.h. eine 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 $(e_1, ..., e_n)$ gibt mit $(e_1, e_2) \in R, (e_2, e_3) \in R, ..., (e_{n-1}, e_n) \in R, (e_n, e_1) \in R$. 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. (Beispiel: „Wurzel von x ist das, was quadriert x ergibt“ ist nicht eindeutig (gar keine Lösung bzw. mehrere))

Wohldefiniertheit von min E gilt, weil R total und E (mindestens) ein kleinstes Element hat ($\mathbb{Z}$ sind z.B. total auf $<$, haben aber kein kleinstes Element).
Das i-te Element aus E (E^i) ist dann für $i \in \mathbb{N}, i \leq |E|$:

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

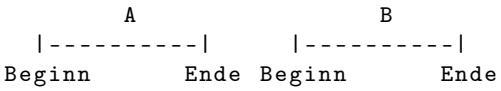
Auch hier ist Wohldefiniertheit zu zeigen.
Projektion auf eine Menge B von Aktionen („Sicht“):

$$\pi_B(E) := e \in E | aktion(e) \in B$$

Zustand zum Zeitpunkt $t \in R$:

$$z_t(E) := e \in E | zeit(e) < t$$

$\rightarrow$ 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 $\alpha(p)$ Menge der Aktionen, die der Thread p „sieht“

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. $\pi_{\alpha(p_1)\cap\alpha(p_2)}(E_1 \oplus E_2) = \emptyset \rightarrow$ symmetrische 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 Ereignisse $\pi_{\alpha(p_i)}(E_1 \cap E_2) = E_i$, 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
$f : \{1, 2, 3\} \rightarrow A$ mit $f(x) = \begin{cases} a & \text{falls } x = 1 \vee x = 3 \\ b & \text{sonst} \end{cases}$
Wertetabelle von f:

x	1	2	3
$f(x)$	a	b	c

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

A^k k-Tupel von Elementen aus A und $\{i \in \mathbb{N} | i \leq k\} \to A$ Folgen der Länge k werden miteinander identifiziert.

Um endliche und unendliche Abläufe einheitlich zu modellieren, bildet man die *Positionenmenge* $\mathbb{N}_\infty := \mathbb{N} \cup \{\infty\}$.

Vergleich $\leq$ auf $\mathbb{N}_\infty$ sei definiert durch

$$m \leq n \Leftrightarrow (n = \mathbb{N}_0) \vee m, n \in \mathbb{N}_0 \wedge m \leq n \text{ (auf } \mathbb{N}_0).$$

Vergleich $m \leq n$ für m, n $\in \mathbb{N}_0$ liefert dasselbe bei $\leq$ auf $\mathbb{N}_\infty$ wie bei $\leq$ auf $\mathbb{N}_0$.
Es gilt: $\leq$ auf $\mathbb{N}_\infty$ ist eine Fortsetzung von $\leq$ auf $\mathbb{N}_0$ und eine Wohlordnung.

$<$ auf $\mathbb{N}_0$ sei definiert durch $m < n \Leftrightarrow m \leq n \wedge m \neq n$.
Es gilt: $<$ auf $\mathbb{N}_\infty$ ist eine Fortsetzung von $<$ auf $\mathbb{N}_0$ und eine Wohlordnung.

Bemerkung: $\leq$ lässt sich weiter fortsetzen: Ordinalzahlen.

Aktionenfolgen

Sei A eine Aktionenmenge. Dann sei $A^* := \cup_{k \in \mathbb{N}_0} A^k$.
$A^\infty := A^* \cup (\mathbb{N} \to A)$, wobei A^* eine endliche Folge ist und $(\mathbb{N} \to A)$ unendliche Folgen.
Es gilt: $A^\infty = \cup_{k \in \mathbb{N}_\infty} (\{i \in \mathbb{N} | i \leq k\} \to A)$.
Die Länge $\#_x$ einer Aktionenfolge x ist definiert durch $\#_x = \begin{cases} \infty & \text{falls } x \in (\mathbb{N} \to A) \\ k & \text{falls } x \in A^k \end{cases}$

Es gilt $\#_x \in \mathbb{N}_\infty$.

$\leq_{\text{pre}} \subseteq A^\infty \times A^\infty$ sei definiert durch $x \leq_{\text{pre}} y \Leftrightarrow \#_x \leq \#_y \wedge \forall 1 \leq i \leq \#_x : x(i) = y(i)$.

Beispiel:
x = (a, b, a, a)
y = (a, b, a, a, b, a, ...)
x ist ein Anfangsstück (auch Präfix, engl. prefix) von y
Es gilt: $\leq_{\text{pre}}$ ist eine Ordnungsrelation (d.h. $\leq_{\text{pre}}$ ist reflexiv, transitiv, antisymmetrisch).

$<_{\text{pre}}$ ist definiert durch: $x <_{\text{pre}} y \Leftrightarrow x \leq_{\text{pre}} y \wedge x \neq y$

„strikter Anteil von $\leq_{\text{pre}}$“.
Es gilt: $<_{\text{pre}}$ ist eine Striktordnung (d.h. irreflexiv, transitiv).

Operationen auf Aktionenfolgen

rest $A^\infty \setminus \{\varepsilon\} \to A^\infty$, wobei $\{\varepsilon\}$ leere Aktionenfolge (= Aktionenfolge der Länge 0), ist definiert durch $rest(x)(i) = x(i+1)$. Es gilt: $\#_{rest(x)} = \infty$, falls $\#_x = \infty$ und $\#_{rest(x)} = \#_x - 1$, sonst:

Konkatenation (Hintereinanderstellen) von Aktionenfolgen ist definiert durch

$$(x \cdot y)(i) = \begin{cases} x(i) & \text{falls } i \leq \#_x \\ y(i - \#_x) & \text{sonst} \end{cases}.$$

Der zweite Fall tritt nicht bei $\#_x = \infty$ auf. Es gilt: $x \cdot y = x$, falls $\#_x = \infty$.

Projektion $\pi_B : A^\infty \to A^\infty$ auf Aktionenmenge B ist rekursiv definiert durch:

$$\pi_B(x) = \begin{cases} \varepsilon & \text{falls } x = \varepsilon \\ x(1) \cdot \pi_B(rest(x)) & \text{falls } 0 < \#_x < \infty \text{ und } x(1) \in B \\ \pi_B(rest(x)) & \text{falls } 0 < \#_x < \infty \text{ und } x(1) \notin B \\ sup\{\pi_B(y) | y <_{\text{pre}} x\} & \text{sonst (Supremum)} \end{cases}$$

Beispiel:
x = (a, b, c, a, b, c, ...)
B = a, b
Behauptung: π_B = (a, b, a, b, ...)
Beweis: Es gilt $\#_x = \infty$. Damit $\pi_B(x) = sup\{\pi_B(y) | y <_{\text{pre}} x\}$.

$$\begin{array}{lll} y_0 := \varepsilon & y_0 <_{\text{pre}} x & \pi_B(y_0) = \varepsilon \\ y_1 := (a) & & \pi_B(y_1) = (a) \\ y_2 := (a, b) & & \pi_B(y_2) = (a, b) \\ y_3 := (a, b, c) & & \pi_B(y_3) = (a, b) \\ y_4 := (a, b, c, a) & & \pi_B(y_4) = (a, b, a) \\ & \vdots & \end{array}$$

Die Anzahl der Vorkommen $\#_B(x)$ von Aktionen aus der Menge B in $x \in A^\infty$ ist definiert durch $\#_B(x) = \#_{\pi_B(x)}$.
Seien $B \subseteq A, i \in \mathbb{N}, x \in A^\infty$. Dann ist die Position des i-ten Vorkommens von einer Aktion aus B in x B^i_x definiert durch $B^i_x = \#_y + 1$, falls $y \cdot (a) \leq_{\text{pre}} x$ und $a \in Bund\#_B(y) = i - 1$.
Wohldefiniertheit ist hier zu zeigen.

Unendliche Wiederholung

Für $x \in A^k$ ist x^∞ definiert durch $x^\infty(i) = x((i-1) mod k + 1)$.
$(i-1) mod k + 1$ ist wie $i mod k$, außer wenn i ein Vielfaches von k ist:
$(i-1) mod k + 1$ liefert dann k statt 0.

y ist ein Zustand der Aktionenfolge x (y ist Vergangenheit von x), wenn $y \leq_{\text{pre}} x$ und $\#_y \in \mathbb{N}_0$ also endlich.

2.3 Faire Mischung

$x \in (A \cup B)^\infty$ heißt eine faire Mischung (engl. fair merge) von $y \in A^\infty$ und $z \in B^\infty$, wenn gilt

$$\pi_A(x) = y \text{ und } \pi_B(x) = z.$$

Bemerkung: „fair“ weil weder y noch z zu kurz kommen.
Bemerkung: Statt „faire Mischung“ sagt man auch „Verschränkung“ oder „shuffle“ (bei Zeichenreihen).

Beispiel:
Sei $y = (0,1)^\infty = (0,1,0,1,0,...)$ und $z = (2,3)^\infty = (2,3,2,3,2,...)$ mit $A = \{0,1\}$ und $B = \{2,3\}$. Dann ist $x_1 = (0,2,1,3,0,2,1,3,...)$ eine faire Mischung, weil
$(0,1,0,1,...) = y = \pi_A(x)$
$(2,3,2,3,...) = z = \pi_B(x)$
. Spezialfall x_1 heißt auch „perfect shuffle“.

Aber auch $x_2 = (2,0,1,3,2,0,1,3,...)$ faire Mischung. x braucht nicht periodisch zu sein oder y und z mit gleicher Geschwindigkeit zu behandeln.

2.4 Sicherheits- und Liveness-Eigenschaften

Spezifikation Beschreibung der gewünschten Eigenschaften des Systems aus Anwendersicht

Verifikation Nachweis, dass das System seine Spezifikation erfüllt

Spezifikation eines sequentiellen Programms $f : Z \to Z$, wobei Z die Menge der Programmzustände sei: $\text{pre}_f(z) \Rightarrow \text{post}_f(f(z)), \forall z \in Z$.

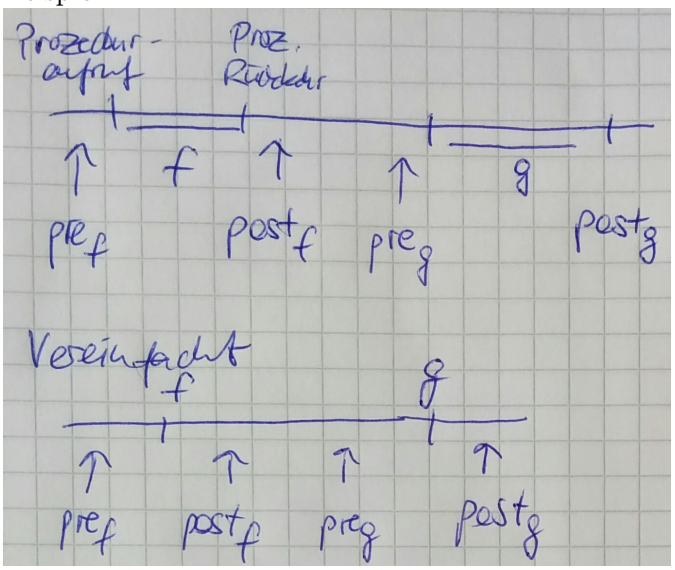
z alter Zustand

$f(z)$ neuer Zustand

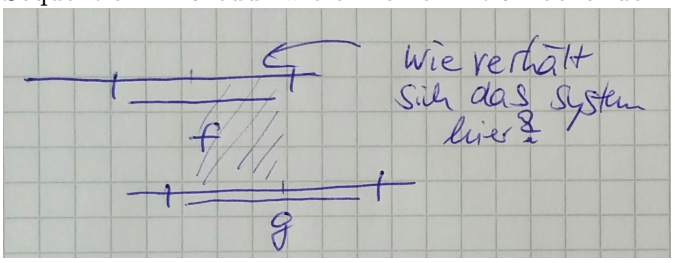
$\text{pre}_f\ Z \to \mathbb{B}$ Vorbedingung

$\text{post}_f\ Z \to \mathbb{B}$ Nachbedingung

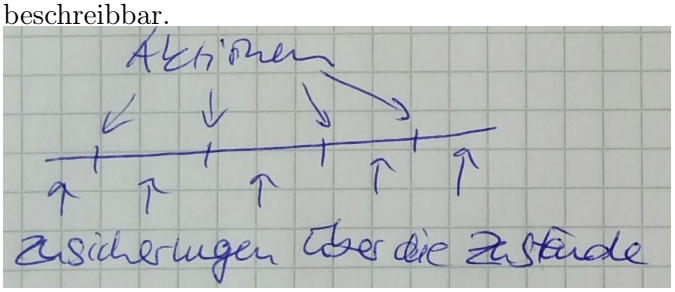
Beispiel:



Sequentiell: Prozedur wie einzelne Aktion behandeln



Problem bei Nebeneinanderablauf: siehe Zeitstrahl, Prozedurausführungen können sich überlappen. Die Ausführung von f und die Ausführung von g können sich über gemeinsame Objekte gegenseitig beeinflussen.
Fazit: Das Verhalten einer Prozedur f ist durch pre_f und post_f nicht mehr genau genug beschreibbar.



Vereinfachungsmöglichkeiten:

- Aktionen, die keine gemeinsamen Objekte betreffen können zusammengefasst werden zu einer Aktion
- Kritische Bereiche können zusammengefasst werden zu einer Aktion

Sicherheitseigenschaft (auch: Konsistenz, Invariante, engl. safety property) Eigenschaft, die für jeden Zustand gelten soll und die sich nur auf vergangene Zustände bezieht.

Liveness-Eigenschaft (auch: Fortschritt, engl. liveness property) alle übrigen Eigenschaften von Abläufen

Grund für die Unterscheidung:

1. Sicherheitseigenschaften sind einfacher zu beweisen.
2. Die meisten Eigenschaften sind Sicherheitseigenschaften.

Intuitiv:
Eine Sicherheitseigenschaft garantiert, dass nichts unerwünschtes geschieht („Verbot“). Eine Liveness-Eigenschaft garantiert, dass *schließlich* etwas Erwünschtes geschieht („Versprechen“). Ein Thread, der nur wartet, erfüllt alle Sicherheitseigenschaften: „Wer schläft, sündigt nicht“.
Bemerkung: Zu sequentiellen Programmen ist partielle Korrektheit eine Sicherheitseigenschaft und Termination eine Liveness-Eigenschaft.

Beweismethode für eine Sicherheitseigenschaft

„Für jeden Zustand gilt S“:

1. S gilt für den Startzustand.
2. S bleibt bei jedem Zustandsübergang erhalten, d.h. wenn S im alten Zustand gilt, dann auch im neuen.

Variante:

1. S gilt nach Initialisierung.
2. S bleibt außerhalb von kritischem Bereich erhalten.
3. Wenn S beim Betreten eines kritischen Bereiches gilt, dann auch beim Verlassen.

Sperren sollen folgende Eigenschaften erfüllen:

1. **Gegenseitiger Ausschluss** Die kritischen Bereiche zweier Threads überlappen nicht.
   Sicherheitseigenschaft: „In jedem Zustand ist verboten, dass zwei Threads im kritischen Bereich sind“
2. **Verklemmungsfreiheit** Wenn ein Thread die Sperre erwerben möchte, dann gibt es einen Thread, der die Sperre bekommt.
   Liveness-Eigenschaft: „Wenn die Sperre zugeteilt werden soll, wird sie schließlich zugeteilt“
3. **Fairness (auch: kein Verhungern)** Jeder Thread, der eine Sperre erwerben möchte, bekommt sie schließlich auch. (Liveness-Eigenschaft)

Es gilt: Aus Fairness folgt Verklemmungsfreiheit.
Bedingungen für Prozeduraufrufe im Zusammenhang mit nebeneinander laufenden Threads

1. **Bounded wait-free** Jeder Prozeduraufruf terminiert nach einer beschränkten Anzahl von Schritten.
2. **Wait-free** Jeder Prozeduraufruf terminiert, d.h. bleibt nicht in einer Warteschleife hängen. (Liveness-Eigenschaft)
3. **Lock-free** Unendlich viele Aufrufe terminieren. (Liveness-Eigenschaft)

2.5 Modellierung

Ausdrucksformen für Sicherheits- und Liveness-Eigenschaften:

- Formale Sprachen, insbesondere reguläre Ausdrücke
- Prozessalgebra
- Temporale Logik: linear oder verzweigend
- Prädikatenlogik mit Abläufen als Objekten

Beispiel: Modellieren von Sperren

Aktionen:

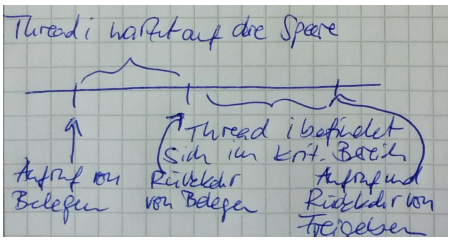
ant_i Thread i beantragt die Sperre

bel_i Thread i belegt die Sperre (betritt den krit. Bereich)

fr_i Thread i gibt die Sperre frei (verlässt den krit. Bereich)

$Bel := \{bel_i | i \in I\}$ die Sperre wird belegt (I ist Menge der Threads)

$Fr := \{fr_i | i \in I\}$ die Sperre wird freigegeben



Gegenseitiger Ausschluss

1. Mit regulären Ausdrücken

Für jeden Ablauf x gilt: Für jedes endliche Anfangsstück y von x gilt:
Die Projektion von y auf die Aktionenmenge $Bel \cup Fr$ ist in der Sprache $(BelFr)^*(Bel + \epsilon) = PRE((BelFr)^*)$ („Präfixabschluss“)
Sei $Ant := \{ant_i | i \in I\}$ und $A := Ant \cup Bel \cup Fr$. Dann kann man gegenseitigen Ausschluss als Formel angeben: $\forall x \in A^{infty} : \forall y \leq_{\text{pre}} x : \#_y <_\infty \Rightarrow \pi_{Bel \cup Fr}(y) \in PRE((BelFr)^*)$
Ausgeschlossen ist z.B. der Ablauf $bel_1 bel_1 fr_2$
Präfixabschluss definiert durch $PRE(x) = \{y \in A^* | y \leq_{\text{pre}} x\}$. Bemerkung: Sicherheitseigenschaften kann man immer mit Präfixabschluss beweisen. Formel für gegenseitigen Ausschluss kompakter: $\pi_{Bel \cup Fr}(PRE(x)) \subseteq PRE((BelFr)^*)$ Zugelassen ist z.B. der Ablauf $bel_1 fr_2 ant_1$. Ausgeschlossen ist z.B. $bel_1 bel_1 fr_2$. Falls $y \leq_{\text{pre}} x$ und $x \in X$, dann $y \in PRE(x)$. $X \subseteq A^*$ ist abgeschlossen unter Präfixen, falls $PRE(X) \subseteq X$. Es gilt: $PRE(X)$ ist abgeschlossen unter Präfixen

2. Mit Formeln der Prädikatenlogik

$\forall y \in PRE(x) \cap A^* : 0 \leq \#_{Bel} y - \#_{Fr} \leq 1$. Oder: $\forall i \in \mathbb{N} : Bel_x^i \leq Fr_x^i \leq Bel_x^{i+1}$ Oder $\forall k, l \in \mathbb{N}. fr_{ix}^k < bel_{jx}^l \vee fr_{jx}^l < bel_{ix}^k$

3. Mit Zustandsautomaten

siehe Bild

Lineare temporale Logik

Hier: Lineare temporale Aussagenlogik.
Syntax: Formeln sind aufgebaut mit:

- true, false
- Variablen
- Verknüpfungen $\wedge, \vee, \neg$ (weitere Verknüpfungen können damit definiert werden, z.B. $\Rightarrow, \Leftrightarrow, \oplus$. Endliche Quantoren $\wedge_{i \in M}$ „für alle $i \in M$“, $\vee_{i \in M}$ „es existiert $i \in M$“ mit M endlich.)
- temporale Operatoren
  - $\bigcirc$ „next“ „im nächten Zustand gilt“
  - $\Box$ „always“ „in allen zukünftigen Zuständen gilt“
  - $\diamond$ „eventually“ „in mind. einem zukünftigen Zustand gilt“

Beispiel: „Wer A sagt, muss auch B sagen“

$$\Box(A \Rightarrow B)$$

„Never change a running system“

$$\Box(R \Rightarrow \Box R)$$

gleichwertig: $\Box(R \Rightarrow \bigcirc R)$

Semantik

σ sei ein serieller Ablauf
j sei eine natürliche Zahl
p sei eine temporal logische Formel

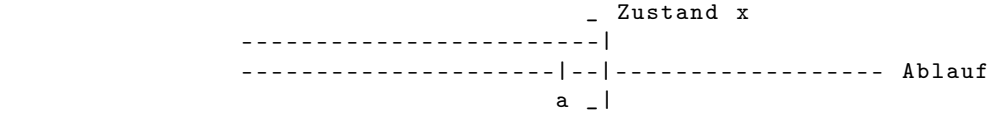
$(\sigma, j) \models p$ „Formel p gilt an Position j des Ablaufs σ“
Das wird rekursiv definiert durch:

$$(\sigma, j) \models p \wedge q :\Leftrightarrow$$
$$(\sigma, j) \models p \wedge (\sigma, j) \models q$$
usw.
$$(\sigma, j) \models \bigcirc p :\Leftrightarrow (\sigma, j+1) \models p$$
$$(\sigma, j) \models \Box p :\Leftrightarrow \forall k \geq j : (\sigma, k) \models p$$
$$(\sigma, j) \models \diamond p :\Leftrightarrow \exists k \geq j : (\sigma, k) \models p$$

Gegenseitiger Ausschluss mit temporal-logischen Formeln:
Beispiel:
$z_x :\Leftrightarrow 0 \leq \#_{Bel} x - \#_{Fr} x \leq 1$
$\Box z$

Kein Verhungern:
$beant_{i\ x} :\Leftrightarrow \#_{ant_i} x > \#_{bel_i} x$
„Thread i hat die Sperre beantragt, aber noch nicht belegt“

a_x soll bedeuten: Aktion a ist im Zustand x soeben ausgeführt worden.



Semantik dazu:
$(\sigma, j) \models a :\Leftrightarrow \sigma(j) = a\ \Box(beant_i \Rightarrow \diamond bel_i)$

3 Synchronisation

3.1 Signale

Synchronisation (hier) dafür sorgen, dass gewisse Abläufe ausgeschlossen sind. Auch: Koordination.

Signal (auch: Handshake, Meldung, engl. Notification) Hinweis an einen anderen Thread, dass er weitermachen kann.

Analogie:

- Startschuss beim Wettlauf
- Staffel beim Staffellauf
- Anschlusszug muss warten
- Becher vor Kaffeezulauf

Ein Signal kann durch eine Sperre implementiert werden:

- signalisieren (auch: melden) = freigeben
- warten = belegen

Das Signal soll garantieren, dass eine gewisse Reihenfolge eingehalten wird.

```
P1:
    S1;
    freigeben(l);
P2:
    belegen(l);
    S2;
```

```
           S1
P1 -----|------|--|-----------
P2 -----|----------|--|------|
           warten        S2
```

l muss freigegeben worden sein, bevor es wieder belegt werden kann, also findet S_1 vor S_2 statt. Durch die Verwendung von Signalen schränkt man die Menge der Abläufe ein. Nachteil: *weniger Parallelität.*

Extremfall: nur noch eine Reihenfolge möglich; der Ablauf wird seriell. Abgesehen vom Koordinationsaufwand zu einem seriellen Programm dann gleichwertig.

3.2 Beispiel: Erzeuger/Verbraucher-Problem, 1. Version

Erzeuger und Verbraucher sind Threads. Der Erzeuger erzeugt Datenblöcke. Der Verbraucher holt die Datenblöcke ab und verarbeitet sie. Die erzeugten aber noch nicht verbrauchten Datenblöcke werden in einem Puffer (:= Warteschlange) zwischengespeichert.

1. Version

Erzeuger	1
Verbraucher	1
Puffergröße	1

Thread *erz*:

```
Wiederhole:
    herstellen(datenblock);
    einreihen(puffer, datenblock);
```

Thread *verb*:

```
Wiederhole:
    abholen(puffer, datenblock);
    verarbeiten(datenblock);
```

Prozeduren:

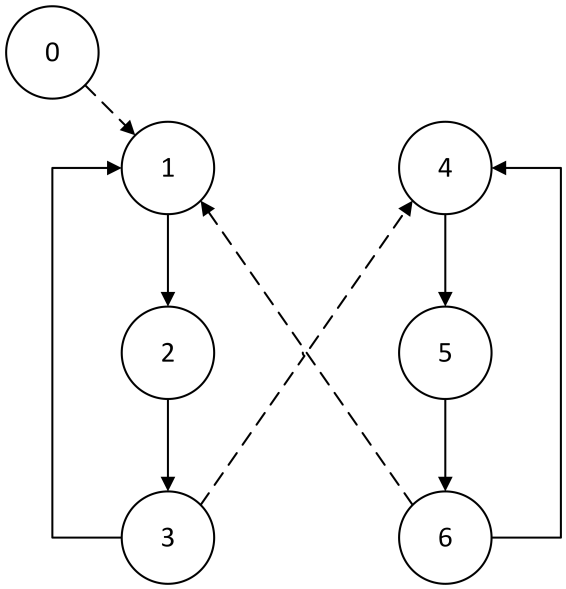
```
    einreihen(puffer, datenblock):
1       belegen(leer);
2       kopieren(datenblock, puffer); // kopiert Datenblock in Puffer
3       freigeben(voll);

    abholen(puffer, datenblock):
4       belegen(voll);
5       kopieren(puffer, datenblock); // kopiert Puffer in Datenblock
6       freigeben(leer);
```

Hauptprogramm:

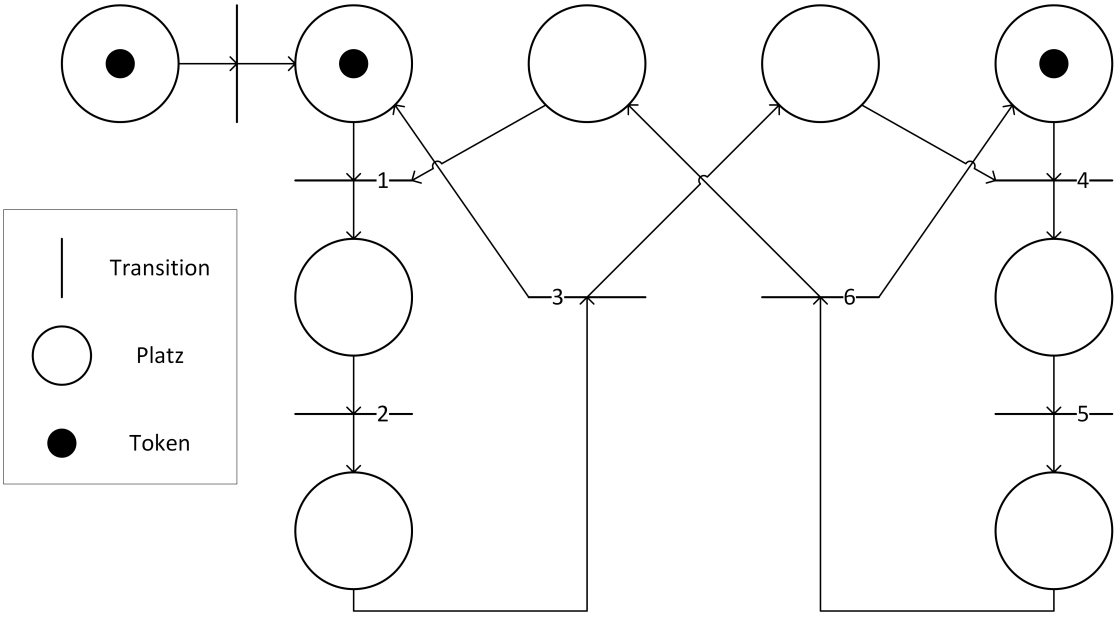
```
    Sperre voll anlegen; // als belegt
    Sperre leer anlegen; // als belegt
    Threads erz und verb anlegen und laufen lassen;
0   freigeben(leer);
```

Kausalitätsgraph



$\longrightarrow$: Programmreihenfolge; $\dashrightarrow$: Reihenfolge erzwungen durch Signal.

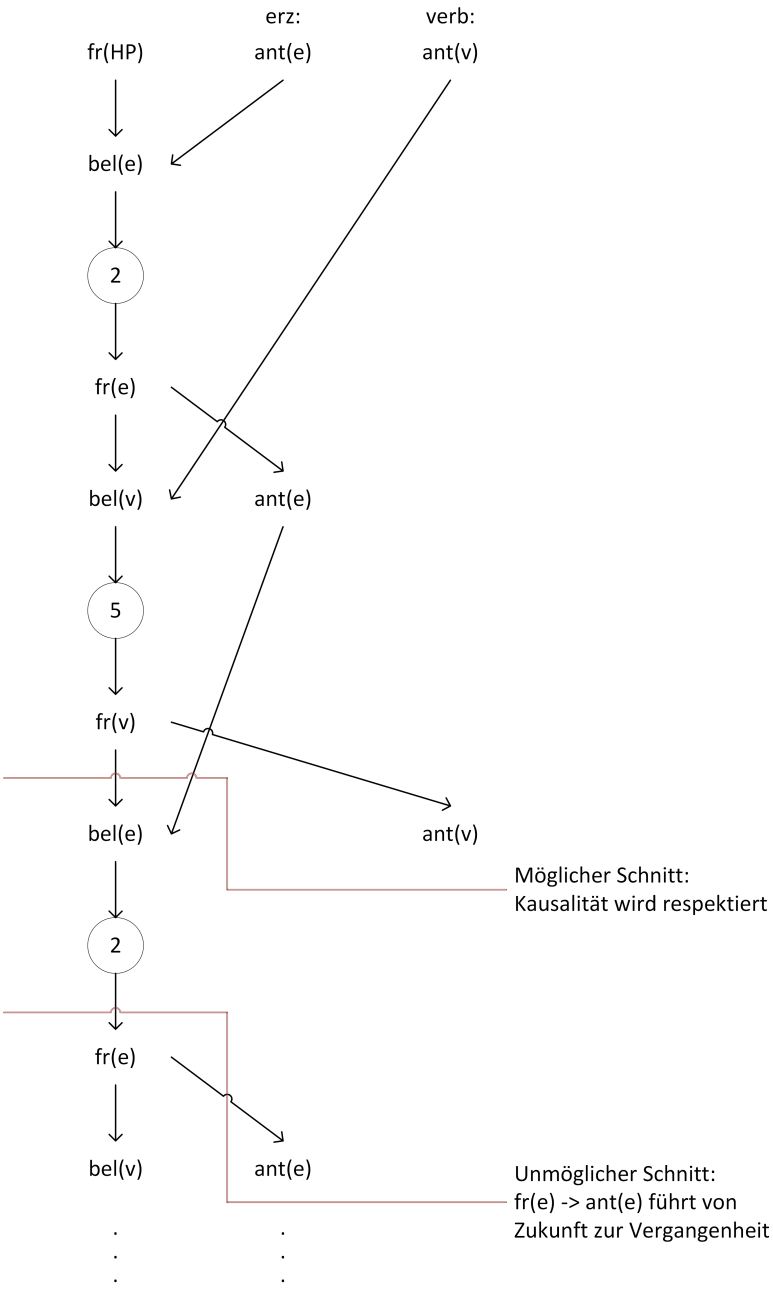
Petri-Netz



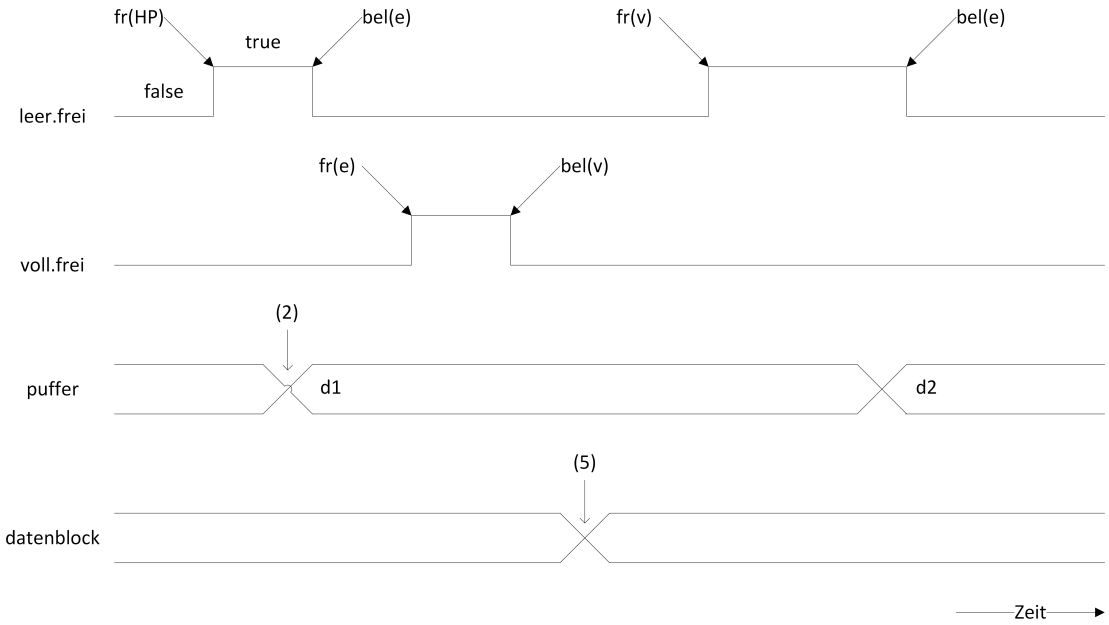
Bipartiter Graph: Zwei Sorten von Knoten; Pfeile nur zwischen verschiedenen Knotensorten.

Elementare Ereignis-Struktur

Durch Abrollen des Kausalitätsgraphen:



Signaldiagramme



- (2) und (5) werden als kritische Bereiche behandelt
- (2) und (5) werden nur abwechselnd ausgeführt

zu 1: Gegenseitiger Ausschluss gilt: $\neg$leer.frei $\vee$ voll.frei ist invariant.
zu 2: Folgt aus Programmierreihenfolge und 1.

3.3 Semaphore

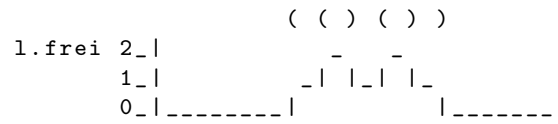
Semaphor Datenstruktur mit Zustand l.frei $\in \mathbb{N}_0$ und Operationen „belegen“ und „freigeben“.

belegen(l) Wartet bis l.frei > 0 und setzt dann l.frei auf l.frei - 1.

freigeben(l) Setzt l.frei auf l.frei + 1

Sperre ist Spezialfall mit l.frei $\in \{0, 1\}$.
Zweck: l.frei verschiedene Kopien eines Betriebsmittels werden verwaltet.
Zusammenhang zu Klammerausdrücken:
„(“ bedeutet „freigeben(l)“
„)“ bedeutet „belegen(l)“
l.frei = Anzahl der noch offenen Klammern

Beispiel:



3.4 Beispiel: Erzeuger/Verbraucher-Problem, 2. Version

2. Version

Erzeuger	1
Verbraucher	1
Puffergröße	N (mit $N > 0$ beliebig)

Threads erz und verb wie in Version 1.

Prozeduren:

```
    einreihen(puffer, datenblock):
1       belegen(nichtvoll); // I' gilt nun wieder
2       stock(puffer, datenblock); // Anfuegen des Datenblocks an Puffer
    hinten
3       freigeben(nichtleer); // I gilt

    abholen(puffer, datenblock):
4       belegen(nichtleer); // I' gilt
5       datenblock := top(puffer); // Liefert vordersten Datenblock des
    Puffers
6       pop(puffer);
7       freigeben(nichtvoll); // I gilt

    main():
        Leeren puffer anlegen;
        Semaphore nichtvoll und nichtleer erzeugen mit nichtvoll.frei = 0
            und nichtleer.frei = 0;
        Threads erg und verb anlegen und laufen lassen;
        freigeben^N(nichtvoll); // nichtvoll wird N-mal freigegeben
        // I gilt
```

Invariante I: $0 \leq$ nichtvoll.frei $\leq N \wedge$ nichtleer.frei $+$ nichtvoll.frei $\leq N$
Invariante I': $0 \leq$ nichtvoll.frei $\leq N \wedge$ nichtleer.frei $+$ nichtvoll.frei $\leq N - 1$
Invariante I'': $0 \leq$ nichtvoll.frei $\leq N \wedge$ nichtleer.frei $+$ nichtvoll.frei $\leq N - 2$
I' gilt, wenn beide Threads belegen, aber noch nicht freigeben aufgerufen haben.

3.5 Bedingte kritische Bereiche

Ein kritischer Bereich soll nur betreten werden, wenn eine gewisse Bedingung B an die gemeinsame Variable gilt. Wie implementiert man das?

1. B vor dem Betreten des kritischen Bereichs überprüfen.
   Problem: B kann beim Betreten des kritischen Bereiches bereits wieder verletzt sein.

2. B im kritischen Bereich überprüfen.
   Problem: Solange B nicht gilt, soll der Thread warten. Weil er sich im kritischen Bereich befindet, können andere Threads die gemeinsame Variable nicht ändern und damit den Wert von B.

Mit kritischen Bereichen kann man das Problem nicht lösen. Abhilfe: neues Konstrukt.

3.6 Beispiel: Erzeuger/Verbraucher-Problem, 3. Version

3. Version

Erzeuger	n
Verbraucher	m
Puffergröße	N (mit $N > 0$ beliebig)

```
einreihen(puffer, datenblock):
    kritisch l {
        warte auf laenge(puffer) < N;
        stock(puffer, datenblock);
    }

abholen(puffer, datenblock):
    kritisch l {
        warte auf laenge(puffer) > 0;
        datenblock := top(puffer);
        pop(puffer);
    }

// laenge(puffer) liefert die Anzahl der Datenbloecke im Puffer

main():
    Erzeugen des Puffers (leer);
    Erzeugen der Sperre l fuer den Puffer;
    Erzeugen und Starten der Threads;

// Version mit Bedingungsvariablen
einreihen(puffer, datenblock):
    belegen(l);
    solange laenge(puffer) < N wiederhole:
```

```
        wait(nichtvoll);
    stock(puffer, datenblock);
    signalAll(nichtleer);
    falls laenge(puffer) < N, dann:
        signalAll(nichtvoll);
    freigeben(l);

abholen(puffer, datenblock):
    belegen(l);
    solange laenge(puffer) > 0 wiederhole:
        wait(nichtleer);
    datenblock := top(puffer);
    pop(puffer);
    signalAll(nichtvoll);
    falls laenge(puffer) > 0, dann:
        signalAll(nichtleer);
    freigeben(l);
```

3.7 Wiederbetretbare Sperren

Wiederbetretbare Sperre (engl. reentrant lock) Ein Thread darf die Sperre mehrfach erwerben.

Zweck: Innerhalb eines kritischen Bereiches darf man eine Prozedur aufrufen, die wieder einen kritischen Bereich für dieselbe gemeinsame Variable enthält. ⇒ bequemere Programmierung.
Die inneren kritischen Bereiche sollen dazu wirkungslos sein.

Beispiel:

```
belegen(l);
S1;
belegen(l);
S2;
freigeben(l);
S3;
freigeben(l);
// Von S1 bis S3 kritischer Bereich. Innere belegen(l) und freigeben(l)
    wirkungslos.
```

Bemerkung: Wiederbetretbare Sperren und Semaphore sind inkompatibel zueinander.

3.8 Leser/Schreiber-Problem

Mehrere Threads greifen lesend oder schreibend auf eine gemeinsame Variable zu (Courtois et al, 1971). Mehrere Threads können lesend auf die gemeinsame Variable zugreifen, ohne sich gegenseitig zu stören.

Lese/Schreib-Konflikt Eine gemeinsame Variable darf nicht gleichzeitig gelesen und geschrieben werden.

Schreib/Schreib-Konflikt Eine gemeinsame Variable darf nicht von mehreren Threads gleichzeitig geschrieben werden.

2 Varianten:

1. Ein Leser muss nur dann warten, wenn gerade ein Schreiber aktiv ist: Leser haben Vorrang.
2. Ein Schreiber muss nur auf Leser und Schreiber warten, die gerade aktiv sind: Schreiber haben Vorrang.

Wenn (1) und (2) abwechselnd verwendet werden, bekommt man Fairness.

		Leser	
		0	≥ 1
Schreiber	0	i. O.	i. O.
	1	i. O.	LS
	≥ 2	SS	LS + SS

4 Implementierung

4.1 Atomare Maschinenbefehle

Wie werden Sperren implementiert?

Naiver Versuch

```
belegen(l):
    Solange !l.frei gilt, wiederhole:     (1)
        warte einen Augenblick;           (2)
    setze l.frei = false;                 (3)
```

Beispielablauf für 2 Threads, die versuchen, belegen(l) aufzurufen. Sei zu Beginn l.frei = true.

P_1	P_2	l.frei
(1)		
	(1)	
(3)		
	(3)	

→ Beide Threads sind im kritischen Bereich!
⇒ (1)(2)(3) muss selber wieder ein kritischer Bereich sein.

Man benötigt einen speziellen Maschinenbefehl, z.B.:

```
getAndSet(c, b, v):
    b := c;
    c := v;
```

Zwei Ausführungen dieses Maschinenbefehls müssen immer unter gegenseitigem Ausschlusds stattfinden, d.h. der Maschinenbefehl muss atomar (komplett und unteilbar) sein. Die Hardware muss dafür sorgen („Arbitrierung“).

Implementierung mit getAndSet

```
belegen(l):
    boolean b;
    getAndSet(l.frei, b, false);
    solange !b wiederhole:
        warte einen Augenblick;
        getAndSet(l.frei, b, false);

freigeben(l):
    boolean b;
    getAndSet(l.frei, b, true);
```

Alternativen:

```
getAndInc(c, b):
    b := c;
    c := c + 1;
getAndDec(c, b):
    b := c;
    c := c - 1;
compareAndSet(c, e, v, b): // b := Wahrheitswert
    Falls c = e, dann:
        c := v;
        b := true;
    Sonst:
        b := false;
```

(Befehl CMPXCHG (compare exchange) auf Intel Pentium)

Implementierung mit compareAndSet

```
belegen(l):
    boolean b;
    compareAndSet(l.frei, true, false, b);
    Solange !b wiederhole:
        warte einen Augenblick;
        compareAndSet(l.frei, true, false, b);

freigeben(l):
    boolean b;
    compareAndSet(l.frei, false, true, b);
```

Volatile (engl. für flüchtig), Schlüsselwort in Java.
Beispiel:

```
volatile int x;
```

x ist damit als gemeinsame Variable gekennzeichnet. Übliche Optimierungen des Compilers für lokale Variablen sind ausgeschlossen. Lese- und Schreibzugriffe auf x sind zueinander atomar („atomares Register“). Die Hardware sorgt für Atomarität.

4.2 Konsenszahlen

Konsensproblem:

1. Im Hauptprogramm: init(c); Gemeinsame Variable c wird initialisiert.
2. Jeder Thread ruft höchstens ein Mal entscheide(c, v, a) auf. (c: gemeinsame Variable, v: Vorschlag vom Typ T, a: Variable vom Typ T)
3. Der Aufruf entscheide(c, v, a) gibt an a einen Wert mit folgenden Eigenschaften:
   - Einigkeit: Jeder Thread bekommt denselben Wert von a
   - Gültigkeit: Der Wert in a wurde von mindestens einem Thread vorgeschlagen.

n-Konsensproblem Konsensproblem mit n beteiligten Threads

Es gilt:

- Mit dem n-Konsensproblem löst man auch das k-Konsensproblem für jedes k $<$ n.
- Das 1-Konsensproblem ist trivial löstbar: a := v implementiert entscheide(c, v, a).

Die Konsenszahl für eine Klasse (Sprache) K ist definiert als:

$$\begin{cases} \infty & \text{falls K das n-Konsensproblem für alle } n \in \mathbb{N} \text{ löst} \\ n & \text{falls } n \in \mathbb{N} \text{ maximal, sodass K das n-Konsensproblem löst} \end{cases} \tag{4.1}$$

Satz: Herlihy, 1991:

Klasse K	Konsenszahl K(K)
atomare Register	1
Warteschlangen	2
Common-2-Operationen	2
compareAndSet	∞

n-Konsens mit *compareAndSet* und *get*:
(Einfaches Konsensproblem: Jeder schlägt sich selber vor)

```
init(c):
    Setze c = -1.

entscheide(c, i, a): // mit i: Thread-ID des Aufrufers
    boolean b;
    compareAndSet(c, -1, i, b);
    Falls b gilt, dann:
        a := i;
    Sonst:
        a := get(c); // Kann auch ohne Fallunterscheidung
                     // angewandt werden, da fuer b true gilt:
                     // i == get(c)
```

Read/Modify/Write-Operation:

rmw(c, b, f): (mit c ist gemeinsame Variable mit Wert vom Typ T, b ist Ergebnisvariable mit Wert von Typ T und f ist Modifikationsfunktion f: T $\rightarrow$ T).
b := c;
c := f(c);

Es gilt:
getAndSet(c, b, v) = rmw(c, b, λx . v)
getAndInc(c, b) = rmw(c, b, λx . x + 1)

Schar F von Funktionen von T nach T heißt *Common2*, falls:

$$f(g(x)) = f(x) \text{ (f absorbiert g) oder} \tag{4.2}$$
$$g(f(x)) = g(x) \text{ oder} \tag{4.3}$$
$$f(g(x) = g(f(x)) \tag{4.4}$$

für alle $f, g \in F, x \in T$ (trivial für f = g).

F heißt *nicht-trivial*, falls $F \neq \{id\}$ mit F ist nichtleer, d.h. $F \backslash \{id\} \neq \emptyset$.

Beispiel: $F = \{\lambda x\ .\ x + 1, \lambda x\ .\ x - 1\} = \{s, p\}$.
Es gilt: $s(p(x) = x = p(s(x))$ für alle $x \in \mathbb{Z}$. Also ist F Common2. Damit Konsenszahl ≤ 2. Da F nicht-trivial, ist Konsenszahl = 2.

4.3 Zwischenspeicher

Zwischenspeicher (ZSP, engl. cache) schneller, kleiner Speicher auf dem Prozessorchip.

Bemerkung: Herkunft des Begriffs „cache“: Versteck der Beute eines Einbrechers.

Verwendung:
Nachdem der Prozessor das erste Mal auf eine gewisse Arbeitsspeicherzelle lesend zugegriffen hat, speichert er den Wert in seinem ZSP. Wenn er das nächste Mal lesend auf dieselbe Adresse zugreifen will, findet er das Ergebnis in seinem ZSP („Treffer“, engl. match). Er braucht dazu nicht auf den BUS zuzugreifen.
Um schreibend auf eine Arbeitsspeicherzelle zuzugreifen, speichert der Prozessor das Wort zunächst in seinen ZSP. Nur wenn ein anderer Prozessor auf dieselbe Speicherzelle lesend zugreifen will, muss das Wort in den Arbeitsspeicher geschrieben werden.

Vorteil des ZSP:

Weniger Zugriffe auf den Arbeitsspeicher nötig, damit schneller und der BUS ist weniger belastet.

Der ZSP lohnt sich, wenn im Programm häufig dicht hintereinander Zugriffe auf dieselbe Adresse vorkommen („Lokalität“).
Um den Verwaltungsaufwand gering zu halten, ist der ZSP in sogenannte *Speicherzeilen* (engl. cache lines) organisiert. Sobald der ZSP voll ist, wird es nötig, manche Zeilen auszuwerfen (engl. to evict) um Platz zu schaffen.

Kohärenz Jeder Lesezugriff auf den ZSP liefert den zuletzt geschriebenen Wert.

Kohärenz bedeutet praktisch, dass sich durch die Einführung des ZSP nichts am Verhalten des Systems ändert.
Um Kohärenz zu erreichen, verwendet man ein Kohärenz-Protokoll, z.B. das MESI-Protokoll.

MESI-Protkoll:

Jede Speicherzeile hat einen Modus:

Modified Zeile wurde verändert. Kein anderer Prozessor hat diese Zeile in seinem ZSP.

Exclusive Zeile ist unverändert. Kein anderer Prozessor hat diese Zeile in seinem ZSP.

Shared Zeile ist unverändert. Andere Prozessoren können diese Zeile in ihrem ZSP haben.

Invalid Zeile enthält eine verwertbaren Daten.

Beispiel-Ablauf:
A, B, C seien Prozessoren, M sei ein Arbeitsspeicherblock.

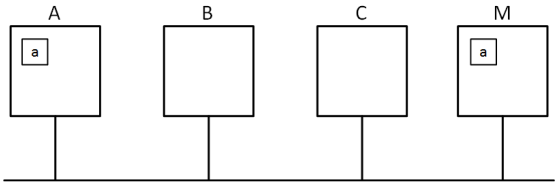


Abbildung 4.1: A liest Adresse von a.

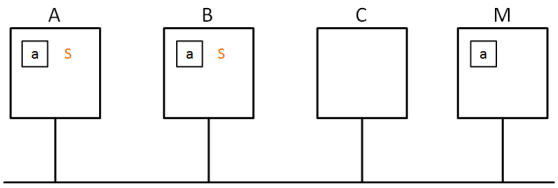


Abbildung 4.2: B liest Adresse von a; A antwortet.

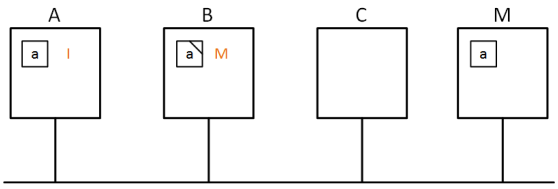


Abbildung 4.3: B schreibt auf Adresse a und informiert alle darüber.

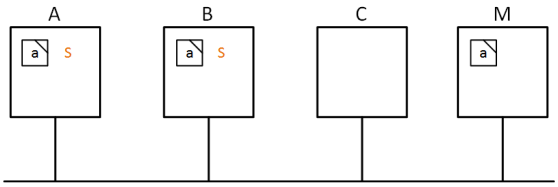


Abbildung 4.4: A liest von Adresse a, das führt zu Anfrage an alle, B sendet Daten.

False Sharing gemeinsame Speicherzelle, obwohl sich die Daten darin nicht überlappen

Im ZSP von B:

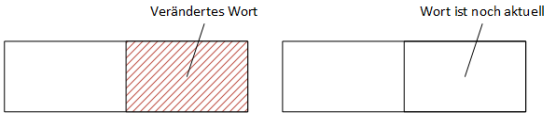


Abbildung 4.5: Speicherzeile für Adresse a.

False Sharing führt unnötig häufig zu Modus I.

Daten, die nebeneinander verwendet werden, sollten in verschiedenen Speicherzeilen liegen.

Verhalten mit getAndSet:
getAndSet(c, b, true)
Dabei ausgeführte Aktionen:

1. c lesen
2. b schreiben
3. c schreiben

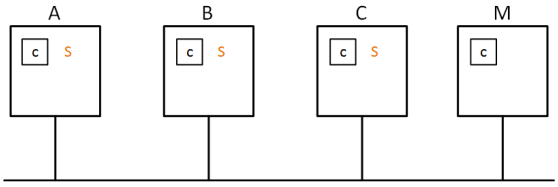


Abbildung 4.6: Zustand vorher.

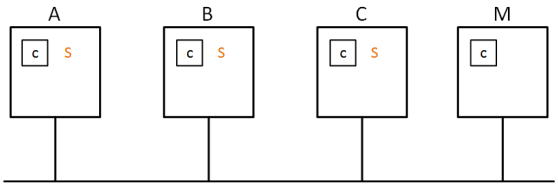


Abbildung 4.7: A führt (1) aus, Wert von c in ZSP von A ist bereits aktuell, keine Änderung.

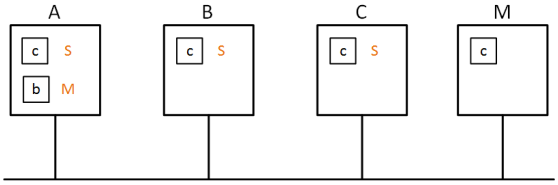


Abbildung 4.8: A führt (2) aus.

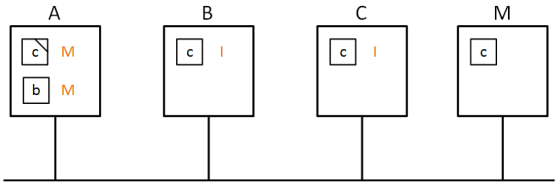


Abbildung 4.9: A führt (3) aus.

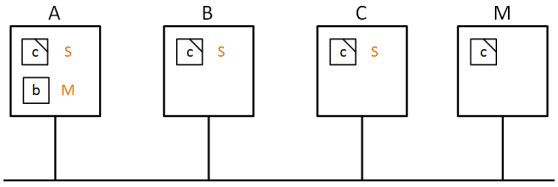


Abbildung 4.10: B führt (1) aus.

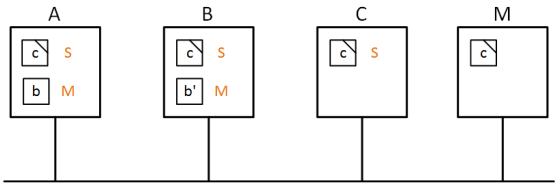


Abbildung 4.11: B führt (2) aus.

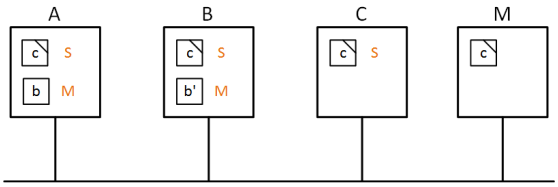


Abbildung 4.12: B führt (3) aus, keine Änderung, denn der Wert in c verändert sich nicht dabei.

Alle getAndSet-Aufrufe der Warteschleife können ohne BUS-Zugriff abgearbeitet werden.

4.4 Bäckerei-Algorithmus

Bäckerei-Algorithmus (engl bakery algorithm) von Leslie Lamport 1974 publiziert; Implementierung von Sperren mit atomaren Registern.

Analogie: Jeder, der (in Amerika) eine Bäckerei betritt, zieht zuerst eine laufende Nummer. Der Kunde mit der niedrigsten Nummer wird als nächste bedient.

Pseudocode mit einer Sperre:

```
Typ Thread ID = {0, ..., n - 1}; // n Threads
volatile flag: boolean[ThreadID]; // initialisiert mit false
volatile label: long[ThreadID]; // initialisiert mit 0

Prozedur belegen():
    int i := Nummer des aufrufenden Threads;
    flag[i] := true;
    label[i] := max(label[0], ..., label[n - 1]) + 1;
    Warte solange  ∃k ≠ i : flag[k] ∧ (label[k], k) <lex (label[i], i)

Prozedur freigeben():
    flag[Nummer des aufrufenden Threads] := false;
```

Behauptung: Der Bäckerei-Algorithmus hat die Fortschritt-Eigenschaft.
Beweis: Der Thread i mit dem kleinsten Paar (label[i], i) wartet nicht. Es gibt so ein i, denn $<_{\text{lex}}$ ist eine Wohlordnung. Damit hat jede nicht-leere Menge ein kleinstes Element.

Behauptung: Der Bäckerei-Algorithmus ist FCFS (First Come First Serve).
Beweis: Falls Thread i den Torweg verlässt, bevor Thread j ihn betritt, dann gilt:

$$\begin{aligned} &w_i\,(label[i], v) \rightarrow \\ &r_j\,(label[i], v) \rightarrow \\ &w_j\left(label[j], v'\right) \text{ mit } v < v' \\ &r_j\,(flag[i], true) \end{aligned}$$

Dabei bedeutet $w_i\,(label[i], v)$: Schreibzugriff von Thread i auf die Variable $label[i]$; der geschriebene Wert ist v.
Es gilt $flag[i] \wedge (flat[i], i) <_{\text{lex}} (label[j], j)$.
Aus Fortschritt und FCFS folgt Fairness.

Behauptung: Der Bäckerei-Algorithmus erfüllt gegenseitigen Ausschluss.
Beweis: Durch Widerspruch (grundsätzliche Methode: Man behauptet, zwei Threads seien simulatan im kritischen Bereich. Herbeiführung von Widerspruch). Angenommen Threads i und j sind nebeneinander im kritischen Bereich. O.B.d.A. gilt: $(label[i], i) <_{\text{lex}} (label[j], j)$. Sobald Thread j die Warteschleife verlassen hat, gilt:

$$flag[i] = false \quad (1)$$

oder

$$(label[j], j) <_{\text{lex}} (label[i], i) \quad (2)$$

Die Werte von i und j sind fest. Der Wert von label[j] ändert sich nicht mehr bis zum Betreten des kritischen Bereichs. Der Wert von label[i] kann höchstens größer werden. Wenn also (2) beim Verlassen der Warteschleife gilt, dann auch im kritischen Bereich. Widerspruch!

Also gilt (1). Deswegen

$$
\begin{aligned}
&r_j\,(\text{label}[i], \_) \rightarrow \ (\_\text{: gelesener Wert ist irrelevant})\\
&w_j\,(\text{label}[j], v) \rightarrow\\
&r_j\,(\text{flag}[i], false) \rightarrow\\
&w_i\,(\text{flag}[i], true) \rightarrow\\
&r_i\,(\text{label}[j], v) \rightarrow\\
&w_i\,\left(\text{label}[i], v'\right)
\end{aligned}
$$

mit $v < v'$, also label$[j]$ < label$[i]$. Widerspruch!

Nachteil: Falls nur atomare Lese- und Schreib-Operationen zur Verfügung stehen („atomare Register“), sind für n Threads Lese- und Schreibzugriffe auf mindestens n Speicherzellen notwendig (Burns/Lynch 1993).
Grund: Jeder Thread benötigt eine Speicherzelle, auf die nur er schreibt. Sonst kann ein anderer Thread das überschreiben, was ein anderer geschrieben hat.
Es muss mindestens n + 1 unterscheidbare Zustände geben:

1. kein Thread befindet sich im kritischen Bereich
2. Thread i befindet sich im kritischen Bereich

5 Feinkörnige Nebenläufigkeit

Problem: Bei einem hohen Grad an Nebenläufigkeit wird der Zugriff auf das gemeinsame Objekt zum Flaschenhals. Die Threads können nur nacheinander zugreifen.
Abhilfe: 4 Techniken.

1. **Feinkörnige Synchronisation:** Statt das Objekt zu sperren, werden nur die betroffenen Komponenten gesperrt. Nur wenn zwei Threads auf dieselbe Komponente zugreifen wollen, muss einer warten.
2. **Optimistische Synchronisation:** Während der Suche nach einer bestimmten Komponente des Objekts werden keine Sperren erworben. Sobald die Komponente gefunden wurde, diese Komponente sperren und überprüfen, ob sich der Kontext inzwischen verändert hat.
3. **Faule Synchronisation:** Löschen einer Komponente wird in zwei Phasen durchgeführt:
   a) Als gelöscht markieren, z.B. durch Setzen eines gewissen Bits („logisches Löschen“).
   b) Aus der Datenstruktur aushängen („physikalisches Löschen“).
4. **Nicht-blockierende Synchronisation:** Statt Sperren werden atomare Operationen verwendet.

5.1 Beispiel: Mengen implementiert durch verkettete Listen

Mengen-Schnittstelle:

```
Typ Set<T>
Methoden
    add(T x) // x zu Menge this hinzufuegen
    remove(T x) // x aus Menge this entfernen
    containes(T x) // Wahrheitswert von "this enthaelt x"
```

Implementierung von Mengen durch verkettete Listen mit Wächtern, sortiert nach Streuwert.

Listenelement (Typ Node<T>) habe folgende Attribute:

T item das Element der Menge

int key der Streuwert des Elements

Node<T> next Zeiger auf das nächste Element

Wächter (engl. Sentinel) „künstliches“ Listenelement, das Anfang oder Ende der Liste markiert.

Beispiel:

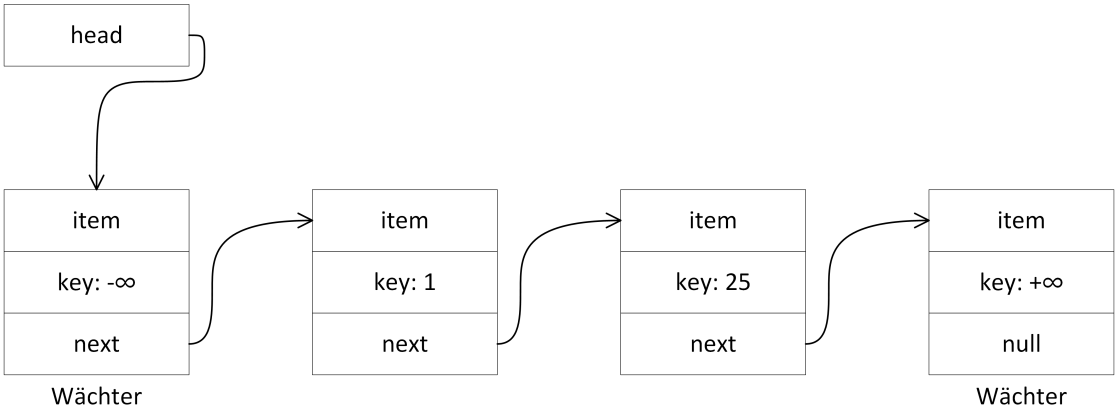


Abbildung 5.1: Visualisierung einer LinkedList.

Die Streuwerte sind sortiert: $-\infty \leq 1 \leq 25 \leq +\infty$.

Klasse List<T> mit Attribut head vom Typ Node<T>.
Invariante: Wächter werden weder hinzugefügt noch gelöscht. Die Listenelemente sind nach Streuwert sortiert.

5.2 Implementierung mit Feinkörniger Synchronisation

In Java:

```
class List<T> {
    ...
    public boolean add(T x) {
        int k = x.hashCode(); // Streuwert
        Node<T> pred, curr;
        try {
            pred = this.head;
            curr = pred.next;
            pred.lock(); // Assume implementation
            curr.lock();
            while (curr.key < k) {
                pred.unlock();
                pred = curr;
                curr = pred.next;
                curr.lock();
            }
            if (curr.key != k) {
                Node<T> node = new Node<>(x);
                node.next = curr;
                pred.next = node;
            }
        } finally {
            curr.unlock();
            pred.unlock();
        }
    }
}
```

Methoden remove und contains ähnlich.

Eine Sperre genügt nicht. Beispiel dazu:

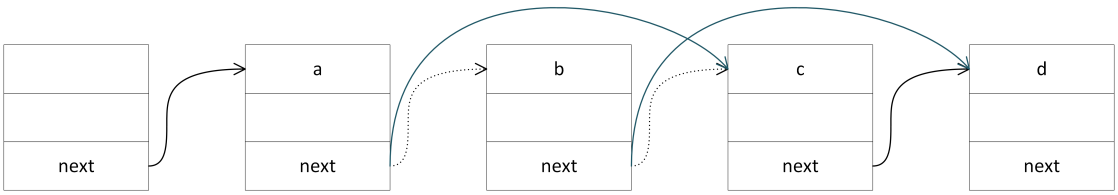


Abbildung 5.2: Visualisierung einer LinkedList.

Thread 1 will b löschen:

- Sperre in a erwerben
- a.next auf c setzen

Thread 2 will c löschen:

- Sperre in b erwerben
- b.next auf d setzen

Wirkung: Nur b wird gelöscht!

Um b zu löschen, muss die Sperre in a und in b erworben werden, und entsprechend um c zu löschen, muss die Sperre in b und in c erworben werden. ⇒ Konflikt!

5.3 Implementierung mit optimistischer Synchronisation

```
class OptList<T> { // (Java)
    ...
    public boolean add(T x) {
        int k = x.hashCode();
        Node<T> pred, curr;
        boolean done = false;
        while (!done) {
            pred = this.head;
            curr = pred.next;
            while (curr.key < k) {
                pred = curr;
                curr = pred.next;
            }
            try {
                pred.lock();
                curr.lock();
                if (this.validate(pred, curr)) {
                    if (curr.key != k) {
                        Node<T> node = new Node<>(x);
                        node.next = curr;
                        pred.next = node;
                    }
                    done = true;
                }
            } finally {
                curr.unlock();
                pred.unlock();
            }
        }
    }

    private boolean validate(Node<T> pred, Node<T> curr) {
        Node<T> node = this.head;
        while (node.key < pred.key) {
            node = node.next;
        }
        return node == pred && curr = node.next;
    }
}
```

Diskussion: Optimistisches Synchronisation lohnt sich, wenn zweimaliges Durchlaufen der Liste billiger ist als einmaliges Durchlaufen mit Setzen von Sperren. Belegen und Freigeben sind aufwendig.

5.4 Implementierung mit fauler Synchronisation

Listenelemente bekommen ein neues Attribut: *marked* drückt aus, dass es zum Löschen markiert wurde.

Neue Version von validate:

```
private boolean validate(Node<T> pred, Node<T> curr) {
    return !pred.marked && !curr.marked && pred.next == curr;
}
```

Löschen:

```
public boolean remove(T item) {
    int k = item.hashCode();
    boolean done = false;
    boolean erg = false;
    while (!done) {
        Node<T> pred = this.head;
        Node<T> curr = pred.next;
        while (curr.key < k) {
            pred = curr;
            curr = pred.next;
        }
        pred.lock();
        try {
            curr.lock();
            try { // falls erste Sperre geht, aber zweite nicht
                if (validate(pred, curr)) {
                    if (curr.key == j) {
                        curr.marked = true;
                        pred.next = curr.next;
                        erg = true;
                    }
                    done = true;
                }
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }
    return erg;
}
```

Contains:

```
public boolean contains(T item) {
    int k = item.hashCode();
    Node<T> curr = this.head;
    while (curr.key < k) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
```

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.