Mutual Exclusion

Annika Külzer
Florian Störkle
Dennis Herzner

Gliederung

Eine kleine Geschichte

Alice und Bob sind Nachbarn, die sich einen Garten teilen, in dem sich Bobs Hund und die Katze von Alice austoben dürfen.

Was passiert erfahrungsgemäß, wenn Katze und Hund gemeinsam im Garten sind?

Eine kleine Geschichte

Quellen:
http://www.graphicsfactory.com/clip-art/image_files/image/8/592878-Animals_ss_c_cartoon010.gif
http://www.jokesduniya.com/wp-content/uploads/2007/03/anygrycat.jpg

Die Moral von der Geschicht’

Alice und Bob müssen also eine Lösung finden, die sicherstellt, dass sich ihre Haustiere niemals gleichzeitig im Garten befinden.

Dieses Prinzip nennt man **Mutual Exclusion**.

Kritischer Bereich: Ein Code-Abschnitt, in dem gemeinsam genutzte Ressourcen verändert werden und der nicht parallel oder verzahnt ausgeführt werden darf.

- kritischer Bereich = Garten
- Threads = Haustiere

Das eigentliche Problem

- Dateninkonsistenz durch unkontrolliertes Lesen und Schreiben
  → fehlerhaftes Verhalten von Software
- allgegenwärtig von CPU Register bis Webanwendung
- Koordination von nebenläufigen Prozessen/Threads
- Sicherstellung von Korrektheit

Mutual Exclusion - Definition

Wenn sichergestellt ist, dass sich zu jedem Zeitpunkt maximal ein einziger Thread im kritischen Bereich befinden kann, so bezeichnet man dies als **Mutual Exclusion**.

→ Verhinderung von Dateninkonsistenz gemeinsam genutzter Datenstrukturen

Lock-Algorithmen

```
1   /**
2    * Zum Schutz des kritischen Bereiches. Jeder kritische Bereich muss
3    * sein eigenes Lock besitzen.
4    */
5   public interface Lock {
6
7       /**
8        * Wird von einem Thread vor dem Eintritt in den kritischen Bereich
9        * aufgerufen.
10       *
11       * Der Thread verweilt hier so lange, bis der Bereich wieder frei
12       * wird (falls besetzt) und er den kritischen Bereich betreten kann.
13       */
14      public void lock();
15
16
17      /**
18       * Wird bei Verlassen des kritischen Bereiches aufgerufen und
19       * gibt diesen wieder frei.
20       */
21      public void unlock();
22  }
```

Lock-Algorithmen: Anforderungen

Was muss für die Threads gelten, die auf das Lock zugreifen?
Welche Anforderungen und welche Eigenschaften sind zu beachten?

- Mutual Exclusion
- Deadlock-Freiheit (Progress)
- Kein Verhungern (Bounded Waiting)
- Fairness

Verwendung eines Locks

```
1     /**
2      * Der Zähler ohne Lock.
3      */
4     public class Counter {
5           private int value;
6
7           public Counter(int c) {
8                 this.value = c;
9           }
10
11          /**
12           * Inkrementiert den Wert des Zählers
13           * und liefert den Wert vor
14           * der Inkrementierung.
15           * @return Wert vor der Inkrementierung
16           */
17          public int getAndIncrement() {
18                // <<kritischer Bereich>>
19                int temp = this.value;
20                this.value = temp + 1;
21                // <<krit. Bereich>>
22
23                return temp;
24          }
25    }
```

Verwendung eines Locks

```java
/**
 * Der Zähler ohne Lock.
 */
public class Counter {
      private int value;

      public Counter(int c) {
            this.value = c;
      }

      /**
       * Inkrementiert den Wert des Zählers
       * und liefert den Wert vor
       * der Inkrementierung.
       * @return Wert vor der Inkrementierung
       */
      public int getAndIncrement() {
            // <<kritischer Bereich>>
            int temp = this.value;
            this.value = temp + 1;
            // <<krit. Bereich>>

            return temp;
      }
}
```

```java
/**
 * Der Zähler mit Lock.
 */
public class CounterWithLock {
      private int value;
      private Lock lock;

      /**
       * Inkrementiert den Wert des Zählers und
       * liefert den Wert vor der Inkrementierung.
       * @return Wert vor der Inkrementierung
       */
      public int getAndIncrement() {
            lock.lock();
            try {
                  //kritischen Bereich betreten
                  int temp = this.value;
                  this.value = temp + 1;
                  return temp;
            } finally {
                  //kritischen Bereich verlassen
                  lock.unlock();
            }
      }}
```

LockOne

Wir versuchen **zwei** Threads zu synchronisieren.

Idee:
A sagt: Ich will (raise the flag)!

A wartet, falls B will und wenn, dann nur so lange bis B nicht mehr will.

```
1   public class LockOne implements Lock {
2       private boolean[] flag = new boolean[2];
3
4       public void lock() {
5           // me ist entweder 0 oder 1
6           int me = ThreadID.get();
7           // other ist entweder 0 oder 1
8           int other = 1 - me;
9           flag[me] = true; // ich will!
10          while (flag[other]) {} // warten
11      }
12
13      public void unlock() {
14          // ich bin fertig!
15          flag[ThreadID.get()] = false;
16      }
17  }
```

LockOne - Analyse

- Mutual Exclusion erfüllt
- Deadlock-Freiheit nicht erfüllt
- Kein Verhungern nicht erfüllt

LockTwo

Idee:

A sagt zu B: Ich warte solange hier, bis du mich hineinlässt.

```
1   public class LockTwo implements Lock {
2       private int victim;
3
4       public void lock() {
5           int me = ThreadID.get();
6           this.victim = me;
7           while (this.victim == me) {
8           }
9       }
10
11      public void unlock() {
12      }
13  }
```

LockTwo: Analyse

- Mutual Exclusion erfüllt (nachfolgender Beweis)
- Deadlock-freiheit nicht erfüllt
- Kein Verhungern nicht erfüllt

LockTwo funktioniert nur sinnvoll, wenn die beiden Threads nebenläufig ausgeführt werden.

Einschub: Notation

Wie beweist man das überhaupt?

Notation:
$a \rightarrow b$:= "a *kommt vor* b"
a^j := die j-te Wiederholung von a

a kann sein:
Event zu einem Zeitpunkt oder ein Intervall zwischen zwei Events

$write_A(x{=}5)$:= Thread A überschreibt x mit dem Wert 5
$read_B(x{=}{=}7)$:= Thread B liest den Wert 7 aus x

Einschub: Formale Definition von Mutex

Formale Definition von Mutex: Gegeben seien zwei Threads A und B und $\forall j, k \in \mathbb{N}$ gilt $CS_B^j \rightarrow CS_A^k$ oder $CS_A^k \rightarrow CS_B^j$.

Informell: Man möchte zeigen, dass sich diese Zeitintervalle nicht überlappen und sich somit nicht zwei Threads gleichzeitig im kritischen Bereich aufhalten können.
Falls die Relation "$\longrightarrow$" nicht gilt, so bedeutet dies, dass sich die Intervalle überlappen.

LockTwo: Beweis von Mutex

Zu zeigen: LockTwo erfüllt Mutex.

Beweis durch Widerspruch: Mutex sei nicht erfüllt.
D. h. es gibt $j, k \in \mathbb{N}$, sodass $CS_A^j \not\to CS_B^k$ oder $CS_B^k \not\to CS_A^j$.

Wir betrachten die letzte Ausführung der `lock()`-Methode in jedem Thread, bevor dieser zum k-ten bzw. j-ten Mal den kritischen Bereich betritt. Mit Blick auf den Code erhält man folgende Gleichungen

(1) $write_A(victim{=}A) \rightarrow read_A(victim{==}B) \rightarrow CS_A$

(2) $write_B(victim{=}B) \rightarrow read_B(victim{==}A) \rightarrow CS_B$

LockTwo: Beweis von Mutex

(1) $write_A(victim{=}A) \rightarrow read_A(victim{==}B) \rightarrow CS_A$
(2) $write_B(victim{=}B) \rightarrow read_B(victim{==}A) \rightarrow CS_B$

1. Thread A schreibt als erstes (victim=A).
2. Damit A in den kritischen Bereich darf, muss victim==B sein.
3. D. h. Thread B muss victim=B geschrieben haben.
4. Daraufhin können keine Schreiboperationen mehr folgen und wir erhalten folgende Gleichung:

$$write_A(victim{=}A) \rightarrow write_B(victim{=}B) \rightarrow read_A(victim{==}B)$$

Nach (2) muss $write_B(victim{=}B) \rightarrow read_B(victim{==}A)$ gelten.
Zwischen $write_B(victim{=}B)$ und $read_B(victim{==}A)$ müsste also noch eine Schreiboperation stattgefunden haben. → Widerspruch

Peterson Lock

Idee:
Kombination von LockOne und LockTwo.

A sagt zu B: Wenn du darfst und du willst, dann kannst du. Ansonsten gehe ich.

```
1   public class Peterson implements Lock{
2       private volatile boolean[] flag = new boolean[2];
3       private volatile int victim;
4
5       public void lock() {
6           int me = ThreadID.get();
7           int other = 1-me;
8           flag[me] = true; //ich würde gerne
9           victim = me;//nach dir :-)
10          while(flag[other] && victim == me){};//warten
11      }
12
13      public void unlock() {
14          flag[ThreadID.get()] = false;
15      }
16  }
```

Peterson Lock: Analyse

- Mutual Exclusion erfüllt
- Deadlock-freiheit erfüllt
- Kein Verhungern erfüllt

Nachteil: nur auf zwei Threads anwendbar → nicht praktikabel

Lamport’s Bakery Algorithmus

Wir erweitern das Lock für den Einsatz mit **n** Threads.

Idee:
Ein Thread signalisiert, dass er den kritischen Abschnitt betreten möchte, zieht eine Wartemarke ("Timestamp") und wartet solange bis keine Threads mehr vor ihm in der Schlange stehen.

Lamport's Bakery Algorithmus: Code

```
1   public class Bakery implements Lock{
2       boolean[] flag;
3       int[] label;
4
5       public Bakery(int n)
6       {
7           //alle flags false
8           //alle labels auf 0
9       }
10      public void lock() {
11          int me = ThreadID.get();
12          flag[me] = true; //ich will!
13          label[me] = max(label[0],...,label[n-1]) +1; //Ziehe Marke!
14          while((∃k != me)(flag[k] && (label[k], k)<<(label[me], me))){}
15      }
16
17      public void unlock() {
18          flag[ThreadID.get()] = false; //ich bin fertig!
19      }
20  }
```

Lamport’s Bakery Algorithmus: Analyse

- Mutual Exclusion erfüllt
- Deadlock-freiheit erfüllt
- Kein Verhungern erfüllt
- Fairness erfüllt

Ist doch alles perfekt?! Wo kann ein Problem auftreten?

Lamport's Bakery Algorithmus: Analyse

- Mutual Exclusion erfüllt
- Deadlock-freiheit erfüllt
- Kein Verhungern erfüllt
- Fairness erfüllt

Ist doch alles perfekt?! Wo kann ein Problem auftreten?

Beim Inkrementieren der Labels besteht die Gefahr eines Overflows.

Timestamps

Timestamps werden benötigt, um Fairness zwischen den wartenden Threads zu garantieren und stellen somit eine Ordnung unter diesen her. Fairness bedeutet hier also "first-come-first-served".

Beispiel:
Bei Bakery fungieren die Labels (Wartemarken) als "timestamps".

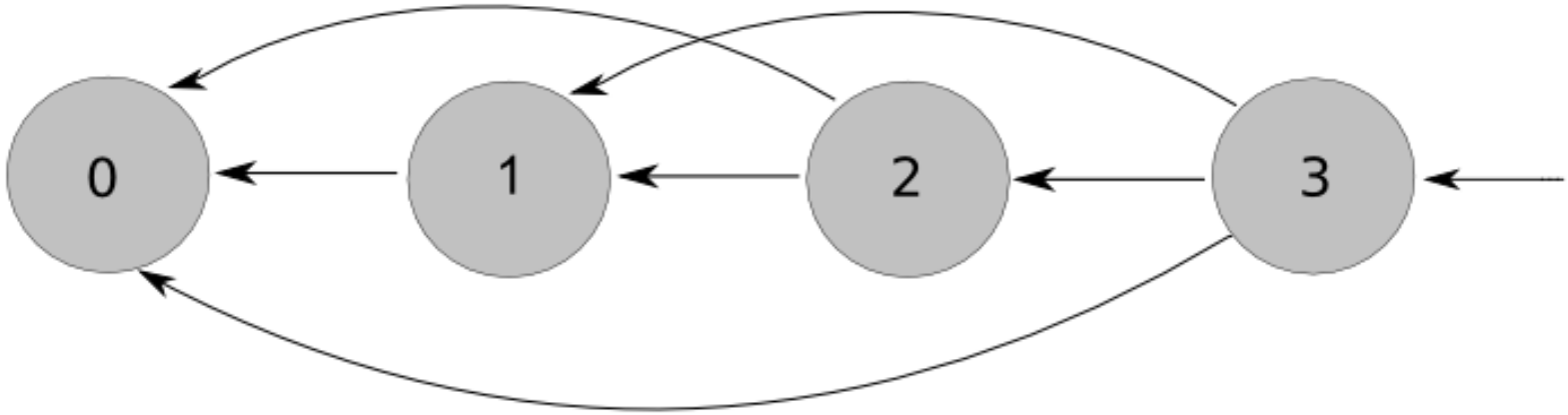
Ein sogenanntes **Timestamping System** besteht aus zwei Methoden:

`label()` - weist einem Thread einen neuen Timestamp zu, der "größer" als alle bereits zugewiesenen Timestamps ist.

`scan()` - liest die Timestamps der anderen Threads aus.

Precedence Graph

- **precedence graph**: gerichteter Graph, bei dem die Kanten eine totale Ordnung unter den Knoten (=Timestamps) herstellen



Eine Kante von 1 nach 0 bedeutet: 1 ist ein jüngerer Timestamp als 0

Eigenschaften dieser Relation über Timestamps:

- irreflexiv (keine Kante von Knoten A nach A)
- antisymetrisch (wenn Kante von A nach B, dann keine Kante von B nach A)

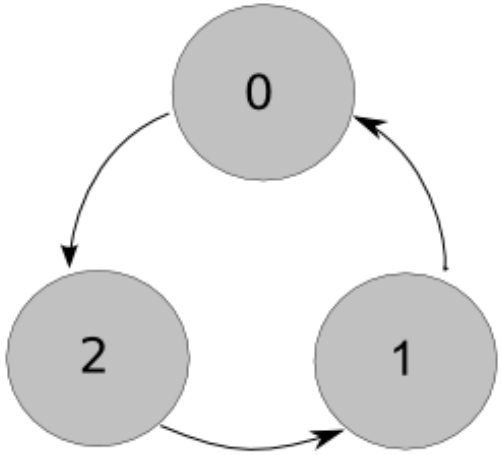
Bounded Timestamps

Lösung des Overflow-Problems beim Bakery-Algorithmus

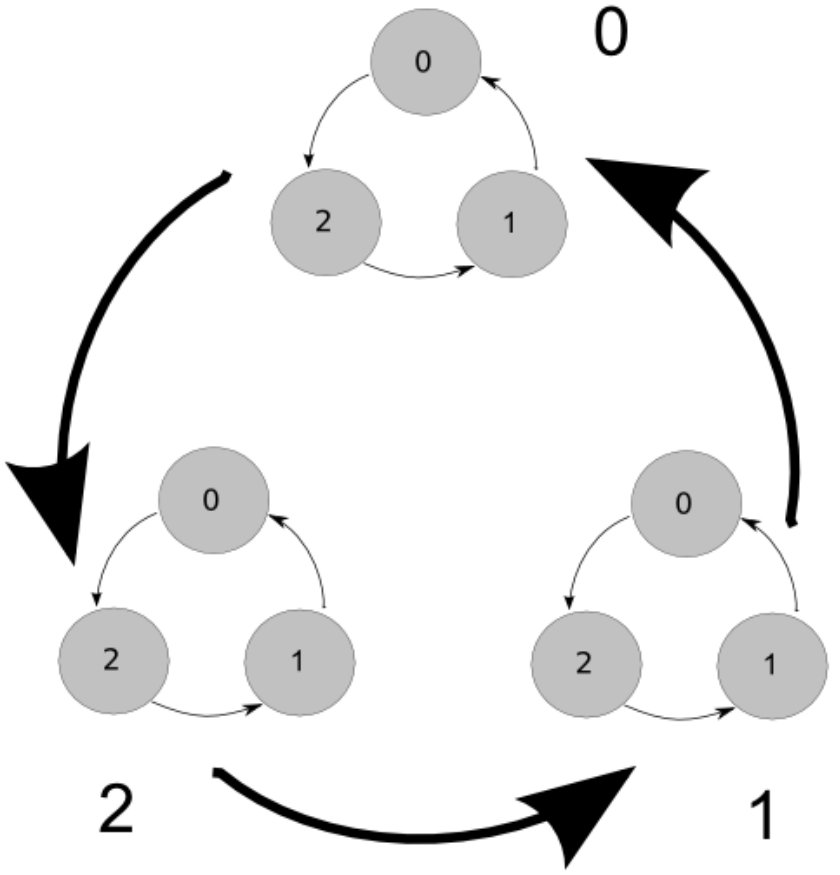
- beschränkte Anzahl an Timestamps
- dadurch kein Overflow möglich

Bounded Timestamps

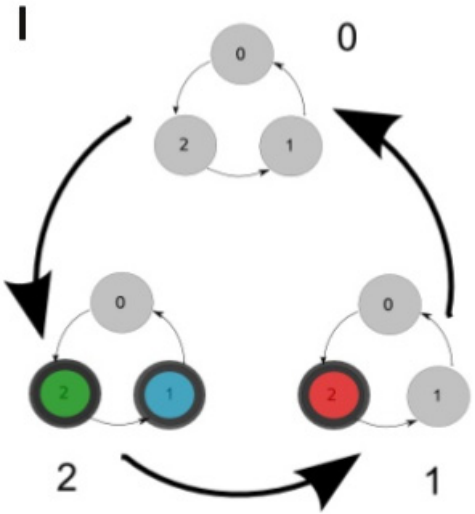
für 2 Threads

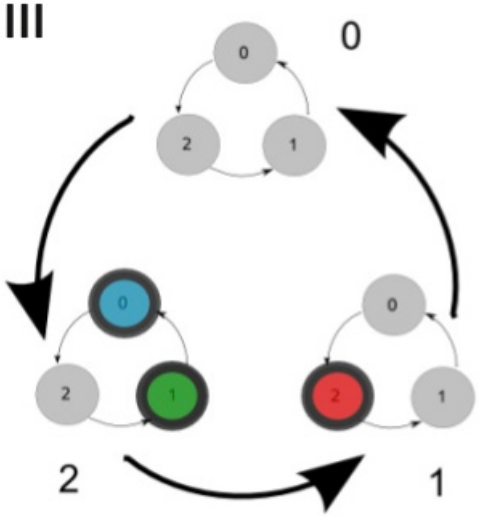


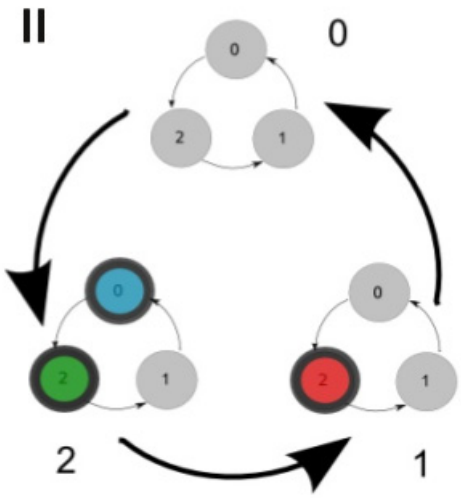
für 3 Threads

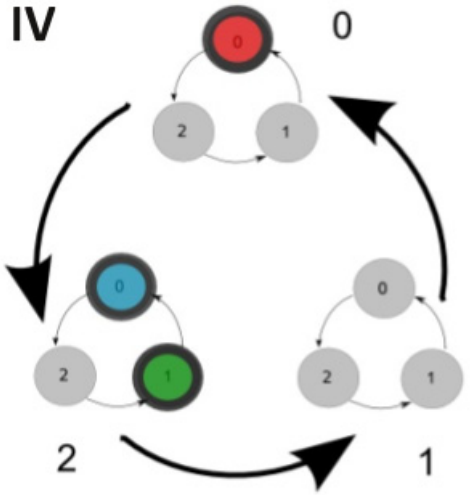


Bounded Timestamps: Beispiel









Bounded Timestamps: Fazit

- Verwaltungsaufwand schon für relativ kleine n recht hoch.
- Speicherplatzbedarf $O(3^{n-1})$

Minimale Zahl an Speicherplätzen

Noch einmal zurück zum Bakery-Algorithmus:
Er ist elegant, fair und knapp.

Aber: Man braucht so viel Speicherplätze wie Threads.

Frage:
Lässt sich dieser Overhead vermeiden?

Anwort:
Nein, jeder Deadlock-freie Lockalgorithmus für n Threads muss mindestens n Speicherplätze zum Lesen und Schreiben reservieren.

Minimale Zahl an Speicherplätzen

Am Zustand eines Lock-Objekts muss erkennbar sein, ob sich ein Thread im kritischen Bereich befindet oder nicht. Ansonsten bezeichnet man den Zustand des Lock-Objekts als **inkonsistent**.

Daraus folgt, dass jeder Thread vor Eintritt in den kritischen Bereich mindestens einen Speicherplatz beschreiben muss.

Man kann zeigen, dass sich ein Deadlock- freier Lock Algorithmus **nie** in einem inkonsistenten Zustand befinden kann.

Minimale Zahl an Speicherplätzen

"Jeder Deadlock-freie Lockalgorithmus für n Threads muss mindestens n Speicherplätze zum Lesen und Schreiben reservieren."

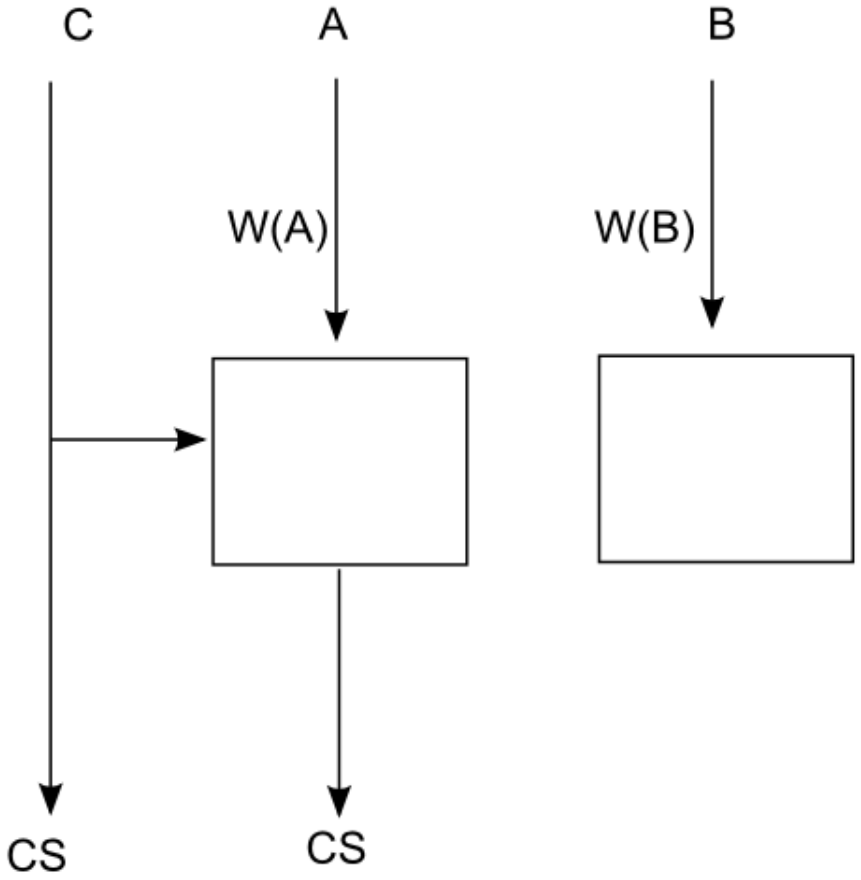
Diese Aussage wollen wir im Folgenden kurz illustrieren.

Der komplette formale Beweis kann im Buch nachgeschlagen werden.

Minimale Zahl an Speicherplätzen

Angenommen: Es gibt nur zwei Speicherplätze für drei Threads.

2. C läuft und **könnte** alle Speicherplätz beschreiben und den kritischen Bereich betreten.



1. A und B sind kurz davor in das Lock zu schreiben (z.B. raise the flag). Das Lock ist unverändert.

3. Jetzt laufen A und B. Sie überschreiben das, was C geschrieben hat und einer von ihnen ***muss*** dann den kritischen Bereich betreten.
Widerspruch!

In a nutshell

1. Eine Hardware-nahe Umsetzung eines Lock-Algorithmus' ist aufgrund des hohen Speicherbedarfs wünschenswert.

2. Sehr schwer zu kontrollierende Lese-/Schreibeoperationen zeigen den Bedarf nach mächtigeren Synchronisationsoperationen auf, um diese als Basis von Mutex-Algorithmen zu nutzen.

3. Mutex bildet die Grundlage, um Nebenläufigkeit zu realisieren.

Quellen und weiterführende Literatur

Wikipedia
http://de.wikipedia.org/wiki/Kritischer_Abschnitt

Betriebsysteme-Skript WS08/09, Prof. Dr. Claudia Linnhoff-Popien
http://www2.mobile.ifi.lmu.de/Vorlesungen/ws0809/bs/skript/Betriebssysteme0809.2.pdf

The Art of Multiprocessor-Programming, Maurice Herlihy & Nir Shavit