Spin Locks & Contention

Proseminar Nebenläufige Programmierung

Janine Mischor

19.05.2010

Überblick

1. Mutual Exclusion
2. Peterson Algorithmus
3. AtomicBoolean Klasse
4. Locks
   I. Test-And-Set Locks
   II. Test-And-Test-And-Set Locks
   III. Exponential Backoff
   IV. Queue Locks
      a) Array Based
      b) CLH Queue Lock
   V. Hierarchical Locks
      a) Hierarchical CLH Queue Lock
5. Fazit

Mutual Exclusion

Mutual Exclusion

- Mutual Exclusion Problem

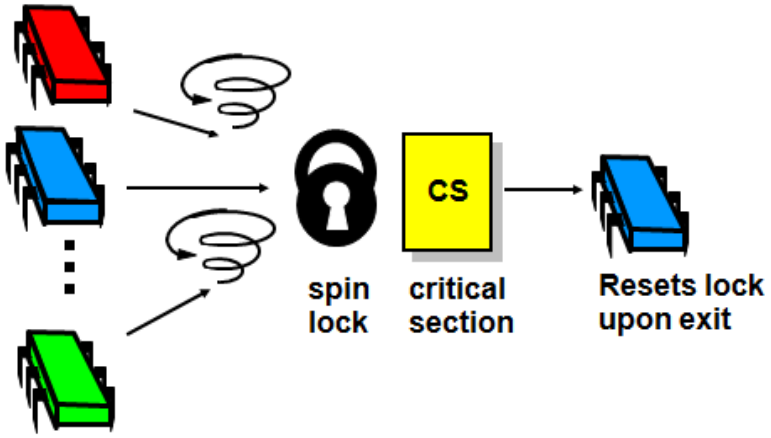
→ Was soll ein Thread machen, wenn er Lock nicht bekommt?

2 Alternativen:

- ***Suspendierung*** (= blocking)
  Scheduler bittet anderen Thread weiterarbeiten zu lassen
- ***Weiter versuchen*** → Lock wird „**spin lock**“ genannt
  „wiederholt versuchen“ = *spinning* oder *busy waiting*

  sinnvoll bei kurzer Verzögerung durch Lock & bei Multiprozessoren

→ ***viele BS mixen beide Alternativen***
spinning für kurze Zeit & dann blocking



Mutual Exclusion

- In Java realisiert durch das Lock Interface aus dem *java.util.concurrent.locks Package*

```
Lock mutex = new LockImpl(...); // Lock implementieren
...
mutex.lock();  // Lock bekommen & Eintritt in krit. Bereich
try {
...
} finally {
        mutex.unlock();
}
```

- 2 Hauptmethoden:
  - lock()
  - unlock()

WICHTIG:

lock() nicht innerhalb des try Blocks aufrufen!

→ lock() könnte Exception werfen

Peterson Algorithmus

Peterson Algorithmus

- Algorithmus für 2 Threads
- Reihenfolge von ***Zeilen 7-9*** wichtig!
- Mögliche Inkonsistenzen bei Multiprozessor-Umgebungen

→ Threads können gleichzeitig in krit. Bereich

- Compiler können Ordnung der Zeilen vertauschen
- Operation muss nicht unbedingt sofort ausgeführt werden

```
1  class Peterson implements Lock {
2          private boolean[] flag = new boolean[2];
3          private int victim;
4          public void lock() {
5                  int i = ThreadID.get(); // entweder 0 oder 1
6                  int j = 1-i;
7                  flag[i] = true;
8                  victim = i;
9                  while (flag[j] && victim == i) {}; // spin
10         }
11 }
```

AtomicBoolean Klasse

AtomicBoolean Klasse

- ***AtomicBoolean*** aus java.util.concurrent Package
  - Enthält Boolean Wert, der atomar geupdatet werden kann, d.h. er wird nicht von anderen, möglicherweise gleichzeitig ablaufenden Vorgängen beeinflusst
  - *Methoden*:
    - compareAndSet()
    - get()
    - getAndSet()
    - set()

Test-And-Set Locks

Test-And-Set Locks

- Anweisung operiert auf einem einzigen Speicherwort → AtomicBoolean → hält binären Wert
- ***getAndSet(true)*** speichert true in Wort & gibt vorherigen Wert zurück (swapping)

  Wert false → lock ist frei

  Wert true → busy

- ***lock()*** Methode wiederholt getAndSet() solange bis false zurückgegeben wird
- ***unlock()*** schreibt den Wert false

```
1  public class TASLock implements Lock {
2          AtomicBoolean state = new AtomicBoolean(false);
3          public void lock() {
4                  while (state.getAndSet(true)) {} // spin
5          }
6          public void unlock() {
7                  state.set(false);
8          }
9  }
```

Test-And-Test-And-Set Locks

Test-And-Test-And-Set Locks

- *Erweiterung von Test-And-Set Algorithmus*:

  getAndSet() wird nicht direkt ausgeführt

  – Warten bis Lock frei ist
  – „Spin“ solange get() true liefert
  – Sobald get() false zurückgibt, getAndSet() aufrufen & das Lock bekommen

```
1  public class TTASLock implements Lock {
2          AtomicBoolean state = new AtomicBoolean(false);
3          public void lock() {
4                  while(true) {
5                      // warten bis lock frei   while (state.get()) {};
6                                                 // false
7                      → tAS, lock bekommen       if (!state.getAndSet(true))
8                                                         return;
9                  }
10         }
11         public void unlock() {
12                 state.set(false);
13         }
14 }
```

TASLock / TTASLock Beurteilung

→ ***TASLock & TTASLock garantieren Deadlock freie Mutual Exclusion***

- ***Aber***: Performance/Laufzeit nicht effizient

  Warum?

  – Prozessoren kommunizieren über gemeinsamen Bus
  → nur ein Prozessor kann zu einem Zeitpunkt senden

  – Jeder Prozessor verfügt über ***Cache*** (= Speicher)
  Prozessor speichert Daten, die von Interesse sein könnten

  – Prozessor will von Adresse im Speicher lesen
  → checkt zuerst im Cache

    - ***cache hit*** falls Suche im Cache erfolgreich → Prozessor kann Daten sofort laden
    - ***cache miss*** falls Suche scheitert → Prozessor muss Daten im Speicher oder Cache eines anderen Prozessors finden

TASLock / TTASLock Beurteilung

- Alle Threads müssen den Bus zum Kommunizeren verwenden → getAndSet() Aufrufe verzögern alle Threads
- andere Prozessoren verwerfen ihre eigenen gecachten Kopien des Locks

  → d.h. es kommt zu cache misses bei spinning Threads

  → Threads müssen neuen aber unveränderten Wert holen

→ „local spinning“

Threads lesen gecachte Werte anstatt wiederholt den Bus zu benutzen

Exponential Backoff

Exponential Backoff

- ***Contention*** (= Wettstreit)

  mehrere Threads versuchen gleichzeitig ein Lock zu bekommen

  schlecht:
  Lock bekommen wollen für welches hoher Wettstreit besteht

  effektiver:
  Thread sollte für gewisse Zeit zurücktreten („back off“)
  konkurrierende Threads haben Chance zum Ende zu kommen

  Für wie lange zurücktreten?

  – je höher Anzahl an nicht erfolgreichen Versuchen, desto höher Wettstreit & desto länger sollte der Thread zurücktreten
  – Thread sieht Lock ist verfügbar, bekommt es nicht → tritt zurück bevor er es wieder versucht
  – Nach jedem gescheiterten Versuch wird die Back-Off Zeit verdoppelt (festgelegtes Maximum)

Exponential Backoff – Backoff Klasse

```
1  public class Backoff {
2          final int minDelay, maxDelay;
3          int limit;
4          final Random random;
5          public Backoff(int min, int max) {
6                  minDelay = min;
7                  maxDelay = max;
8                  limit = minDelay;
9                  random = new Random();
10         }
11         public void backoff() throws InterruptedException {
12                 int delay = random.nextInt(limit);      // zufällig generierte Verzögerung
13                 limit = Math.min(maxDelay, 2*limit);    // Verdoppeln falls nicht erfolgreich
14                 Thread.sleep(delay);
15         }
16 }
```

Exponential Backoff – BackoffLock Klasse

```
1  public class BackoffLock implements Lock {
2          private AtomicBoolean state = new AtomicBoolean(false);
3          private static final int MIN_DELAY = ...;
4          private static final int MAX_DELAY = ...;
5
6          public void lock() {
7                  Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
8                  while(true) {
9                          while(state.get() {};  // warten bis lock frei ist – get() = false
10                                 if (!state.getAndSet(true)) {  // Lock bekommen falls erfolgreich
11                                         return;
12                                 } else {
13                                         backoff.backoff();   // zurücktreten
14                                 }
15                 }
16         }
17         public void unlock() {
18                 state.set(false);
19         }
20         ...
21 }
```

- Verwendet *Backoff* Objekt
- minimum & maximum Backoff Dauer von *minDelay* & *maxDelay* verwaltet
- Thread tritt nur zurück, wenn er das zuvor als frei beobachte Lock nicht bekommt

BackoffLock – Performance & Probleme

→ besser als TASLock

ABER: Performance hängt von minDelay, maxDelay ab

Lösung: Experimentieren mit verschiedenen Werten → optimale Werte hängen von Anzahl der Prozessoren & ihrer Geschwindigkeit ab

- *2 weitere Probleme*:
  - ***Cache-coherence traffic***

    = Problem den Überblick über mehrere Kopien derselben Daten zu behalten
    Threads „drehen“ auf derselben gemeinsamen Umgebung → cache-coherence traffic bei allen erfolgreichen Lock Zugriffen
  - ***Critical Section Underutilization***

    Threads verzögern länger als nötig
    → kritischer Bereich wird nicht genügend genutzt

Queue Locks

Queue Locks

- Mögliche Lösung der Exponential Backoff Probleme durch ***Queue-Bildung*** der Threads

  → jeder Thread kann herausfinden, ob er an der Reihe ist indem er prüft, ob sein Vorgänger fertig ist

- *Vorteile*:
  - Bessere Verwendung des krit. Bereichs
  - Queue → garantiert FCFS Fairness

Array Based Locks

Queue Locks – Array Based Locks

- Auf Array basierte Queue Locks
- Threads teilen sich eine AtomicInteger „***tail***“ Variable
- Thread-lokale Variable „***mySlotIndex***“

  → jeder Thread hat seine eigenen, unabhängig initialisierten Kopien von jeder Variablen

  → müssen nicht in shared memory gelagert werden, brauchen keine Synchronisation, verursachen keinen coherence traffic

  → Wert durch get() und set() Methoden geregelt

Array Based Locks – Methoden & Beurteilung

```
16        public void lock() {
17                int slot = tail.getAndIncrement() % size;  // Thread erhöht tail um Lock zu bekommen
18                mySlotIndex.set(slot);  // in Thread-lokale Variable speichern – Index in einem Boolean Array
19                while (! flag[slot]) {};    // slot = true – Lock bekommen
20        }
21        public void unlock() {
22                int slot = mySlotIndex.get();
23                flag[slot] = false;    // Lock loslassen
24                flag[(slot + 1) % size] = true;    // flag beim nächsten slot true setzen
25        }
```

Vorteile:

- Wettstreit minimiert
- Zeitabstand zwischen Lock-Freigabe und Lock bekommen minimiert
- Kein Verhungern
- FCFS Fairness

- ***ABER***: Laufzeit ist nicht effizient

CLH Queue Lock

Queue Locks – CLH Queue Lock

- Status eines Threads in ***QNode*** Objekt gespeichert → enthält „locked“ Variable
- *locked = true*

  → Thread hat Lock oder wartet darauf
- Lock selbst wird als virtuelle verkettete Liste von QNode Objekten dargestellt

```
20          public void lock() {
21          QNode qnode = myNode.get();
22          qnode.locked = true;  // Thread hat lock oder wartet auf lock
23          QNode pred = tail.getAndSet(qnode);
24          myPred.set(pred);
25          while (pred.locked) {}  // spin
26          }
27          public void unlock() {
28          QNode qnode = myNode.get();
29          qnode.locked = false;  // lock release
30          myNode.set(myPred.get());
31          }
32  }
```

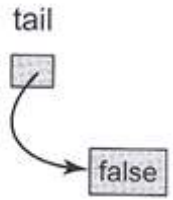
CLH Queue Lock – Lock Acquisition

Thread A führt getAndSet() auf tail aus → fügt seinen QNode ans Ende der Queue & bekommt Referenz des Vorgänger-QNodes

(a)

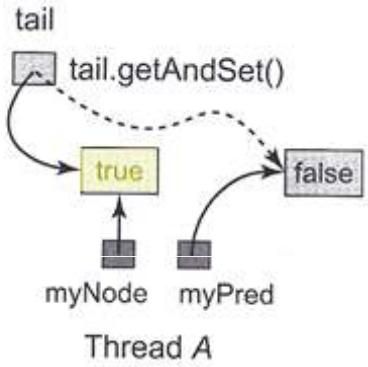
Initially

tail zeigt auf QNode dessen locked Variable false ist



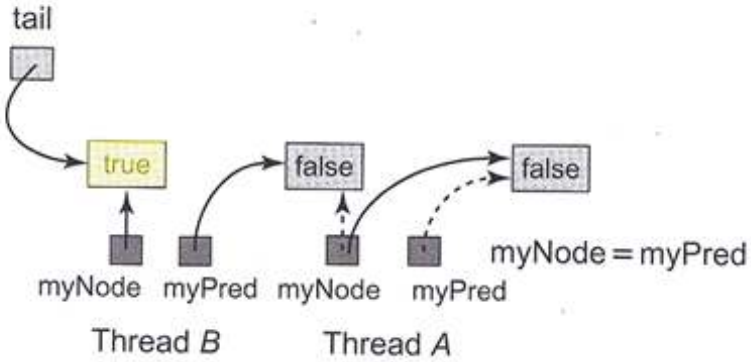
(b)

A:lock()



(c)

A:unlock()
B:lock()



Thread B macht dasselbe
Thread A lässt Lock los indem locked Variable auf false gesetzt wird → Vorgänger-QNode ist nun dessen neuer QNode

CLH Queue Lock - Beurteilung

- ***Vorteile***:
  - weniger Speicheraufwand als Array Based Lock
  - FCFS

- ***Nachteil***:

  schlechte Performance auf cache-freien Architekturen

Hierarchical Locks

Hierarchical Locks

- Viele moderne Architekturen organisieren Prozessoren in ***Clustern*** (=Bündel/Gruppen)

  → Kommunikation innerhalb Cluster bedeutend *schneller* als Kommunikation zwischen Clustern

- ***Hierarchisch*** → berücksichtigen die Speicherhirarchie von Architekturen & Zugriffskosten

- Annahme: jeder Cluster hat eigene cluster id → erreichbar über ThreadID.getCluster()

Hierarchical CLH Queue Lock

Hierarchical CLH Queue Lock

- ***Idee***: Threads vom selben Cluster, aber auch Fairness

  → Planung von *Request-Sequenzen* von Nachfragen desselben Clusters

- HCLHLock besteht aus einer Reihe von *lokalen* Queues (eine pro Cluster) & einer einzigen *globalen* Queue

- Jede Queue ist eine ***verkettete Liste*** von Knoten

  → 2 thread-lokale Variablen: *myQNode*, *myPred*

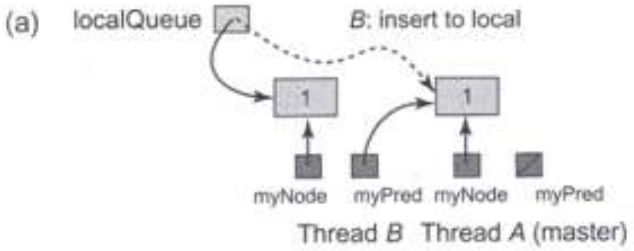
  → Thread besitzt seinen eigenen *myQNode* Knoten → Vorgänger ist der Besitzer seines myPred Knoten

Hierarchical CLH Queue Lock

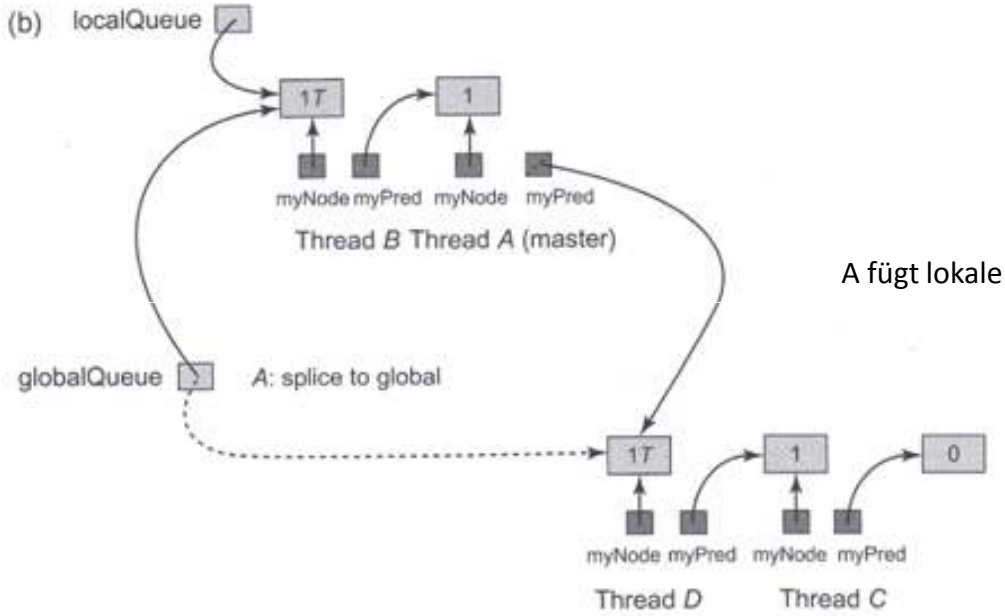
- Jeder Knoten besitzt 3 virtuelle Variablen
  - ***ClusterID*** des aktuellen Besitzers
  - 2 Boolean Variablen
    - ***successorMustWait***
      true vor Einreihen in Queue
      false wenn Lock losgelassen wird
    - ***tailWhenSpliced***
      zeigt an, ob Knoten der Letzte in der Sequenz ist, die an globale Queue geknüpft wird

    → virtuell, da sie automatisch geupdatet werden müssen

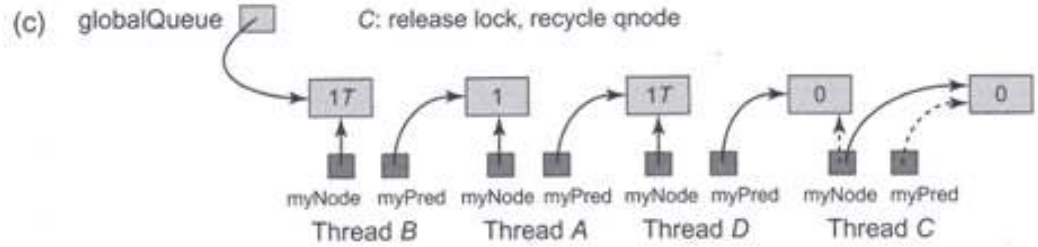
- Thread an Spitze der lokalen Schlange ist ***Cluster Master*** → ist verantwortlich für Anfügen der lokalen an die globale Schlange



successorMustWait
0 = false, 1 = true
B fügt seinen Knoten in lokale Schlange



A fügt lokale Schlange an globale Schlange



C löst Lock → successorMustWait = false
myQNode zeigt auf Vorgänger-Knoten

Hierarchical CLH Queue Lock - Beurteilung

- Bevorzugt Sequenzen von lokalen Threads (einer wartet auf den anderen innerhalb der Warteliste der globalen Schlange)

- Wie bei CLH Queue Lock: Verwendung von impliziten Referenzen minimalisiert cache misses & Thread „dreht“ auf lokalen gecachten Kopien des Zustands seines Nachfolgerknoten

Fazit

Fazit

- *Kein einziger Algorithmus ist ideal für alle Anwendungen*
- *Beste Wahl hängt ab von verschiedenen Aspekten der Anwendung und der Zielarchitektur*

Vielen Dank für die Aufmerksamkeit!

Quellen:

Herlihy, M. & Shavit, N. (2008). The Art Of Multiprocessor Programming. Morgan Kaufmann Publishers.

Elsevier Inc. http://www.elsevierdirect.com/companion.jsp?ISBN=9780123705914