

# Lock-freie Konzepte für parallele Software

## Ein Überblick

*Klaus Kusche, September 2014*

# Mein Bezug zum Thema

*Ich habe Lock-freie Algorithmen und Datenstrukturen als zentrale Komponente eines Betriebssystems für industrielle Echtzeit-Steuerungs-Systeme entworfen, implementiert,  
getestet und optimiert.*

Mein Code (entwickelt: 2006 / 2007) läuft derzeit in einigen tausend produktiven Systemen.

(Ich habe das Lock-freie Grundkonzept nicht erfunden und nicht wissenschaftlich weiterentwickelt.)

# Inhalt

- **Motivation:**  
Welche Probleme wollen wir lösen?
- **Lock-freie Algorithmen:**  
Definition, typische Grundstruktur  
Implementierung (CAS)
- **Fortgeschrittene Konzepte:**  
Multiword-CAS  
Wait-freie Algorithmen
- **Alternative Ansätze**

# Motivation

- x86-CPU's haben Hyperthreading und mehrere Cores (sogar die Low-End-Prozessoren in Embedded Systems)
  - Multi-Core ARM-, MIPS- und PowerPC-Prozessoren sind ebenfalls allgegenwärtig (< 25 \$, < 2 W !) u.a. in Steuerungen und Kommunikationssystemen
  - Selbst auf Single-Core-CPU's:  
Viele Dinge laufen (pseudo-) „gleichzeitig“
- Heutiger Code ist hoch-parallel, enthält viele Threads!

# Das Problem

- Viele Threads wollen

**gleichzeitig**  
auf **gemeinsame Daten** zugreifen

- Wenn man das „ganz normal“ macht  
(wie in sequentiellen Code),

gehen Daten **verloren**  
oder werden **inkonsistent**

- Beispiele:

- Erhöhen eines gemeinsamen Zählers
- Anhängen an eine verkettete Liste

# Die „klassische“ Lösung: Locks

Seit mehr als 50 Jahren:

**Locks** (Mutexes, Semaphoren, ...)

...

**Lock();**

*// Zugriff auf gemeinsame Daten („critical region“)*

**Unlock();**

...

→ Zu jedem Zeitpunkt darf nur maximal ein Thread zwischen **Lock()** und **Unlock()** sein!

→ Alle anderen müssen beim **Lock()** warten!

# Probleme der Locks (1)

- Locks sind langsam und aufwändig:
  - Oft ein System Call pro **Lock()** und **Unlock()** ...
  - ... und mindestens ein Taskwechsel, wenn das **Lock()** warten muss.
- Sie limitieren den Multi-Core-Leistungsgewinn:

Max. ein Core kann im Code kritischer Regionen sein!

→ „Amdahl's Law“ (bzw. die „50 %-Regel“)
- Bei mehreren Locks: Gefahr von Deadlocks:

A wartet auf B, B wartet auf A ... für ewig!

# Probleme der Locks (2)

## „Priority Inversion“ in Echtzeit-Systemen:

- Der „unwichtige“ Thread C belegt ein Lock
  - Der „halbwichtige“ Thread B verdrängt C von der CPU
    - C wird mit seiner kritischen Region nicht fertig.
    - Das Lock bleibt gesperrt, bis C von B die CPU bekommt.
  - Der „hochwichtige“ Thread A möchte das Lock, das C belegt, und muss darauf warten.
- Der „halbwichtige“ B kann den „hochwichtigen“ A (indirekt über C) beliebig lange blockieren!
- Die maximalen Antwortzeiten sind kaum noch berechenbar!



# Probleme der Locks (3)

Locks sind nicht für Device Driver geeignet:

Interrupt-Handler dürfen  
keine Locks verwenden

... denn Interrupt-Handler dürfen  
niemals blockiert werden / warten müssen:  
*Was ist, wenn das Lock beim Interrupt belegt ist?*

→ *Wie greift man  
in Interrupt-Handlern sicher  
auf komplexe gemeinsame Datenstrukturen zu?*

# Probleme der Locks (4)

- Wenn ein Thread in einer kritischen Region hängenbleibt / abstürzt / gekillt wird
  - oder der Programmierer am **UnLock()** vorbei eine Exception wirft oder ein **return** macht
  - oder schlicht das **UnLock()** vergisst
- ... dann bleibt das Lock ewig gesperrt!
- Alle anderen Threads,  
die dieses Lock verwenden,  
bleiben früher oder später ewig hängen!

# Was ist „Lock-frei“ ?

Jeder Thread darf zu jeder Zeit laufen:

Kein Thread wird je blockiert (zeitlich)  
oder von einem Stück Code ausgesperrt (örtlich).

Weitere Anforderung:

Mindestens einer der laufenden Threads  
macht immer echten Fortschritt  
(= sinnvolle Arbeit).

(das vermeidet „Livelocks“,  
d.h. dass sich mehrere Threads  
ewig sinnlos gegenseitig beschäftigen)

# Eine „Lock-freie“ Lösungsidee

Ein Beispiel für Lock-freies Vorgehen:

Der „optimistische“ Ansatz („*wird schon nicht kollidieren*“):

- Jeder macht seine Berechnungen „ganz normal“, als ob er der einzige Thread wäre...
- ... und erst am Ende beim Speichern der Ergebnisse in die gemeinsamen Daten:

Jeder prüft, ob ein anderer inzwischen die Daten „*hinter seinem Rücken*“ verändert hat.

Wenn ja (*selten!*): „Nochmal probieren!“  
(*Berechnung mit den neuen Daten wiederholen, wieder prüfen und speichern*)

# Die Grund-Operation dafür: CAS

Jede Art der Synchronisation paralleler Threads  
braucht Hardware-Unterstützung!

- Single-Core-Systeme: Sperre der Interrupts
- Multi-Core-Systeme:
  - „Atomare“ Befehle  
(atomic Increment, atomic Exchange, ...)
  - Hardware: Exklusiver, synchroner Speicherzugriff
- In unserem Fall:  
**Atomares „Compare and Swap“ (CAS)**

# Was macht `CAS(xPtr, old, new)` ?

- `xPtr` ... Zeiger auf zu ändernde Variable `x`
- `old` ... erwarteter alter Wert von `x`
- `new` ... zu schreibender neuer Wert von `x`

```
int CAS(int *xPtr, int old, int new) {  
    if (*xPtr == old) { // Diese beiden *xPtr ...  
        *xPtr = new; // ... müssen atomar sein!  
        return new; // new heißt „Erfolg“  
    } else { // Verändere x nicht!!!  
        return *xPtr; // Nicht new = „Misserfolg“  
    }  
}
```

# Einfachste Verwendung von CAS

```
// hol den alten Wert der gemeinsamen Variable x  
old = x;  
do {  
    // berechne den neuen Wert von x in new  
    // nur auf old basierend (nicht mehr auf x,  
    // denn x könnte inzwischen geändert werden!)  
    new = ... old ... ;  
    // versuche das Ergebnis zu speichern  
    old = CAS(&x, old, new)  
    // wiederhole Berechnung bis erfolgreich gespeichert  
} while (old != new);
```

# Die Implementierung von CAS

... muss atomar sein:

*Kein anderer Thread darf x  
zwischen Vergleich und Zuweisung ändern!*

- **IBM 370 Großrechner**: Seit 1970 eigener Befehl  
(ganz zentral für das Betriebssystem!)
- **x86**: Befehl "**CMPXCHG**" (und weitere)  
(*"COMpare and eXCHanGe", seit i486)*)
- **ARM, PowerPC, ...**: Zwei Befehle ohne Lock in HW!  
"**LL**" = „Load Linked“ + "**SC**" = „Store Conditional“
- Auf **Single-Core-Systemen**:  
Normaler Code innerhalb Interrupt-Sperre.



# Komplexe Datenstrukturen & CAS

Komplexe Datenstrukturen (verkettete Listen, Bäume, ...)

benötigen mehrere Schreibzugriffe  
um von einem konsistenten Zustand  
zum nächsten zu kommen

„Einfach mehrere CAS nacheinander“ ist keine Lösung:

- Das erste CAS endet erfolgreich
  - Das zweite CAS schlägt fehl
- Die Datenstruktur ist inkonsistent!
- Es gibt keine einfache Möglichkeit zum Wiederholen  
oder zum rückgängig Machen der kaputten Änderung!

# Ad-hoc Lösung: Nachdenken!

Für manche Datenstrukturen schafft man es,

clevere Algorithmen mit CAS zu „erfinden“!

Beispiel: Einfach & doppelt verkettete Listen  
haben wir gelöst!

- Es gibt kein mechanisches „Kochrezept“  
zur Umformung bestehender Algorithmen!
- Sehr schwierig (*unser Hirn tut sich sehr schwer  
bei der gedanklichen Simulation paralleler Abläufe*):  
Meist fällt einem keine / eine falsche Lösung ein...
- Die Algorithmen sind schwer zu lesen & zu verstehen  
und kaum formal zu verifizieren!

# Systematische Lösung: Multiword CAS

- Nimm an, es gibt eine CAS-Grundoperation, die  
atomar mehrere Variablen prüft und setzt:

```
MultiCAS(xPtr1, old1, new1,  
          xPtr2, old2, new2,  
          xPtr3, old3, new3, ...)
```

- Zuerst alle Werte prüfen  
(alle Prüfungen müssen erfolgreich sein!)
  - Dann alle Werte auf einmal speichern, oder gar keinen  
... und das alles atomar
- Hinterlässt nie teilweise geänderte Daten!!!

# Multiword CAS Algorithmen

- Gleiche Struktur wie Single-Word-CAS-Algorithmen: Alte Daten holen, neue ausrechnen, CAS machen bis ok
- Den Code der meisten gebräuchlichen Datenstrukturen kann man fast mechanisch in Lock-freien Code mit Multiword CAS transformieren  
  
(Balancierte Bäume sind eine Ausnahme, aber normale Bäume funktionieren problemlos ...).
- Die entstehenden Algorithmen sind einfach, elegant und offensichtlich korrekt.
- **Aber** (weil Multiword CAS so lange dauert): Die Wahrscheinlichkeit von Kollisionen und Retries steigt deutlich!

# Multiword CAS Implementierung

- Multiword CAS ist nicht in Hardware verfügbar ...
- ... aber kann basierend auf Single-Word CAS in Software implementiert werden
- Erstes praktisch umsetzbares Paper:  
"Timothy L. Harris, Keir Fraser, Ian A. Pratt:  
*A Practical Multi-Word Compare-and-Swap Operation*" (2002)
- Das „Innenleben“ von Multiword CAS ist kompliziert und sehr langsam (mindestens 3 CAS pro Variable)
- Gegen „halbe“ Änderungen: Auch Lesezugriffe müssen via Multiword-CAS-Library erfolgen!

# Vergleich

Library (produktiver C-Code) für

## doppelt verkettete Listen

- **Sequentieller Code: ~ 830 Lines of Code**
- Lösung mit **Multiword CAS: ~ 1920 LoC**  
(inclusive 900 LoC Multiword CAS Library)
- **Ad-hoc-Algorithmus mit einfachem CAS: ~ 1510 LoC**
  - Ist mehr als doppelt so schnell!
  - Aber brauchte vierfache Entwicklungs- & Test-Zeit!
  - ... und hat schlechtere Daten-Konsistenz-Eigenschaften.

# Alle Probleme gelöst?

Lock-freie Algorithmen ...

- verhalten sich zwar ähnlich „Amdahl's Law“ (*Kollisionen und Retry's nehmen zu*), aber

*skalieren in der Praxis  
viel besser als Locks!*

- kennen keine Deadlocks  
(zumindest einer kommt immer voran)
- sind für Interrupt-Handler geeignet
- brauchen keine Syscalls und keine Taskwechsel

# Ungelöste Probleme

Theoretisch sind

**unendlich viele Retries möglich!**

A stört B immer wieder

→ B wird trotz CPU-Verbrauch nie fertig!

Bei „echter“ Parallelität = Multi-Core:

Auch wenn A niedrigere Priorität als B hat

→ Effekt ähnlich „Priority Inversion“

→ Nicht für (beweisbar) „harte“ Echtzeit geeignet!

Und bei manchen Algorithmen:

Nicht resistant gegen Abstürze, Exceptions, ...!



# Lösung aller Probleme: „Wait-frei“

Formale Anforderung:

Die gesamte noch verbleibende Arbeit muss zu jedem Zeitpunkt weniger werden!

- Alle Threads (nicht nur einer wie bei „Lock-frei“)
- machen garantiert Fortschritte bzw. sinnvolle Arbeit, wenn sie CPU brauchen
  - und enden in endlicher, deterministischer CPU-Zeit!

(Auch bei maximaler Konkurrenz wird keine Arbeit verschwendet, und kein Thread wird mit unabsehbar viel Zusatzarbeit belastet)

→ „Probier's nochmal“-Schleifen sind verboten!!!

# *Gute* „Wait-freie“ Algorithmen ...

- ... muss man individuell „erfinden“: Kein „Kochrezept“
- Sehr schwierig, bisher nur für wenige Probleme gelöst

## Grundidee:

- Mach nur Arbeit, die garantiert „nützlich“ ist!
- Wenn du mit deiner Arbeit nicht mehr weiterkannst, weil zuerst andere ihre Arbeit fertig machen müssen:
  - Entweder: Mach **zuerst deren Arbeit**, dann deine!  
(Die merken das dann bei ihrem CAS und freuen sich...)
  - Oder: Sorge dafür, dass andere deine Arbeit fertigstellen, nachdem sie ihre beendet haben!  
(„Der Letzte räumt alles auf!“)

# Alternative #1: RCU (1) „Read-Copy-Update“

Besonders einfacher Spezialfall Lock-freier Datenstrukturen.

Voraussetzung:

Zugriff auf die Datenstruktur (bzw. ihre Elemente)  
über einen einfachen Pointer  
(Beispiel: Einfach verkettete Liste ohne Tail)

→ Es genügt, einen einzelnen Pointer auszutauschen,  
um die Struktur bzw. ein Element atomar zu ändern!

Sehr häufig verwendet in vielen Betriebssystemen  
für Daten, die viel gelesen und nur selten geändert werden  
(Linux enthält über 5000 RCU-Datenstrukturen!)

# Alternative #1: RCU (2)

## „Read-Copy-Update“

- Leser:
  - Müssen **read\_begin()** und **read\_end()** aufrufen, um über aktive Leser Buch zu führen (in Linux: No-Op)
  - Lesen ganz normal (ohne Locks, ohne CAS, ...).
- Schreiber:
  - Machen eine private Kopie des Original-Elementes.
  - Ändern ihre private Kopie.
  - Ersetzen das alte Element durch ihre Kopie, indem sie den Pointer auf ihre Kopie ändern.
  - Schützen das Ändern des Pointers mit CAS oder Locks.
- Aufräumen:
  - Wenn der letzte Leser die alten, ausgehängten Daten verlassen hat: Speicher freigeben.

# Alternative #2: STM (1)

## „Software Transactional Memory“

Abstraktes Modell von Daten im Speicher,  
ähnlich wie Datenbanken:

1) „Begin transaction“

2) Gemeinsame Daten beliebig lesen & schreiben

3) „Commit“ (oder „Abort“)

„Commit“ ist wie bei Datenbanken garantiert atomar:  
„Alles oder nichts“

Die Änderungen sind davor für andere nicht sichtbar!

→ Sehr universell, leicht zu verstehen / zu verwenden!

„Commit“ kann bei konkurrierenden Schreibzugriffen  
fehlschlagen! (→ *automatisches oder explizites Retry*)

# Alternative #2: STM (2)

## „Software Transactional Memory“

### Implementierung:

- Für fast alle Sprachen:  
Zahlreiche Libraries verfügbar
- Basierend auf verschiedenen Grund-Operationen:  
Locks, Lock-frei (CAS), oder MMU- bzw. Paging-basiert
- Meist komplex und langsam,  
einige closed-source und Patent-geschützt!
- Demnächst: **Standardisierte Sprachkonstrukte**  
in C / C++ (derzeit im Beta-Test von gcc und icc)

# Hardware Transactional Memory

Implementiert seit Intel Haswell / Haswell E:

„TSX“ = *Transactional Synchronization Extensions*

- Eigene Befehle
- Implementierung: Lock-frei!

Basierend auf erweiterter Cache-Logik:  
Mehrere „private“ Cache-Kopien  
desselben Speicherbereichs

- Noch buggy, nicht freigeschaltet  
(Freischalt-Microcode-Update gegen NDA von Intel)
- TSX-Software-Emulation u.a. in QEMU

**“The end”**

*Fragen?*