Lockfreie Konzepte zum parallelen Zugriff auf gemeinsame Daten

Klaus Kusche

Mai 2016

Mein Bezug zum Thema

Ich habe Lock-freie, CAS-basierte
Algorithmen & 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

- Wiederholung / Motivation:
  - Problematik paralleler Datenzugriffe
  - klassisches Locking
  - Probleme von Locking
- CAS = “Compare and Swap”
- RCU = “Read-Copy-Update”
- Multiword CAS
- STM = “(Software) Transactional Memory”

Ziele

Verständnis

- … der zu lösenden *Problematik*,
- … verschiedener *Lösungskonzepte*,
- … ihrer *Implementierung*,
- … ihrer *Vor- und Nachteile*

Kenntnis

- … der wichtigsten *Fachbegriffe*

Voraussetzungen

Grundkenntnisse von

- *Paralleler* Programmierung:

  Wechselseitiger Ausschluss
  (Mutex, Lock, Semaphore, …)

  incl. Deadlock, Priority Inversion, …

- *Datenstrukturen:*

  Verkettete Liste (wegen des Beispiels)

Zwei simple Additionen ...

Prozessor A will zu **x** 10 dazuzählen,
Prozessor B will zu **x** 15 dazuzählen,
der bisherige Wert von **x** sei 100

Prozessor A	**Prozessor B**	**Speicher x**
		100
liest **x** aus Speicher: 100		100
	liest **x** aus Speicher: 100	100
zählt intern 10 dazu: 110		100
	zählt intern 15 dazu: 115	100
speichert Ergebnis in **x**: 110		110
	speichert Ergebnis in **x**: 115	115
		115

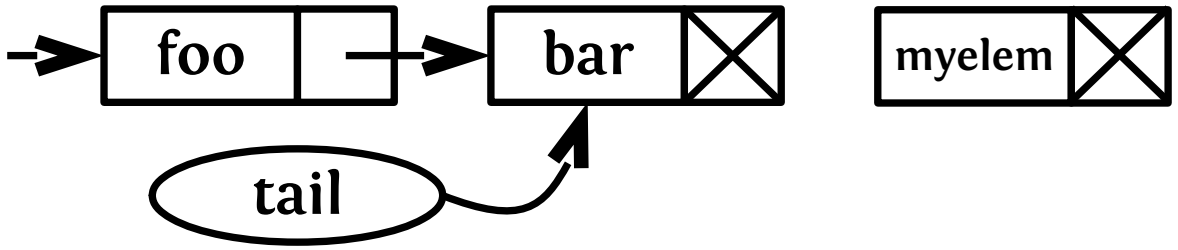
==> Ergebnis 115 statt 125, Addition von A ging verloren!

Dynamische Datenstrukturen (1)

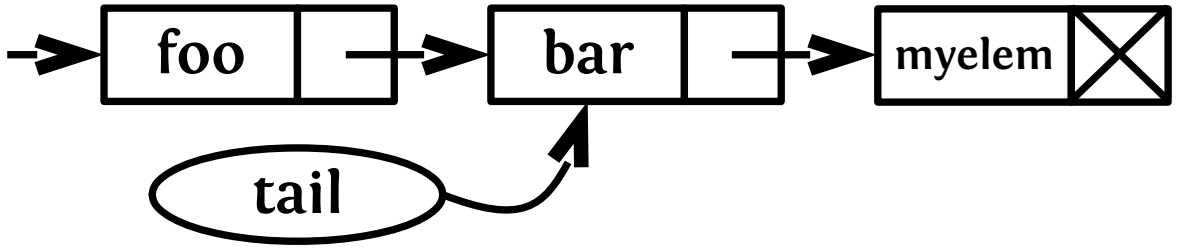
Anhängen von **myelem** an eine *einfach verkettete Liste:*

```
tail->next = myelem;
tail = myelem;
```

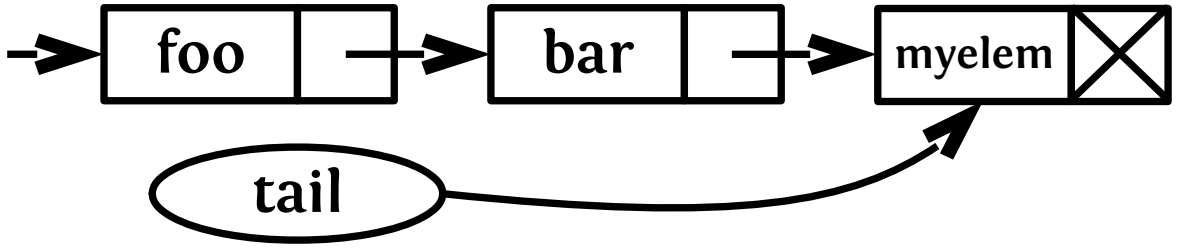
Ausgangslage:



Nach Zeile 1:



Nach Zeile 2:

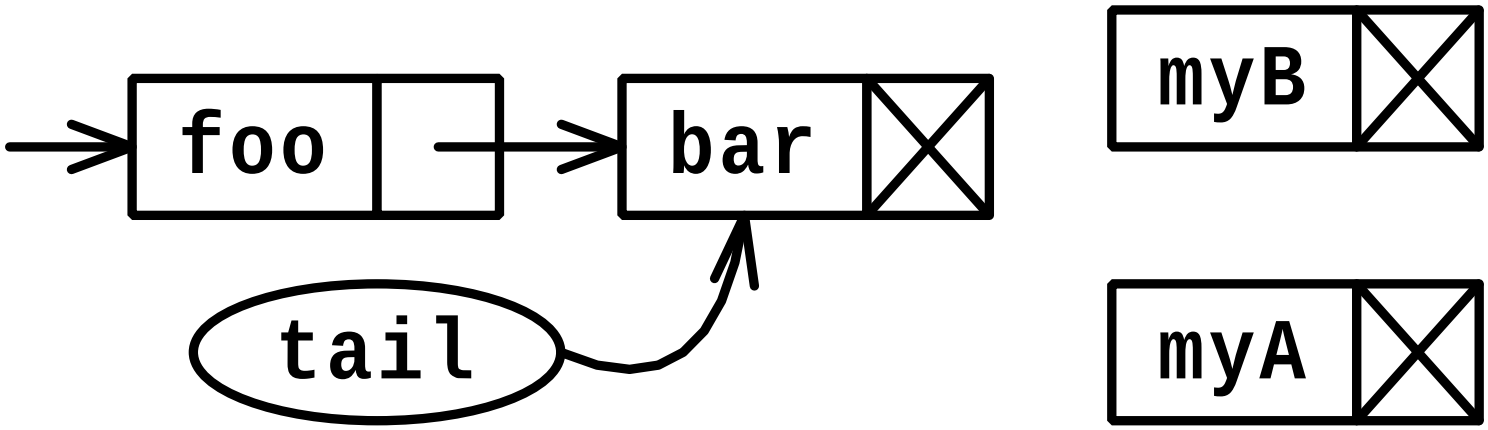


Dynamische Datenstrukturen (2)

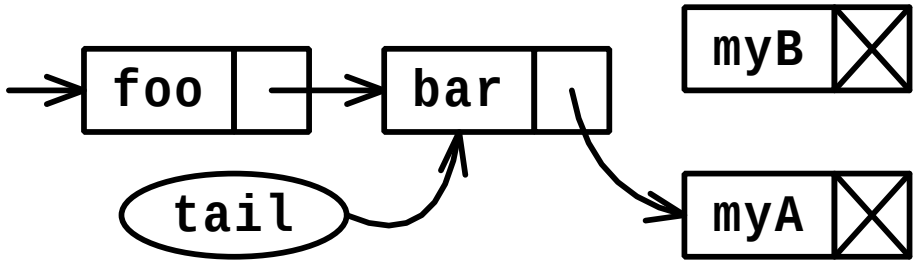
... und das wieder *zwei Mal gleichzeitig:*

Zwei neue Elemente **myA** und **myB**

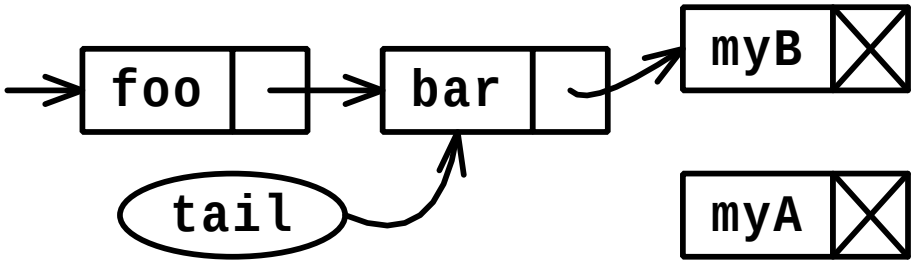
Ausgangslage:



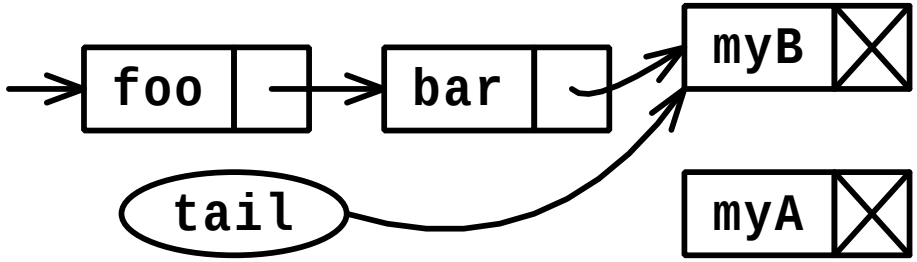
Dynamische Datenstrukturen (3)



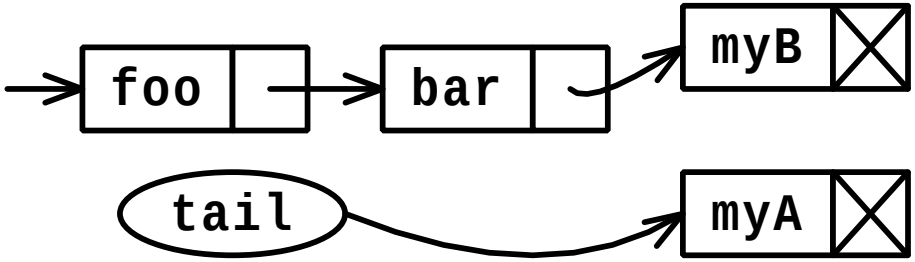
1: Task A hat Listenende geändert.



2: Task B hat Listenende geändert.



3: Task B hat Tailpointer geändert.



4: Task A hat Tailpointer geändert.

*==> Die Liste ist in **zwei <u>getrennte</u> Teile** zerbrochen!*

Konzept “Kritische Region”

= ein Stück Code,

das “allein und ungestört”
durchlaufen muss,

um vom *alten konsistenten Zustand* der Daten
in einen *neuen konsistenten Zustand* der Daten
zu kommen.

Bei genauerer Betrachtung:

Während der kritischen Region sind die *Daten inkonsistent*
==› Dürfen von anderen *nicht einmal gelesen* werden!
==› Auch *reine Lese-Zugriffe* gehören *in eine kritische Region*

“Wechselseitiger Ausschluss” …

… garantiert ***exklusiven Zugriff***
für kritische Regionen:

Für jede Datenstruktur

darf sich zu jedem Zeitpunkt

nur ein einziger Thread

innerhalb einer kritischen Region befinden,

die auf diese Datenstruktur zugreift!

Konzepte und Konstrukte dafür

Konstrukte zur Erreichung dieses Ziels:

- ***Mutex = "Mutual Exclusion"***

  (bezeichnet im Englischen
  sowohl das *allgemeine Konzept*
  als auch das *konkrete Konstrukt* zur Lösung!)

- ***Lock***
- ***Semaphore***
- ***Monitor***

... beschreiben alle *in etwa dasselbe*!

Code-Struktur

Kritische Regionen werden im Code irgendwie

"eingerahmt"

(durch eigene Sprachkonstrukte oder Library-Aufrufe,
die *exklusive Ausführung* sicherstellen)

`...` ("unkritischer" Code)

`Lock(...)` (oder **`Wait(...)`** oder **`Acquire(...)`**)

Code der kritischen Region
bzw. Zugriffe auf gemeinsame Daten

`Unlock(...)` (oder **`Signal(...)`** oder **`Release(...)`**)

`...` ("unkritischer" Code)

Problem: Performance

Locks sind **langsam und aufwändig**:

- Wenn das **Lock()** warten muss:

  1-2 zusätzliche ***Taskwechsel***

  ==> Das *"Aufwecken"* reagiert langsam!

- **Lock()** auf Anwendungs-Ebene:

  Oft ein ***System Call***
  pro **Lock()** und **Unlock()**

Problem: Race Conditions

"Race Condition" =

Korrektheit / Ergebnis / Verhalten
eines Programms
hängen von ***zeitlichen Zufällen*** ab.

Ursache:

Kritische Regionen vergessen oder zu klein!

==> Zugriffe auf *gemeinsame Daten*
je nach zufälliger zeitlicher Verzahnung
in *verschiedener Reihenfolge*.

Problem: Deadlocks

"Deadlock" = Verklemmung = Zyklisches Warten mehrerer Threads aufeinander:

A wartet auf B, B wartet auf A
(„Dining Philosophers")

==> Keiner kann mehr weiterrechnen, *System steht!*
==> *Gewaltsamer Eingriff* von außen nötig!

Ursache (ganz unwissenschaftlich gesprochen):

***Zu viele / schlecht entworfene** kritische Regionen!*

Aber: Theoretisch gut untersucht & *gelöst!*

Problem: Amdahl's Law

„Amdahl's Law":

Der nicht parallelisierbare Code-Anteil beschränkt die durch Parallelisierung erreichbare Beschleunigung des Gesamtsystems.

Anschaulich: ***„50 % Regel"***

„Wenn (zeitlich) die *Hälfte* des Codes *rein sequentiell* berechnet werden muss (d.h. *innerhalb von kritischen Regionen* steht), wird es selbst mit unendlich vielen Prozessoren *höchstens doppelt so schnell*"

Problem: “Übergebliebene” Locks

Wenn ein Thread
innerhalb einer kritischen Region

- eine ***Exception*** wirft
  (die Exception “fliegt” am **`Unlock`** vorbei!),
- ***abstürzt***, gekillt wird oder hängenbleibt,
- oder z.B. versehentlich ein **`return`** enthält,

dann ***bleibt die kritische Region gesperrt***

==> *Alle anderen hängen “ewig” beim* **Lock** !

Problem: Priority Inversion

"Priority Inversion": *(Problem in Echtzeit-Systemen!)*

- **Unwichtiger Thread C**
  ist gerade *innerhalb* einer kritischen Region.
- **Halbwichtiger Thread B** nimmt C die *CPU weg*
  ==> C wird nicht fertig, sperrt die Region *"ewig"!*
- **Hochwichtiger Thread A**
  wartet auf die von C gesperrte kritische Region.

==> Halbwichtiger B kann hochwichtigen A

beliebig lange blockieren!

Problem: Interrupt Handler

Interrupt Handler
(= jene Codeteile des Betriebssystems,
die hardwaremäßig durch Interrupts aktiviert werden)

dürfen nie blockiert
(wartend gestellt) werden!

==> Daher kann man in Interrupt Handlern
keine kritische Region bzw. Mutex verwenden.

==> Sie erfordern *spezielle Techniken*
zum Datenaustausch mit dem Rest des Systems!

Ziel der Alternativen

Lock-Probleme ==> Forderung ***"Lock-freier" Code***

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

Weitere notwendige / sinnvolle 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)

Geht, praktische Erfahrungen sehr gut!!!

Eine „Lock-freie“ Lösungsidee

Der “*optimistische*” Ansatz:

“wird schon keiner gleichzeitig zugreifen”

- Jeder macht seine Berechnungen “*ganz normal*”, als ob er der einzige Thread wäre…
- … und erst ***am Ende beim Speichern*** der Ergebnisse in die gemeisamen Daten:

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

  Wenn ja (sehr *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!

In unserem Fall für den

"prüfen-und-wenn-ok-speichern"-Schritt:

Maschinenbefehl

***"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
- **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*.

Read-Copy-Update (1)

Einfachste “Lock-freie” Datenstruktur, **Voraussetzung:**

Zugriff auf die Datenstruktur (bzw. ihre Elemente)
über einen ***einfachen Pointer***

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

==> *Bestehende* Daten werden *nie* *“in Place” geändert*,
nur als Ganzes durch woanders gespeicherte *ersetzt*!

Sehr häufig verwendet in *vielen Betriebssystemen*
(in IBM-Betriebssystemen seit ~1970 zentrales Konzept,
auch Linux enthält über 5000 RCU-Datenstrukturen!)

Read-Copy-Update (2)

- **Leser**:
  - Müssen **`read_begin()`** und **`read_end()`** aufrufen,
    um über *aktive Leser Buch zu führen* (in Linux: unnötig!)
  - Lesen *ganz normal* (ohne Locks, ohne CAS, …).
- **Schreiber**:
  - Machen eine *private Kopie* des Original-Elementes.
  - *Ändern* ihre private Kopie ganz normal (ohne CAS).
  - *Ersetzen* das Original durch ihre Kopie,
    indem sie den *Pointer darauf auf ihre Kopie umbiegen*.
  - Schützen das Ändern des Pointers *mit CAS* (oder Locks).
- **Aufräumen**:
  - Wenn der *letzte Leser* die alten, ausgehängten Daten
    verlassen hat: Speicher freigeben.

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*:

  Meist fällt einem keine / eine falsche Lösung ein...
- Die Algorithmen sind *schwer zu lesen & zu verstehen*,
  ihre *Korrektheit* kaum mit Kopf & Papier überprüfbar!

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 Implementierung

- Multiword CAS ist *nicht in Hardware* verfügbar …
- … aber kann *basierend auf Single-Word CAS* in *Software* implementiert werden
- Erstes praktisch umsetzbares Paper (Vorlage für mich)
  "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 relativ *langsam* (mindestens *3 CAS* pro Variable)
- *Gegen "halbe" Änderungen:* Auch alle *Lesezugriffe* müssen über die Multiword-CAS-Library erfolgen!!!

Multiword CAS Algorithmen

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

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
  - Läuft mehr als *doppelt so schnell!*
  - Aber brauchte *vierfache Entwicklungs- & Test-Zeit!*
  - … und hat *schlechtere Daten-Konsistenz-Eigenschaften*.

Transactional Memory (1)

Abstraktes Modell von Daten im Speicher,

Idee ähnlich wie bei Datenbanken:

1) "**Begin transaction**"
2) Gemeinsame Daten im Speicher beliebig lesen & schreiben
3) "**Commit**" (oder "**Abort**")

Transactional Memory (2)

"Commit" ist wie bei Datenbanken

garantiert atomar:
"Alles oder nichts"

==> Die Änderungen sind *davor für andere **nicht** sichtbar*!

==> Alle Änderungen werden *auf einmal* sichtbar,
andere sehen *nie "**halbe**" Änderungen!*

==> **Sehr universell, leicht zu verstehen / zu verwenden!**

Aber:
"Commit" kann bei konkurrierenden Schreibzugriffen
fehlschlagen! *(==> automatisches oder explizites Retry)*

Transactional Memory in SW

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!*
  ==> *Unbefriedigend!!!*

- Demnächst: ***Standardisierte Sprachkonstrukte***
  in **C / C++** (derzeit im Beta-Test von **`gcc`** und **`icc`**)

Transactional Memory in HW

Implementiert seit **Intel Skylake**
(+ gefixte Steppings von Broadwell, Haswell E):

“TSX” =
Transactional Synchronization Extensions

- *Eigene Maschinenbefehle*
- Implementierung: ***Lock-frei****!*

  Basierend auf erweiterter ***Cache-Logik***:
  Mehrere “private” Cache-Schattenkopien
  desselben Speicherbereichs
- *TSX-Software-Emulation* u.a. in QEMU

“The end”

Fragen?