Wechselseitiger Ausschluss

Klaus Kusche

Juni 2016

Inhalt

- Ziele & Voraussetzungen
- Der “einfachste Fall”
  und seine Lösung
- Der “Normalfall”
  und seine Lösung
- Probleme des Konzeptes
- 2. Teil: Neue Ansätze

Ziele

Verständnis

- … der zu lösenden *Problematik*,
- … des *Lösungskonzeptes*,
- … sowie dessen *Implementierung*
  (incl. notwendiger HW-Voraussetzungen)
- … und dessen potentieller *Probleme*

Kenntnis

- … der wichtigsten *Fachbegriffe*

Voraussetzungen

Grundkenntnisse von

- *Hardware:*

  Arbeitsweise Prozessor und Speicher
  Multicore- / Multiprozessor-Systeme
  Interrupts

- *Betriebssystem:*

  Prozesse / Threads & Scheduling, ...

- *Programmierung, Datenstrukturen:*

  Verkettete Liste (wegen des Beispiels)

Computerei heute ...

... ist fast immer

parallel !

Auf **Hardware-Ebene**:

Multicore-Prozessoren, Multiprozessor-Server

Auf **Betriebssystem-Ebene**:

Viele Anwendungen & Dienste laufen “gleichzeitig”
(= “Quasi-Parallel”, z.B. Round-Robin-Scheduling)

Auf **Anwendungs-Ebene**:

Viele Anwendungen (und vor allem Server-Dienste)
starten intern mehrere / viele *parallele Threads*

Die zentrale Frage

Kann man parallele Systeme

"ganz normal"

(= wie rein sequentielle Systeme)

programmieren?

Oder müssen *neue Effekte* berücksichtigt werden,
die bei sequentieller Programmierung
nicht auftreten?

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!

Die Lösung

Erfordert zwingend **Hardware-Unterstützung**:

- Auf ***Single****-Core-Systemen*:

  Interrupt-Sperre

  rund um den Befehl / die Befehle

  ==> Unterbindet Prozesswechsel & Interrupt Handler
  ==> *Alleiniger, ungestörter Ablauf* garantiert
  ==> Dann reichen "normale" Prozessorbefehle

- Auf ***Multi****-Core-Systemen*:

  "Atomare" Maschinenbefehle
  bzw. Speicherzugriffe

Ein “Atomarer” Befehl …

… ist ein Maschinenbefehl, der eine Speicherzelle (= **`int`**)

exklusiv liest und anschließend schreibt,

ohne dass ein *anderer Core*
zwischen dem Lesen und dem Schreiben
denselben Speicherinhalt modifizieren kann

- *Atomic Increment, “Test-and-Set”, Atomic Exchange, …*
- x86: “*Bus Lock Prefix*” vor einem Befehl
- RISC-Architekturen: *Load Linked / Store Conditional*

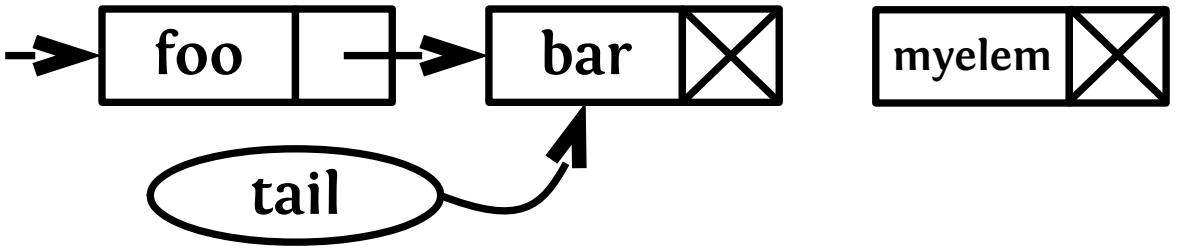
Technisch sehr *aufwändig* und relativ *langsam*:
Früher Pin am Prozessorbus, heute durch *Cache Coherence Hardware*!

Reicht das für Anwendungen? (1)

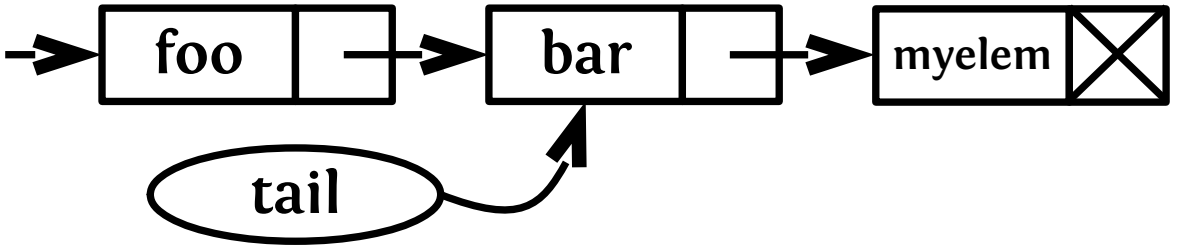
Anhängen von **myelem** an eine *einfach verkettete Liste:*

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

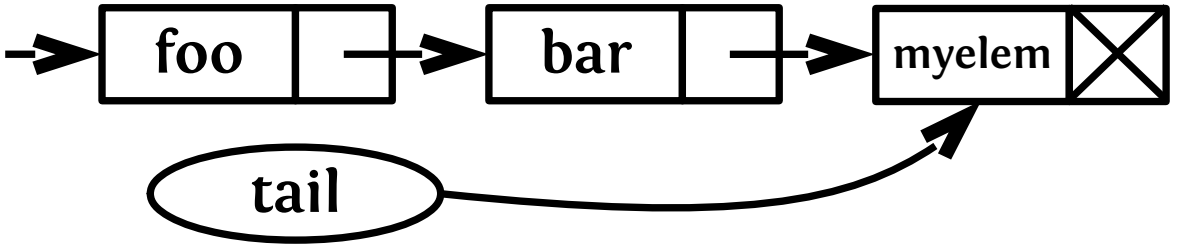
Ausgangslage:



Nach Zeile 1:



Nach Zeile 2:

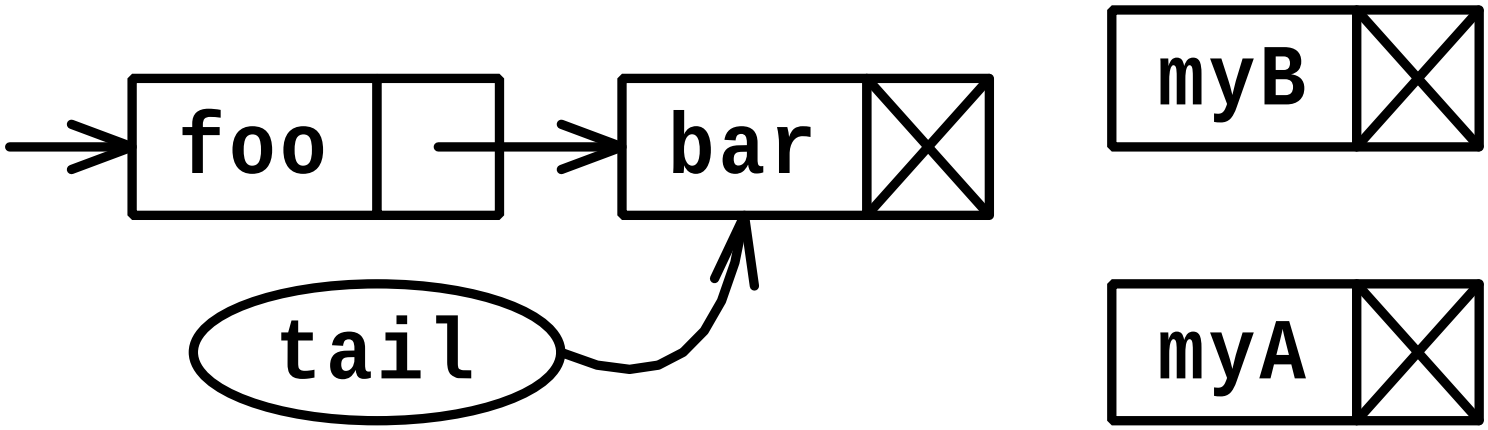


Reicht das für Anwendungen? (2)

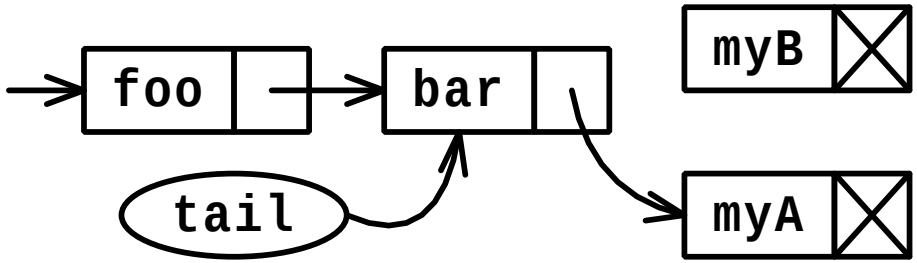
... und das wieder *zwei Mal gleichzeitig:*

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

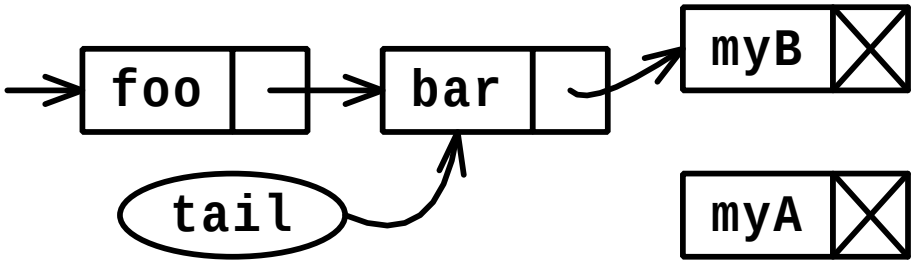
Ausgangslage:



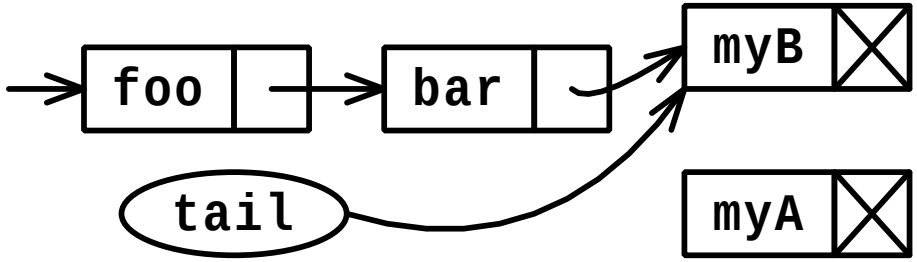
Reicht das für Anwendungen? (3)



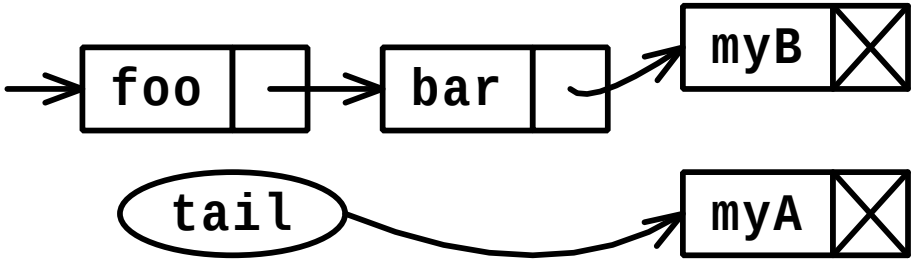
1: Task A hat Listenende geändert.



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!*

Erkenntnis

*Atomare Befehle helfen hier **nicht**:*

***Inkonsistente Daten** entstehen*

*durch **mehrere Zugriffe** pro Thread*

*auf **verschiedene Variablen***

==> *Bestimmte Reihenfolge der Zugriffe erzwingen*

==> *Parallelität einschränken*

==> *Threads koordinieren bzw. synchronisieren*

Konzept “Kritische Region” (1)

= 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*

Konzept "Kritische Region" (2)

Eine kritische Region besteht aus

mehreren <u>logisch zusammengehörenden</u>
Lese- und Schreibzugriffen *auf die Daten*

Beispiel Liste: Genau die beiden Zeilen!

<u>Aber:</u> Wichtiger **<u>Grundsatz</u>** für den Code innerhalb einer kritischen Region:

So viel wie nötig,
so wenig wie möglich!

==> Kritische Regionen möglichst *<u>kurz</u>* halten!

<u>Weil:</u> ***Parallelität so wenig wie möglich einschränken!***

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

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 eigenes Sprachkonstrukt oder Library-Aufrufe)

```
...
Lock(...)
```
(“unkritischer” Code)

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

Code der kritischen Region
bzw. Zugriffe auf gemeinsame Daten

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

```
...
```
(“unkritischer” Code)

Naive Vorstellung (1)

- Die kritische Datenstruktur
  befindet sich in einer versperrten Kiste
- Es gibt nur einen ***einzigen Schlüssel***
- Nur einer kann den Schlüssel haben
- Nur wer den Schlüssel hat,
  darf auf die Daten zugreifen
  (und muss nachher wieder zusperren)
- Der Hausmeister (= z.B. das Betriebssystem)
  verwaltet den Schlüssel

Typischerweise ein separater “Schlüssel”
für jeden zu schützenden Datenbereich ==> mehr Parallelität!

Naive Vorstellung (2)

`Lock` bzw. **`Wait`** bzw. **`Acquire`**:

- ***Hol*** *dir den Schlüssel* vom Hausmeister
- Wenn gerade *jemand anderer* den Schlüssel hat: ***Warte!***

`Unlock` bzw. **`Signal`** bzw. **`Release`**:

- *Gib den Schlüssel* ***zurück*** an den Hausmeister
- Wenn jemand auf den Schlüssel *wartet*:
  Der Hausmeister ***weckt*** *einen auf*
  und *gibt* ihm den Schlüssel

Implementierung eines Mutex (1)

Grundsätzlich:

Eine ***binäre Variable*** (0 / 1)
(+ ev. eine ***Queue*** für die Wartenden)

Lock:

- ***Atomares Test-and-Set*** der Variable
- ***Warten*** wenn Test-and-Set meldet
  "war schon gesetzt"

Unlock:

- ***Clear*** der Variable

Implementierung eines Mutex (2)

Auf **Anwendungs- bzw. Library-Ebene:**

In vielen Fällen *nicht direkt!*

Stattdessen:

Nutzung von
Betriebssystem-Funktionalität!

==> Jedesmal ein System Call, dauert!!!

Häufiger **Mittelweg:**

- Lock-Variable setzen / löschen: Direkt
- Nur wenn Warten / Aufwecken nötig: Syscall

Implementierung des Wartens (1)

Im Normalfall:

Aufruf des Schedulers

==> Thread *gibt die CPU ab*
und wird bei Freiwerden der kritischen Region
wieder in die Bereit-Queue des Schedulers gereiht

==> Scheduler- bzw. Prozesswechsel-Overhead

==> Längere Warte- bzw. Reaktionszeit

==> *Kein CPU-Verbrauch* während des Wartens

Implementierung des Wartens (2)

- Nur bei
  - *ganz kurzer* zu erwartender Sperrzeit
  - und echten Multicore-Systemen

Spin Lock bzw. ***Busy Waiting***

Warten durch ***ständiges Prüfen*** des Mutex-Status in einer ***Schleife***

==> *Verheizt* während des Wartens sinnlos eine CPU

==> Dafür kein Betriebssystem-Aufruf,
kein Prozesswechsel

==> *Reagiert* in Nanosekunden auf das Freiwerden!

Wo braucht man das?

Innerhalb des **Betriebssystems**

... für die *internen Datenstrukturen*
des Betriebssystems:

Scheduler Queres,
Filesystem-Caches und -Metadaten, ...

In **Multithread-Anwendungen**

... für von mehreren Threads
gemeinsam benutzte Datenstrukturen

Problem: Race Conditions

"Race Condition" =

Korrektheit / Ergebnis / Verhalten
des 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: “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!

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*"

Themenwechsel

-

Fragen?

Alternativen zum wechselseitigen Ausschluss

Klaus Kusche

Juni 2016

Inhalt

- Motivation
- CAS = “Compare and Swap”
- RCU = “Read-Copy-Update”
- Multiword CAS
- STM = “(Software) Transactional Memory”

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.)

Motivation

Probleme (siehe zuvor) von wechselseitigem Ausschluss

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

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

Weitere 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)

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`**

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

Einfachste Verwendung von CAS

```
do {
   // hol den alten Wert der gemeinsamen Variable x
   old = x;   // nur 1 Zugriff auf x pro Durchlauf!
   // berechne den neuen Wert für x in new
   // nur auf old basierend (nicht mehr auf x,
   // denn x könnte inzwischen geändert werden!)
   new = ... old ... ;
   // versuche das Ergebnis zu speichern
   // wiederhole Berechnung bis erfolgreich gespeichert
} while (!CAS(&x, 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 *Lesezugriffe*
  müssen über die Multiword-CAS-Library erfolgen!!!

Multiword CAS Algorithmen

- Gleiche Struktur wie Single-Word-CAS-Algorithmen:
  Alte Daten *holen*, neue *ausrechnen*,
  ein *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 deutlich!

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*.

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 resistent 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!“*)

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!*

- 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 Befehle*
- Implementierung: ***Lock-frei****!*

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

“The end”

Fragen?