Read-Copy-Update (RCU)

Seminar *"Konzepte von Betriebssystem-Komponenten: Konkurrenz und Koordinierung in Manycore-Systemen"*

Alexander Richardson

Friedrich-Alexander-Universität Erlangen-Nürnberg

11. Juli 2013

Was ist RCU?

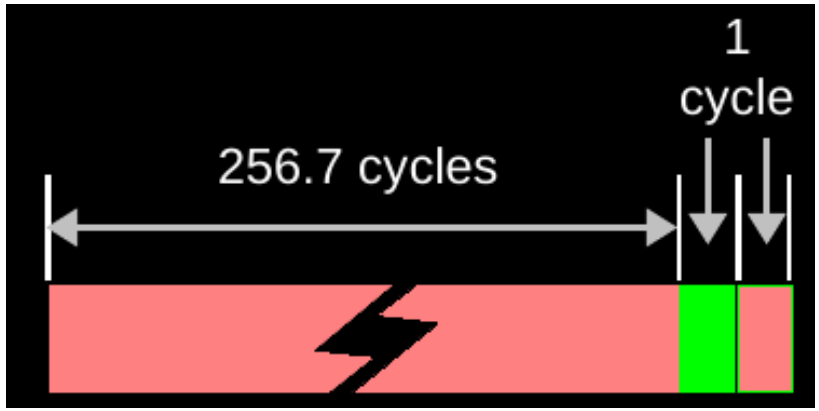
- Synchronisationsmechanismus
- Primäre Verwendung in Betriebssystemen: DYNIX/ptx, Linux
- Inzwischen auch als Bibliothek im Benutzermodus [1]
- Relativ neu: erste RCU-ähnliche Arbeit 1980 [2]
- 1995 von John Slingwine und Paul McKenney patentiert[3]
- Seit 15. Oktober 2002 Teil des Linux Kernels [4]

Warum noch ein Synchronisationsmechanismus?

- Anwendungsgebiet: Betriebssystemkern
- Auf viele Daten in Betriebssystemen primär lesender Zugriff
- Bei anderen Synchronisationsmechanismen oft Gleichberechtigung von Lesern und Schreibern (z.B. Mutex)
- Ziel: Bei Lesern möglichst wenig Aufwand, bei Schreibern Mehraufwand akzeptabel

Probleme bei vielen Synchronisationsmechanismen I

- In der Regel hoher zeitlicher Synchronisationsaufwand
  - Beispiel: *spinlock*



Anfordern eines freien Lock bei Intel X5550 [5]

- Bei Alternativen mit geringerem Zeitaufwand stattdessen hoher Speicherverbrauch
  - Beispiel: *brlock*

Probleme bei vielen Synchronisationsmechanismen II

- Zeitaufwand des Lockerwerbs nicht deterministisch
  - Blockieren oder Aktives Warten

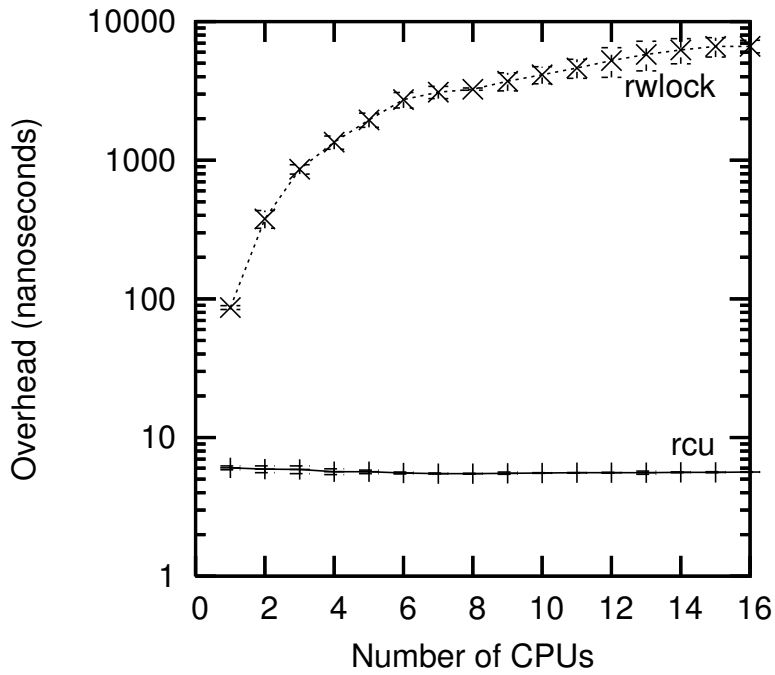
Konkurrierendes Anfordern eines Locks bei Intel X5550 [5]

Probleme bei vielen Synchronisationsmechanismen III

- Nur ein Prozess im kritischen Abschnitt $\Rightarrow$ geringe Parallelität
- Mit *rwlock* immerhin alle Leser parallel
  - Schreiber blockiert jedoch alle Leser

Probleme bei vielen Synchronisationsmechanismen IV

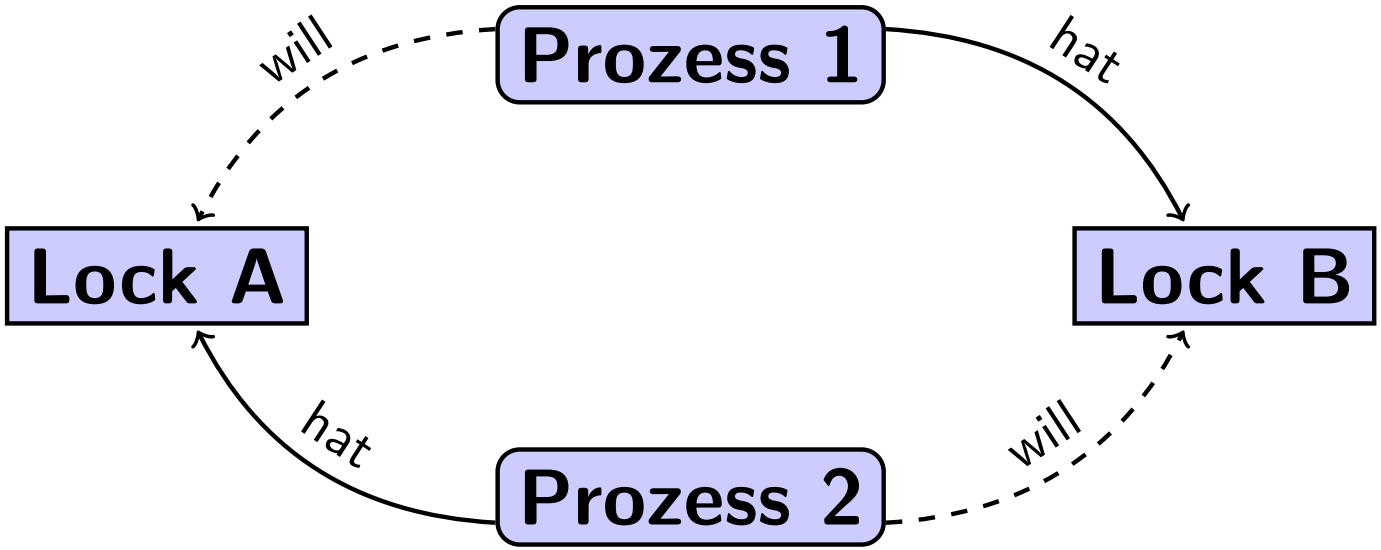
$\Rightarrow$ Schlechte Skalierbarkeit bei vielen CPUs



Mehraufwand bei RCU und *readers–writer lock* für Leser [6]

Probleme bei vielen Synchronisationsmechanismen V

- Deadlocks
  - Auch wegen nicht-rekursiver Locks
  - Verhinderung von Deadlocks eines der Primärziele von RCU [6]



Deadlock Zyklus bei zwei Prozessen

Funktionsweise von RCU

Voraussetzung:

- Datenstruktur mit nur einem atomar zu ändernden Wert (z.B. verkettete Liste)
  - Gegebenenfalls durch zusätzliche Indirektion
- Leser müssen kritischen Abschnitt kennzeichnen (`lock()`/`unlock()` Operation)
- Nur ein Schreiber (Sicherstellung z.B. mit Locks)

Funktionsweise von RCU

1. Atomares Ändern des Wertes
   - Bestehende Leser sehen alten Wert
   - Neu hinzukommende Leser sehen neuen Wert
   - Grund: Prozessorlokale Caches
2. Warten bis bestehende Leser fertig sind
   - Erkennung erfolgt **indirekt**
3. Falls dynamisch allozierte Elemente: Freigabe des alten Elements

$\Rightarrow$ Verzögern von Schreibern um Leser zu beschleunigen

Funktionsweise von RCU

- Alter Wert noch für manche Leser sichtbar
- Nach Ablauf einer bestimmten Zeitspanne neuer Wert für alle sichtbar

grace period (dt. Gnadenfrist)

Der Zeitraum von Beginn der Änderung bis hin zu dem Punkt, an dem der Schreiber sicher sein kann, dass kein vorher existierender Leser mehr aktiv ist, nennt sich *grace period*.

- Dauer der *grace period* **indirekt** durch sogenannte *quiescent states* erkennbar

Quiescent states

quiescent state (dt. Ruhezustand)

Ein *quiescent state* ist ein Zustand, in dem garantiert ist, dass vorherige Operationen vollendet wurden.

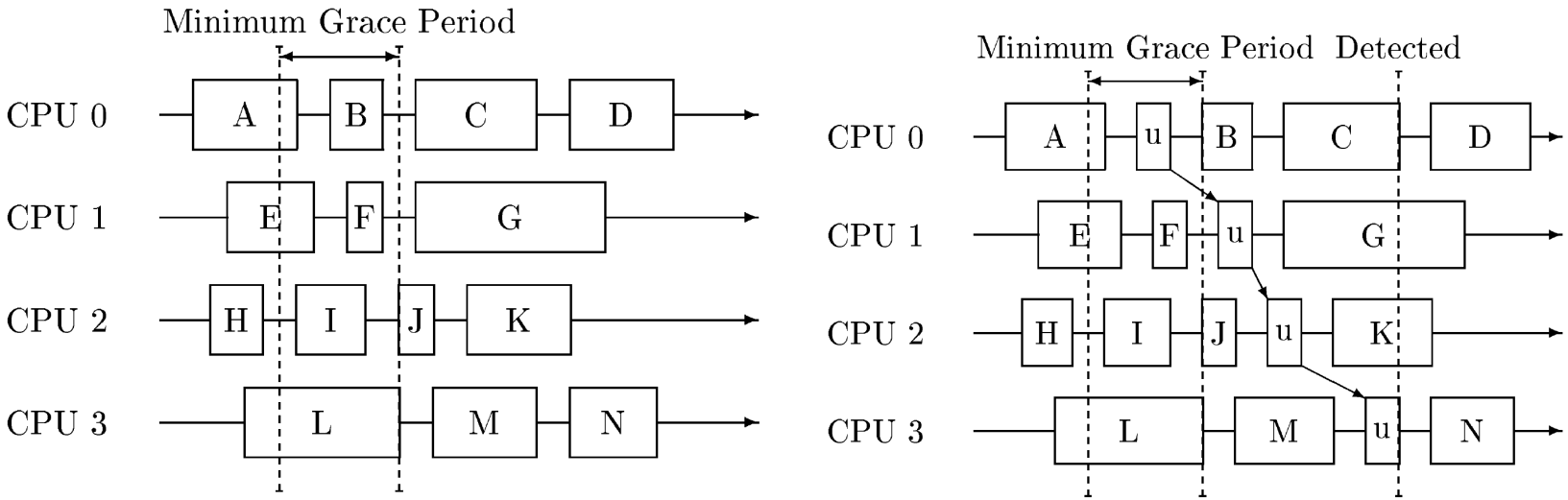
- Operationen, die einen *quiescent state* darstellen:
  - Prozesswechsel
  - Wechsel in Benutzermodus
  - Ausführen der Idle Loop
- In Betriebssystemkernen regelmäßig auftretend und leicht erkennbar

Vorgehen um Ende der *grace period* erkennen

- Schreiber muss warten, bis **jede andere CPU** einen *quiescent state* durchlaufen hat.
- Am einfachsten zu erreichen, indem Schreiber den Scheduler anweist, ihn auf jeder CPU der Reihe nach auszuführen.
- Sobald Schreiber auf jeder CPU ausgeführt wurde, ist die *grace period* vorbei.

- **Wichtig:** Leser darf innerhalb von kritischen Abschnitten **niemals blockieren**

Grace period

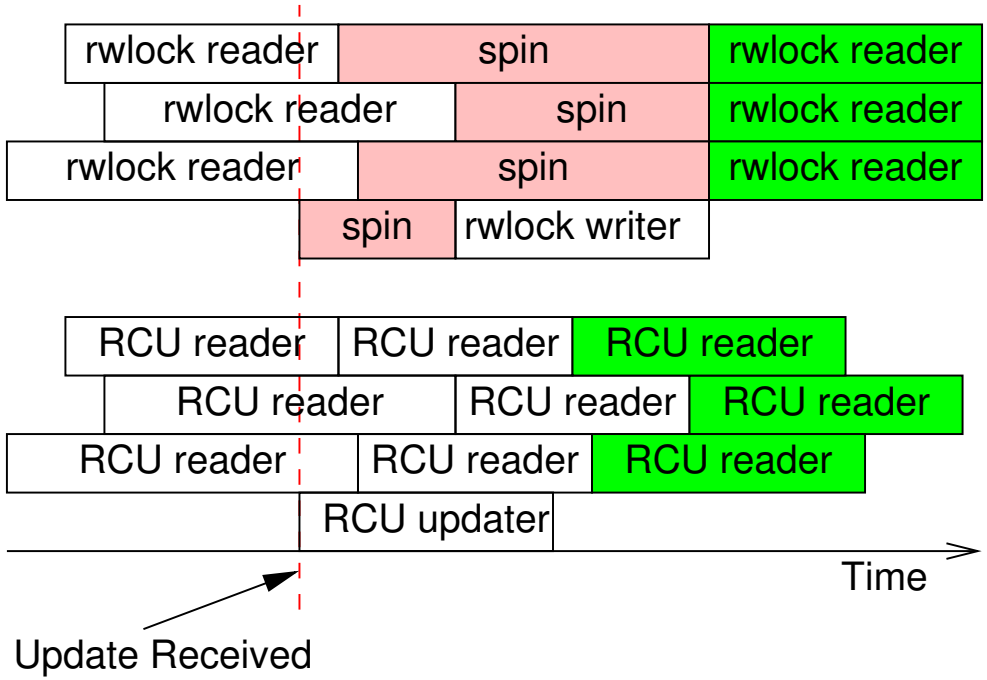


Erkennung der *grace period* durch Prozesswechsel auf jede CPU

Nachteile von RCU

- Warten auf Leser aufwändig: im Millisekundenbereich
- Bei vielen Schreiboperationen hoher Zeitaufwand oder Speicherverbrauch
- Lese- und Schreiboperationen parallel zueinander
- Nur bedingt nutzbar im Benutzermodus
- Sinnvoll wenn Leser:Schreiber $\geq$ #CPUs:1
  - Da indirekte Erkennung der *grace period* dort schwieriger
- Leser sehen teilweise veraltete Werte

Veraltete Werte für Leser



Updateverlauf RCU vs. rwlock [7]

RCU-API

Für Leser:

- `rcu_read_lock()`
- `rcu_read_unlock()`
- `rcu_dereference()`

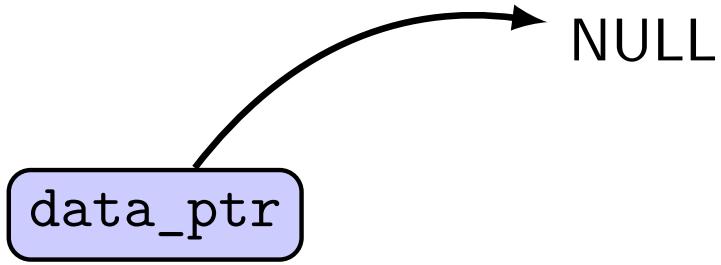
Für Schreiber:

- `synchronize_rcu()`
- `rcu_assign_pointer()`

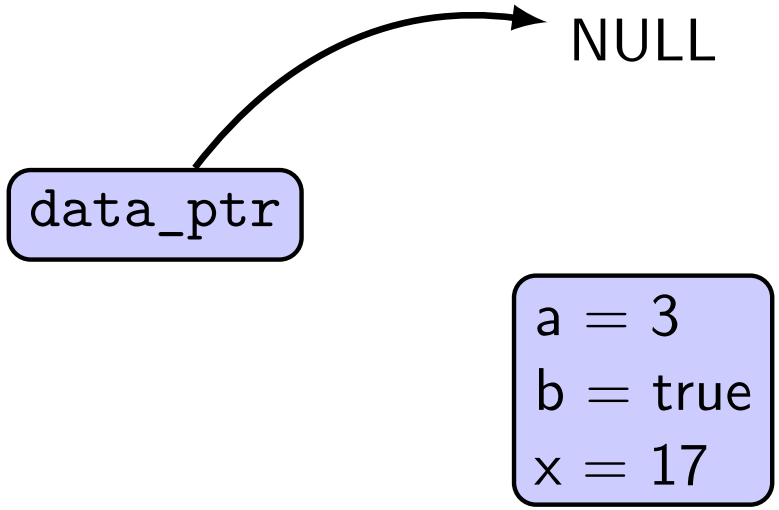
Beispiel: einzelner Zeiger

Beispiel: RCU-geschützter Zeiger

Beispiel: einzelner Zeiger

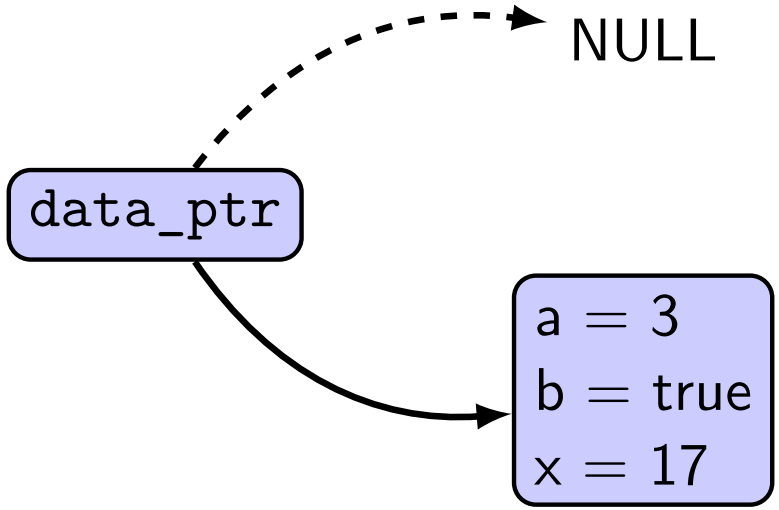


Beispiel: einzelner Zeiger



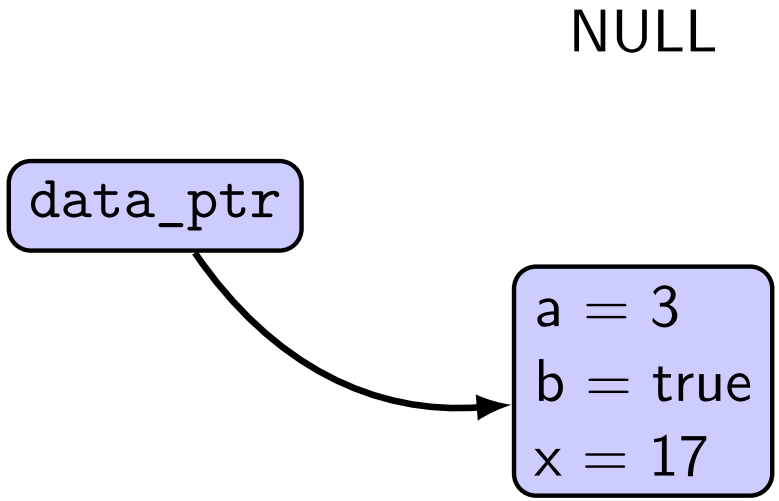
Vor `rcu_assign_pointer()`

Beispiel: einzelner Zeiger



Nach `rcu_assign_pointer()`

Beispiel: einzelner Zeiger

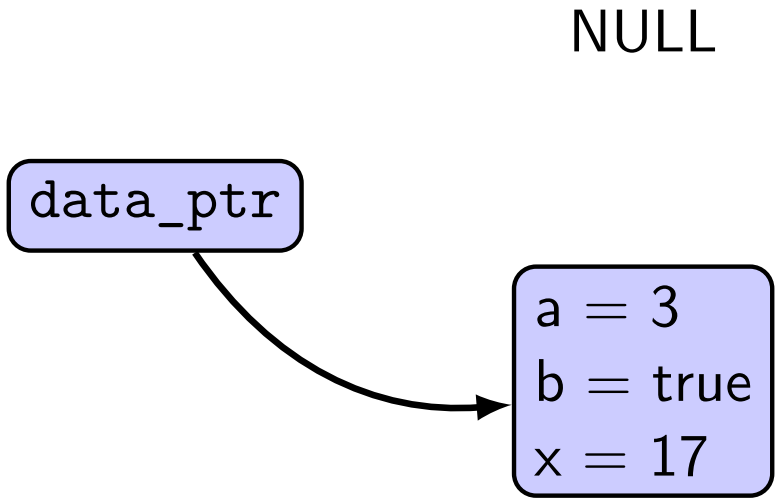


Nach Ablauf einer gewissen Zeitspanne

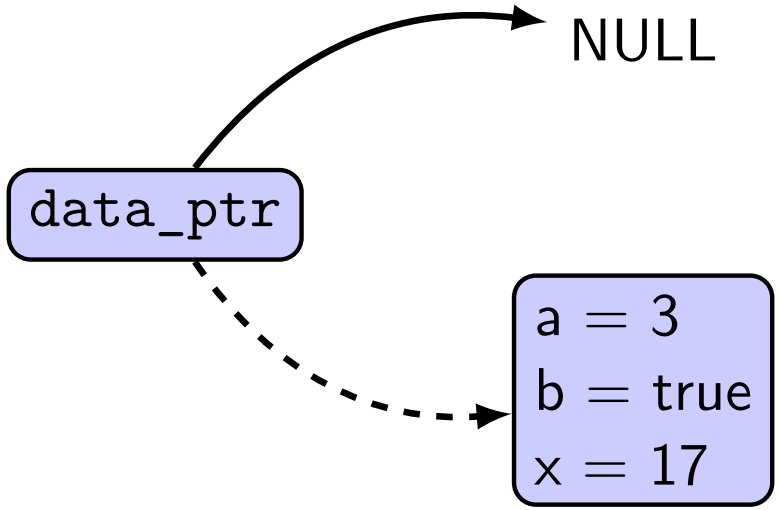
Beispiel: einzelner Zeiger

- Einmaliges Setzen des Zeigers unproblematisch
  - Auch ohne RCU möglich
- **Problem:** Freigeben eines RCU-geschützen Zeigers
  - Wann sieht niemand mehr den alten Wert?

Beispiel: einzelner Zeiger

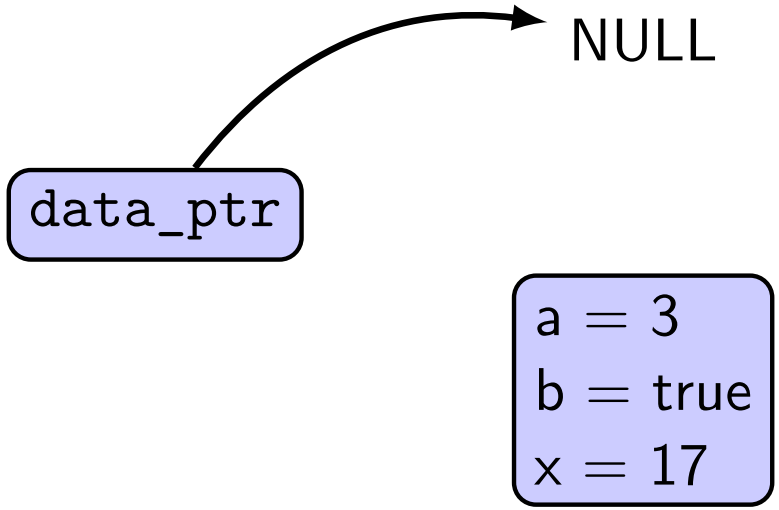


Beispiel: einzelner Zeiger



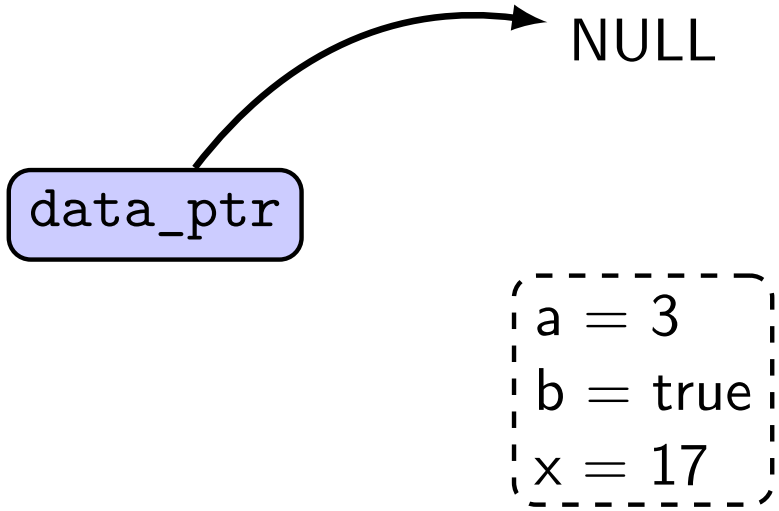
Nach `rcu_assign_pointer()`, vor `synchronize_rcu()`

Beispiel: einzelner Zeiger



Nach `synchronize_rcu()`

Beispiel: einzelner Zeiger

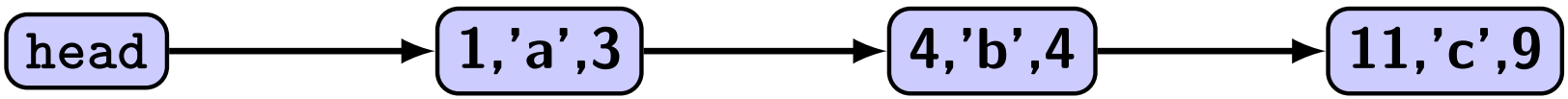


Nach `kfree()`

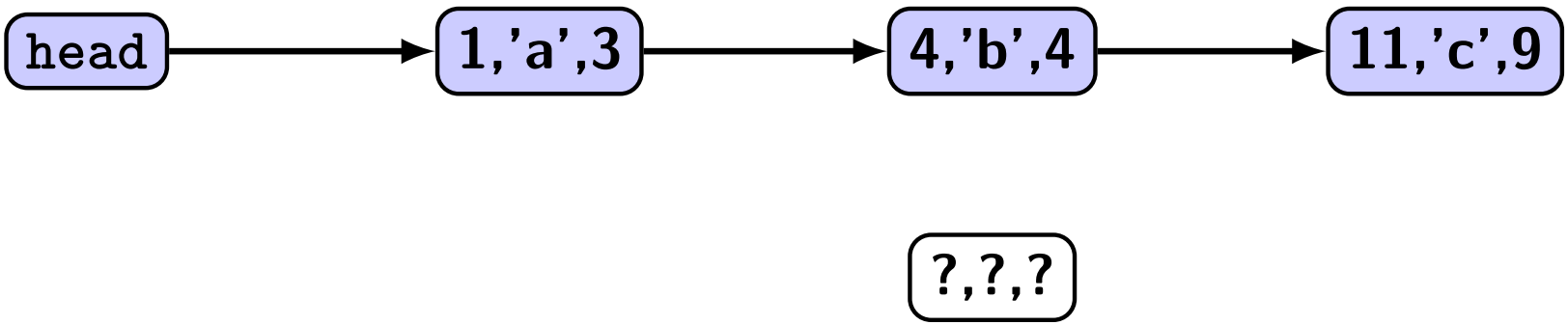
Ändern eines Elements

Ändern eines Elements aus der Liste [8]

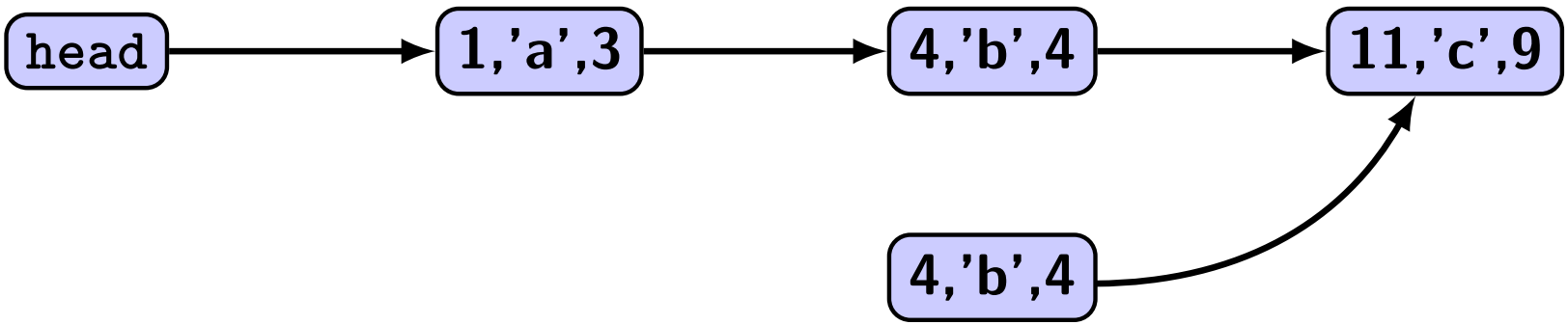
Ändern eines Elements



Ändern eines Elements



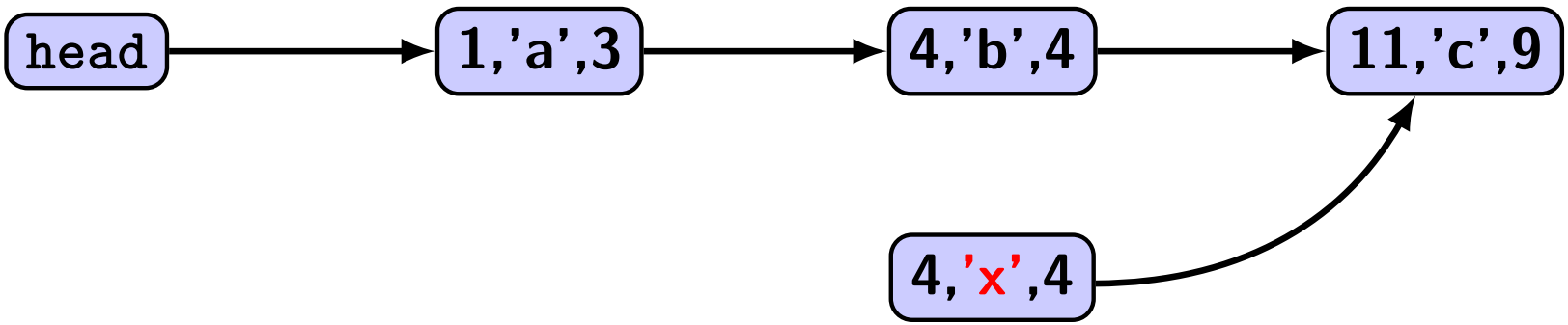
Ändern eines Elements



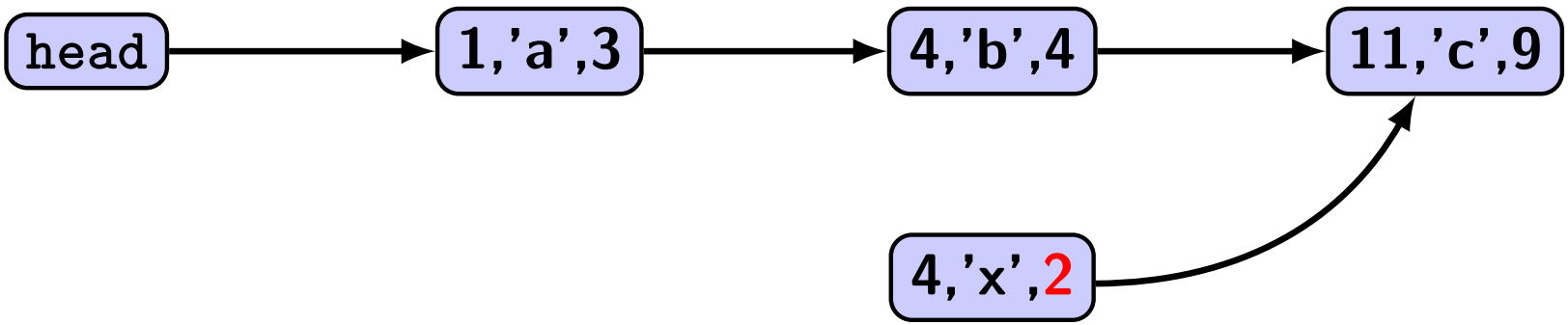
Namensgebende Operation:

The structure is read concurrently with a thread copying in order to do an update, hence the name "read-copy update" [9]

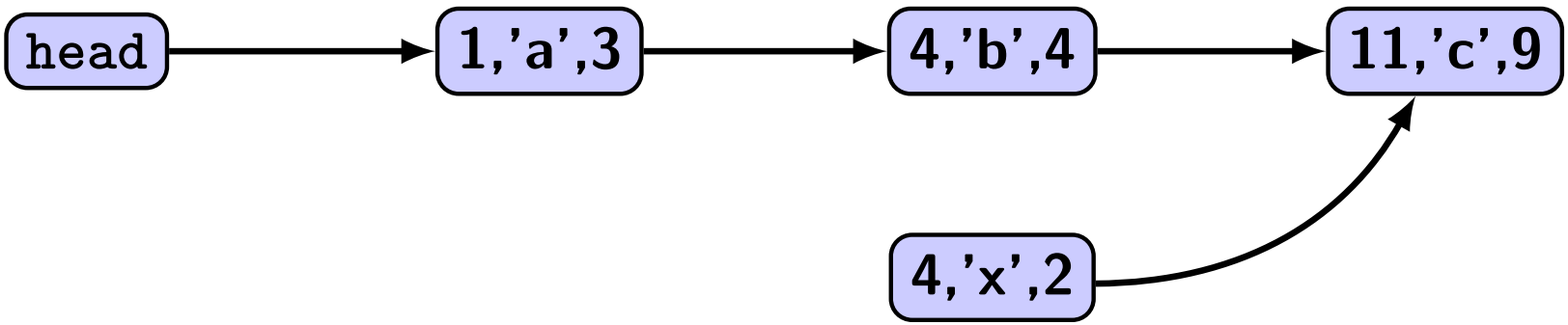
Ändern eines Elements



Ändern eines Elements

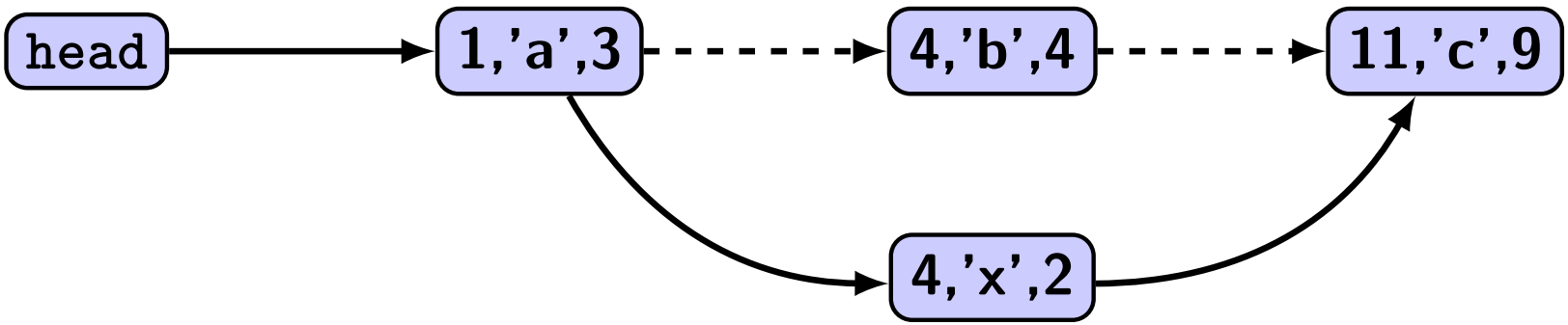


Ändern eines Elements



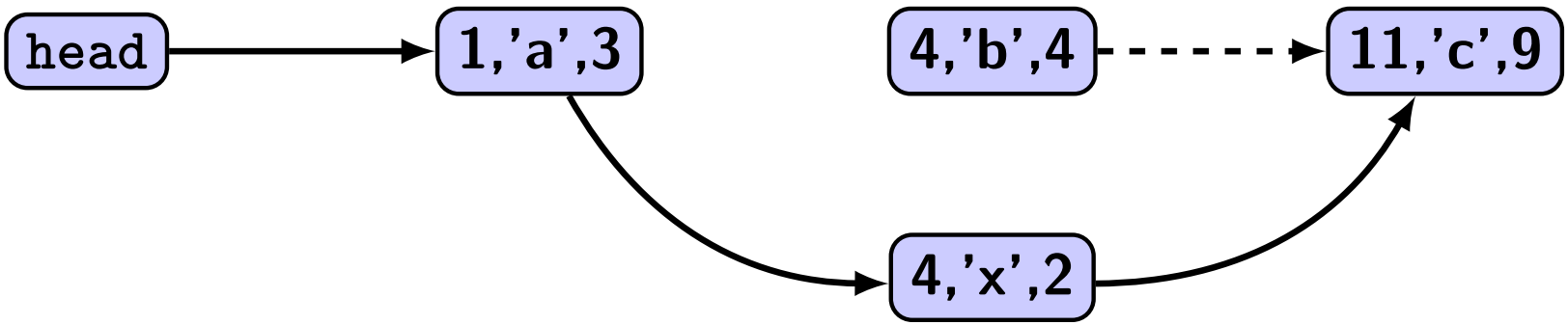
Vor `rcu_assign_pointer()`

Ändern eines Elements



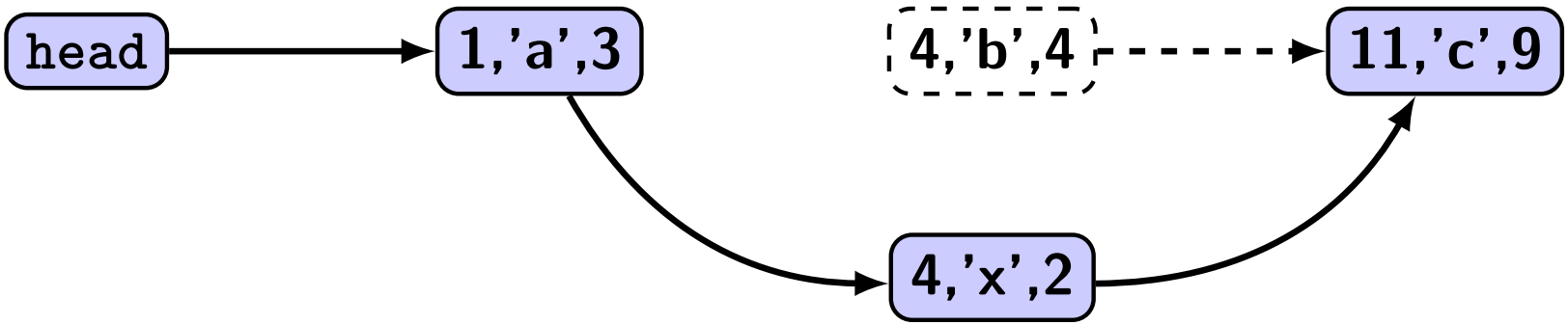
Nach `rcu_assign_pointer()`, vor `synchronize_rcu()`

Ändern eines Elements



Nach `synchronize_rcu()`

Ändern eines Elements



Nach `kfree()`

Implementierung der Leser-API ohne Verdrängung

```
inline void rcu_read_lock() {
}

inline void rcu_read_unlock() {
}

template<typename T>
inline T* rcu_dereference(T* p) {
  //Compiler darf Code nicht über Barriere verschieben
  barrier();
  //Speicherzugriff erzwingen
  return ACCESS_ONCE(p);
}
```

- **Kein Aufwand** für Leser

Implementierung der Leser-API mit Verdrängung

```
inline void rcu_read_lock() {
 preempt_disable();
}

inline void rcu_read_unlock() {
 preempt_enable();
}

template<typename T>
inline T* rcu_dereference(T* p) {
  //Compiler darf Code nicht über Barriere verschieben
  barrier();
  //Speicherzugriff erzwingen
  return ACCESS_ONCE(p);
}
```

- Aufwand für Leser lediglich **Inkrementieren einer prozesslokalen Variablen**

Implementierung der Schreiber-API mit/ohne Verdrängung

```
void synchronize_rcu() {
  int cpu;
  for_each_online_cpu(cpu)
   run_on(cpu);
}
template<typename T>
inline void rcu_assign_pointer(T*& p, T* v) {
  //write memory barrier für CPU
  smp_wmb();
  //Speicherzugriff erzwingen
  ACCESS_ONCE(p) = v;
}
```

- Sehr hoher Aufwand für `synchronize_rcu()`
- Recht geringer Aufwand für `rcu_assign_pointer()`

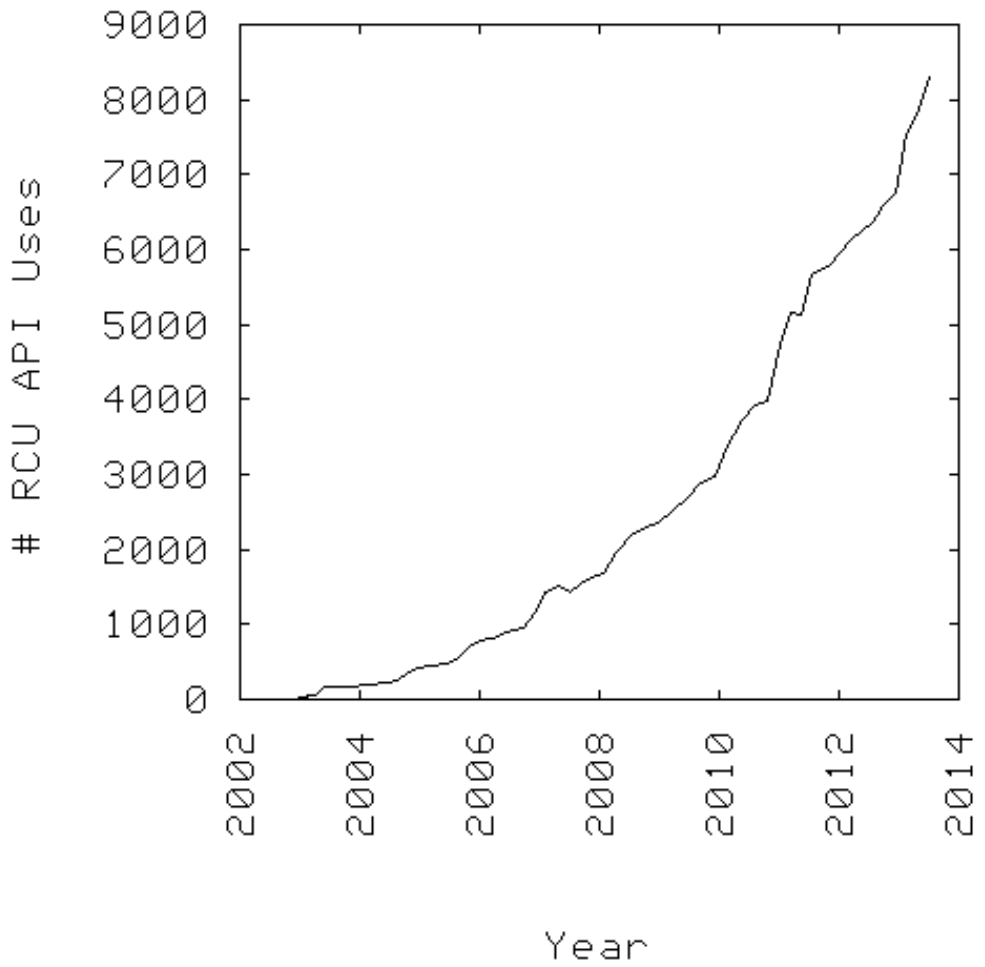
Asynchrone Schnittstelle

- `synchronize_rcu()` pro gelöschtem Element zu aufwändig
- Lösung: Liste mit Callbacks $\Rightarrow$ asynchrones Freigeben
  - `call_rcu()`

Andere Anwendungsgebiete von RCU

- Eine Art Referenzzähler
- Existenzgarantie für Elemente in kritischem Abschnitt
- Warten auf Beendingung einer Operation
  - Benachrichtigung aber **nicht sofort nach Ende**
- Eine Art Garbage Collector

Verwendung in der Praxis



Verwendung von RCU im Linux Kernel [10]

Verwendung in der Praxis

PID	Name	Benutzername	Prozessorzeit
11	rcu_preempt	root	0:12
12	rcu_bh	root	0:00
13	rcu_sched	root	0:00

- Seit Linux 3.6 Sichtbarkeit für Benutzer [11]
  - Grund: Bessere Echtzeitlatenz + einfacherer Code

Weiterführende Informationen

- Verzeichnis `Documentation/RCU` im Linux Kernel Quellcode
- Paul McKenneys Website über RCU
  `http://www2.rdrop.com/users/paulmck/RCU/`
- Linux Quellcode lesen (viele hilfreiche Kommentare vorhanden)
  - Am leichtesten in einer IDE oder online auf
    `http://lxr.free-electrons.com/`

Literaturverzeichnis I

[1] Mathieu Desnoyer. *Userspace RCU*. 2013. URL: http://lttng.org/urcu.

[2] H. T. Kung und Q. Lehman. „Concurrent Maintenance of Binary Search Trees“. In: *ACM Transactions on Database Systems* 5.3 (Sep. 1980), S. 354–382.

[3] John D. Slingwine und Paul E. Mckenney. „Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring“. Pat. 5442758. 1995. URL: http://www.freepatentsonline.com/5442758.html.

[4] Dipankar Sarma (IBM). *Linux kernel-history.git commit 1477a825d7e6486a077608c7baf6abbb6f27ed95*. Okt. 2002.

Literaturverzeichnis II

[5] Paul E. McKenney. *Accommodating the Laws of Physics: RCU*. Jan. 2013. URL: http://www2.rdrop.com/users/paulmck/RCU/RCU.2013.01.22d.PLMW.pdf.

[6] Paul E. McKenney. *What is RCU? Part 2: Usage*. 2007. URL: http://lwn.net/Articles/263130/.

[7] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Available: http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html. Corvallis, OR, USA: kernel.org, 2011.

[8] Paul E. McKenney. *What is RCU, Fundamentally?* 2007. URL: http://lwn.net/Articles/262464/.

[9] Paul E. McKenney. *What is RCU, Really?* URL: http://www.rdrop.com/users/paulmck/RCU/whatisRCU.html.

Literaturverzeichnis III

[10] Paul E. McKenney. *Linux RCU Usage*. 2013. URL: http://www2.rdrop.com/users/paulmck/RCU/linuxusage.html.

[11] Paul E. McKenney. *The new visibility of RCU processing*. 2012. URL: http://lwn.net/Articles/518953/.