Read-copy-update und seine Anwendungen

Ein Vortrag im KVBK-Seminar

Christian Bewermeyer

Friedrich-Alexander-Universität Erlangen-Nürnberg

Agenda

Einführung
- Grundkonzept
- animiertes Beispiel
- Schonfristen

RCU in Linux
- API
- Vereinfachte Implementierung
- Anwendungsbeispiele

Vor- und Nachteile

Userlevel-RCU

Einführung
- Grundkonzept
- animiertes Beispiel
- Schonfristen

RCU in Linux
- API
- Vereinfachte Implementierung
- Anwendungsbeispiele

Vor- und Nachteile

Userlevel-RCU

Anforderungen an Synchronisationsmechanismen

- Performance
  - geringer Ausführungsoverhead
  - geringer Speicherverbrauch
  - möglichst geringe Verzögerung von Lesern (auch *während* Updates)
- deterministische Ausführungszeit für Leseoperationen

Anforderungen an Synchronisationsmechanismen

- Performance
  - geringer Ausführungsoverhead
  - geringer Speicherverbrauch
  - möglichst geringe Verzögerung von Lesern (auch *während* Updates)
- deterministische Ausführungszeit für Leseoperationen

Problem

Kein lockbasierter Synchronisationsmechanismus erfüllt diese Anforderungen.

RCU: Grundkonzept

- Read-Copy-Update: Synchronisationsverfahren
- Hauptziel: möglichst geringe Verlangsamung von Lesern

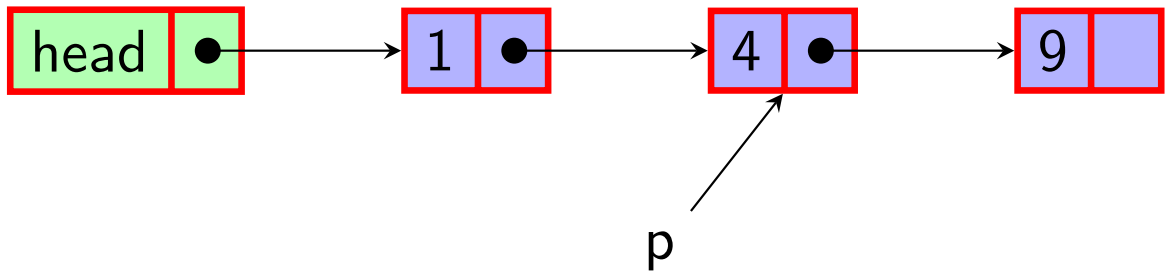
RCU: Grundkonzept

- Read-Copy-Update: Synchronisationsverfahren
- Hauptziel: möglichst geringe Verlangsamung von Lesern
- Grundidee:
  - verwalte mehrere Versionen von Datenobjekten
  - stelle sicher, dass Versionen intakt bleiben solange Leser sie referenzieren

RCU: Grundkonzept

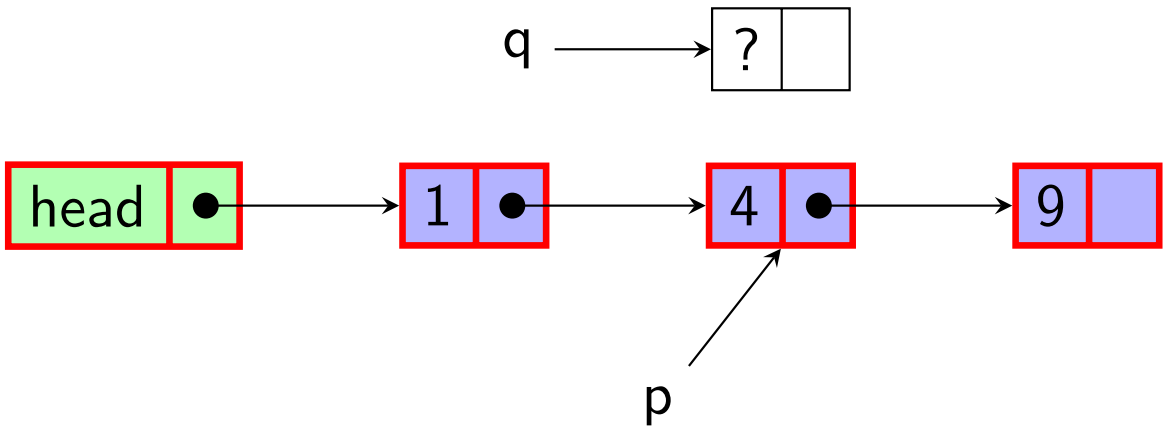
- Read-Copy-Update: Synchronisationsverfahren
- Hauptziel: möglichst geringe Verlangsamung von Lesern
- Grundidee:
  - verwalte mehrere Versionen von Datenobjekten
  - stelle sicher, dass Versionen intakt bleiben solange Leser sie referenzieren
- Vorgehen bei Änderungen:
  - Updater erzeugt neue Version des zu ändernden Datenelements
  - nachfolgende Leser sehen nur diese neue Version
  - vorher existierende Leser können weiter auf alte Version zugreifen
  - sobald kein Leser mehr Referenz auf alte Version hält: Freigabe des Speichers

Beispiel: Ändern von verketteter Liste I



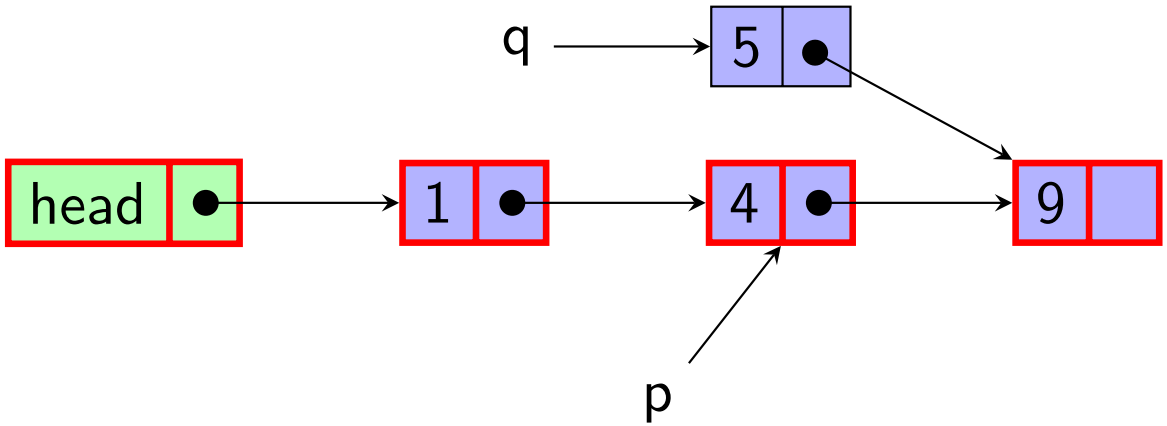
- Modifizieren eines Elements einer verketteten Liste
- Listenelemente sind Integerwerte
- p: Zeiger auf zu modifizierendes Element
- roter Rand: Leser können Referenzen auf jeweiliges Element halten

Beispiel: Ändern von verketteter Liste II



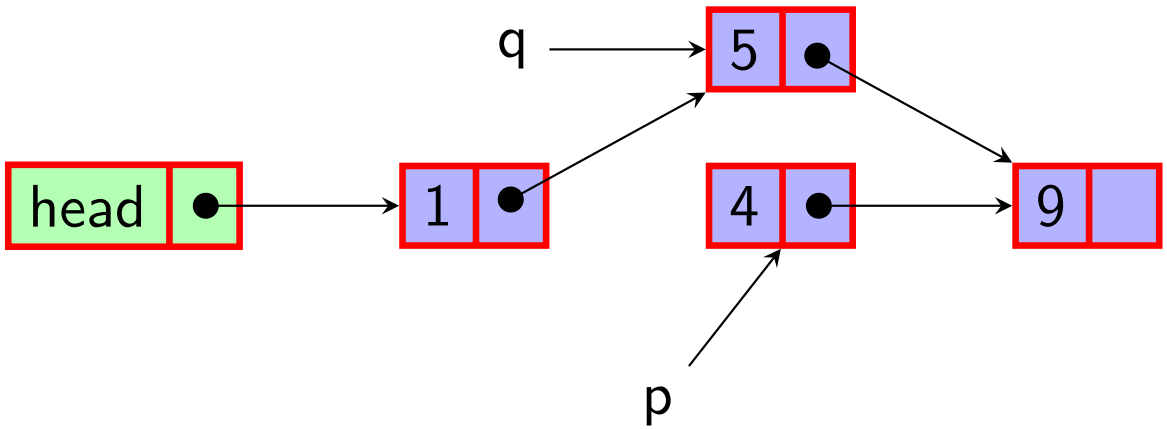
- Allozieren einer Kopie des zu modifizierenden Elements

Beispiel: Ändern von verketteter Liste III



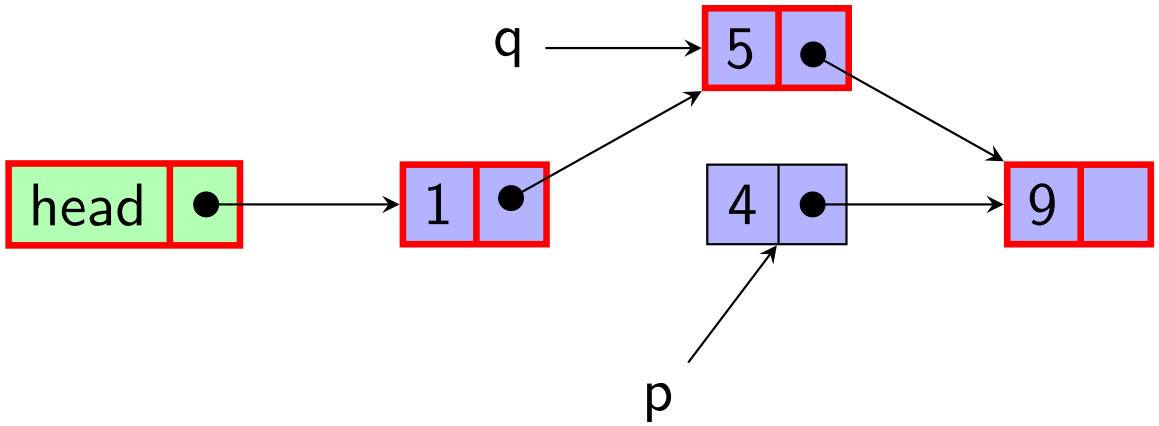
- Setze Membervariablen der Kopie auf ihre neuen Werte
- Setze next-Pointer

Beispiel: Ändern von verketteter Liste IV



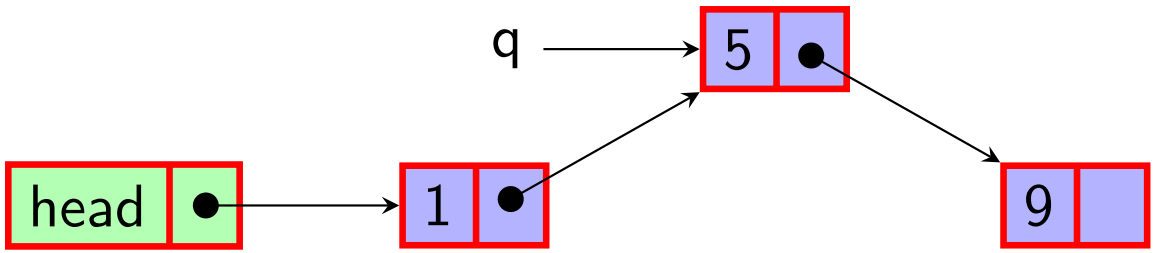
- eigentliche Ersetzung: Setze next-Pointer des Vorgängers
- Ab jetzt: 2 Versionen der Liste

Beispiel: Ändern von verketteter Liste V



- nach gewisser Zeit: kein Leser hält mehr Referenzen auf altes Listenelement
- für Leser existiert nur noch eine Version der Liste

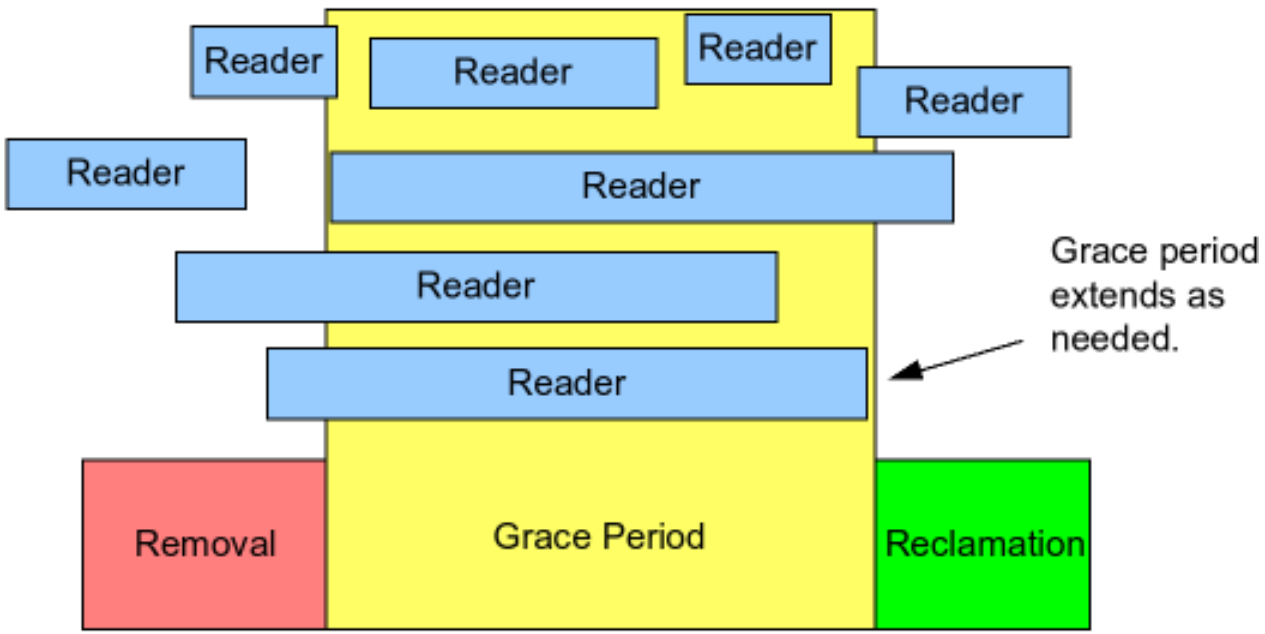
Beispiel: Ändern von verketteter Liste VI



- Zum Schluss: freigeben des Speichers

Das Konzept der Schonfrist I

- Problem: Wie erkennt man das kein Leser mehr Referenzen auf veraltete Version hält?
- Ansatz: Leseoperationen werden in kritischem Abschnitt ausgeführt
- Folgerung: Wenn alle vor Beginn der Schreiboperation begonnenen kritischen Abschnitte *vollständig* ausgeführt sind, hält kein Leser mehr Referenzen auf alte Version



Das Konzept der Schonfrist II

Schonfrist (engl. *grace period*)

Intervall bis alle vorher existierenden leserseitgen kritischen Abschnitte abgeschlossen sind

- Ausführungszeit kritischer Abschnitte endlich $\rightsquigarrow$ Schonfrist endlich
- Wie erkennt man das Ende der Schonfrist?
  - z.B. über Thread-Kontextwechsel
  - Vorraussetzung: kritische Abschnitte sind nicht-präemptiv und nicht-blockierend
  - Dann: Schonfrist endet, wenn jede CPU mindestens 1 Kontextwechsel ausgeführt hat

Einführung
Grundkonzept
animiertes Beispiel
Schonfristen

RCU in Linux
API
Vereinfachte Implementierung
Anwendungsbeispiele

Vor- und Nachteile

Userlevel-RCU

Überblick RCU-API

- *rcu_read_lock*: Beginn von kritischem Abschnitt
- *rcu_read_unlock*: Ende von kritischem Abschnitt
- *synchronize_rcu:* Warte bis alle vorher angefangenen leserseitgen kritischen Abschnitte beendet sind
- *rcu_dereference/rcu_assign_pointer*: verhindern das Leser auf uninitialisierte Daten zugreifen können (mittels Speicherbarrieren/Compilerdirektiven)
- uvm. . . (z.B. Abstraktionen für Listenzugriff)

Implementierung

Implementierungsskizze für präemptive Systeme

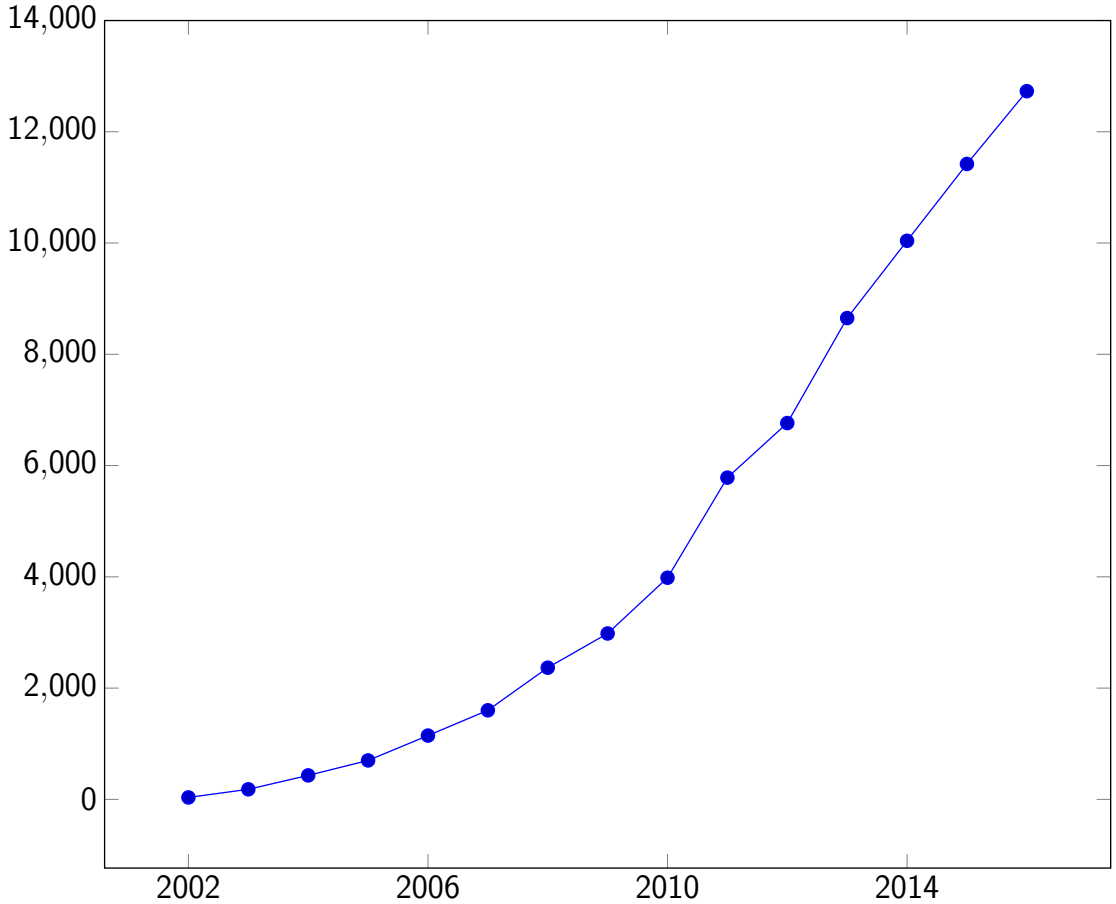
```
void rcu_read_lock()
{
      preempt_disable[cpu_id()]++;
}

void rcu_read_unlock()
{
      preempt_disable[cpu_id()]--;
}

void synchronize_rcu(void)
{
      for_each_cpu(int cpu)
            run_on(cpu);
}
```

- Kontextwechsel dienen zur Erkennung des Endes von Schonfristen
- bei nicht-präemptiven Systemen: leserseitigen Primitiven generieren keinen Code!

Anwendungsbeispiele

- Dynamisches Laden von Kernelmodulen
  - RCU verhindert Wettlaufsituation bei Entfernung eines Moduls, das noch in Benutzung ist
  - ersetzt Referenzzähler, die atomar modifiziert werden
- Non-maskable Interrupts (NMIs)
  - erlaubt dynamisch deregistrieren von NMI-Handlern
  - früher nicht möglich (Deadlock-Gefahr!)
- Netzwerk (z.B. Routingtabellen)
- Syscall-Tabellen, System-V-API, Kernel-basierte Virtualisierung uvm.

Statistik



Anzahl RCU-Aufrufe nach Jahren

- über die Hälfte der Aufrufe von RCU-Primitiven im Netzwerk-Stack

Einführung
- Grundkonzept
- animiertes Beispiel
- Schonfristen

RCU in Linux
- API
- Vereinfachte Implementierung
- Anwendungsbeispiele

Vor- und Nachteile

Userlevel-RCU

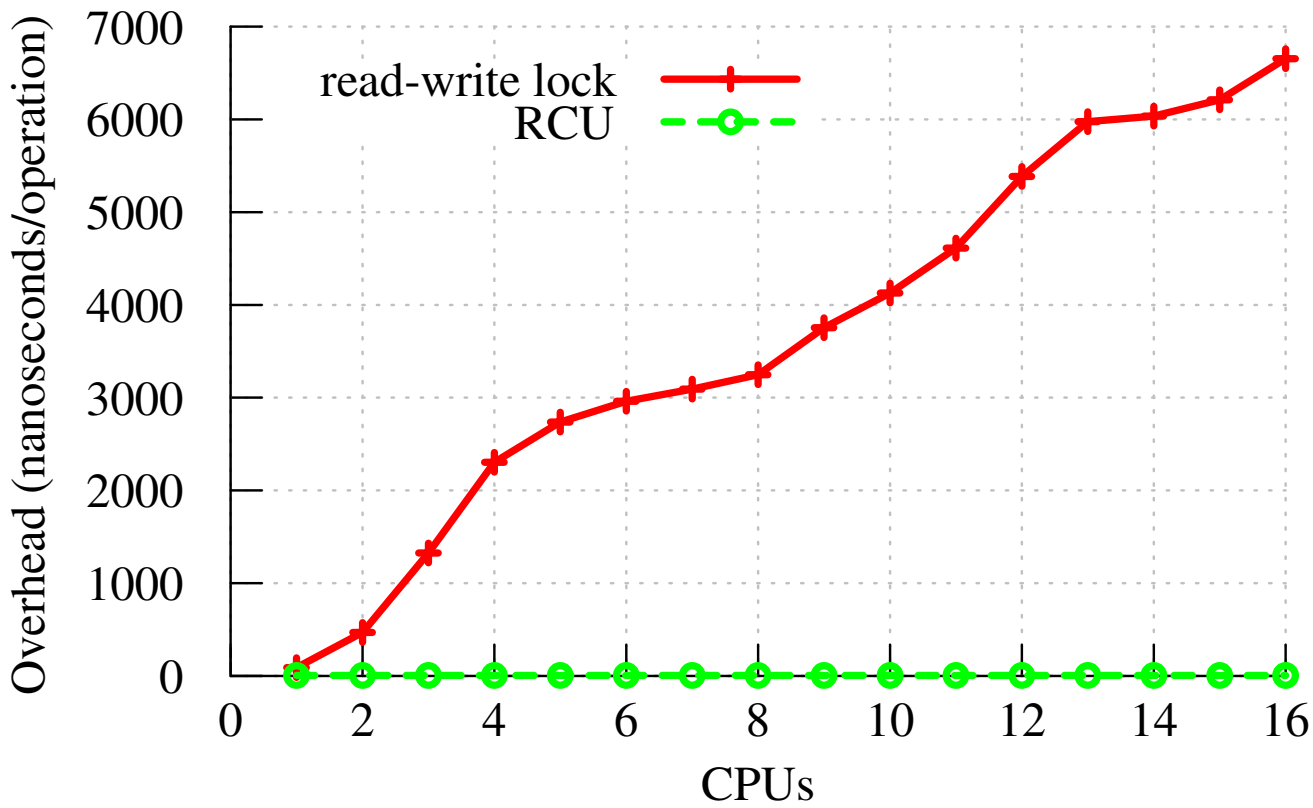
Vorteile

- führt zu großen Performancevorteilen bei *lese-intensiven* Datenstrukturen
- Faustregel: Eine Datenstruktur ist lese-intensiv, wenn

$$\frac{\#\text{Schreibzugriffe}}{\#\text{Zugriffe}} \ll \frac{1}{\#\text{CPUs}}$$

- Geringer Speicheroverhead für Leser
- Deterministische Ausführungszeit bei Lesern (unabh. von Anzahl der CPUs)
- Prioritäten: hochpriore Leser werden nicht von Schreibern mit niedriger Priorität verzögert

Vergleich: Locks vs. RCU



Nachteile

- RCU ist pessimistisch:
  - Updater werden länger als nötig verzögert
  - Schonfristen dauern u.U. mehrere Millisekunden
- Anwendungen müssen veraltete Daten tolerieren können
- Anderes Verfahren muss Schreiber serialisieren (z.B. Spinlocks)

Einführung
- Grundkonzept
- animiertes Beispiel
- Schonfristen

RCU in Linux
- API
- Vereinfachte Implementierung
- Anwendungsbeispiele

Vor- und Nachteile

Userlevel-RCU

Userlevel-RCU

- Userspace RCU Library (URCU, seit 2009): Implementierung von RCU für Benutzung in normalen Userspace-Anwendungen
- in Debian als Paket `liburcu2`
- Herausforderung: Annahmen aus Kernelspace nicht 1:1 auf Userspace übertragbar
  - Threads können im kritischen Abschnitt verdrängt werden
  - direkter Zugriff auf RCU-Primitiven des Kernels keine gute Idee (Syscall-Overhead)
- Tradeoff: Performance vs. wenig Einschränkungen für Anwender
- Beispiel für Userlevel-Implementierung: *quiescent-state-based RCU*
  - verlangt von Threads regelmäßig bekanntzugeben, dass sie nicht in krit. Abschnitt sind
  - selbe Performance wie Kernel-RCU

Schluss

- RCU: Synchronisationsverfahren mit hohen Performancevorteilen bei *lese-intensiven* Datenstrukturen
- sehr gut geeignet für Betriebssytemkernel
- eingesetzt in diversen Linux-Subsystemen
- mehr Informationen: `http://www.rdrop.com/~paulmck/RCU/`