What Is RCU?
What is RCU?

Overview

- Mutual Exclusion
- Example Application
- Performance of Synchronization Mechanisms
- Making Software Live With Current (and Future) Hardware
- Implementing RCU
- RCU Grace Periods: Conceptual and Graphical Views
- Performance
- RCU Area of Applicability
- The Issaquah Challenge
- Summary
What is RCU?

Mutual Exclusion
Can you create a trivial lock-based deque allowing concurrent pushes and pops at both ends?
- Coordination required if the deque contains only one or two elements
- But coordination is not required for three or more elements
What is RCU?

Mutual Exclusion Challenge: Double-Ended Queue

- Can you create a trivial lock-based deque allowing concurrent pushes and pops at both ends?
  - Coordination required if the deque contains only one or two elements
  - But coordination is not required for three or more elements

![Diagram of deque with elements A, B, C, D showing different states of the deque]

Pointless problem, but solution on later slide...
What is RCU?

Mutual Exclusion Question

- What mechanisms can enforce mutual exclusion?
What is RCU?

Example Application
What is RCU?

Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example in upcoming ACM Queue)
  - Births result in insertions, deaths in deletions
  - Queries from those interested in Schrödinger's animals
  - Lots of short-lived animals such as mice: High update rate
  - Great interest in Schrödinger's cat (perhaps queries from mice?)
What is RCU?

Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example in upcoming ACM Queue)
  - Births result in insertions, deaths in deletions
  - Queries from those interested in Schrödinger's animals
  - Lots of short-lived animals such as mice: High update rate
  - Great interest in Schrödinger's cat (perhaps queries from mice?)

- Simple approach: chained hash table with per-bucket locking
What is RCU?

Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example in upcoming ACM Queue)
  - Births result in insertions, deaths in deletions
  - Queries from those interested in Schrödinger's animals
  - Lots of short-lived animals such as mice: High update rate
  - Great interest in Schrödinger's cat (perhaps queries from mice?)

- Simple approach: chained hash table with per-bucket locking

```
0: lock
1: lock
2: lock
3: lock
```

```
mouse
zebra
boa
cat
gnu
```

Will holding this lock prevent the cat from dying?
What is RCU?

Read-Only Bucket-Locked Hash Table Performance

2GHz Intel Xeon Westmere-Ex, 1024 hash buckets

Why the dropoff???
What is RCU?

Varying Number of Hash Buckets

2GHz Intel Xeon Westmere-EX

Some improvement, but...
NUMA Effects???

- /sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list: 0,32
- /sys/devices/system/cpu/cpu0/cache/index1/shared_cpu_list: 0,32
- /sys/devices/system/cpu/cpu0/cache/index2/shared_cpu_list: 0,32
- /sys/devices/system/cpu/cpu0/cache/index3/shared_cpu_list: 0-7,32-39

- Two hardware threads per core, eight cores per socket
- Try using only one CPU per socket: CPUs 0, 8, 16, and 24
What is RCU?

Bucket-Locked Hash Performance: 1 CPU/Socket

This is not the sort of scalability Schrödinger requires!!!
What is RCU?

Locking is BAD: Use Non-Blocking Synchronization!
What is RCU?

Use Non-Blocking Synchronization!

- Big issue: Lookups run concurrently with deletions
  - Bad form for a lookup to hand back a pointer to free memory
What is RCU?

Use Non-Blocking Synchronization!

- Big issue: Lookups run concurrently with deletions
  - Bad form for a lookup to hand back a pointer to free memory
  - Results in lookups writing to shared memory, usually atomically

![Diagram of Hash Table with Lookups and Deletions]
**Performance of Synchronization Mechanisms**

16-CPU 2.8GHz Intel X5550 (Nehalem) System

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (ns)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period</td>
<td>0.4</td>
<td>1</td>
</tr>
<tr>
<td>“Best-case” CAS</td>
<td>12.2</td>
<td>33.8</td>
</tr>
<tr>
<td>Best-case lock</td>
<td>25.6</td>
<td>71.2</td>
</tr>
<tr>
<td>Single cache miss</td>
<td>12.9</td>
<td>35.8</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>7.0</td>
<td>19.4</td>
</tr>
<tr>
<td>Single cache miss (off-core)</td>
<td>31.2</td>
<td>86.6</td>
</tr>
<tr>
<td>CAS cache miss (off-core)</td>
<td>31.2</td>
<td>86.5</td>
</tr>
<tr>
<td>Single cache miss (off-socket)</td>
<td>92.4</td>
<td>256.7</td>
</tr>
<tr>
<td>CAS cache miss (off-socket)</td>
<td>95.9</td>
<td>266.4</td>
</tr>
</tbody>
</table>

And these are best-case values!!! (Why?)
What is RCU?

Why All These Low-Level Details???
What is RCU?

Why All These Low-Level Details???

- Would you trust a bridge designed by someone who did not understand strengths of materials?
  - Or a ship designed by someone who did not understand the steel-alloy transition temperatures?
  - Or a house designed by someone who did not understand that unfinished wood rots when wet?
  - Or a car designed by someone who did not understand the corrosion properties of the metals used in the exhaust system?
  - Or a space shuttle designed by someone who did not understand the temperature limitations of O-rings?
What is RCU?

Why All These Low-Level Details???

- Would you trust a bridge designed by someone who did not understand strengths of materials?
  - Or a ship designed by someone who did not understand the steel-alloy transition temperatures?
  - Or a house designed by someone who did not understand that unfinished wood rots when wet?
  - Or a car designed by someone who did not understand the corrosion properties of the metals used in the exhaust system?
  - Or a space shuttle designed by someone who did not understand the temperature limitations of O-rings?

- So why trust algorithms from someone ignorant of the properties of the underlying hardware???
What is RCU?

But What Do The Operation Timings Really Mean???
What is RCU?

But What Do The Operation Timings Really Mean???

- Single-instruction critical sections protected by data locking

Uncontended

256.7 cycles

1 cycle

Release

Critical Section

Acquire

Contended, No Spinning

256.7 cycles

1 cycle

256.7 cycles

So, what does this mean?
What is RCU?

But What Do The Operation Timings Really Mean???

- Single-instruction critical sections protected by data locking

Uncontended:

- 256.7 cycles

Contended, No Spinning:

- 256.7 cycles
- 1 cycle
- 256.7 cycles

258 CPUs to break even with single CPU!

514 CPUs to break even with single CPU!!!
What is RCU?

But What Do The Operation Timings Really Mean???

- Single-instruction critical sections protected by data locking

Uncontended

Contended, No Spinning

Contended, Spinning

Arbitrarily large number of CPUs to break even with single CPU!!

258 CPUs to break even with single CPU!

514 CPUs to break even with single CPU!!

© 2017 IBM Corporation
What is RCU?

Reader-Writer Locks Are Even Worse!
What is RCU?

Reader-Writer Locks Are Even Worse!

CPU 0

Acquire

Critical Section

Wait for Lock Data

Release

266.4 cycles

1 cycle

266.4 cycles

266.4 cycles

800 CPUs to break even with a single CPU!!!
What is RCU?

But What About Scaling With Atomic Operations? Non-Blocking Synchronization For The Win!!!
What is RCU?

If You Think Single Atomic is Expensive, Try Lots!!!
What is RCU?

Why So Slow???
What is RCU?

System Hardware Structure and Laws of Physics

Electrons move at 0.03C to 0.3C in transistors and, so lots of waiting. 3D???
Atomic Increment of Global Variable

Lots and Lots of Latency!!!
What is RCU?

Atomic Increment of Per-CPU Counter

Little Latency, Lots of Increments at Core Clock Rate
What is RCU?

Can't The Hardware Do Better Than This???
What is RCU?

**HW-Assist Atomic Increment of Global Variable**

SGI systems used this approach in the 1990s, expect modern CPUs to optimize. Still not as good as per-CPU counters.
What is RCU?

**HW-Assist Atomic Increment of Global Variable**

Put an ALU near memory to avoid slowdowns due to latency. Still not as good as per-CPU counters.
What is RCU?

Problem With Physics #1: Finite Speed of Light
What is RCU?

Problem With Physics #2: Atomic Nature of Matter

Source

Base

Drain

No complaints for eons, and now, suddenly, we're too $\ast\ast\ast\ast$ big?!

I feel so fat!

And our dielectric constant isn't big enough for them! They can go find some other $\ast\ast\ast\ast$ atom! Sheesh!
What is RCU?

How Can Software Live With This Hardware???
What is RCU?

Design Principle: Avoid Bottlenecks

Only one of something: bad for performance and scalability. Also typically results in high complexity.
Many instances of something good!
Avoiding tightly coupled interactions is an excellent way to avoid bugs. Hazard pointers uses this trick with reference counting.
What is RCU?

Design Principle: Avoid Bottlenecks

Many instances of something good!
Avoiding tightly coupled interactions is an excellent way to avoid bugs.
But NUMA effects defeated this for per-bucket locking!!!
### Design Principle: Avoid Expensive Operations

#### 16-CPU 2.8GHz Intel X5550 (Nehalem) System

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (ns)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period</td>
<td>0.4</td>
<td>1</td>
</tr>
<tr>
<td>“Best-case” CAS</td>
<td>12.2</td>
<td>33.8</td>
</tr>
<tr>
<td>Best-case lock</td>
<td>25.6</td>
<td>71.2</td>
</tr>
<tr>
<td>Single cache miss</td>
<td>12.9</td>
<td>35.8</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>7.0</td>
<td>19.4</td>
</tr>
<tr>
<td>Single cache miss (off-core)</td>
<td>31.2</td>
<td>86.6</td>
</tr>
<tr>
<td>CAS cache miss (off-core)</td>
<td>31.2</td>
<td>86.5</td>
</tr>
<tr>
<td>Single cache miss (off-socket)</td>
<td>92.4</td>
<td>256.7</td>
</tr>
<tr>
<td>CAS cache miss (off-socket)</td>
<td>95.9</td>
<td>266.4</td>
</tr>
</tbody>
</table>

*Typical synchronization mechanisms do this a lot*

---

*Need to be here! (Partitioning/RCU/hazptr)*

*But can't always!*

*Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)*
What is RCU?

Design Principle: Avoid Contention

- Locking
  - Spin-wait
  - Acquire contention
  - Release contention

- Non-Blocking Synchronization
  - Preparation
  - Attempt
  - Preparation
  - Retry

- Desired State

*Simple* non-blocking synchronization does very well.
What is RCU?

Design Principle: Get Your Money's Worth

- If synchronization is expensive, use large critical sections
- On Nehalem, off-socket CAS costs about 260 cycles
  - So instead of a single-cycle critical section, have a 26000-cycle critical section, reducing synchronization overhead to about 1%
What is RCU?

Design Principle: Get Your Money's Worth

- If synchronization is expensive, use large critical sections
- On Nehalem, off-socket CAS costs about 260 cycles
  - So instead of a single-cycle critical section, have a 26000-cycle critical section, reducing synchronization overhead to about 1%
- Of course, we also need to keep contention low, which usually means we want short critical sections
  - Resolve this by applying parallelism at as high a level as possible
  - Parallelize entire applications rather than low-level algorithms!
- This does not work for Schrödinger: The overhead of hash-table operations is too low
  - Which is precisely why we selected hash tables in the first place!!!
What is RCU?

**Design Principle: Leverage Read-Mostly Situations**

Read-only data remains replicated in all caches

---

Read-mostly access dodges the laws of physics!!!
What is RCU?

Updates Hit Hard By Unforgiving Laws of Physics

Read-only data remains replicated in all caches, but each update destroys other replicas!
What is RCU?

Design Principle: Leverage Locality!!!

Each CPU operates on its own “shard” of the data, preserving cache locality and performance.
What is RCU?

Updates: Just Say “No”???

- “Doing updates is slow and non-scalable!”
- “Then don't do updates!”
What is RCU?

Updates: Just Say “No”???

- “Doing updates is slow and non-scalable!”
- “Then don't do updates!”

OK, OK, don't do *unnecessary* updates!!!
For example, read-only traversal to update location
What is RCU?

Design Principle: Avoid Mutual Exclusion!!!
### What is RCU?

**Design Principle: Avoiding Mutual Exclusion**

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>Reader</th>
<th>Reader</th>
<th>Reader</th>
<th>Reader</th>
<th>Reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1</td>
<td>Reader</td>
<td>Reader</td>
<td>Reader</td>
<td>Reader</td>
<td>Reader</td>
</tr>
<tr>
<td>CPU 2</td>
<td>Reader</td>
<td>Reader</td>
<td>Reader</td>
<td>Reader</td>
<td>Reader</td>
</tr>
<tr>
<td>CPU 3</td>
<td>Reader</td>
<td>Reader</td>
<td>Updater</td>
<td>Reader</td>
<td>Reader</td>
</tr>
</tbody>
</table>

No Dead Time!
What is RCU?

But How Can This Possibly Be Implemented???
What is RCU?

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()
What is RCU?

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()

- Advantages:
- Disadvantage:
What is RCU?

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_readUnlock()

- Advantages: Best possible performance, scalability, real-time response, wait-freedom, and energy efficiency

- Disadvantage: How can something that does not affect machine state possibly be used as a synchronization primitive???
What is RCU?

What Is RCU?

- Publishing of new data
- Subscribing to the current version of data
- Waiting for pre-existing RCU readers: Avoid disrupting readers by maintaining multiple versions of the data
  - Each reader continues traversing its copy of the data while a new copy might be being created concurrently by each updater *
    - Hence the name read-copy update, or RCU
  - Once all pre-existing RCU readers are done with them, old versions of the data may be discarded

* This backronym expansion provided by Jonathan Walpole
What is RCU?

Publication of And Subscription to New Data

Key:  
- Red: Dangerous for updates: all readers can access
- Yellow: Still dangerous for updates: pre-existing readers can access (next slide)
- Green: Safe for updates: inaccessible to all readers

```
cptr
->a=?
->b=?
->c=?
```

```
cptr
->a=1
->b=2
->c=3
```

```
cptr = tmp
```

```
cptr
->a=1
->b=2
->c=3
```

```
reader
```
What is RCU?

Memory Ordering: Mischief From Compiler and CPU
What is RCU?

Memory Ordering: Mischief From Compiler and CPU

- Original updater code:
  ```
  p = malloc(sizeof(*p));
  p->a = 1;
  p->b = 2;
  p->c = 3;
  cptr = p;
  ```

- Original reader code:
  ```
  p = cptr;
  foo(p->a, p->b, p->c);
  ```

- Mischievous updater code:
  ```
  p = malloc(sizeof(*p));
  cptr = p;
  p->a = 1;
  p->b = 2;
  p->c = 3;
  ```

- Mischievous reader code:
  ```
  retry:
  p = guess(cptr);
  foo(p->a, p->b, p->c);
  if (p != cptr)
    goto retry;
  ```
What is RCU?

Memory Ordering: Mischief From Compiler and CPU

▪ Original updater code:
  
  ```c
  p = malloc(sizeof(*p));
  p->a = 1;
  p->b = 2;
  p->c = 3;
  cptr = p;
  ```

▪ Original reader code:
  
  ```c
  p = cptr;
  foo(p->a, p->b, p->c);
  ```

▪ Mischievous updater code:
  
  ```c
  p = malloc(sizeof(*p));
  cptr = p;
  p->a = 1;
  p->b = 2;
  p->c = 3;
  ```

▪ Mischievous reader code:
  
  ```c
  retry:
  p = guess(cptr);
  foo(p->a, p->b, p->c);
  if (p != cptr)
    goto retry;
  ```

But don't take my word for it on HW value speculation:
http://www.openvms.compaq.com/wizard/wiz_2637.html
What is RCU?

Preventing Memory-Order Mischief

- Updater uses `rcu_assign_pointer()` to publish pointer:
  ```c
  #define rcu_assign_pointer(p, v) 
  ({{ 
    smp_wmb(); /* SMP Write Memory Barrier */ 
    (p) = (v); 
  }})
  ```

- Reader uses `rcu_dereference()` to subscribe to pointer:
  ```c
  #define rcu_dereference(p) 
  ({{ 
    typeof(p) _p1 = (*(volatile typeof(p)*)&(p)); 
    smp_read_barrier_depends(); 
    _p1; 
  }})
  ```

- The Linux-kernel definitions are more ornate: Debugging code
What is RCU?

Preventing Memory-Order Mischief

- “Memory-order-mischief proof” updater code:
  ```c
  p = malloc(sizeof(*p));
  p->a = 1;
  p->b = 2;
  p->c = 3;
  rcu_assign_pointer(cptr, p);
  ```

- “Memory-order-mischief proof” reader code:
  ```c
  p = rcu_dereference(cptr);
  foo(p->a, p->b, p->c);
  ```
What is RCU?

Publication of And Subscription to New Data

Key:  
- Dangerous for updates: all readers can access
- Still dangerous for updates: pre-existing readers can access (next slide)
- Safe for updates: inaccessible to all readers

But if all we do is add, we have a big memory leak!!!
What is RCU?

RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())
What is RCU?

RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())

But how can software deal with two different versions simultaneously???
What is RCU?

Two Different Versions Simultaneously???

I think the poor thing has expired.

No!

Where there is a brain-wave, there is a way!
What is RCU?

RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())

But if readers leave no trace in memory, how can we possibly tell when they are done???
What is RCU?

How Can RCU Tell When Readers Are Done???
What is RCU?

How Can RCU Tell When Readers Are Done???

That is, without re-introducing all of the overhead and latency inherent to other synchronization mechanisms...
What is RCU?

But First, Some RCU Nomenclature

- **RCU read-side critical section**
  - Begins with `rcu_read_lock()`, ends with `rcu_read_unlock()`, and may contain `rcu_dereference()`

- **Quiescent state**
  - Any code that is not in an RCU read-side critical section

- **Extended quiescent state**
  - Quiescent state that persists for a significant time period

- **RCU grace period**
  - Time period when every thread was in at least one quiescent state
What is RCU?

But First, Some RCU Nomenclature

- **RCU read-side critical section**
  - Begins with `rcu_read_lock()`, ends with `rcu_read_unlock()`, and may contain `rcu_dereference()`

- **Quiescent state**
  - Any code that is not in an RCU read-side critical section

- **Extended quiescent state**
  - Quiescent state that persists for a significant time period

- **RCU grace period**
  - Time period when every thread was in at least one quiescent state

- OK, names are nice, but how can you possibly implement this???
What is RCU?

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - RCU readers are not permitted to block
  - Same rule as for tasks holding spinlocks
What is RCU?

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - RCU readers are not permitted to block
  - Same rule as for tasks holding spinlocks

- CPU context switch means all that CPU's readers are done

- *Grace period* ends after all CPUs execute a context switch
What is RCU?

Synchronization Without Changing Machine State???

- But rcu_read_lock() does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections
What is RCU?

Synchronization Without Changing Machine State???

- But `rcu_read_lock()` does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections

- RCU is therefore *synchronization via social engineering*
What is RCU?

Synchronization Without Changing Machine State???

- But rcu_read_lock() does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections

- RCU is therefore *synchronization via social engineering*

- As are all other synchronization mechanisms:
  - “Avoid data races”
  - “Protect specified variables with the corresponding lock”
  - “Access shared variables only within transactions”
What is RCU?

Synchronization Without Changing Machine State???

- But `rcu_read_lock()` does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections

- RCU is therefore *synchronization via social engineering*

- As are all other synchronization mechanisms:
  - “Avoid data races”
  - “Protect specified variables with the corresponding lock”
  - “Access shared variables only within transactions”

- RCU is unusual is being a purely social-engineering approach
  - But RCU implementations for preemptive environments do use lightweight code in addition to social engineering

Userspace RCU: http://liburcu.org
What is RCU?

Toy Implementation of RCU: 20 Lines of Code

- Read-side primitives:
  
  ```c
  #define rcu_read_lock()
  #define rcu_read_unlock()
  #define rcu_dereference(p) (
      typeof(p) _p1 = (*(volatile typeof(p)*)&(p)); \
      smp_read_barrier_depends(); \
      _p1;
  )
  ```

- Update-side primitives
  
  ```c
  #define rcu_assign_pointer(p, v) (
    smp_wmb(); \
    (p) = (v);
  )
  
  void synchronize_rcu(void)
  {
    int cpu;

    for_each_online_cpu(cpu)
      run_on(cpu);
  }
  ```
What is RCU?

Toy Implementation of RCU: 20 Lines of Code, Full Read-Side Performance!!!

- Read-side primitives:
  ```c
  #define rcu_read_lock()
  #define rcu_read_unlock()
  #define rcu_dereference(p) \
  ({ \
    typeof(p) _p1 = (*(volatile typeof(p))*&(p)); \
    smp_read_barrier_depends(); \
    _p1;
  })
  ``

- Update-side primitives
  ```c
  #define rcu_assign_pointer(p, v) \
  ({ \
    smp_wmb(); \
    (p) = (v);
  })
  
  void synchronize_rcu(void)
  {
    int cpu;

    for_each_online_cpu(cpu)
      run_on(cpu);
  }
  ``

Only 9 of which are needed on sequentially consistent systems...
And some people still insist that RCU is complicated... ;-)
RCU Usage: Readers

- Pointer to RCU-protected object guaranteed to exist throughout RCU read-side critical section
  ```c
  rcu_read_lock(); /* Start critical section. */
  p = rcu_dereference(cptr);
  /* *p guaranteed to exist. */
  do_something_with(p);
  rcu_read_unlock(); /* End critical section. */
  /* *p might be freed!!! */
  ```

- The `rcu_read_lock()`, `rcu_dereference()` and `rcu_read_unlock()` primitives are very light weight

- However, updaters must take care...
What is RCU?

RCU Usage: Updaters

- Updaters must wait for an *RCU grace period* to elapse between making something inaccessible to readers and freeing it
  ```c
  spin_lock(&updater_lock);
  q = cptr;
  rcu_assign_pointer(cptr, new_p);
  spin_unlock(&updater_lock);
  synchronize_rcu(); /* Wait for grace period. */
  kfree(q);
  ```
- RCU grace period waits for all pre-exiting readers to complete their RCU read-side critical sections
What is RCU?

Complex Atomic-To-Reader Updates, Take 1
What is RCU?

RCU Replacement Of Item In Linked List

1 Version
boa

1 Version
boa

1 Version
boa

1 Version
boa

2 Versions
boa
cat'
synchronize_rcu()

1 Version
boa

cat

? kmalloc()

1 Version
boa

cat

update

1 Version
boa

cat'

1 Version
boa

cat

kfree()

1 Version
boa

cat'

2 Versions
boa
cat'

1 Version
boa

cat

gnu

gnu

gnu

gnu

gnu

gnu

gnu

gnu

gnu

gnu

gnu

gnu

gnu

gnu

gnu

gnu

Readers?

Readers?

Readers?

Readers?

Readers?

Readers?

Readers?
What is RCU?

RCU Grace Periods: Conceptual and Graphical Views
What is RCU?

RCU Grace Periods: A Conceptual View

- **RCU read-side critical section**
  - Begins with rcu_read_lock(), ends with rcu_read_unlock(), and may contain rcu_dereference()

- **Quiescent state**
  - Any code that is not in an RCU read-side critical section

- **Extended quiescent state**
  - Quiescent state that persists for a significant time period

- **RCU grace period**
  - Time period when every thread is in at least one quiescent state
  - Ends when all pre-existing readers complete
  - Guaranteed to complete in finite time iff all RCU read-side critical sections are of finite duration

- But what happens if you try to extend an RCU read-side critical section across a grace period?
What is RCU?

RCU Grace Periods: A Graphical View

So what happens if you try to extend an RCU read-side critical section across a grace period?
What is RCU?

RCU Grace Period: A Self-Repairing Graphical View

A grace period is not permitted to end until all pre-existing readers have completed.

Grace period extends as needed.

synchronize_rcu()
What is RCU?

RCU Grace Period: A Lazy Graphical View

But it is OK for RCU to be lazy and allow a grace period to extend longer than necessary
What is RCU?

RCU Grace Period: A Really Lazy Graphical View

And it is also OK for RCU to be even more lazy and start a grace period later than necessary. But why is this useful?
Starting a grace period late can allow it to serve multiple updates, decreasing the per-update RCU overhead. But...
The Costs and Benefits of Laziness

- Starting the grace period later increases the number of updates per grace period, reducing the per-update overhead
- Delaying the end of the grace period increases grace-period latency
- Increasing the number of updates per grace period increases the memory usage
  - Therefore, starting grace periods late is a good tradeoff if memory is cheap and communication is expense, as is the case in modern multicore systems
    - And if real-time threads avoid waiting for grace periods to complete
  - However...
RCU Grace Period: A Too-Lazy Graphical View

And it is OK for the system to complain (or even abort) if a grace period extends too long. Too-long grace periods are likely to result in death by memory exhaustion anyway.
What is RCU?

RCU Asynchronous Grace-Period Detection
What is RCU?

**RCU Asynchronous Grace-Period Detection**

- The call_rcu() function registers an RCU callback, which is invoked after a subsequent grace period elapses.

**API:**

```c
void (*func)(struct rcu_head *rcu));
```

- The `rcu_head` structure:

```c
define struct rcu_head {
    struct rcu_head *next;
    void (*func)(struct rcu_head *rcu);
};
```

- The `rcu_head` structure is normally embedded within the RCU-protected data structure.
What is RCU?

RCU Grace Period: An Asynchronous Graphical View

Reader
Reader
Reader
Reader
Reader
Reader
Reader
Reader

Change
Grace Period
Change Visible to All Readers

call_rcu(&p->rcu, func);
func(&p->rcu);
What is RCU?

Performance
What is RCU?

**Theoretical Performance**

RCU (wait-free)

- Uncontended
  - 71.2 cycles
  - 1 cycle
  - Full performance, linear scaling, real-time response

- Contended, No Spinning
  - 71.2 cycles
  - 1 cycle
  - 71.2 cycles
  - 73 CPUs to break even with a single CPU!
  - 144 CPUs to break even with a single CPU!!!
What is RCU?

Measured Performance
What is RCU?

Schrödinger's Zoo: Read-Only

RCU and hazard pointers scale quite well!!!
What is RCU?

Schrödinger's Zoo: Read-Only Cat-Heavy Workload

RCU handles locality, hazard pointers not bad, bucket locking horrible!
What is RCU?

Schrödinger's Zoo: Reads and Updates

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Reads</th>
<th>Failed Reads</th>
<th>Cat Reads</th>
<th>Adds</th>
<th>Deletes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Locking</td>
<td>799</td>
<td>80</td>
<td>639</td>
<td>77</td>
<td>77</td>
</tr>
<tr>
<td>Per-Bucket Locking</td>
<td>13,555</td>
<td>6,177</td>
<td>1,197</td>
<td>5,370</td>
<td>5,370</td>
</tr>
<tr>
<td>Hazard Pointers</td>
<td>41,011</td>
<td>6,982</td>
<td>27,059</td>
<td>4,860</td>
<td>4,860</td>
</tr>
<tr>
<td>RCU</td>
<td>85,906</td>
<td>13,022</td>
<td>59,873</td>
<td>2,440</td>
<td>2,440</td>
</tr>
</tbody>
</table>
What is RCU?

Real-Time Response to Changes
What is RCU?

RCU vs. Reader-Writer-Lock Real-Time Latency

- RWLock reader
- Spin
- RWLock reader
- RWLock reader
- Spin
- RWLock reader
- RWLock reader
- Spin
- RWLock reader
- RWLock reader
- Spin
- RWLock writer
- RWLock reader
- RWLock reader
- RWLock reader
- RWLock reader
- Spin
- RWLock reader
- RWLock reader
- Spin
- RWLock reader
- RWLock reader
- Spin
- RCU reader
- RCU reader
- RCU reader
- RCU reader
- RCU reader
- RCU reader
- RCU reader
- RCU reader
- RCU reader
- RCU reader
- RCU reader
- RCU updater

External Event

RCU Latency

RWLock Latency
What is RCU?

RCU Performance: “Free is a Very Good Price!!!”
What is RCU?

RCU Performance: “Free is a Very Good Price!!!”
And Nothing Is Faster Than Doing Nothing!!!
What is RCU?

RCU Area of Applicability

Need fully fresh and consistent data

Stale and inconsistent data OK

1. RCU provides ABA protection for update-friendly mechanisms
2. RCU provides bounded wait-free read-side primitives for real-time use
## What is RCU?

<table>
<thead>
<tr>
<th></th>
<th>Reference Counting</th>
<th>Hazard Pointers</th>
<th>Sequence Locks</th>
<th>RCU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Existence Guarantees</strong></td>
<td>Complex</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Updates and Readers</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Progress Concurrently</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Contention Among Readers</strong></td>
<td>High</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td><strong>Reader Per-Critical-Section Overhead</strong></td>
<td>N/A</td>
<td>N/A</td>
<td>Two (smp_mb())</td>
<td>Ranges from none to two (smp_mb())</td>
</tr>
<tr>
<td><strong>Reader Per-Object Traversal Overhead</strong></td>
<td>Read-modify-write atomic operations, memory-barrier instructions, and cache misses</td>
<td>(smp_mb())</td>
<td>None, but unsafe</td>
<td>None (volatile accesses)</td>
</tr>
<tr>
<td><strong>Reader Forward Progress Guarantee</strong></td>
<td>Lock free</td>
<td>Lock free</td>
<td>Blocking</td>
<td>Bounded wait free</td>
</tr>
<tr>
<td><strong>Reader Reference Acquisition</strong></td>
<td>Can fail (conditional)</td>
<td>Can fail (conditional)</td>
<td>Unsafe</td>
<td>Cannot fail (unconditional)</td>
</tr>
<tr>
<td><strong>Memory Footprint</strong></td>
<td>Bounded</td>
<td>Bounded</td>
<td>Bounded</td>
<td>Unbounded</td>
</tr>
<tr>
<td><strong>Reclamation Forward Progress</strong></td>
<td>Lock free</td>
<td>Lock free</td>
<td>N/A</td>
<td>Blocking</td>
</tr>
<tr>
<td><strong>Automatic Reclamation</strong></td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td><strong>Lines of Code</strong></td>
<td>94</td>
<td>79</td>
<td>79</td>
<td>73</td>
</tr>
</tbody>
</table>

**Table 9.5:** Which Deferred Technique to Choose?
What is RCU?

Existence Guarantees

- Purpose: Avoid data being yanked from under reader
- Reference counting (also non-blocking synchronization)
  - Possible, but complex and error-prone
- Hazard pointers: Yes
- Sequence locks: No
  - You just get told later that something might have been yanked
- RCU: Yes
What is RCU?

Reader/Writer Concurrent Forward Progress

- Purpose: Avoid starvation independent of workload
- Reference counting: Yes
- Hazard pointers: Yes
- Sequence locks: No, updates roll back readers
- RCU: Yes
What is RCU?

Avoid Read-Side Contention

- Purpose: Scalability, performance, forward progress
- Reference counting: No, high memory contention
- Hazard pointers: Yes
- Sequence locking: Yes
- RCU: Yes
What is RCU?

Degree of Read-Side Critical-Section Overhead

- **Purpose:** Low overhead means faster execution
- **Reference counting:** None (no critical sections)
- **Hazard pointers:** None (no critical sections)
- **Sequence locks:** Two full memory barriers
- **RCU:**
  - Ranges from none (QSBR) to two full memory barriers (SRCU)
What is RCU?

Read-Side Per-Object Traversal Overhead

- Purpose: Low overhead for faster execution
- Reference counting: RMW atomic operations, memory-barrier instructions, and cache misses
- Hazard pointers: smp_mb(), but can eliminate with operating-system membarrier support
- Sequence locking: Kernel panic!!!
- RCU: None (except on DEC Alpha)
What is RCU?

Read-Side Forward Progress Guarantee

- Purpose: Meet response-time commitments
- Reference counting: Lock free
- Hazard pointers: Lock free
- Sequence locks: Blocking (can wait on updater)
- RCU: Population-oblivious bounded wait-free
What is RCU?

Read-Side Reference Acquisition

- Purpose: Must client code retry read-side traversals?
- Reference counting: Traversals can fail, requiring retry
- Hazard pointers: Traversals can fail, requiring retry
- Sequence locking: Kernel can panic
- RCU: Traversals guaranteed to succeed, no retry needed
What is RCU?

Memory Footprint

- Purpose: Small memory footprints are good!
  - Especially if you are as old as I am!!!
- Reference counting: Bounded (number of active references)
- Hazard pointers: Bounded (number of active references, though tight bound incurs CPU overhead)
- Sequence locks: Bounded (especially given unsafe traversal)
- RCU: Unbounded or updaters delayed
Reclamation Forward Progress

- Purpose: Tight memory footprint independent of workload
- Reference counting: Lock free
- Hazard pointers: Lock free
- Sequence locking: N/A
- RCU: Blocking: Single reader can block reclamation
Automatic Reclamation

- Purpose: Simplify memory management
- Reference counting: Yes
- Hazard pointers: No, but working on it
- Sequence locking: N/A
- RCU: No, but working on it
What is RCU?

Lines of Code for Pre-BSD Routing Table

- Reference counting: 94 (but buggy)
- Hazard pointers: 79
- Sequence locks: 79 (but buggy)
- RCU: 73
What is RCU?

Different Design Points!

- Locking is still the workhorse for production software
- Non-blocking synchronization where it works well
- Reference counting OK on small systems or for rarely accessed portions of larger systems, and provide tight bounds on memory. Traversals subject to retry.
- Hazard pointers handle large systems, provide tight bounds on memory, excellent scalability, and decent traversal performance. Traversals subject to retry.
- Sequence locks need one of the other approaches
- RCU handles huge systems, excellent scalability and traversal overhead, no-retry traversals. Unbounded memory footprint.
What is RCU?

RCU Applicability to the Linux Kernel
What is RCU?

Complex Atomic-To-Reader Updates, Take 2
What is RCU?

Complex Atomic-To-Reader Updates, Take 2
Atomic Multi-Structure Update: Issaquah Challenge
What is RCU?

Atomic Multi-Structure Update: Issaquah Challenge

Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree
What is RCU?

Atomic Multi-Structure Update: Issaquah Challenge

Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree
Without contention between the two move operations!
What is RCU?

Atomic Multi-Structure Update: Issaquah Challenge

Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree
Without contention between the two move operations!
Hence, most locking solutions “need not apply”
What is RCU?

Recall Applicable Laws of Physics...

- The finite speed of light
- The atomic nature of matter

- We therefore avoid *unnecessary* updates!!!
What is RCU?

Update-Heavy Workloads Painful for Parallelism!!!
But There Are Some Special Cases...
What is RCU?

But There Are Some Special Cases

- Per-CPU/thread processing (perfect partitioning)
  - Huge number of examples, including the per-thread/CPU stack
  - We will look at split counters

- Read-only traversal to location being updated
  - Key to solving the Issaquah Challenge

- Trivial Lock-Based Concurrent Deque???
What is RCU?

Split Counters
What is RCU?

Split Counters Diagram

Counter 0
Counter 1
Counter 2
Counter 3
Counter 4
Counter 5

Increment only your own counter
What is RCU?

Split Counters Diagram

Counter 0
Counter 1
Counter 2
Counter 3
Counter 4
Counter 5

Sum all counters
While they continue changing
Split Counters Lesson

- Updates need not slow us down – if we maintain good locality
- For the split counters example, in the common case, each thread only updates its own counter
  - Reads of all counters should be rare
  - If they are not rare, use some other counting algorithm
  - There are a lot of them, see “Counting” chapter of “Is Parallel Programming Hard, And, If So, What Can You Do About It?” (http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html)
What is RCU?

Trivial Lock-Based Concurrent Dequeue
What is RCU?

Trivial Lock-Based Concurrent Dequeue

- Use two lock-based dequeues
  - Can always insert concurrently: grab dequeue's lock
  - Can always remove concurrently unless one or both are empty
    - If yours is empty, grab both locks in order!

![Diagram of Trivial Lock-Based Concurrent Dequeue]
Trivial Lock-Based Concurrent Dequeue

- Use two lock-based dequeues
  - Can always insert concurrently: grab dequeue's lock
  - Can always remove concurrently unless one or both are empty
    - If yours is empty, grab both locks in order!

- But why push all your data through one dequeue???
What is RCU?

Trivial Lock-Based Concurrent Dequeue Performance

- Dalessandro et al., “Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory”, ASPLOS'11, March 5-11, Newport Beach, California, USA
  - See "Deque benchmark" subsection of section 4.2 on page 6, especially Figure 7a (next slide)
  - Lock-based dequeue beats all STM algorithms
What is RCU?

Dalessandro et al. Figure 7a:

- Normalized Throughput
- Threads

- McKenney

- 7.1 ops/usec

- 42% Standard Deviation
What is RCU?

Trivial Lock-Based Concurrent Dequeue Performance

  - See Figure 1 and discussion in Section 3 on page 2
  - Lock-based dequeue beats all HTM algorithms at some point

- Both sets of authors were exceedingly gracious, without the need for a Code of Conflict
What is RCU?

Dice et al., Figure 1
What is RCU?

**Read-Only Traversal To Location Being Updated**
What is RCU?

Why Read-Only Traversal To Update Location?

- Lock root
- Lock child, unlock root
- Lock child, unlock parent
- Lock child, unlock parent
- Lock child, retain parent's lock

Lock contention despite read-only accesses!
What is RCU?

And This Is Another Reason Why We Have RCU!

- (You can also use garbage collectors, hazard pointers, reference counters, etc.)
- Design principle: Avoid expensive operations in read-side code
- As noted earlier, lightest-weight conceivable read-side primitives
  /* Assume non-preemptible (run-to-block) environment. */
  #define rcu_read_lock()
  #define rcu_read_unlock()
What is RCU?

Better Read-Only Traversal To Update Location
What is RCU?

Deletion-Flagged Read-Only Traversal

RCU

Lockless RCU-protected traversal

Acquire locks, recheck state, retry if concurrent update

Marked deleted
What is RCU?

**Read-Only Traversal To Location Being Updated**

- Focus contention on portion of structure being updated
  - And preserve locality of reference to different parts of structure

- Of course, full partitioning is better!

- Read-only traversal technique citations:
  - Arbel & Attiya, “Concurrent Updates with RCU: Search Tree as an Example”, PODC’14 (very similar lookup, insert, and delete)
  - McKenney, Sarma, & Soni, “Scaling dcache with RCU”, Linux Journal, January 2004
  - And maybe also: Kung & Lehman, “Concurrent Manipulation of Binary Search Trees”, ACM TODS, September, 1980
What is RCU?

Issaquah Challenge: One Solution
What is RCU?

Locking Regions for Binary Search Tree

In many cases, can implement existence as simple wrapper!
What is RCU?

Possible Upsets While Acquiring Locks...

Before

After

What to do?
Drop locks and retry!!!
What is RCU?

Existence Structures
What is RCU?

Existence Structures

- Solving yet another computer-science problem by adding an additional level of indirection...
What is RCU?

Example Existence Structure Before Switch

Data Structure A
- Existence
- Offset=0

Data Structure B
- Existence
- Offset=1

Existence Switch

1
0
0
1
What is RCU?

Example Existence Structure After Switch

- Data Structure A
  - Existence
  - Offset=0

- Data Structure B
  - Existence
  - Offset=1

- Existence Switch
  - 1
  - 0
  - 0
  - 1
What is RCU?

But Levels of Indirection Are Expensive!

- And I didn't just add one level of indirection, I added three!
- But most of the time, elements exist and are not being moved
- So represent this common case with a NULL pointer
  - If the existence pointer is NULL, element exists: No indirection needed
  - Backwards of the usual use of a NULL pointer, but so it goes!
- In the uncommon case, traverse existence structure as shown on the preceding slides
  - Expensive, multiple cache misses, but that is OK in the uncommon case
- There is no free lunch:
  - With this optimization, loads need smp_load_acquire() rather than READ_ONCE(), ACCESS_ONCE(), or rcu_dereference()
- Can use low-order pointer bits to remove two levels of indirection
  - Kudos to Dmitry Vyukov for this trick
What is RCU?

Example Existence Structure: Dmitry's Approach

- Data Structure A
  - Existence | 0

- Data Structure B
  - Existence | 1

Existence Switch 0/1
What is RCU?

Example Existence Structure: Dmitry's Approach

- Data Structure A
  - Existence | 0

- Data Structure B
  - Existence | 1

- Existence Switch 0

0 1
1 0
What is RCU?

Example Existence Structure: Dmitry's Approach

Data Structure A
Existence | 0

Data Structure B
Existence | 1

Existence Switch 1

1 0

© 2017 IBM Corporation
What is RCU?

Abbreviated Existence Switch Operation (1/6)

Initial state: First tree contains 1,2,3, second tree contains 2,3,4. All existence pointers are NULL.
What is RCU?

Abbreviated Existence Switch Operation (2/6)

First tree contains 1,2,3, second tree contains 2,3,4.
After insertion, same: First tree contains 1,2,3, second tree contains 2,3,4.
After existence switch: First tree contains 2,3,4, second tree contains 1,2,3. Transition is single store, thus atomic! (But lookups need barriers in this case.)
What is RCU?

Abbreviated Existence Switch Operation (5/6)

Unlink old nodes and allegiance structure
After waiting a grace period, can free up existence structures and old nodes
And data structure preserves locality of reference!
What is RCU?

Existence Structures

- Existence-structure reprise:
  - Each data element has an existence pointer
  - NULL pointer says “member of current structure”
  - Non-NULL pointer references an existence structure
    - Existence of multiple data elements can be switched atomically

- But this needs a good API to have a chance of getting it right!
  - Especially given that a NULL pointer means that the element exists!!!
What is RCU?

Existence Data Structures

```c
struct existence_group {
    uintptr_t eg_state;
    struct cds_list_head eg_outgoing;
    struct cds_list_head eg_incoming;
    struct rcu_head eg_rh;
};

struct existence_head {
    uintptr_t eh_egi;
    struct cds_list_head eh_list;
    int (*eh_add)(struct existence_head *ehp);
    void (*eh_remove)(struct existence_head *ehp);
    void (*eh_free)(struct existence_head *ehp);
    int eh_gone;
    spinlock_t eh_lock;
    struct rcu_head eh_rh;
};
```
What is RCU?

Existence APIs

- `void existence_init(struct existence_group *egp);`
- `uintptr_t existence_group_outgoing(struct existence_group *egp);`
- `uintptr_t existence_group_incoming(struct existence_group *egp);`
- `void existence_set(struct existence **epp, struct existence *ep);`
- `void existence_clear(struct existence **epp);`
- `int existence_exists(struct existence_head *ehp);`
- `int existence_exists_relaxed(struct existence_head *ehp);`
- `int existence_head_init_incoming(struct existence_head *ehp, struct existence_group *egp, int (*eh_add)(struct existence_head *ehp), void (*eh_remove)(struct existence_head *ehp), void (*eh_free)(struct existence_head *ehp))`
- `int existence_head_set_outgoing(struct existence_head *ehp, struct existence_group *egp)`
- `void existence_flip(struct existence_group *egp);`
- `void existence_backout(struct existence_group *egp);`
What is RCU?

Existence Data Structures: Multiple Membership

User data element atomically moving from data structure 1 to 2, which can be different types of data structures.
What is RCU?

Pseudo-Code for Atomic Move

- Allocate and initialize existence_group structure (existence_group_init())

- Add outgoing existence structure to item in source tree (existence_head_set_outgoing())
  - If operation fails, existence_backout() and report error to caller
  - Or maybe retry later

- Insert new element (with source item's data pointer) to destination tree (existence_head_init_incoming())
  - If operation fails, existence_backout() and error to caller
  - Or maybe retry later

- Invoke existence_flip() to flip incoming and outgoing
  - And existence_flip() automatically cleans up after the operation
  - Just as existence_backout() does after a failed operation
What is RCU?

Existence Structures: Performance and Scalability

100% lookups
Super-linear as expected based on range partitioning
(Hash tables about 3x faster)
What is RCU?

Existence Structures: Performance and Scalability

90% lookups, 3% insertions, 3% deletions, 3% full tree scans, 1% moves
(Workload approximates Gramoli et al. CACM Jan. 2014)
What is RCU?

Existence Structures: Performance and Scalability

100% moves (worst case)
What is RCU?

Existence Structures: Performance and Scalability

100% moves: Still room for improvement!
What is RCU?

But Requires Modifications to Existing Algorithms
What is RCU?

But Requires Modifications to Existing Algorithms
New Goal: Use RCU Algorithms Unchanged!!!
What is RCU?

Rotate 3 Elements Through 3 Hash Tables (1/4)
What is RCU?

Rotate 3 Elements Through 3 Hash Tables (2/4)
What is RCU?

Rotate 3 Elements Through 3 Hash Tables (3/4)

Existence Structure 1
What is RCU?

Rotate 3 Elements Through 3 Hash Tables (4/4)

- HT 1
  - permanent
  - EL 1
- HT 2
  - permanent
  - EL 2
- HT 3
  - permanent
  - EL 3
What is RCU?

Data to Rotate 3 Elements Through 3 Hash Tables

```c
struct keyvalue {
    unsigned long key;
    unsigned long value;
    atomic_t refcnt;
};

struct hash_exists {
    struct ht_elem he_hte;
    struct hashtab *he_htp;
    struct existence_head he_eh;
    struct keyvalue *he_kv;
};
```
What is RCU?

**Code to Rotate 3 Elements Through 3 Hash Tables**

```c
egp = malloc(sizeof(*egp));
BUG_ON(!egp);
existence_group_init(egp);
rcu_read_lock();
heo[0] = hash_exists_alloc(egp, htp[0], hei[2]->he_kv, ~0, ~0);
heo[1] = hash_exists_alloc(egp, htp[1], hei[0]->he_kv, ~0, ~0);
heo[2] = hash_exists_alloc(egp, htp[2], hei[1]->he_kv, ~0, ~0);
BUG_ON(existence_head_set_outgoing(&hei[0]->he_eh, egp));
BUG_ON(existence_head_set_outgoing(&hei[1]->he_eh, egp));
BUG_ON(existence_head_set_outgoing(&hei[2]->he_eh, egp));
rcu_read_unlock();
existence_flip(egp);
call_rcu(&egp->eg_rh, existence_group_rcu_cb);
```

BUG_ON()s become checks with calls to existence_backout() if contention possible
What is RCU?

**Code to Rotate 3 Elements Through 3 Hash Tables**

```c
egp = malloc(sizeof(*egp));
BUG_ON(!egp);
existence_group_init(egp);
rcu_read_lock();
heo[0] = hash_exists_alloc(egp, htp[0], hei[2]->he_kv, ~0, ~0);
heo[1] = hash_exists_alloc(egp, htp[1], hei[0]->he_kv, ~0, ~0);
heo[2] = hash_exists_alloc(egp, htp[2], hei[1]->he_kv, ~0, ~0);
BUG_ON(existence_head_set_outgoing(&hei[0]->he_eh, egp));
BUG_ON(existence_head_set_outgoing(&hei[1]->he_eh, egp));
BUG_ON(existence_head_set_outgoing(&hei[2]->he_eh, egp));
rcu_read_unlock();
existence_flip(egp);
call_rcu(&egp->eg_rh, existence_group_rcu_cb);
```

*BUG_ON()*s become checks with calls to *existence_backout()* if contention possible

Works with an RCU-protected hash table that knows nothing of atomic move!!!
What is RCU?

Performance and Scalability of New-Age Existence Structures?
What is RCU?

Performance and Scalability of New-Age Existence Structures?

- For readers, as good as ever
- For update-only triple-hash rotations, not so good!
What is RCU?

Triple-Hash Rotations are Pure Updates: Red Zone!

- Stale and inconsistent data OK
- Read-Mostly, Stale & Inconsistent Data OK (RCU Works Great!!)
- Read-Mostly, Need Consistent Data (RCU Works OK)
- Read-Write, Need Consistent Data (RCU Might Be OK)
- Update-Mostly, Need Fresh Consistent Data (RCU Not So Good)

Need fully fresh and consistent data

Opportunity to improve the infrastructure!
What is RCU?

Existence Structures: Towards Update Scalability

- “Providing perfect performance and scalability is like committing the perfect crime. There are 50 things that might go wrong, and if you are a genius, you might be able to foresee and forestall 25 of them.” — Paraphrased from Body Heat, w/apologies to Kathleen Turner fans

- Issues thus far:
  - Data structure alignment (false sharing) – easy fix
  - User-space RCU configuration (need per-thread call_rcu() handling, also easy fix)
  - The “perf” tool shows massive futex contention, checking locking design finds nothing
    - And replacing all lock acquisitions with “if (!trylock()) abort” never aborts
    - Other “perf” entries shift suspicion to memory allocators
  - Non-scalable memory allocators: More complex operations means more allocations!!!
    - The glibc allocator need not apply for this job
    - The jemalloc allocator bloats the per-thread lists, resulting in ever-growing RSS
    - The tcmalloc allocator suffers from lock contention moving to/from global pool
    - A tcmalloc that is better able to handle producer-consumer relations in the works, but I first heard of this a few years back and it still has not made its appearance
What is RCU?

Existence Structures: Towards Update Scalability

- "Providing perfect performance and scalability is like committing the perfect crime. There are 50 things that might go wrong, and if you are a genius, you might be able to foresee and forestall 25 of them." – Paraphrased from Body Heat, w/apologies to Kathleen Turner fans

- Issues thus far:
  - Data structure alignment (false sharing) – easy fix
  - User-space RCU configuration (need per-thread call_rcu() handling, also easy fix)
  - The "perf" tool shows massive futex contention, checking locking design finds nothing
    - And replacing all lock acquisitions with "if (!trylock()) abort" never aborts
    - Other "perf" entries shift suspicion to memory allocators
  - Non-scalable memory allocators: More complex operations means more allocations!!!
    - The glibc allocator need not apply for this job
    - The jemalloc allocator bloats the per-thread lists, resulting in ever-growing RSS
    - The tcmalloc allocator suffers from lock contention moving to/from global pool
    - A tcmalloc that is better able to handle producer-consumer relations in the works, but I first heard of this a few years back and it still has not made its appearance

- Fortunately, I have long experience with memory allocators
  - McKenney & Slingwine, "Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors", 1993 USENIX
  - But needed to complete implementation in one day, so chose quick hack
What is RCU?

Specialized Producer/Consumer Allocator

- RCU Callbacks
- Lockless Memory Queue
- Lockless Memory Queue
- Lockless Memory Queue
- Worker Threads
What is RCU?

New Age Existence Structures: Towards Scalability

- “Providing perfect performance and scalability is like committing the perfect crime. There are 50 things that might go wrong, and if you are a genius, you might be able to foresee and forestall 25 of them.” – Paraphrased from Body Heat, with apologies to Kathleen Turner fans

- Issues thus far:
  - Data structure alignment (false sharing) – easy fix
  - User-space RCU configuration (need per-thread call_rcu() handling, also easy fix)
  - The “perf” tool shows massive futex contention, checking locking design finds nothing
    - And replacing all lock acquisitions with “if (!trylock()) abort” never aborts
    - Other “perf” entries shift suspicion to memory allocators
  - Non-scalable memory allocators: More complex operations means more allocations!!!
    - Lockless memory queue greatly reduces memory-allocator lock contention
  - Profiling shows increased memory footprint is an issue: caches and TLBs!
  - Userspace RCU callback handling appears to be the next bottleneck
    - Perhaps some of techniques from the Linux kernel are needed in userspace
What is RCU?

Performance and Scalability of New-Age Existence Structures for Triple Hash Rotation?

Some improvement, but still not spectacular
But note that each thread is rotating concurrently
But What About Skiplists?
What is RCU?

Rotate 3 Elements Through 3 Skiplists (1/4)
What is RCU?

Rotate 3 Elements Through 3 Skiplists (2/4)
What is RCU?

Rotate 3 Elements Through 3 Skiplists (3/4)

Existence Structure 1
What is RCU?

Rotate 3 Elements Through 3 Skiplists (4/4)

SL 1
- permanent
  - EL 1

SL 2
- permanent
  - EL 2

SL 3
- permanent
  - EL 3
What is RCU?

Data to Rotate 3 Elements Through 3 Skiplists

```c
struct keyvalue {
    unsigned long key;
    unsigned long value;
    atomic_t refcnt;
};

struct hash_exists {
    struct skiplist se_sle;
    struct skiplist *se_slh;
    struct existence_head se_eh;
    struct keyvalue *se_kv;
};
```
What is RCU?

**Code to Rotate 3 Elements Through 3 Skiplists**

```c
egp = malloc(sizeof(*egp));
BUG_ON(!egp);
existence_group_init(egp);
rcu_read_lock();
seo[0] = skiplist_exists_alloc(egp, &slp[0], sei[2]->se_kv, ~0, ~0);
seo[1] = skiplist_exists_alloc(egp, &slp[1], sei[0]->se_kv, ~0, ~0);
seo[2] = skiplist_exists_alloc(egp, &slp[2], sei[1]->se_kv, ~0, ~0);
BUG_ON(existence_head_set_outgoing(&sei[0]->se_eh, egp));
BUG_ON(existence_head_set_outgoing(&sei[1]->se_eh, egp));
BUG_ON(existence_head_set_outgoing(&sei[2]->se_eh, egp));
rcu_read_unlock();
existence_flip(egp);
call_rcu(&egp->eg_rh, existence_group_rcu_cb);
```

As with hash table: RCU-protected skiplist that knows nothing of atomic move
What is RCU?

Performance and Scalability of New-Age Existence Structures for Triple Skiplist Rotation?

This skiplist is a random tree, so we have lock contention.
But Can We Atomically Rotate More Elements?

- Apply batching optimization!
- Instead of rotating three elements through three hash tables, rotate three pairs of elements
- Then three triplets of elements
- And so on, rotating ever larger sets through the three tables
What is RCU?

But Can We Atomically Rotate More Elements?

- Apply batching optimization!
- Instead of rotating three elements through three hash tables, rotate three pairs of elements
- Then three triplets of elements
- And so on, rotating ever larger sets through the three tables
- It can be done, but there is a performance mystery
What is RCU?

Large-Hash-Rotation Performance Mystery

Many additional optimizations are possible, but...
What is RCU?

Even Bigger Mystery: Why Rotate This Way???
Even Bigger Mystery: Why Rotate This Way???

- Every third rotation brings us back to the original state
- So why bother with allocation, freeing, and grace periods?
Even Bigger Mystery: Why Rotate This Way???

- Every third rotation brings us back to the original state
- So why bother with allocation, freeing, and grace periods?
- Just change the existence state variable!!!
  - But we need not be limited to two states
  - Define *kaleidoscopic data structure* as one updated by state change
  - Data structures and algorithms are very similar to those for existence
What is RCU?

Rotate Through Hash Table & Skiplist (1/3)

Hash Table

permanent

EL 1

Skiplist

permanent

EL 2
What is RCU?

Rotate Through Hash Table & Skip List (2/3)

Kaleidoscope Structure 0

Hash Table

State 0

State 1

EL 1

Skip List

State 0

State 1

EL 2
What is RCU?

Rotate Through Hash Table & Skiplist (3/3)

Kaleidoscope Structure 1

Hash Table
State 0
State 1
EL 1

Skiplist
State 0
State 1
EL 2
What is RCU?

Rotate Through Hash Table & Skiplist (2/3)

Kaleidoscope Structure 0
What is RCU?

Rotate Through Hash Table & Skiplist (3/3)

Hash Table

State 0

State 1

EL 1

Skiplist

State 0

State 1

EL 2

Kaleidoscope Structure 1
What is RCU?

Very Tight Loop...

```c
while (ACCESS_ONCE(goflag) == GOFLAG_RUN) {
    kaleidoscope_set_state(kgp, nrotations % 2);
    nrotations++;
}
```
What is RCU?

Kaleidoscopic Rotation Performance Results

This is more like it!!! Too bad about the specificity...
What is RCU?

Kaleidoscopic Rotation Performance Results

This is more like it!!! Too bad about the specificity...
As always, be wary of benchmarks!!!
What is RCU?

Existence Advantages and Disadvantages

- Existence requires focused developer effort
- Existence specialized to linked structures (for now, anyway)
- Existence requires explicit memory management
- Existence-based exchange operations require linked structures that accommodate duplicate elements
  - Current prototypes disallow duplicates, explicit check for hash tables
- Existence permits irrevocable operations
- Existence can exploit locking hierarchies, reducing the need for contention management
- Existence achieves semi-decent performance and scalability
- Flip/backout automation significantly eases memory management
- Existence's use of synchronization primitives preserves locality of reference
- Existence is compatible with old hardware
- Existence is a downright mean memory-allocator and RCU test case!!!
What is RCU?

When Might You Use Existence-Based Update?

- We really don't know yet
  - But similar techniques are used by Linux-kernel filesystems

- Best guess is when one or more of the following holds and you are willing to invest significant developer effort to gain performance and scalability:
  - Many small updates to large linked data structure
  - Complex updates that cannot be efficiently implemented with single pointer update
  - Read-mostly to amortize higher overhead of complex updates
  - Need compatibility with hardware not supporting transactional memory
    - Side benefit: Dispense with the need for software fallbacks!
  - Need to be able to do irrevocable operations (e.g., I/O) as part of data-structure update
What is RCU?

Existence Structures: Production Readiness
What is RCU?

Existence Structures: Production Readiness

- No, it is **not** production ready (but was getting there)
What is RCU?

Existence Structures: Production Readiness

- No, it is *not* production ready (but was getting there)

<table>
<thead>
<tr>
<th>Production: 1T Instances</th>
<th>Need this for Internet of Things, Validation is a <strong>big</strong> unsolved problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production: 1G Instances</td>
<td>Formal verification for RCU!!!</td>
</tr>
<tr>
<td>Production: 1M Instances</td>
<td></td>
</tr>
<tr>
<td>Production: 1K Instances</td>
<td></td>
</tr>
<tr>
<td>R&amp;D Prototype</td>
<td></td>
</tr>
<tr>
<td>Benchmark Special</td>
<td></td>
</tr>
<tr>
<td>Limping</td>
<td></td>
</tr>
<tr>
<td>Builds</td>
<td></td>
</tr>
</tbody>
</table>

RCU

© 2017 IBM Corporation
What is RCU?

Existence Structures: Known Antecedents

- Fraser: “Practical Lock-Freedom”, Feb 2004
  - Insistence on lock freedom: High complexity, poor performance
  - Similarity between Fraser's OSTM commit and existence switch

  - Block concurrent operations while large update is carried out

- Triplett: “Scalable concurrent hash tables via relativistic programming”, Sept 2009

  - Similarity between Triplett's key switch and allegiance switch
  - Could share nodes between trees like Triplett does between hash chains, but would impose restrictions and API complexity

- Some filesystem algorithms in Linux kernel
What is RCU?

Summary

- Complex atomic updates can be applied to unmodified RCU-aware concurrent data structures
  - Need functions to add, remove, and free elements
  - Free to use any synchronization mechanism
  - Free to use any memory allocator

- Flip/backout processing can be automated

- High update rates encounter interesting bottlenecks in the infrastructure: Memory allocation and userspace RCU
  - Read-mostly workloads continue to perform and scale well
  - As do kaleidoscopic updates

- Lots of opportunity for collaboration and innovation!
What is RCU?

Graphical Summary

Not only are they lazy, they get more work done than I do!

"yawn"
What is RCU?

To Probe Deeper (1/4)

- Hash tables:
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Chapter 10

- Split counters:
  - http://events.linuxfoundation.org/sites/events/files/slides/BareMetal.2014.03.09a.pdf

- Perfect partitioning
  - Candide et al: “Dynamo: Amazon's highly available key-value store”
    • http://doi.acm.org/10.1145/1323293.1294281
    • http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Section 6.5
  - McKenney: “Retrofitted Parallelism Considered Grossly Suboptimal”
    • Embarrassing parallelism vs. humiliating parallelism
    • https://www.usenix.org/conference/hotpar12/retro%EF%AC%81tted-parallelism-considered-grossly-sub-optimal
  - McKenney et al: “Experience With an Efficient Parallel Kernel Memory Allocator”
  - Bonwick et al: “Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources”
    • http://static.usenix.org/event/usenix01/full_papers/bonwick/bonwick_html/
  - Turner et al: “PerCPU Atomics”
What is RCU?

To Probe Deeper (2/4)

- **Stream-based applications:**
  - Sutton: “Concurrent Programming With The Disruptor”
    - [http://www.youtube.com/watch?v=UvE389P6Er4](http://www.youtube.com/watch?v=UvE389P6Er4)
  - Thompson: “Mechanical Sympathy”
    - [http://mechanical-sympathy.blogspot.com/](http://mechanical-sympathy.blogspot.com/)

- **Read-only traversal to update location**
  - Arcangeli et al: “Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel”
    - [https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli_html/index.html](https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli_html/index.html)
  - Corbet: “Dcache scalability and RCU-walk”
    - [https://lwn.net/Articles/419811/](https://lwn.net/Articles/419811/)
  - Xu: “bridge: Add core IGMP snooping support”
  - Triplett et al., “Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming”
  - Howard: “A Relativistic Enhancement to Software Transactional Memory”
  - McKenney et al: “URCU-Protected Hash Tables”
    - [http://lwn.net/Articles/573431/](http://lwn.net/Articles/573431/)
What is RCU?

To Probe Deeper (3/4)

- Hardware lock elision: Overviews
  - Kleen: “Scaling Existing Lock-based Applications with Lock Elision”
    - http://queue.acm.org/detail.cfm?id=2579227

- Hardware lock elision: Hardware description
  - POWER ISA Version 2.07
    - http://www.power.org/documentation/power-isa-version-2-07/
  - Intel® 64 and IA-32 Architectures Software Developer Manuals
  - Jacobi et al: “Transactional Memory Architecture and Implementation for IBM System z”

- Hardware lock elision: Evaluations
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Section 16.3

- Hardware lock elision: Need for weak atomicity
  - Herlihy et al: “Software Transactional Memory for Dynamic-Sized Data Structures”
  - Shavit et al: “Data structures in the multicore age”
    - http://doi.acm.org/10.1145/1897852.1897873
  - Haas et al: “How FIFO is your FIFO queue?”
    - http://dl.acm.org/citation.cfm?id=2414731
  - Gramoli et al: “Democratizing transactional programming”
    - http://doi.acm.org/10.1145/2541883.2541900
To Probe Deeper (4/4)

- **RCU**
  - Desnoyers et al.: “User-Level Implementations of Read-Copy Update”
    - [http://www.computer.org/cms/Computer.org/dl/trans/td/2012/02/extras/ttd2012020375s.pdf](http://www.computer.org/cms/Computer.org/dl/trans/td/2012/02/extras/ttd2012020375s.pdf)
  - McKenney et al.: “RCU Usage In the Linux Kernel: One Decade Later”
  - McKenney: “Structured deferral: synchronization via procrastination”
    - [http://doi.acm.org/10.1145/2483852.2483867](http://doi.acm.org/10.1145/2483852.2483867)
  - McKenney et al.: “User-space RCU” [https://lwn.net/Articles/573424/](https://lwn.net/Articles/573424/)

- **Possible future additions**
  - Boyd-Wickizer: “Optimizing Communications Bottlenecks in Multiprocessor Operating Systems Kernels”
  - McKenney: “N4037: Non-Transactional Implementation of Atomic Tree Move”
  - McKenney: “C++ Memory Model Meets High-Update-Rate Data Structures”
What is RCU?

Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.
- Credits:
  - This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.
  - Joint work with Mathieu Desnoyers, Alan Stern, Michel Dagenais, Manish Gupta, Maged Michael, Phil Howard, Joshua Triplett, Jonathan Walpole, and the Linux kernel community.
  - Additional reviewers: Carsten Weinhold and Mingming Cao.
What is RCU?

Questions?

Use the right tool for the job!!!

Image copyright © 2004 Melissa McKenney