Introduction to RCU Concepts

Liberal application of procrastination for accommodation of the laws of physics – for more than two decades!
Mutual Exclusion

- What mechanisms can enforce mutual exclusion?
Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example from CACM article)
  - 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?)
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
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

Will holding this lock prevent the cat from dying?
Read-Only Bucket-Locked Hash Table Performance

2GHz Intel Xeon Westmere-EX (64 CPUs)
1024 hash buckets
Read-Only Bucket-Locked Hash Table Performance

Why the dropoff???

Number of CPUs/Threads

Lookups per Millisecond

2GHz Intel Xeon Westmere-EX, 1024 hash buckets
Varying Number of Hash Buckets

2GHz Intel Xeon Westmere-EX

Still a dropoff...

Number of CPUs/Threads

Lookups per Millisecond

16384
8192
4096
2048
1024
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
Bucket-Locked Hash Performance: 1 CPU/Socket

2GHz Intel Xeon Westmere-EX: This is not the sort of scalability Schrödinger requires!!!
Performance of Synchronization Mechanisms
Problem With Physics #1: Finite Speed of Light
Problem With Physics #2: Atomic Nature of Matter

Source

I feel so fat!

Base

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

Drain

And our dielectric constant isn’t big enough for them! They can go find some other $\$\$\$\$\$\$\$ atom! Sheesh!
How Can Software Live With This Hardware???
Design Principle: Avoid Bottlenecks

Only one of something: bad for performance and scalability. Also typically results in high complexity.
Design Principle: Avoid Bottlenecks

Many instances of something good! Full partitioning even better!!!
Avoiding tightly coupled interactions is an excellent way to avoid bugs. But NUMA effects defeated this for per-bucket locking!!!
Design Principle: Get Your Money's Worth

- If synchronization is expensive, use large critical sections
- On Nehalem, off-socket atomic operation costs ~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!
Design Principle: Get Your Money's Worth

- If synchronization is expensive, use large critical sections
- On Nehalem, off-socket atomic operation costs ~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!
  - But the low overhead hash-table insertion/deletion operations do not provide much scope for long critical sections...
Design Principle: Avoid Mutual Exclusion!!!

Plus lots of time waiting for the lock's cache line...
**Design Principle: Avoiding Mutual Exclusion**

```
CPU 0
Reader  Reader  Reader  Reader  Reader
CPU 1
Reader  Reader  Reader  Reader  Reader
CPU 2
Reader  Reader  Reader  Reader  Reader
CPU 3
Reader  Reader  Updater  Reader  Reader
```

No Dead Time!
But How Can This Possibly Be Implemented???
But How Can This Possibly Be Implemented???

I think the poor thing has expired.

No!

Where there is a brain-wave, there is a way!
But How Can This Possibly Be Implemented???

Hazard Pointers and RCU!!!
RCU: Keep It Basic: Guarantee Only Existence

- 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...
RCU: How Updaters Guarantee Existence

- 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 = cpotr;
  rcu_assign_pointer(cpotr, 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

- Next slides give diagram representation
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!!!
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???
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
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* begins after synchronize_rcu() call and ends after all CPUs execute a context switch
Performance
Theoretical Performance

RCU (wait-free)

Uncontended

71.2 cycles

1 cycle

Full performance, linear scaling, real-time response

73 CPUs to break even with a single CPU!

Contended, No Spinning

71.2 cycles

1 cycle

144 CPUs to break even with a single CPU!!!

71.2 cycles
Measured Performance
Schrödinger's Zoo: Read-Only

RCU and hazard pointers scale quite well!!!
Schrödinger's Zoo: Read-Only Cat-Heavy Workload

RCU handles locality quite well, hazard pointers not bad, bucket locking horribly.
**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>
RCU Performance: “Free is a Very Good Price!!!”
And Nothing Is Faster Than Doing Nothing!!!
RCU Area of Applicability

- **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 Consistent Data (RCU is Really Unlikely to be the Right Tool For The Job, But It Can: (1) Provide Existence Guarantees For Update-Friendly Mechanisms (2) Provide Wait-Free Read-Side Primitives for Real-Time Use)**

Schrodinger's zoo is in blue: Can't tell exactly when an animal is born or dies anyway! Plus, no lock you can hold will prevent an animal's death...
RCU Applicability to the Linux Kernel

![Graph showing the increase in RCU API uses from 2002 to 2014. The graph has a y-axis labeled with numbers from 0 to 9000 and an x-axis labeled with years from 2002 to 2014. The graph shows a steady increase in RCU API uses over time.]
Summary
Summary

- Synchronization overhead is a big issue for parallel programs
- Straightforward design techniques can avoid this overhead
  - Partition the problem: “Many instances of something good!”
  - Avoid expensive operations
  - Avoid mutual exclusion
- RCU is part of the solution, as is hazard pointers
  - Excellent for read-mostly data where staleness and inconsistency OK
  - Good for read-mostly data where consistency is required
  - Can be OK for read-write data where consistency is required
  - Might not be best for update-mostly consistency-required data
    - Provide existence guarantees that are useful for scalable updates
    - Used heavily in the Linux kernel
- Much more information on RCU is available...
Graphical Summary

Not only are they lazy, they get more work done than I do!
To Probe Further:

- [https://queue.acm.org/detail.cfm?id=2488549](https://queue.acm.org/detail.cfm?id=2488549) — “Structured Deferral: Synchronization via Procrastination”
- [http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159](http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159) and [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) — “User-Level Implementations of Read-Copy Update”
- [git://lttng.org/userspace-rcu.git](git://lttng.org/userspace-rcu.git) (User-space RCU git tree)
- [http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf](http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf) — Comparison of RCU and NBS (later appeared in JPDC)
- [http://doi.acm.org/10.1145/1400097.1400099](http://doi.acm.org/10.1145/1400097.1400099) — History of RCU in Linux (Linux changed RCU more than vice versa)
- [http://read.seas.harvard.edu/cs261/2011/rcu.html](http://read.seas.harvard.edu/cs261/2011/rcu.html) — Harvard University class notes on RCU (Courtesy Eddie Koher)
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.
Questions?

Use the right tool for the job!!!