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
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

```plaintext
0: lock  →  mouse  →  zebra
1: lock  →  boa   →  cat   →  gnu
2: lock
3: lock
```
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

Diagram:

- 0: lock
- 1: lock
- 2: lock
- 3: lock

0: mouse → zebra
1: boa → cat → gnu

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???

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

![Graph showing the performance of hash table lookups with varying numbers of hash buckets. The graph plots the number of lookups per millisecond against the number of CPUs/Threads. The chart shows a peak performance with a certain number of hash buckets, followed by a dropoff. The labels indicate that the performance improves with fewer hash buckets, but still shows a dropoff. The graph is labeled with the number of hash buckets: 1024, 2048, 4096, 8192, 16384. The processor used is a 2GHz Intel Xeon Westmere-EX.](image-url)
NUMA Effects???

- `/sys/devices/system/cpu/cpu00/cache/index00/shared_cpu_list: -0,32`
- `/sys/devices/system/cpu/cpu00/cache/index1/shared_cpu_list: -0,32`
- `/sys/devices/system/cpu/cpu00/cache/index2/shared_cpu_list: -0,32`
- `/sys/devices/system/cpu/cpu00/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

Faster!
Faster!
Problem With Physics #2: Atomic Nature of Matter

Source

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

I feel so fat!

Base

And our dielectric constant isn’t big enough for them! They can go find some other $%^*@ atom! Sheesh!

Drain
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

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 = 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
- 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)

Full performance, linear scaling, real-time response

73 CPUs to break even with a single CPU!

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

Uncontended

Contended, No Spinning

71.2 cycles

1 cycle

71.2 cycles

71.2 cycles

71.2 cycles

1 cycle

Local (blocking)

RCU (wait-free)
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
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
  - “Structured Deferral: Synchronization via Procrastination”
- http://doi.ieeeecomputersociety.org/10.1109/TPDS.2011.159 and
  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 (User-space RCU git tree)
  - Applying RCU and weighted-balance tree to Linux mmap_sem.
  - RCU-protected resizable hash tables, both in kernel and user space
  - Combining RCU and software transactional memory
- http://wiki.cs.pdx.edu/rp/: Relativistic programming, a generalization of RCU
- http://lwn.net/Articles/262464/, http://lwn.net/Articles/263130/, http://lwn.net/Articles/264090/
  - “What is RCU?” Series
  - RCU motivation, implementations, usage patterns, performance (micro+sys)
  - System-level performance for SELinux workload: >500x improvement
  - Comparison of RCU and NBS (later appeared in JPDC)
- http://doi.acm.org/10.1145/1400097.1400099
  - History of RCU in Linux (Linux changed RCU more than vice versa)
  - 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!!!
Introduction to Userspace RCU Data Structures
Mathieu Desnoyers

EfficiOS Inc.

- [http://www.efficios.com](http://www.efficios.com)

Author/Maintainer of

- Userspace RCU,
- LTTng kernel and user-space tracers,
- Babeltrace.
Introduction to major Userspace RCU (URCU) concepts,

URCU memory model,

URCU APIs
- Atomic operations, helpers, reference counting,

URCU Concurrent Data Structures (CDS)
- Lists,
- Stacks,
- Queues,
- Hash tables.
Userspace RCU hands-on tutorial
Data Structure Characteristics

- Scalability
- Real-Time Response
- Performance
Non-Blocking Algorithms

✅ Progress Guarantees

⬆️ Lock-free

- guarantee of system progress.

⬆️⬆️⬆️ Wait-free

- also guarantee per-thread progress.
Weakly ordered architectures can reorder memory accesses

Initial conditions
\[ x = 0 \]
\[ y = 0 \]

CPU 0
\[ x = 1; \]
\[ y = 1; \]

CPU 1
\[ r1 = y; \]
\[ r2 = x; \]

If r2 loads 0, can r1 have loaded 1?
Weakly ordered architectures can reorder memory accesses

Initial conditions
x = 0
y = 0

CPU 0
x = 1;
y = 1;

CPU 1
r1 = y;
r2 = x;

If r2 loads 0, can r1 have loaded 1?

**YES**, at least on many weakly-ordered architectures.
# Memory Model

Summary of Memory Ordering

Paul E. McKenney, Memory Ordering in Modern Microprocessors, Part II,


<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>AMD64</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>IA64</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>(PA–RISC)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>PA–RISC CPUs</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>POWER</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SPARC RMO</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>(SPARC PSO)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SPARC TSO</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>x86</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>(x86 O2Store)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>zSeries</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>
Memory Model

? But how comes we can usually expect those accesses to be ordered?

🔒 Mutual exclusion (locks) are the answer,

⇓¹ They contain the appropriate memory barriers.

いますが なにが発生するか考えて 互いに 排他（ロック）が答えです、

⇓¹ それが 恰当な 間の記憶 障壁です。

❑ But what happens if we want to do synchronization without locks?

⇓¹ Need to provide our own memory ordering guarantees.

傘 なにが 発生するか考えて 互いに 排他（ロック）が 答えです、

⇓¹ それが 自分の 記憶 障壁 保証です。
Memory Model

- Userspace RCU
  - Similar memory model as the Linux kernel, for user-space.
  - For details, see Linux Documentation/memory-barriers.txt
Userspace RCU Memory Model

- urcu/arch.h
  - memory ordering between processors
    - cmm_smp_{mb,rmb,wmb}()
  - memory mapped I/O, SMP and UP
    - cmm_{mb,rmb,wmb}()
  - eventual support for architectures with incoherent caches
    - cmm_smp_{mc,rmc,wmc}()

- urcu/compiler.h
  - compiler-level memory access optimisation barrier
    - cmm_barrier()
Userspace RCU Memory Model (cont.)

- urcu/system.h
  - Inter-thread load and store
    - CMM_LOAD_SHARED(),
    - CMM_STORE_STORE_SHARED(),
  - Semantic:
    - Ensures aligned stores and loads to/from word-sized, word-aligned data are performed atomically,
    - Prevents compiler from merging and refetching accesses.
    - Deals with architectures with incoherent caches,
Atomic operations and data structure APIs have their own memory ordering semantic documented.
• Similar to the Linux kernel atomic operations,
• urcu/uatomic.h
  • uatomic_{add,sub,dec,inc}_return(), uatomic_cmpxchg(), uatomic_xchg() imply full memory barrier (smp_mb()).
  • uatomic_{add,sub,dec,inc,or,and,read,set}() imply no memory barrier.
  • cmm_smp_mb__{before,after}_uatomic_*() provide associated memory barriers.
Userspace RCU Helpers

• urcu/compiler.h
  • Get pointer to structure containing a given field from pointer to field.
    • caa_container_of()

• urcu/compat-tls.h
  • Thread-Local Storage
    • Compiler __thread when available,
    • Fallback on pthread keys,
    • DECLARE_URCU_TLS(),
    • DEFINE_URCU_TLS(),
    • URCU_TLS().
Userspace RCU Reference Counting

- Reference counting based on Userspace RCU atomic operations,
- urcu/ref.h
  - urcu_ref_{set,init,get,put}()
URCU Concurrent Data Structures

- Navigating through URCU CDS API and implementation
- Example of wait-free concurrent queue
  - urcu/wfcqueue.h: header to be included by applications,
    - If _LGPL_SOURCE is defined before include, functions are inlined, else implementation in liburcu-cds.so is called,
  - urcu/wfcqueue.h and wfcqueue.c implement exposed declarations and LGPL wrapping logic,
  - Implementation is found in urcu/static/wfcqueue.h.
URCU lists

- Circular doubly-linked lists,
- Linux kernel alike list API
  - urcu/list.h
  - cds_list_{add,add_tail,del,empty,replace,splice}()
  - cds_list_for_each*()
- Linux kernel alike RCU list API
  - Multiple RCU readers concurrent with single updater.
  - urcu/rculist.h
  - cds_list_{add,add_tail,del,replace,for_each*}_rcu()
URCU hlist

- Linear doubly-linked lists,
- Similar to Linux kernel hlists,
- Meant to be used in hash tables, where size of list head pointer matters,
- urcu/hlist.h
  - cds_hlist_{add_head,del,for_each*}()
- urcu/rcuhlist.h
  - cds_hlist_{add_head,del,for_each*}_rcu()
Stack (Wait-Free Push, Blocking Pop)

- urcu/wfstack.h
  - N push / N pop
  - Wait-free push
    - cds_wfs_push()
  - Wait-free emptiness check
    - cds_wfs_empty()
  - Blocking/nonblocking pop
    - __cds_wfs_pop_blocking()
    - __cds_wfs_pop_nonblocking()
  - subject to existence guarantee constraints
    - Can be provided by either RCU or mutual exclusion on pop and pop all.
Stack (Wait-Free Push, Blocking Pop)

- urcu/wfstack.h (cont.)
  - Wait-free pop all
    - \texttt{__cds_wfs_pop_all()}
    - subject to existence guarantee constraints
      - Can be provided by either RCU or mutual exclusion on pop and pop all.
  - Blocking/nonblocking iteration on stack returned by pop all
    - \texttt{cds_wfs_for_each_blocking*()}
    - \texttt{cds_wfs_first()}, \texttt{cds_wfs_next_blocking()}, \texttt{cds_wfs_next_nonblocking()}
Lock-Free Stack

• urcu/lfstack.h
  • N push / N pop
  • Wait-free emptiness check
    • cds_lfs_empty()
  • Lock-free push
    • cds_lfs_push()
  • Lock-free pop
    • __cds_lfs_pop()
    • subject to existence guarantee constraints
      • Can be provided by either RCU or mutual exclusion on pop and pop all.
Lock-Free Stack

- urcu/lfstack.h (cont.)

- Lock-free pop all and iteration on the returned stack
  - __cds_lfs_pop_all()
  - subject to existence guarantee constraints
    - Can be provided by either RCU or mutual exclusion on pop and pop all.
  - cds_lfs_for_each*()
Wait-Free Concurrent Queue

- urcu/wfcqueue.h
  - N enqueue / 1 dequeue
  - Wait-free enqueue
    - cds_wfcq_enqueue()
  - Wait-free emptiness check
    - cds_wfcq_empty()
  - Blocking/nonblocking dequeue
    - __cds_wfcq_dequeue_blocking()
    - __cds_wfcq_dequeue_nonblocking()
      - Mutual exclusion of dequeue, splice and iteration required.
Wait-Free Concurrent Queue

- urcu/wfcqueue.h (cont.)
  - Blocking/nonblocking splice (dequeue all)
    - __cds_wfcq_splice_blocking()
    - __cds_wfcq_splice_nonblocking()
      - Mutual exclusion of dequeue, splice and iteration required.
Wait-Free Concurrent Queue

- urcu/wfcqueue.h (cont.)
  - Blocking/nonblocking iteration
    - __cds_wfcq_first_blocking()
    - __cds_wfcq_first_nonblocking()
    - __cds_wfcq_next_blocking()
    - __cds_wfcq_next_nonblocking()
    - __cds_wfcq_for_each_blocking()*
      - Mutual exclusion of dequeue, splice and iteration required.
Wait-Free Concurrent Queue

- urcu/wfcqueue.h (cont.)
  - Splice operations can be chained, so N queues can be merged in N operations.
    - Independent of the number of elements in each queue.
Lock-Free Queue

- urcu/rculfqueue.h
- Requires RCU synchronization for queue nodes
- Lock-Free RCU enqueue
  - cds_lfq_enqueue_rcu()
- Lock-Free RCU dequeue
  - cds_lfq_dequeue_rcu()
- *No* splice (dequeue all) operation
- Requires a destroy function to dispose of queue internal structures when queue is freed.
  - cds_lfq_destroy_rcu()
RCU Lock-Free Hash Table

- urcu/rculfhash.h
- Wait-free lookup
  - Lookup by key,
  - cds_lfht_lookup()
- Wait-free iteration
  - Iterate on key duplicates
    - cds_lfht_next_duplicate()
  - Iterate on entire hash table
    - cds_lfht_first()
    - cds_lfht_next()
    - cds_lfht_for_each*()
RCU Lock-Free Hash Table

• Lock-Free add
  • Allows duplicate keys
  • cds_lfht_add().

• Lock-Free del
  • Remove a node.
  • cds_lfht_del().

• Wait-Free check if deleted
  • cds_lfht_is_node_deleted().
RCU Lock-Free Hash Table

- Lock-Free add_unique
  - Add node if node's key was not present, return added node,
  - Acts as a lookup if key was present, return existing node,
  - cds_lfht_add_unique().
RCU Lock-Free Hash Table

• Lock-Free replace
  • Replace existing node if key was present, return replaced node,
  • Return failure if not present,
  • cds_lfht_replace().

• Lock-Free add_replace
  • Replace existing node if key was present, return replaced node,
  • Add new node if key was not present.
  • cds_lfht_add_replace().
RCU Lock-Free Hash Table

- Uniqueness guarantee
  - Lookups/traversals executing concurrently with add_unique, add_replace, replace and del will never see duplicate keys.

- Automatic resize and node accounting
  - Pass flags to cds_lfht_new()
    - CDS_LFHT_AUTO_RESIZE
    - CDS_LFHT_ACCOUNTING
  - Node accounting internally performed with split-counters, resize performed internally by call_rcu worker thread.
Userspace RCU Hands-on Tutorial

RCU Island Game

http://urcu.so
Downloads required

- Userspace RCU library 0.8.0
  - [http://urcu.so](http://urcu.so)
  - Follow README file to install

- RCU Island game
  - git clone [git://github.com/efficios/urcu-tutorial](http://github.com/efficios/urcu-tutorial)
  - Run `./bootstrap`
  - Solve exercises in exercises/questions.txt
Thank you!

EfficiOS

http://www.efficios.com

http://urcu.so
lttng-dev@lists.lttng.org
@lttng_project