Read-Log-Update

```
What is Read-Copy-Update (RCU)?
  An API for synchronizing access to read-mostly data structures
  Basic API for reader:
    rcu_read_lock()   - start read-side critical section (can nest)
    rcu_dereference() - get RCU-protected global pointer
    rcu_read_unlock() - end read-side critical section
  Basic API for writer:
    rcu_assign_pointer() - set RCU-protected pointer to new value
    synchronize_rcu()    - wait for all readers to be done with old ptr
    call_rcu()           - register async. callback for when readers are done
Example:
  Global: global_obj *gp; spinlock gp_lock;
  Reader: rcu_read_lock(); use(rcu_dereference(gp)); rcu_read_unlock();
  Writer: lock(&gp_lock);
          global_obj *old = gp;
          rcu_assign_pointer(gp, alloc_updated(old));
          unlock(&gp_lock);
          synchronize_rcu();
          obj_free(old);
What guarantees does RCU make?
  See http://www.efficios.com/pub/rcu/urcu-main.pdf (Desnoyers) fig. 2
  A read-side critical section can overlap at most two of following in writer:
      +----------------+--------------+-------------+
      | assign_pointer | grace period | free object |
      +----------------+--------------+-------------+
How would you implement RCU?  Multiple strategies:
  1. Disable preemption in readers, run on all CPUs in synchronize
       rcu_read_lock() disables preemption
       rcu_dereference() is memory_order_consume load
       rcu_synchronize() { for (i = 0; i < ncpus; ++i) run_on_cpu(i); }
  2. Periodically call rcu_quiescent() in readers not in critical section
       _Atomic uint64_t rcu_ctr; mutex_t rcu_lock;
       thread_local _Atomic uint64_t local_ctr;
       void rcu_synchronize() {
         lock(&rcu_lock);
         local_ctr = ++rcu_ctr;
         for_each_thread { while (thread's local_ctr < local_ctr) yield(); }
         unlock(&rcu_lock);
       }
       void rcu_quiescent() {
         local_ctr = global_ctr;
       }
  3. Use fences or synchronized atomics inside rcu_read_lock/rcu_read_unlock
       rcu_read_lock() looks like rcu_quiescent() which no longer exists
       Last rcu_read_unlock() sets local_ctr = MAXINT

Note you don't have to copy the whole data structure with RCU
  Can you remove item from linked list without copying list?
    Yes.  See RLU Figure 1 and Desnoyers Figure 3
  What happens if reader traverses linked list forwards & backwards?
    Writer can't atomically swap two pointers simultaneously
    Risk seeing invariant violations like item->next->prev != item -- oops
  So works for some structures (singly-linked list) & not others (doubly)
  What about concurrent updates?  Also quite dangerous
  The upshot:  Need to be very clever for each data structure
  Goal of read-LOG-update (RLU) is to fix all of this
    Instead of swapping one pointer atomically, swap many items atomically
    Maybe even atomically make updates by multiple concurrent writers!

If every problem in CS can be fixed with a level of indirection
    ... what's the magic level of indirection in RLU?
  We will actually update a bunch of data structures non-atomically
```

```
  But, some GLOBAL CLOCK makes readers ignore all of these updates
  Then atomically update the clock and new readers will see new values
  In-progress readers will continue to use old clock value!

What is RLU API?
  rlu_reader_lock() / rlu_dereference(obj) - like RCU
  rlu_reader_unlock() - Like RCU, but might also commit writes you made
  rlu_try_lock(ptr) - May return locked private object copy you can modify
  rlu_cmp_objs(ptr, ptr) - because unlike RCU "obj1 == obj2" may not work
  rlu_assign_pointer(lptr, ptr) - deals with multiple pointer values properly
  rlu_abort() - abort current transaction and undo any pending writes
  rlu_malloc()/rlu_free()?  (Reserve space for log_ptr)

What data structures do we need (Sec. 3.4)?
  * Per thread:
    - 2 Write logs: one active, one left over from previous epoch
    - run_counter: odd when using RLU, even when outside critical section
    - local_clock: Time at which your view of data should be frozen
    - write_clock: Infinity until you want to commit
    - is_writer: Flag is true if you have made modifications
  * Global:
    - global_clock: Used to initialize local clock each transaction
    - List of threads providing access to per-thread info for all threads
  * Per object in heap
    - log_ptr: pointer to copy of object in a log (or NULL if none)
  * Per object in log
    - Thread identifier of thread that owns log
    - A pointer to the original object of which this is a copy
    - The size of the object (so you know where next log entry is)
    - Special reserved value (where log_ptr would be in heap object)

How does RLU work with serial writers (assume external big write lock)?
  For reader:
    rlu_reader_lock()
      - bump run_counter to something odd
      - local_clock = global_clock - this determines version of data you see
      - is_writer = false
    rlu_reader_unlock() - bump run_counter to something even
    rlu_dereference(ptr)
      - unlocked heap object? return as-is
      - log object? return as-is (you had the pointer, must be okay)
      Otherwise, we must have a locked heap object
      - Did we lock it?  Then return corresponding log object
      - Is local_clock >= log owner's write_clock?  Then return log object
      - Otherwise, locked after rlu_reader_lock(), so return heap object
    rlu_assign_pointer(lptr, ptr) - always translate ptr to heap if log
  For writer:
    rlu_try_lock(ptr)
      - Initialize log header for object copy in our log
      - Set heap object log_ptr to point to our header
      - Copy heap object into log
    rlu_synchronize()
      - Record run_counter of all active threads
      - For each thread T, spin while all of:
          1. T's run_counter is odd (so T using RLU), and
          2. T's run_counter is the same as we recorded before the loop, and
          3. T's local_clock < our write_clock (T ignoring our log)
    rlu_reader_unlock()
      - set write_clock = global_clock + 1 (was infinity)
      - ++global clock - now new readers will use our log
      - rlu_synchronize()
      - set write_clock = infinity
      - Swap our two logs (readers might still use recent one - 3.6.2)

What happens if you add concurrent writers?
```

  We can use log_ptr as a per-object lock
  Just modify rlu_try_lock(ptr)
      - Already locked by us? return log_ptr
      - Already lockec by another thread? rlu_abort()
      - Initialize log header for object copy in our log
      - Try to lock with atomic compare-and-swap on heap header
      - Success? Copy object from heap to log. Failure? rlu_abort()
  Drawbacks of concurrent writer version?
    Aborts (don't happen in serial writer version)
    More complicated semantics with concurrent writers, e.g.:
      With serial writer RLU, following code guarantees a == b:
        T1: reg = *rlu_dereference(a); *rlu_try_lock(b) = reg;
        T2: reg = *rlu_dereference(b); *rlu_try_lock(a) = reg;
      With concurrent writer version, might just swap a's and b's values

Why do you want RLU deferring (3.7) and what is it?
  rlu_synchronize() is expensive. When do you really need it?
    After copying object into heap (to ensure old log_ptr not used)
    Therefore if another thread wants to write same object
    Also if you want another thread to see changes--why?
      Say just you wrote back log object w/o incrementing counter
      Then set log_ptr to NULL to release the lock
      Now readers whose local_clock < your write_clock will see new version
      Can't set all log_ptrs simultaneously - violates atomicity
      A readers may see both versions in one transaction - violates isolation
  How to reduce calls to rlu_synchronize()?
    Omit bumping global_clock and calling rcu_synchronize in rcu_read_unlock
    Instead send "sync request" to other thread after write-write conflict
    Anticipate fewer sync requests than is_writer transactions
  Bonus: reduces contention on global clock (to write-write conflicts)
  Bonus 2: fewer conflict cache messes in readers
    Yes, but because they are seeing stale data
  Drawbacks?
    Usage scenario must tolerate added update latency
    What if sync request takes too long? Maybe write back other thread's log?
    Only a big win with one benchmark at (Citrus to 80 cores)
      More modest win with only 16 cores

What evaluation questions should be asking?
  1. How is performance compared to RCU today?
     On average systems today
     When scaled to many cores
  2. How is correctness?
     Is the approach correct (these things are tricky)?
     Is it easy to implement correctly, or bug prone?
  3. Does it simplify/allow more data structures than RCU?
What RCU do they compare to--is this fair?
  "State-of-the art" urcu library (Desnoyers) and linux kernel RCU
  Include numbers for incorrect but faster Harris linked list w. mem. leak
  Authors even fixed a performance bug in linux list_entry_rcu (4.6)
  But... authors don't support all features of more mature RCUs
* Fig. 4--What is the best linked list?
  RLU unless you want memory leaks
  Note how concurrent updates kill RCU scalability!
  Don't show deferred RLU but claim it is the same
* Fig. 5 (hash table)
  Why does RCU dominate?
    RCU readers do less constant work than Harris readers (and RLU)
    Use RCU on individual buckets--means very few actual write conflicts
  Why does deferred update win here but not in linked list?
    Fewer write-write conflicts in bucketed hash table than in single list
* Fig. 6 (resizable hash table)
  How does a resizable hash table even work in RCU?
    Super complicated because can only update one pointer at a time
    So first create expanded hash table where two buckets share each list

```
      Assumes you aren't completely rehashing, but just splitting buckets
      Not useful, yet, but at least lookups in new table correct
    Then go through and "unzip" the buckets so they share tails
    Eventually tails will be NULL and you are done
  How does a resizable hash table work in RLU?
    Just like a regular hash table + some rcu_dereference/rcu_try_lock
  Which performs better RCU or RLU for resizable hash table?
    Figure 6 appears to show slight win for RCU.  Is this fair?
    Resizable hash table doesn't support concurrent insert/remove
    Benchmarks is from other paper, so designed to make RCU look good
    Might well see win for RLU under different workload
* Fig. 7 (stress test)--Why is RLU twice as slow even for 1 CPU?
  100% update, one item per bucket is best case for RCU
  Cost of copying objects in RLU causes 2x slowdown
  At least this likely bounds worst-case for RLU
* What do we learn from Citrus search tree (Sec 4.5)?
  Complex structures possible with RCU, but complicated
  Compare listing 3 to listing 4--RLU wins on simplicity
  Plus RLU performs beter
Do authors answer questions 2 and 3?
  2 (correctness): passes applicable subset of kernel torture test
  3 (functionality):
    Already saw how much simpler Citrus code got
    Kyoto cabinet Cache DB
      RLU speeds up application where RCU is intractable
```