RCU and C++
What Is RCU, Really?

- Publishing of new data: `rcu_assign_pointer()`
- Subscribing to the current version of data: `rcu_dereference()`
- Waiting for pre-existing RCU readers: Avoid disrupting readers by maintaining multiple versions of the data
  - Reader begins with `rcu_read_lock()` and ends at matching `rcu_read_unlock()`
  - The time an updater must wait is a grace period
  - Blocking wait for a grace period: `synchronize_rcu()`
  - Asynchronous wait for a grace period: `call_rcu()`
    - Specified function invoked at the end of a grace period
But if all we do is add, we have a big memory leak!!!

Publication of And Subscription to New Data

Key:
- Dangerous for updates: all readers can access
- 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 and C++

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???
Two Different Versions Simultaneously???

I think the poor thing has expired.

No!

Where there is a brain-wave, there is a way!
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); 
  }})
  ```

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

Grace period extends as needed.

synchronize_rcu()
But it is OK for RCU to be lazy and allow a grace period to extend longer than necessary.
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.
  - In the Linux kernel, can be thousands of updates per grace period!
- 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.
RCU and C++

RCU Asynchronous Grace-Period Detection

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

- **API:**

  ```c
  call_rcu(struct rcu_head head,
           void (*func)(struct rcu_head *rcu));
  ```

- **The rcu_head structure:**
  ```c
  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.**
RCU and C++

RCU Grace Period: An Asynchronous Graphical View

```c
func(&p->rcu);
call_rcu(&p->rcu, func);
```

change_visible_to_all_readers

```
func(&p->rcu);
```
RCU and C++

Destructors Not Necessarily Known At Construction

f() Sometimes Known Here

f() Always Known Here

Allocation → Construction → Add To Structure → Remove From Structure → call_rcu(f) → f(): Destruction → Deallocation

Accessible To All Readers

Grace Period

Accessible Only To Old Readers
Destructors Not Necessarily Known At Construction

When \( f() \) is not known until \( \text{call\_rcu()} \) time, need fixed-width storage!!!
Any Type Any Time Any Translation Unit Anywhere
Must get to the right translation-unit context: Some sort of function pointer...
Underlying C-Language RCU API

```c
1 void std::rcu_read_lock();
2 void std::rcu_read_unlock();
3 void std::synchronize_rcu();
4 void std::call_rcu(struct std::rcu_head *rhp,
5     void cbf(class rcu_head *rhp));
6 void std::rcu_barrier();
7 void std::rcu_register_thread();
8 void std::rcu_unregister_thread();
9 void std::rcu_quiescent_state();
10 void std::rcu_thread_offline();
11 void std::rcu_thread_online();
```
class rcu_domain {
  public:
    virtual void register_thread() = 0;
    virtual void unregister_thread() = 0;
    static inline bool register_thread_needed() { return true; }
    virtual void read_lock() noexcept = 0;
    virtual void read_unlock() noexcept = 0;
    virtual void synchronize() noexcept = 0;
    virtual void call(class rcu_head *rhp,
                      void cbf(class rcu_head *rhp)) = 0;
    virtual void barrier() noexcept = 0;
    virtual void quiescent_state() noexcept = 0;
    virtual void thread_offline() noexcept = 0;
    virtual void thread_online() noexcept = 0;
};
RCU and C++

RCU Scoped Readers

```cpp
class rcu_scoped_reader {
public:
  rcu_scoped_reader();
  rcu_scoped_reader(class rcu_domain *rd);
  rcu_scoped_reader(const rcu_scoped_reader &) = delete;
  rcu_scoped_reader& operator=(const rcu_scoped_reader &) = delete;
  ~rcu_scoped_reader();
}
```

Derived concrete class for each “flavor” in userspace RCU library
Tracking RCU Callbacks: Approach #0
Tracking RCU Callbacks: Approach #1 (Work In Progress)

1 // Isabella Muerte approach
2 template <class T>
3 struct default_deleter;
4
5 template<class T, class Deleter=default_deleter<T>>
6 struct rcu_head_delete2: rcu_head, Deleter {
7
8   Deleter& get_deleter () { return *this; } 
9
10  void call ();
11  void call (rcu_domain& rd);
12 };
Tracking RCU Callbacks: Approach #2 (Work In Progress)

1 // Arthur O'Dwyer approach
2 template<typename T,
3     typename D = default_delete<T>,
4     bool E = is_empty<D>::value>
5 class rcu_head_delete {
6 public:
7     void call(D d = {});
8     void call(rcu_domain &rd, D d = {});
9 };
Schrödinger's Zoo: Read-Only

RCU and hazard pointers scale quite well!!!
RCU Area of Applicability

- **Read-Write, Need Consistent Data (RCU Might Be OK...)**
- **Read-Mostly, Need Consistent Data (RCU Works OK)**
- **Read-Mostly, Stale & Inconsistent Data OK (RCU Works Great!!!)**
- **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...
Future

- Add Hazard Pointers and RCU to Concurrency TS
  - And then to the C++ Standard

- Working drafts:
  - Hazard Pointers: P0233R1
  - RCU memory_order_consume semantics: P0190R2
  - RCU marked dependency chains: TBD
  - RCU C++ bindings: TBD