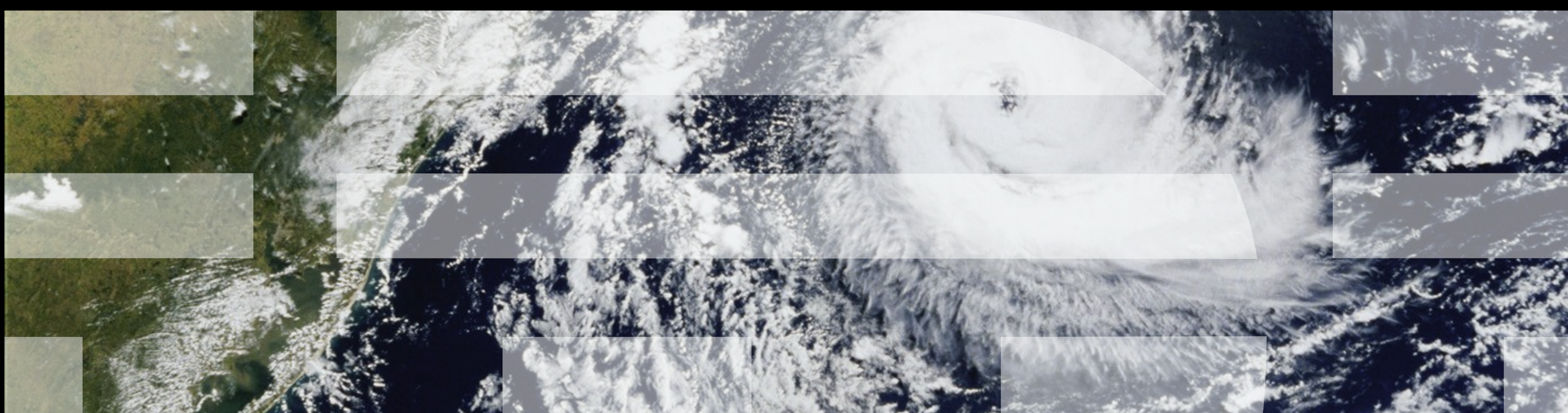
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center

Member, IBM Academy of Technology

CPPCON, September 23, 2016

RCU and C++

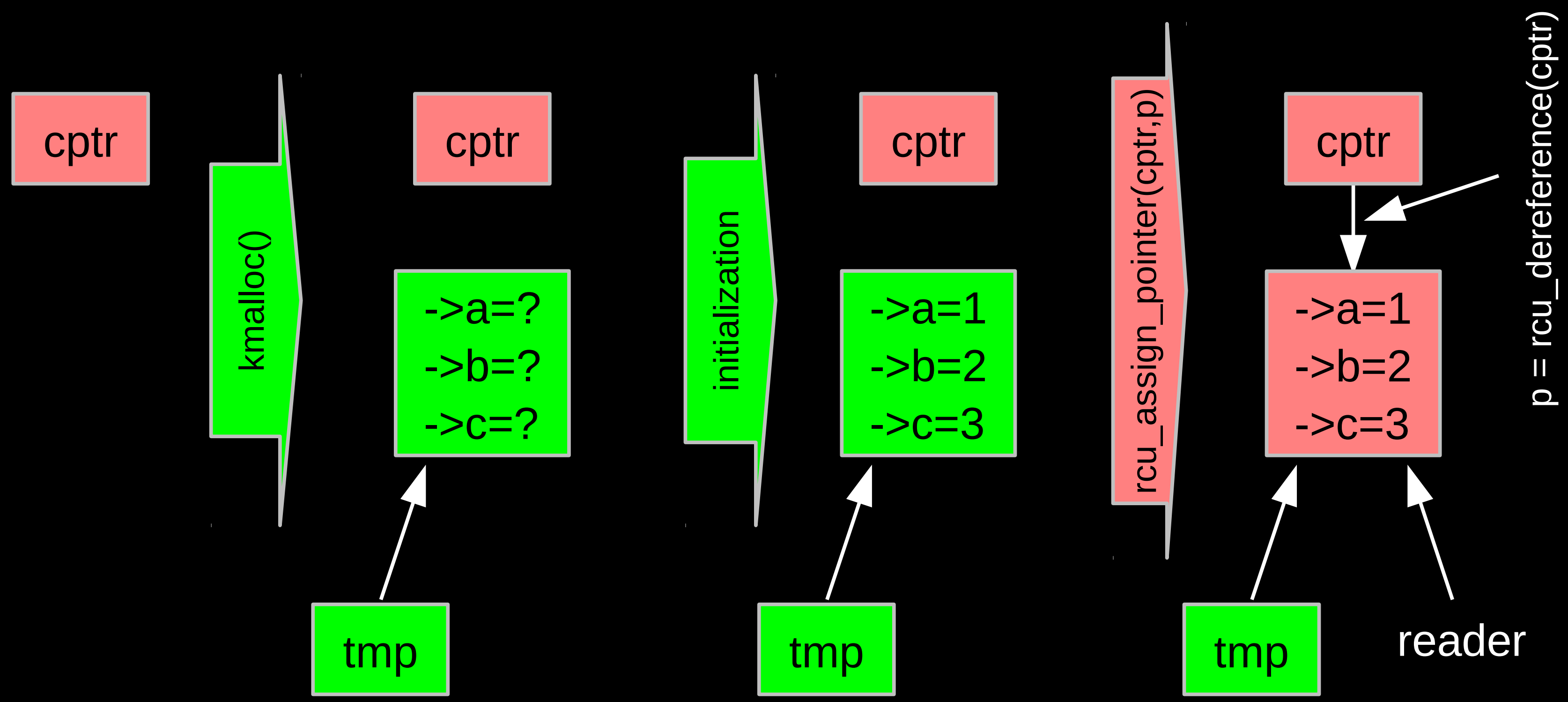


What Is RCU, Really?

- 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

Publication of And Subscription to New Data

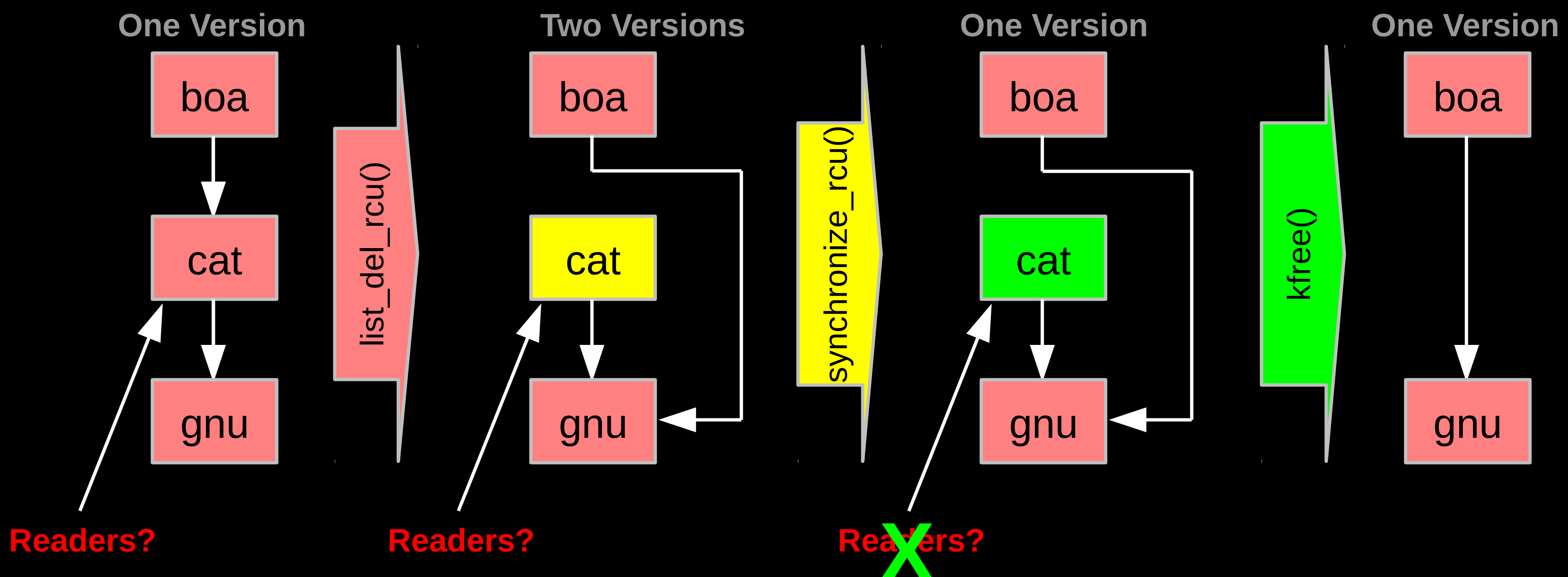
Key:
- Dangerous for updates: all readers can access
- Still dangerous for updates: pre-existing readers can access (next slide)
- Safe for updates: inaccessible to all readers



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 how can software deal with two different versions simultaneously???

Two Different Versions Simultaneously???

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

- Read-side primitives:

```
#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

```
#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

```
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...

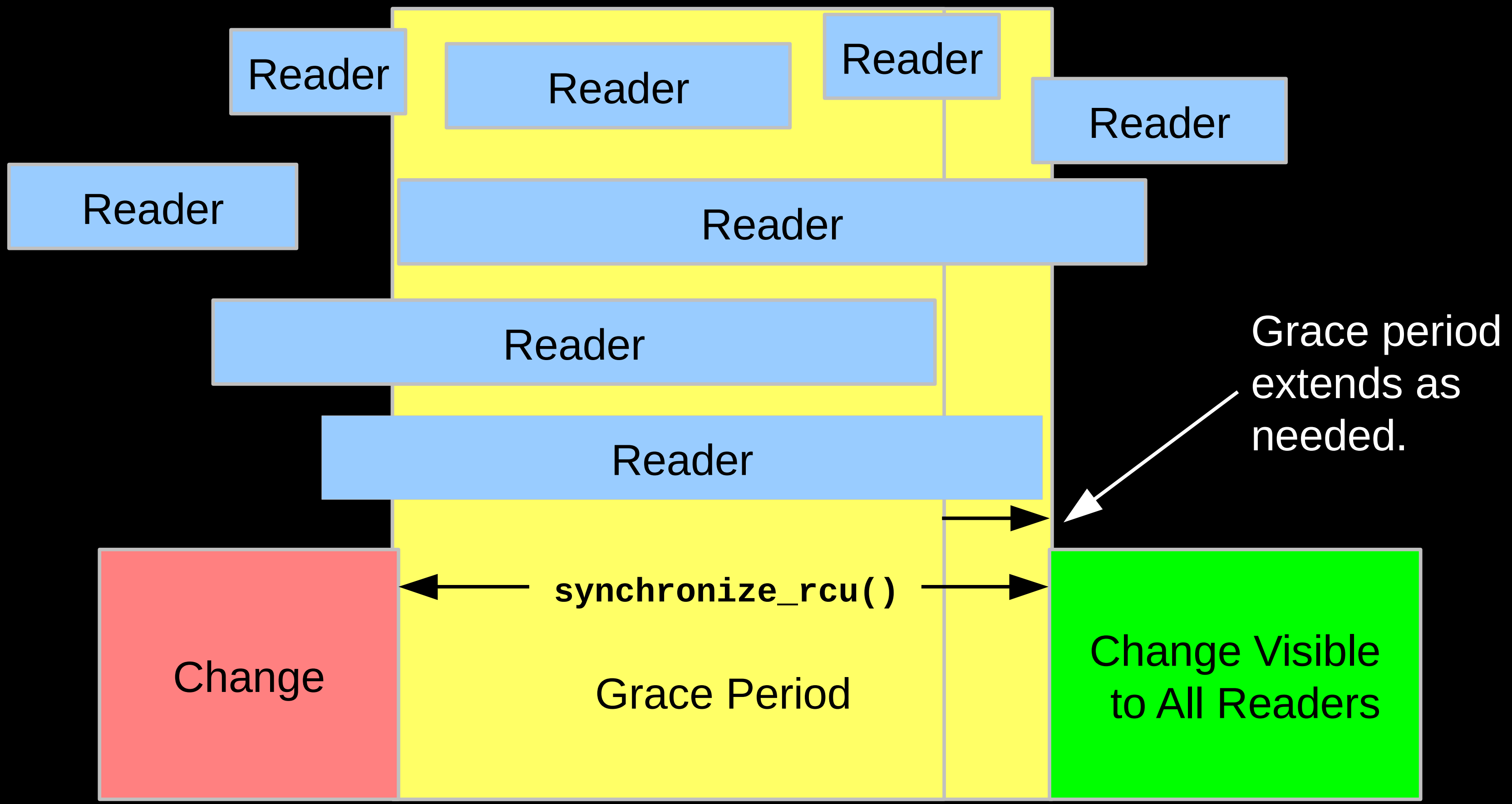
RCU Usage: Updaters

- Updaters must wait for an *RCU grace period* to elapse between making something inaccessible to readers and freeing it

```
spin_lock(&updater_lock);
q = cptr;
rcu_assign_pointer(cptr, new_p);
spin_unlock(&updater_lock);
synchronize_rcu(); /* Wait for grace period. */
kfree(q);
```

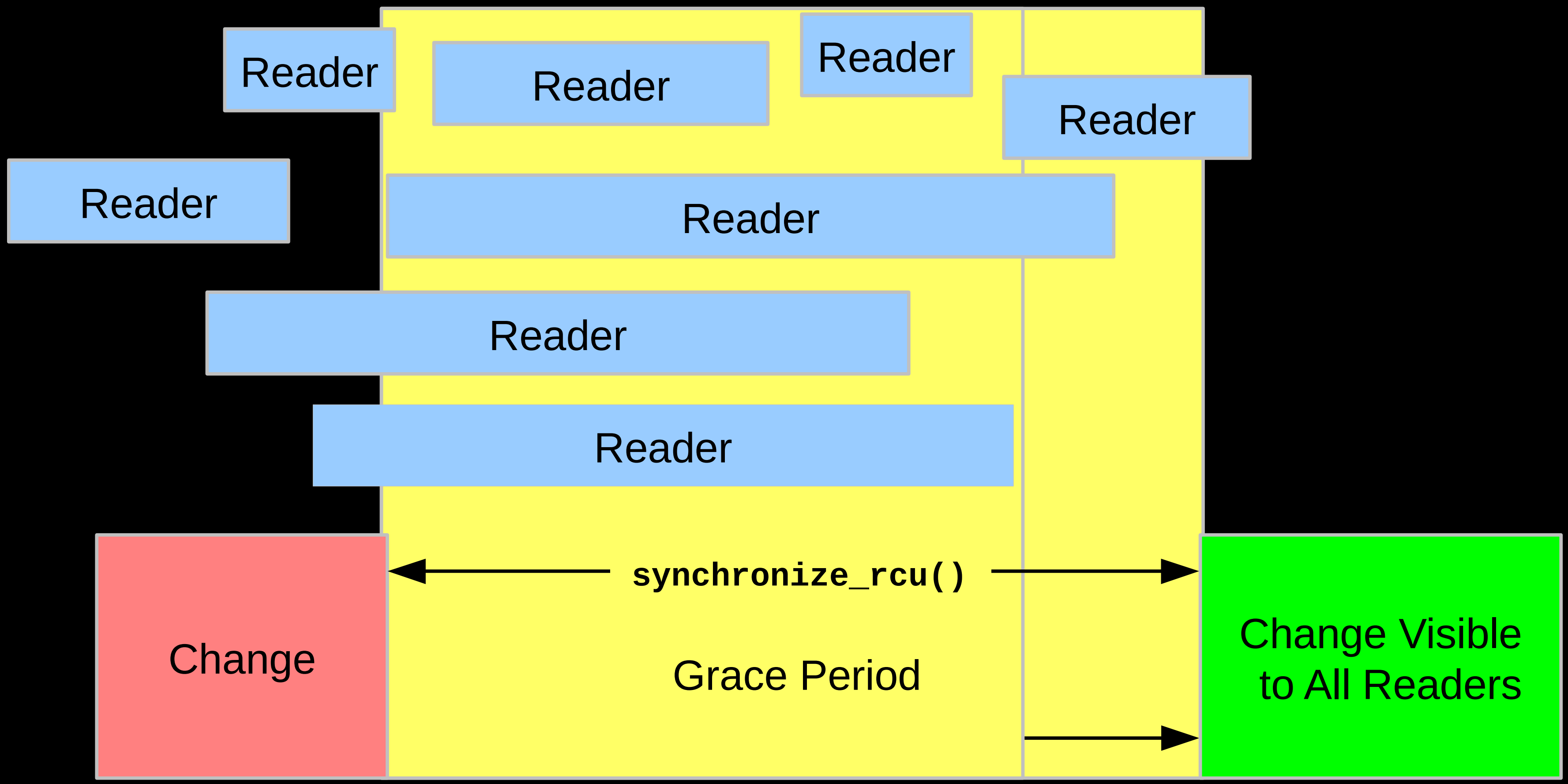
- RCU grace period waits for all pre-exiting readers to complete their RCU read-side critical sections

RCU Grace Period: A Self-Repairing Graphical View



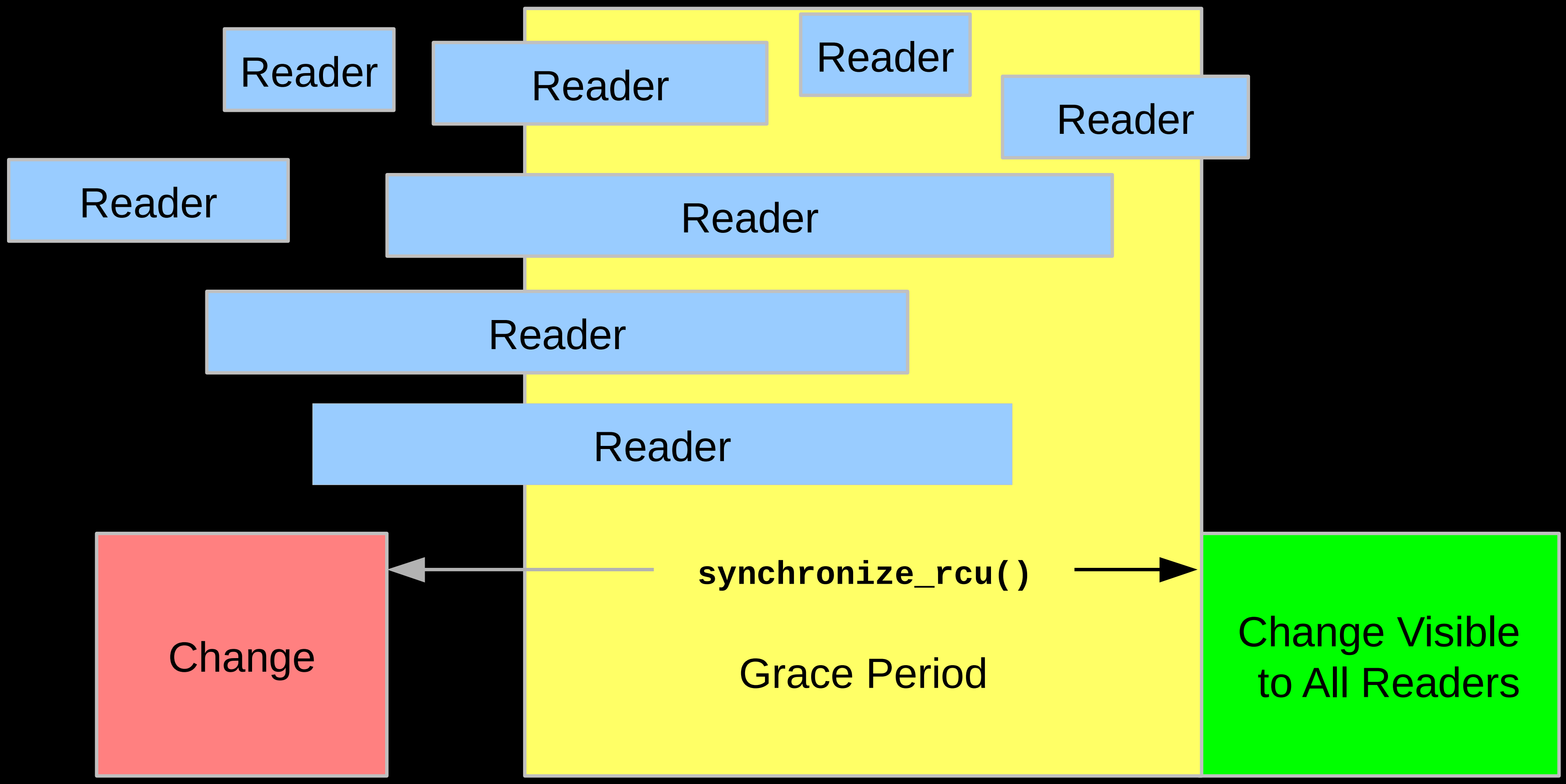
A grace period is not permitted to end until all pre-existing readers have completed.

RCU Grace Period: A Lazy Graphical View



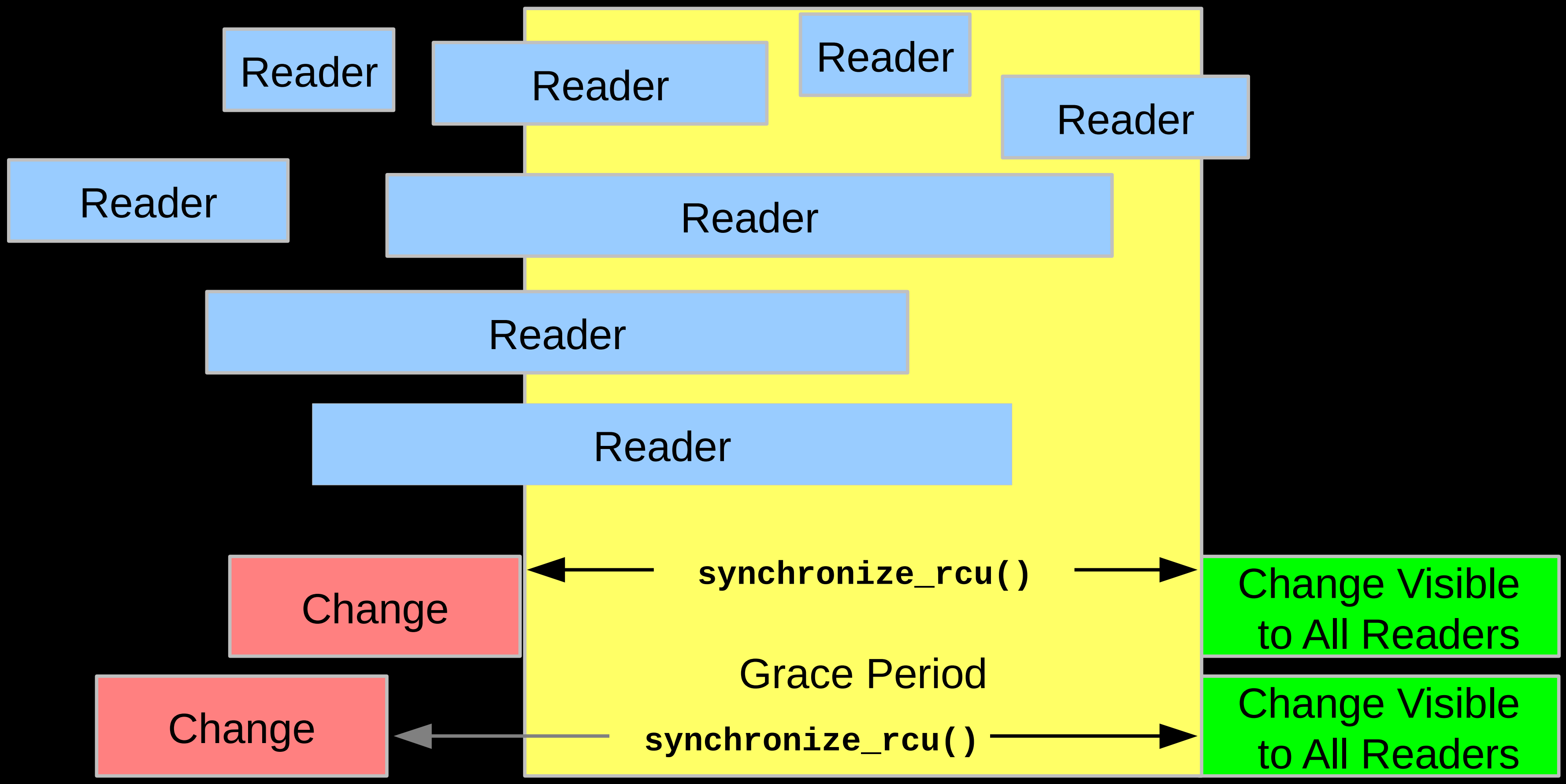
But it is OK for RCU to be lazy and allow a grace period to extend longer than necessary

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?

RCU Grace Period: A Usefully Lazy Graphical View



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 Asynchronous Grace-Period Detection

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

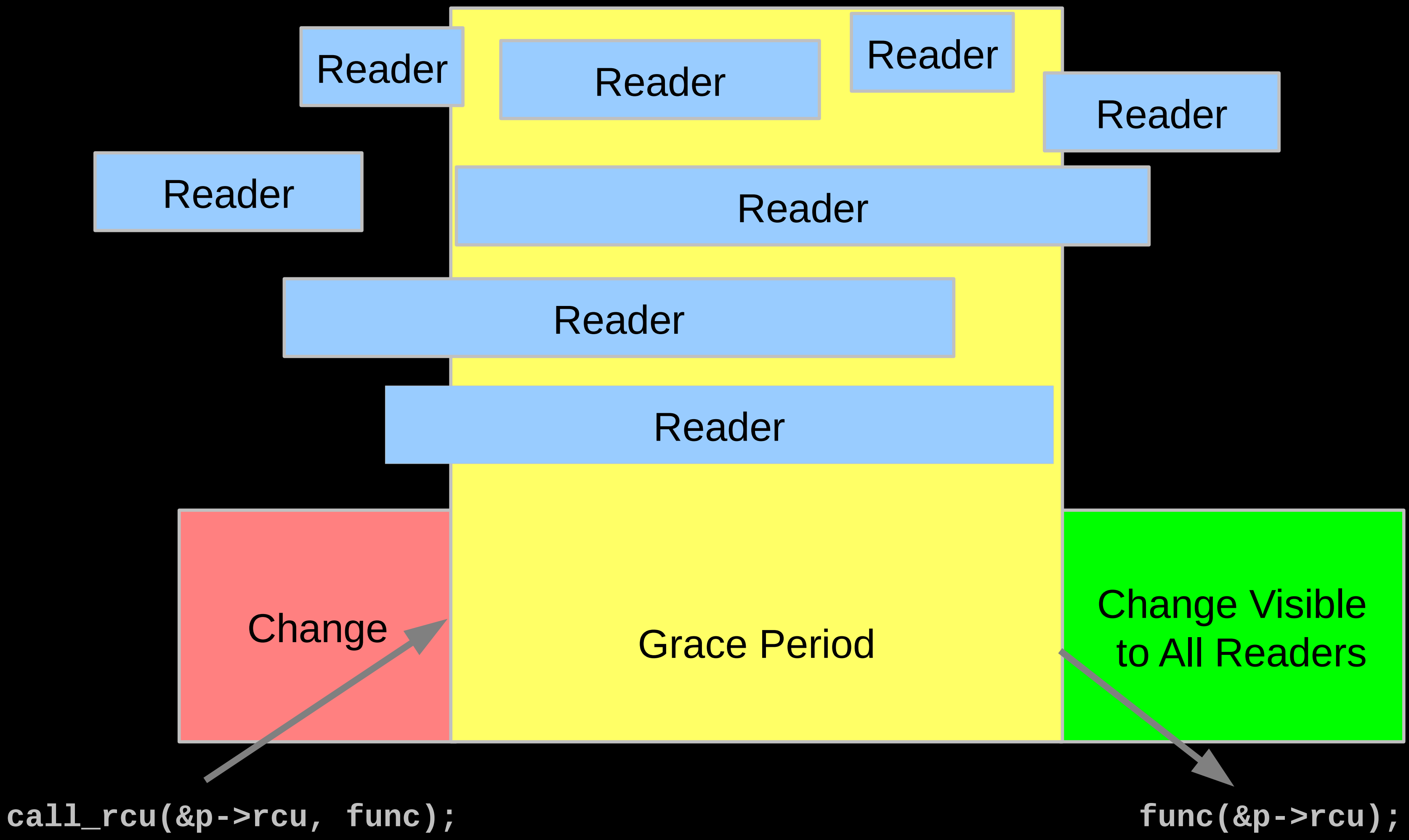
```
call_rcu(struct rcu_head head,
         void (*func)(struct rcu_head *rcu));
```

- The `rcu_head` structure:

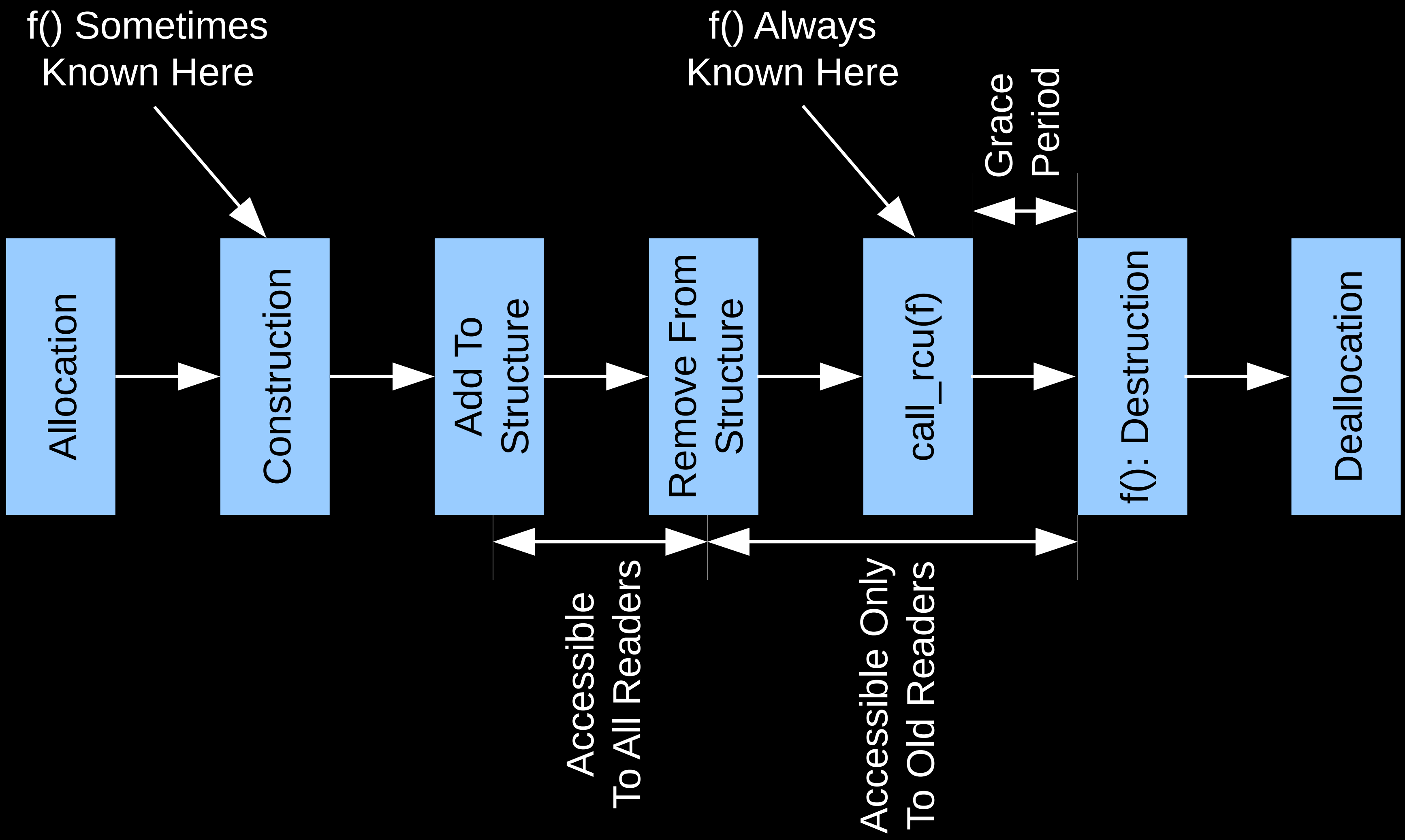
```
struct rcu_head {
        struct rcu_head *next;
        void (*func)(struct rcu_head *rcu);
};
```

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

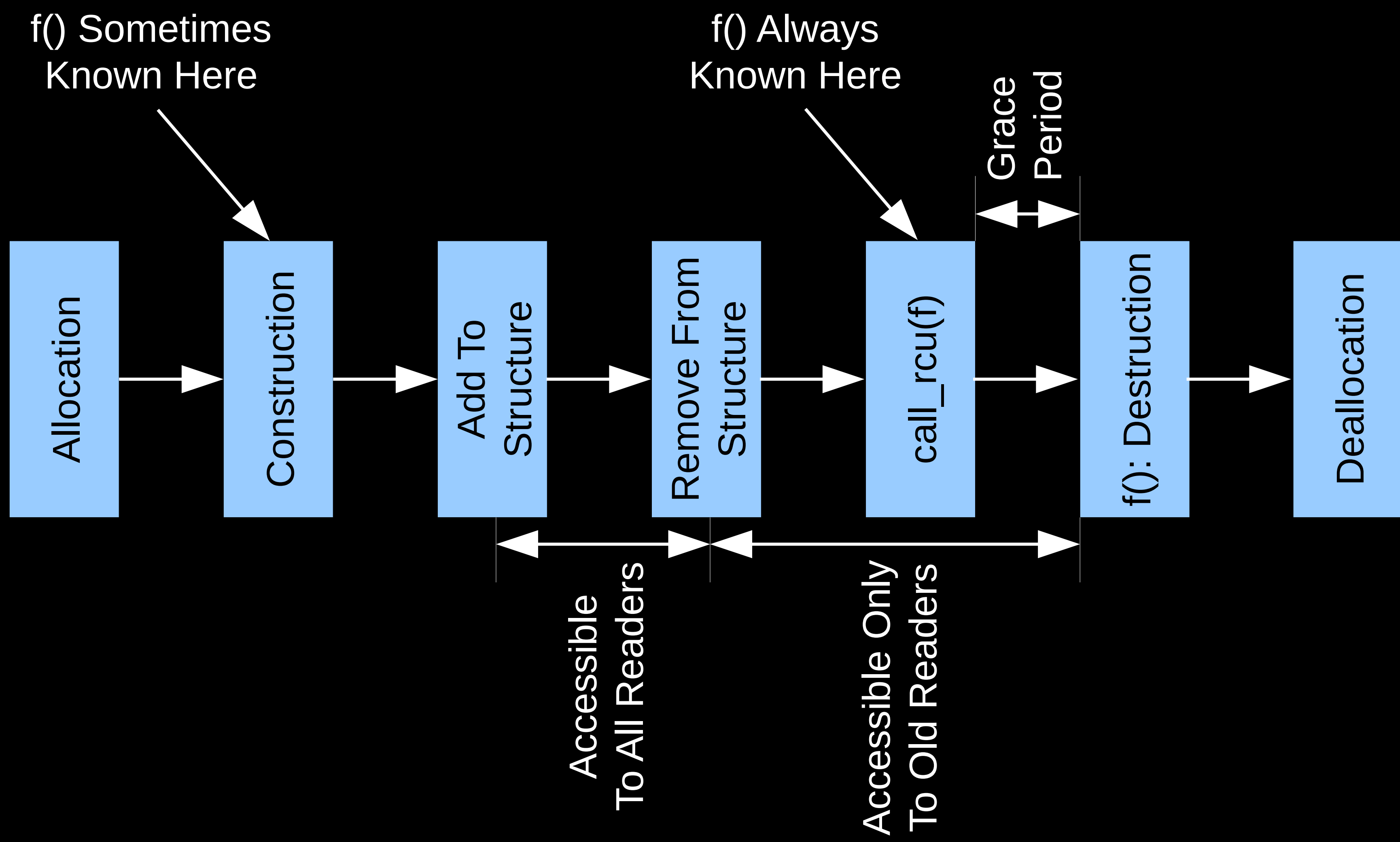
RCU Grace Period: An Asynchronous Graphical View



Destructors Not Necessarily Known At Construction

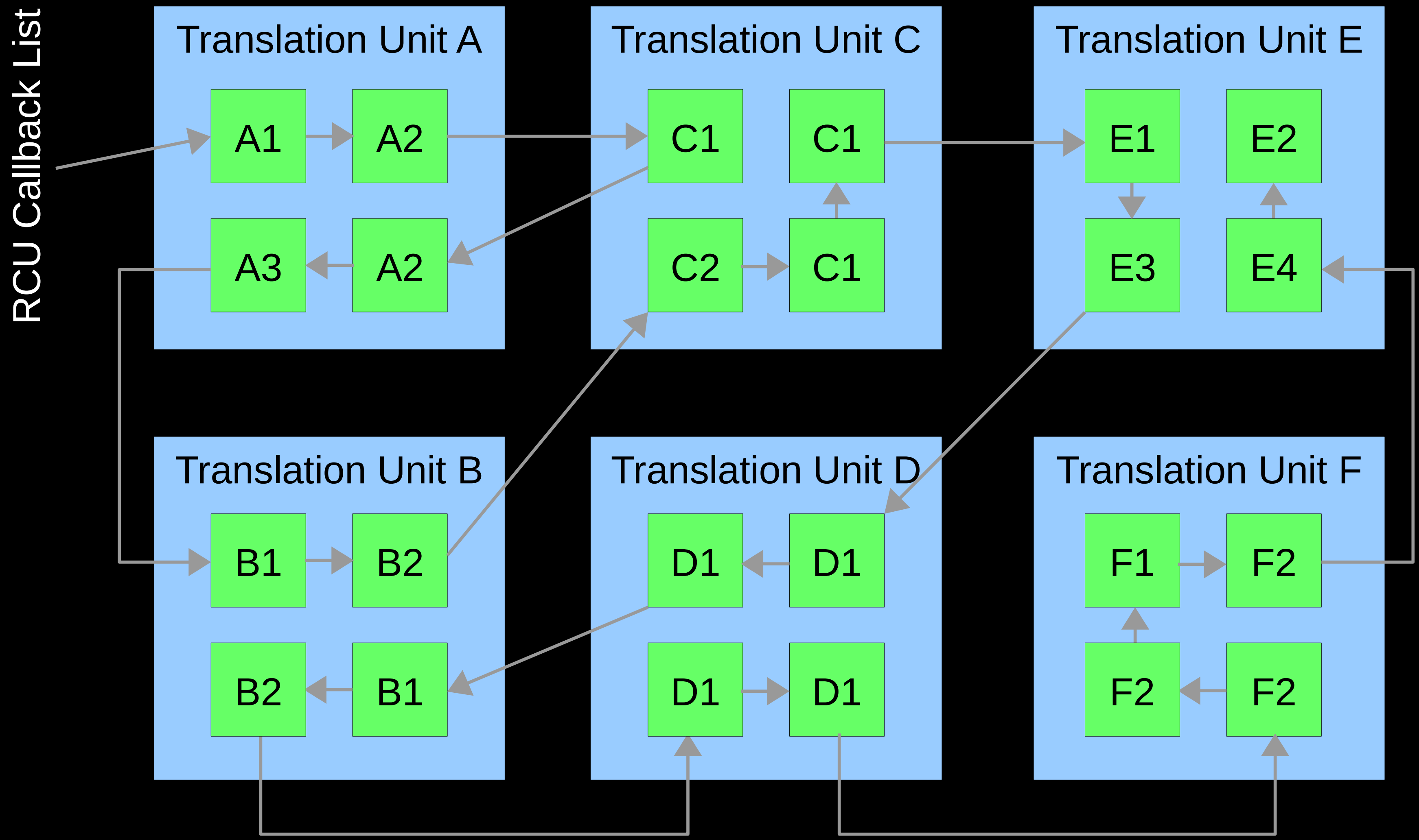


Destructors Not Necessarily Known At Construction

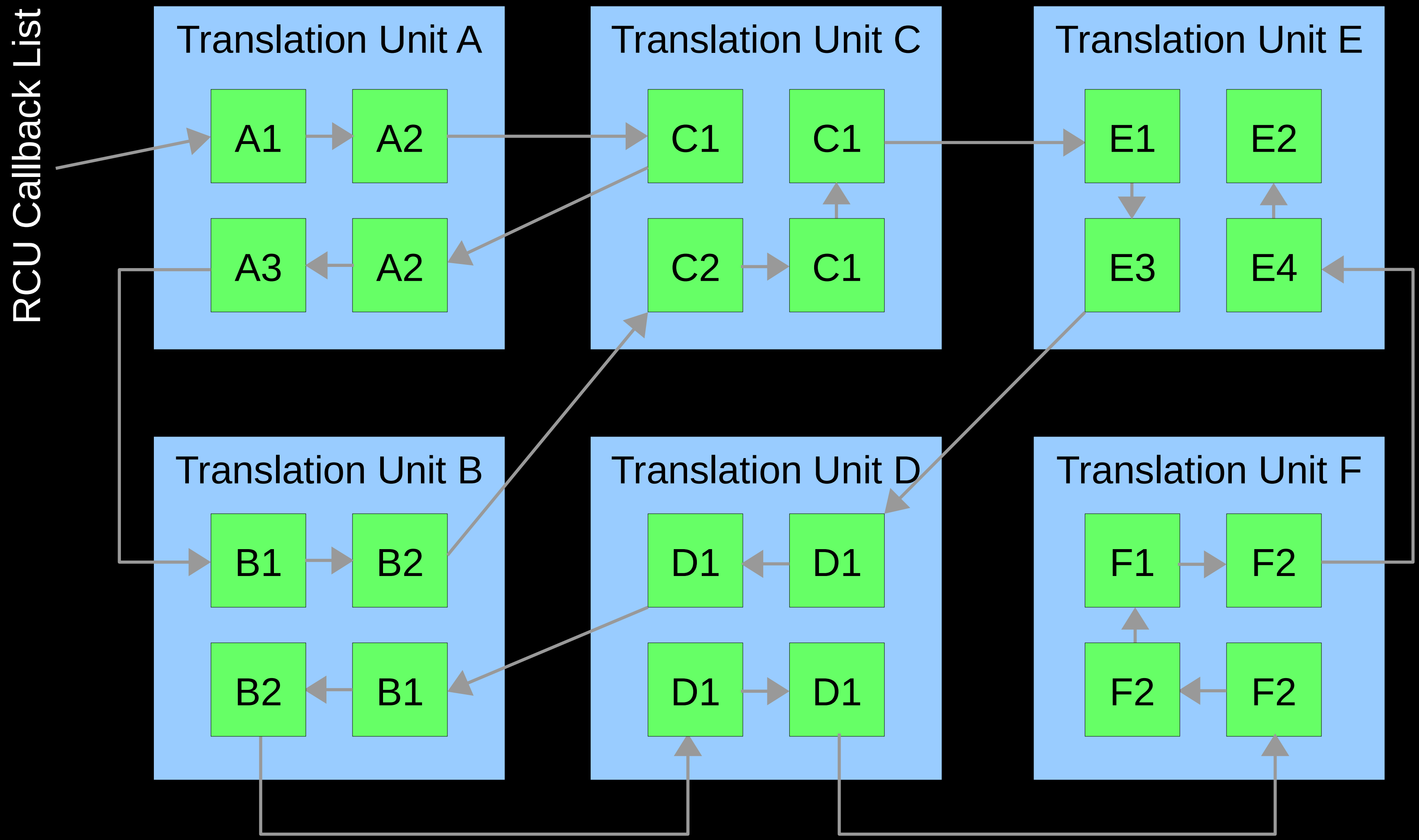


When f() is not known until call_rcu() time, need fixed-width storage!!!

Any Type Any Time Any Translation Unit Anywhere



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

```
void std::rcu_read_lock();
void std::rcu_read_unlock();
void std::synchronize_rcu();
void std::call_rcu(struct std::rcu_head *rhp,
                   void cbf(class rcu_head *rhp));
void std::rcu_barrier();
void std::rcu_register_thread();
void std::rcu_unregister_thread();
void std::rcu_quiescent_state();
void std::rcu_thread_offline();
void std::rcu_thread_online();
```

Defining an RCU Domain

```
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;
};
```

Derived concrete class for each “flavor” in userspace RCU library

RCU Scoped Readers

```
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
That Would Be Mine: The Less Said, The Better!!!

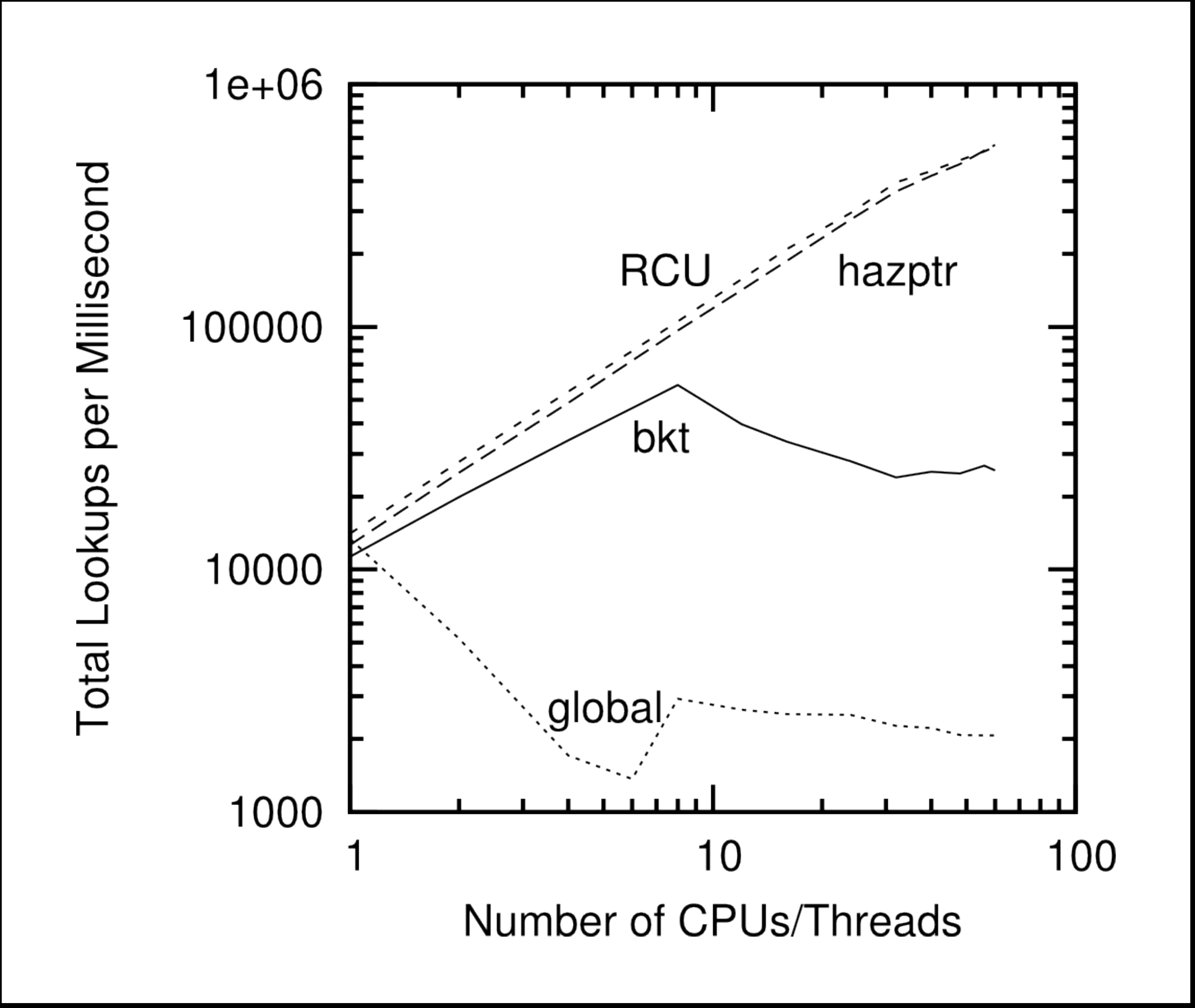
Tracking RCU Callbacks: Approach #1 (Work In Progress)

```
// Isabella Muerte approach
template <class T>
struct default_deleter;

template<class T, class Deleter=default_deleter<T>>
struct rcu_head_delete2: rcu_head, Deleter {

  Deleter& get_deleter () { return *this; }

  void call ();
  void call (rcu_domain& rd);
};
```

Tracking RCU Callbacks: Approach #2 (Work In Progress)

```
// Arthur O'Dwyer approach
template<typename T,
         typename D = default_delete<T>,
         bool E = is_empty<D>::value>
class rcu_head_delete {
public:
  void call(D d = {});
  void call(rcu_domain &rd, D d = {});
};
```

Schrödinger's Zoo: Read-Only



RCU and hazard pointers scale quite well!!!

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...

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