Concurrency TS 2
Use Cases and Future Direction

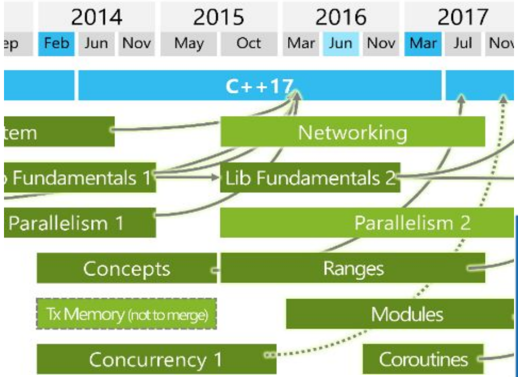
MICHAEL WONG, MAGED MICHAEL
& PAUL MCKENNEY

Cppcon
The C++ Conference
2022
September 12th-16th

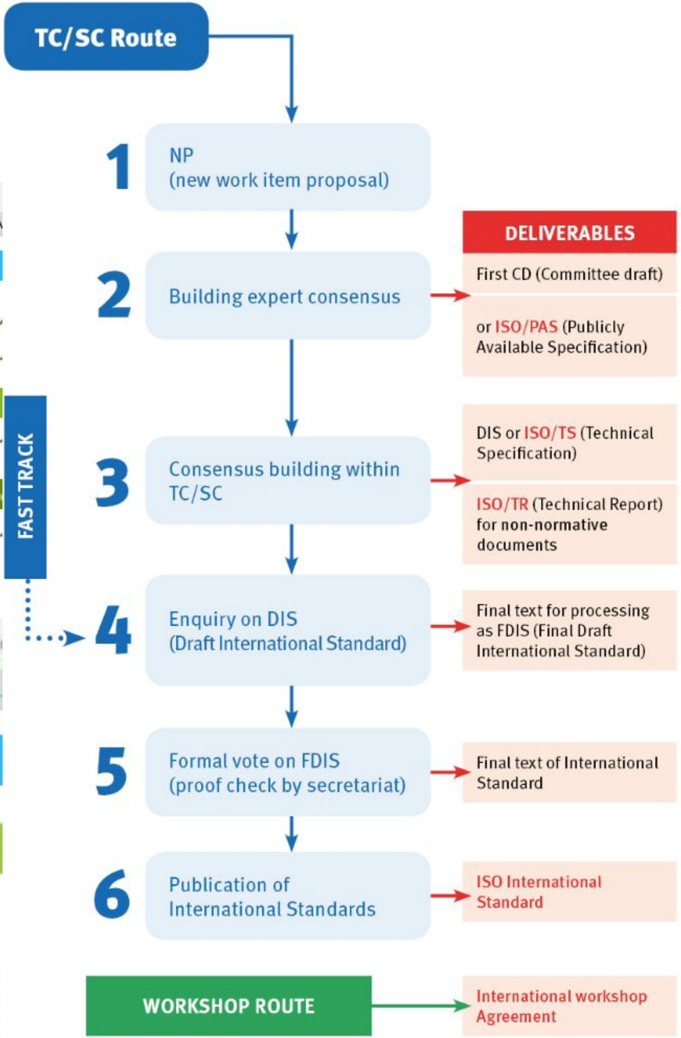
TS2 Tricks and Tips

TS road to C++ Standard

- Concurrency TS1
  - improvements to std::future
  - Latches and barriers
  - Atomic smart pointers



- Concurrency TS2



The cart before the horse?

- TS 2 likely will close in 2023 in N4895
  - 2 initial items - Hazard Pointers, RCU
    - https://github.com/cplusplus/concurrency-ts2
    - HP: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1121r3.pdf
    - RCU: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1122r4.pdf
  - A few others possible: snapshot, asymmetric fences
  - *Then usually a few more years for experience, so could miss C++26*
- But Hazard Pointers and RCU already have a lot of C++ experience, since 2016
  - **ARE WE REBELS?** Why wait?
    - Committee agrees and is pushing it forward even before TS2 is out
  - Aiming for C++26 now
    - SG1 approved for C++26 for HP and RCU, soon LEWG, then LWG
    - HP: https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2022/p2530r0.pdf
    - RCU: https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2022/p2545r0.pdf
  - So what’s changed from TS to IS 26
    - Actually not much

Read Copy Update (RCU) TS2->IS 26

- RCU - no change for C++26 based on Folly experience
  - For after C++26, there will be some ideas for additions

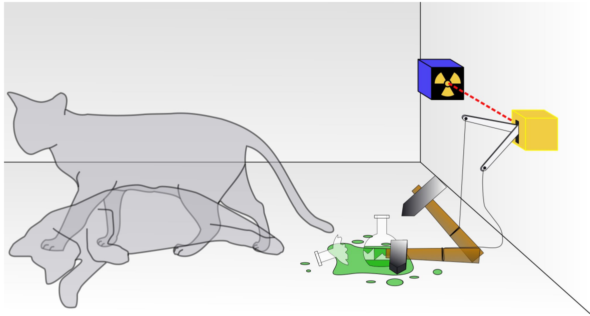
Hazard Pointers (HP) TS2->IS26

- Omits custom domains
- Omits global cleanup
  - enables synchronous reclamation
  - Maged's talk from cppcon 2021

5.1.2 Header <hazard_pointer> synopsis [saferecl.hp.syn]

```
namespace std::experimental::inline concurrency_v2 /* [Omitted] */ {
  // 5.1.3, class hazard_pointer_domain [Omitted]
  class hazard_pointer_domain;

  // 5.1.4, Default hazard_pointer_domain [Omitted]
  hazard_pointer_domain& hazard_pointer_default_domain() noexcept;

  // 5.1.5, Clean up [Omitted]
  void hazard_pointer_clean_up(hazard_pointer_domain& domain = hazard_pointer_default_domain())
    noexcept;
  // 5.1.6, class template hazard_pointer_obj_base
  template <typename T, typename D = default_delete<T>> class hazard_pointer_obj_base;

  // 5.1.7, class hazard_pointer
  class hazard_pointer;

  // 5.1.8, Construct non-empty hazard_pointer
  hazard_pointer make_hazard_pointer(
      hazard_pointer_domain& domain = hazard_pointer_default_domain() // [Omitted]
      );
```

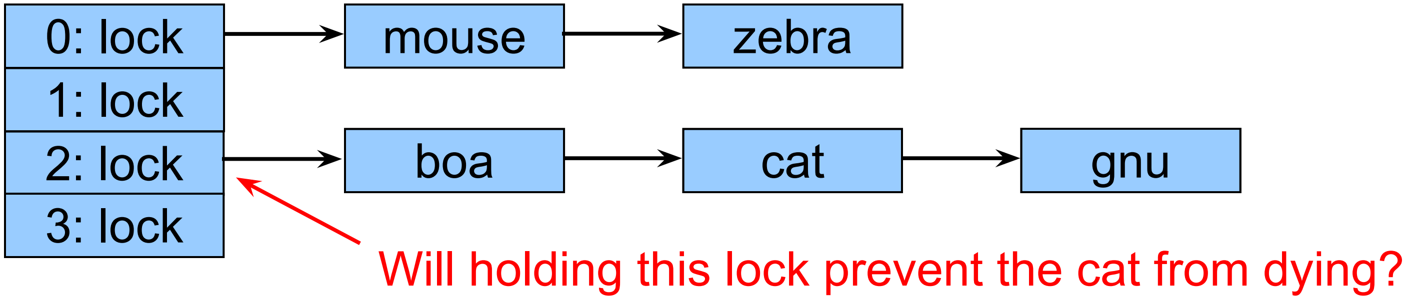
Deferred Reclamation! What is it?



- TS 2 will have several Deferred Reclamation facilities
  - 2 low level APIs: HP and RCU
  - May be 1 high level for deferred reclamation
  - So what is Deferred Reclamation and why is it important
    - It is Heisenberg’s Uncertainty Principle married with Schrödinger’s Cat in Lock-free algo
    - Readers access data while holding reader locks or data is protected
      - Guarantee data will remain live while lock is held or data is protected
    - One or more updaters update data by replacing it with newly allocated data
      - All subsequent readers will see new value
      - Old values is not destroyed until all readers access it have released their locks
      - Here is where you can have 2 views of Schrödinger’s Cat: one alive and one dead
    - Benefits; readers never block the updater or other readers
      - Updaters never block readers
    - What you pay: Updates have extra cost, could be very small
      - They need allocation and new values construction
      - OK if updates are rare

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



Trading Certainty for Performance and Scalability in Life

A Common Problem

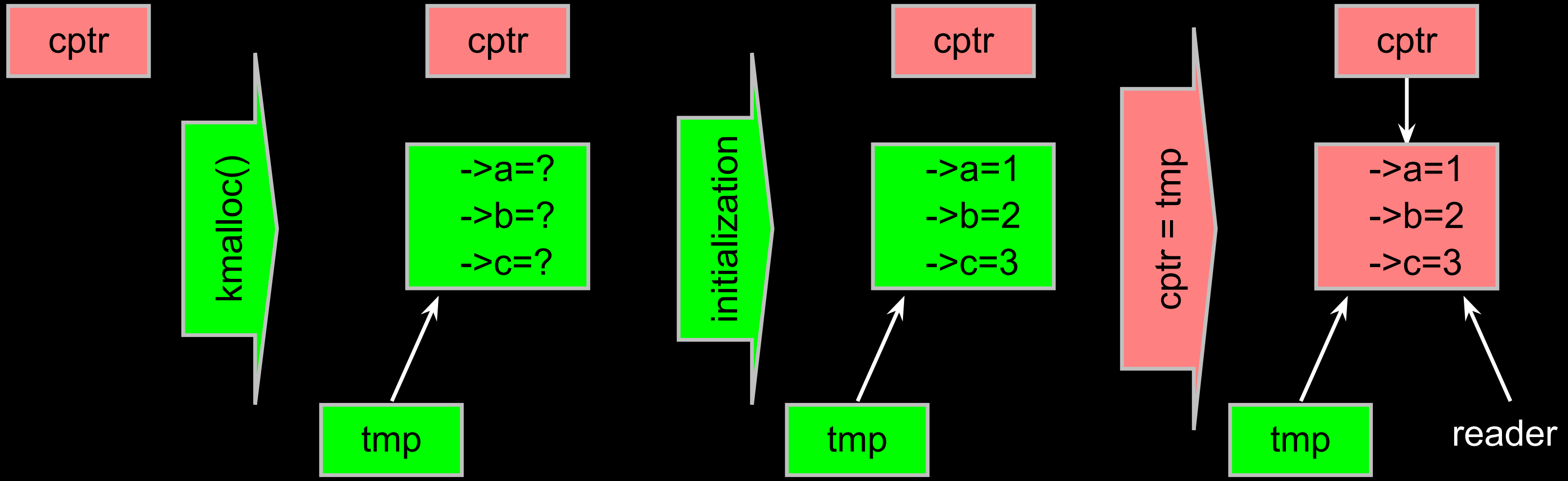
1. Acquire a lock
2. While holding the lock, compute some property of data protected by that lock
3. Release the lock
4. Use the computed property

Several Approaches

1. Reader Writer Lock
2. Reference count
3. RCU
4. Hazard pointers

Publication of And Subscription to New Data

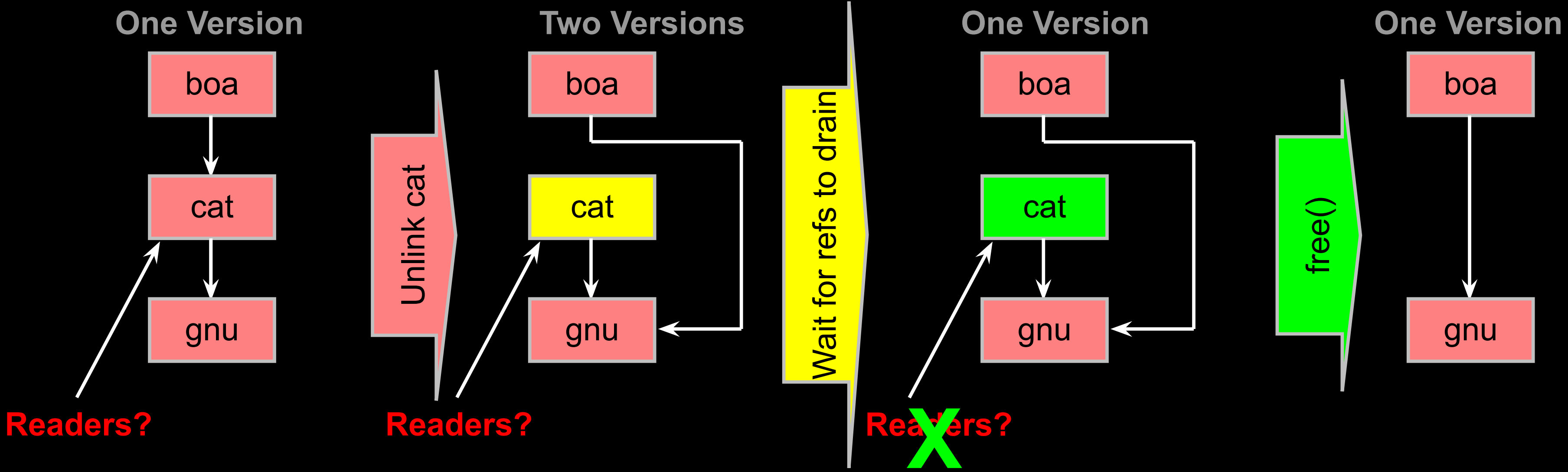
Key:
- Dangerous for updates: all readers can access
- Still dangerous for updates: pre-existing readers can access (next slide)
- Safe for updates: inaccessible to all readers



But if all we do is insert, we have a big memory leak!

Deferred Removal via Reference Counting

- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (Unlink cat)
  - Writer waits for all readers to finish
  - Writer can then free the cat's element



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

Beyond performance, you also need to choose from other properties of lock-free programming	Reader Writer Locks	Reference Counting	RCU	Hazard Pointers
Readers	Slow and unscalable	Slow and unscalable	**Fast and scalable**	**Fast and Scalable**
Unreclaimed objects	**None**	**None**	Unbounded	**Bounded**
Traversal speed	No or low overhead	Atomic RMW updates	**No or low overhead**	**Low overhead**
Reference acquisition	**Unconditional**	Depends on use case	**Unconditional**	Conditional
Contention among readers	Can be very high	Can be very high	**No contention**	**No contention**
Automatic reclamation	No	**Yes**	No	No
Reclamation timing	**Immediate**	**Immediate**	Deferred	Deferred
Non-blocking traversal *	Blocking	Either blocking or lock free with limited reclamation	**Bounded population oblivious wait free**	**Lock free.**
Non-blocking reclamation (no memory allocator) *	Blocking	Either blocking or lock free with limited reclamation	Blocking	**Bounded wait free**

* Typically of theoretical interest

What else could be in TS2?

- Could be more, but we are likely to close it in 2023, which limits it
  - A high level interface for deferred reclamation: SNAPSHOT P0561
  - Asymmetric fences P1202

SNAPSHOT: An RAII interface for Deferred Reclamation

```
 class Server {

public:

   void SetConfig(Config new_config) {
config_.update(std::make_unique<const
Config>(std::move(new_config)));    }

   void HandleRequest() {

     snapshot_ptr<const Config> config =
config_.get_snapshot();

     // Use `config` like a unique_ptr<const
Config>    }

private:

   snapshot_source<Config> config_;

 };
```

```
template <typename T, typename Alloc = allocator<T>>   class raw_snapshot_source {

public:

  // Not copyable or movable

  raw_snapshot_source(raw_snapshot_source&&) = delete;

  raw_snapshot_source& operator=(raw_snapshot_source&&) = delete;

  raw_snapshot_source(const raw_snapshot_source&) = delete;

  raw_snapshot_source& operator=(const raw_snapshot_source&) = delete;

  raw_snapshot_source(nullptr_t = nullptr, const Alloc& alloc = Alloc());

  raw_snapshot_source(std::unique_ptr<T> ptr, const Alloc& alloc = Alloc());

  void update(nullptr_t);

  void update(unique_ptr<T> ptr);

  bool try_update(const snapshot_ptr<T>& expected, std::unique_ptr<T>&& desired);

  snapshot_ptr<T> get_snapshot() const;

};

template <typename T>   using snapshot_source = raw_snapshot_source<see below>;
```

Asymmetric Fences

```
namespace std::experimental::inline concurrency_v2 { // ?.2.1
  asymmetric_thread_fence_heavy void
asymmetric_thread_fence_heavy(memory_order order) noexcept; // ?.2.2
  asymmetric_thread_fence_light void
asymmetric_thread_fence_light(memory_order order) noexcept;
}
```

How to use TS2 (or IS26) safely

Deferred reclamation can be applied readily to most concurrent linked data structures

- HP
  - Not hard to convert ref count to HP
  - No blocking concerns as Reclamation objects are bounded
  - because we removing the cleanup in the IS26, your code should be aware of any dependency on destructors
- RCU
  - Reader might block reclamation if unbounded, so an unbounded amount of memory might remain unclaimed
  - But in safety critical, memory is bounded by the maximum duration of RCU read-side critical section X max amount of memory retired per unit of time
  - In safety if you use static allocation then you will not have new injections and this is actually good as it will not block reclamation
  - If you recycle a fixed number of statically allocated blocks, then blocking in an RCU reader is less damaging to updates than blocking in a reader-writer-locking reader.
  - An RCU reader typically only blocks recycling of memory, allowing updates to proceed concurrently with RCU readers.
  - In contrast, a reader-writer-locking reader blocks updates entirely.
- Coroutines:
  - Similar to things like std::mutex, RCU readers should not span a coroutine suspension point (unless special non-standard extensions or use cases are applied).
  - Similar to reference counting, hazard pointers can be held across coroutine suspension points, and further can be passed from one thread to another.
- Both hazard pointers and RCU can have debugging issues due to thread switching

Hazard-Pointer Tricks and Tips

Hazard Pointers in a Nutshell

Protect access to objects that may be concurrently removed.
A hazard pointer is a single-writer multi-reader pointer.
If a hazard pointer points to an object
before its removal,
then the object will not be reclaimed
as long as the hazard pointer remains unchanged

<u>Protect object A</u>
Set a hazard pointer to point to A
if A is not removed
then it is safe to use A

<u>Remove and reclaim object A</u>
Remove A
if no hazard pointers point to A
then it is safe to reclaim A

Features:

- Fast and scalable protection
- Supports arbitrarily long protection

Concurrency TS2 Hazard Pointers Interface

Custom Domains

```
class hazard_pointer_domain {
public:
  hazard_pointer_domain() noexcept;
  explicit hazard_pointer_domain(
      pmr::polymorphic_allocator<byte> poly_alloc) noexcept;
  hazard_pointer_domain(const hazard_pointer_domain&) = delete;
  hazard_pointer_domain& operator=(const hazard_pointer_domain&) = delete;
  ~hazard_pointer_domain();
};

hazard_pointer_domain& hazard_pointer_default_domain() noexcept;
```

Global Cleanup

```
// For synchronous reclamation
void hazard_pointer_clean_up(
    hazard_pointer_domain& domain = hazard_pointer_default_domain()) noexcept;
```

Protectable Objects

```
template <typename T, typename D = default_delete<T>>
class hazard_pointer_obj_base {
public:
  void retire(
    D d = D(),
    hazard_pointer_domain& domain = hazard_pointer_default_domain()) noexcept;
  void retire(hazard_pointer_domain& domain) noexcept;
};
```

Hazard Pointers

```
class hazard_pointer {
public:
  hazard_pointer() noexcept; // Empty
  hazard_pointer(hazard_pointer&&) noexcept;
  hazard_pointer& operator=(hazard_pointer&&) noexcept;
  ~hazard_pointer();
  [[nodiscard]] bool empty() const noexcept;
  template <typename T> T* protect(const atomic<T*>& src) noexcept;
  template <typename T> bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
  template <typename T> void reset_protection(const T* ptr) noexcept;
  void reset_protection(nullptr_t = nullptr) noexcept;
  void swap(hazard_pointer&) noexcept;
};

hazard_pointer make_hazard_pointer(
    hazard_pointer_domain& domain = hazard_pointer_default_domain());

void swap(hazard_pointer&, hazard_pointer&) noexcept;
```

Hazard Pointers TS2 Interface Essential Subset

```
template <typename T> class hazard_pointer_obj_base {
  void retire() noexcept; // Object must be already removed
};

class hazard_pointer {
  hazard_pointer() noexcept; // Construct an empty hazard pointer
  hazard_pointer(hazard_pointer&&) noexcept;
  hazard_pointer& operator=(hazard_pointer&&) noexcept;
  ~hazard_pointer();
  template <typename T> bool try_protect(T*& ptr, const atomic<T*>& src)
noexcept;
  template <typename T> T* protect(const atomic<T*>& src) noexcept;
  template <typename T> void reset_protection(const T* ptr) noexcept;
};
hazard_pointer make_hazard_pointer(); // Construct a non-empty hazard pointer
void swap(hazard_pointer&, hazard_pointer&) noexcept;
```

Hazard Pointers TS2 Interface Essential Subset

`hazard_pointer_obj_base` : base type of objects protectable by hazard pointers

- **`retire`** : removed object is to be reclaimed when no longer protected

`hazard_pointer` : hazard pointer object, may be empty, a nonempty hazard pointer object owns a hazard pointer

- **`hazard_pointer()`** : constructs an empty hazard pointer object
- **`operator=(hazard_pointer&&)`** : moves hazard pointer objects,
  ends moved to and continues moved from protection if any,
  moved from becomes empty
- **`~hazard_pointer()`** : destroys the hazard pointer object, ends protection by the owned hazard pointer if any
- **`try_protect(ptr, src)`** : protects **`ptr`** only if **`src`** equals **`ptr`**
- **`protect(src)`** : protects a pointer from **`src`**
- **`reset_protection(ptr)`** : ends current protection if any, starts protecting **`ptr`** if not null and not removed

`make_hazard_pointer` : constructs a nonempty hazard pointer object

`swap` : swaps two hazard pointer objects

3 Use Case Examples of Hazard Pointers TS2 Interface

1. **Protecting arbitrarily-long access**

2. **Hand-over-hand traversal**

3. **Iteration**

(1) Protecting Arbitrarily-Long Access

Protecting Arbitrarily-Long Access

```
class Foo : public hazard_pointer_obj_base<Foo> { /* Foo members */ };

void access(const std::atomic<Foo*>& src, Func fn) { // Called frequently
  hazard_pointer h = make_hazard_pointer(); // Construct a non-empty
  Foo* ptr = h.protect(src);  // ptr is now protected
  fn(ptr); // fn is also allowed to block and/or take long time
  // End of scope destroys h and ends the protection of ptr
}

void update(std::atomic<Foo*>& src, Foo* newptr) { // Called infrequently
  Foo* oldptr = src.exchange(newptr);  // oldptr is now removed
  oldptr->retire(); // oldptr will be reclaimed only when unprotected
}
```

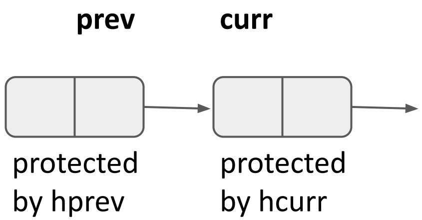
(2) Hand-over-Hand Traversal

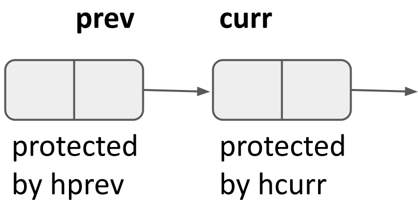
Concurrent Linked List Example 1/2

```
class Node : public hazard_pointer_obj_base<Node>
  { T value_;  atomic<Node*> next_; /* etc */ };

atomic<Node*> head_; // Pointer to the head of the linked list

// Single (or synchronized) writer
void remove(Node* prev, Node* target) {
  prev->next_.store(target->next_.load());
  target->next_.store(nullptr);
  target->retire();// target will be reclaimed only when unprotected
}
```

Concurrent Linked List Example 2/2

```
// May be called by multiple concurrent readers
bool find(const T& val) {
  hazard_pointer hprev = make_hazard_pointer();
  hazard_pointer hcurr = make_hazard_pointer();
  while (true) {
    atomic<Node*>* prev = &head_;
    atomic<Node*> curr = prev->load();
    while (true) {
      if (!curr) return false; // not found
      if (!hcurr->try_protect(curr, *prev)) break;
      auto next = curr->next_.load();
      if (prev->load() != curr) break; // start over
      if (curr->value_ == val) return true; // found
      swap(hcurr, hprev); // hprev protects curr the next prev
      prev = &(curr->next_); // advance prev
      curr = next; // advance curr
    }
  }
}
```

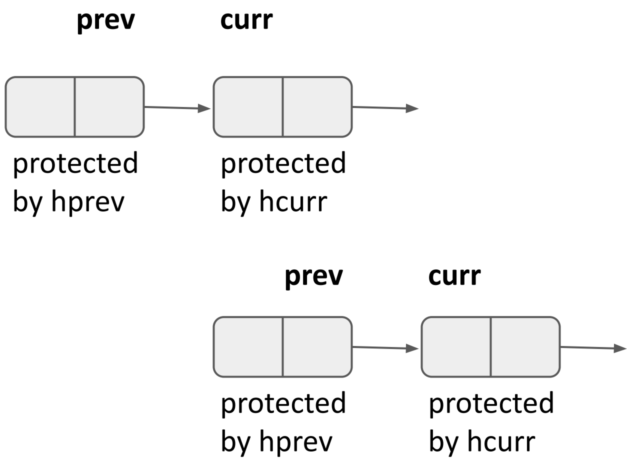




Example of Incorrect Protection

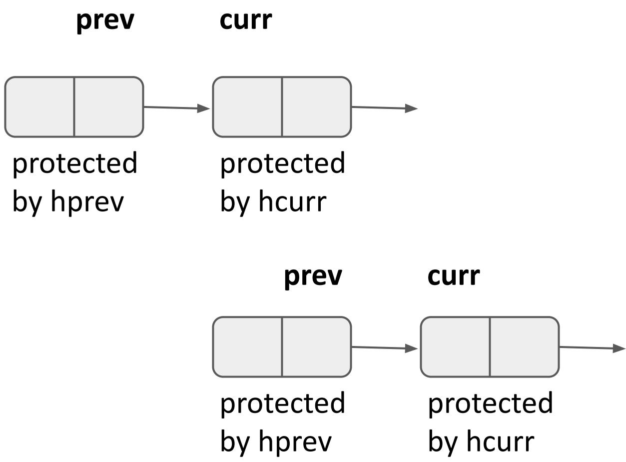
```
// May be called by multiple concurrent readers
bool find(const T& val) {
  hazard_pointer hprev = make_hazard_pointer();
  hazard_pointer hcurr = make_hazard_pointer();
  while (true) {
    atomic<Node*>* prev = &head_;
    atomic<Node*> curr = prev->load();
    while (true) {
      if (!curr) return false; // not found
      if (!hcurr->try_protect(curr, *prev)) break;
      auto next = curr->next_.load();
      if (prev->load() != curr) break; // start over
      if (curr->value_ == val) return true; // found
      swap(hcurr, hprev); hprev.reset_protection(curr);
      prev = &(curr->next_); // advance prev
      curr = next; // advance curr
    }
  }
}
```

INCORRECT: `curr` may be already retired
Can't start protecting a retired object



Example of Incorrect Handling of Hazard Pointer Objects

```
// May be called by multiple concurrent readers
bool find(const T& val) {
  hazard_pointer hprev = make_hazard_pointer();
  hazard_pointer hcurr = make_hazard_pointer();
  while (true) {
    atomic<Node*>* prev = &head_;
    atomic<Node*> curr = prev->load();
    while (true) {
      if (!curr) return false; // not found
      if (!hcurr->try_protect(curr, *prev)) break;
      auto next = curr->next_.load();
      if (prev->load() != curr) break; // start over
      if (curr->value_ == val) return true; // found
      swap(hcurr, hprev); hprev = hcurr; // move
      prev = &(curr->next_); // advance prev
      curr = next; // advance curr
    }
  }
}
```



INCORRECT: `hcurr` becomes empty after move
Can't use an empty hazard pointer object for protection

(3) Iteration

Hash Table Iterator Example 1/4

```
class Node : public hazard_pointer_obj_base<Node> {
  K key_; atomic<Node*> next_; atomic<int> linkcount_; /* etc */
  void acquire_link() { ++linkcount_; }
  void release_link() { if (--linkcount_ == 0) this->retire(); }
  ~Node() {
    // releases link to successor, retire it if its link count is down to zero
    Node* next = curr->next_.load(); if (node) node->release_link();
  }
};
class Bucket { atomic<Node*> head_; /* etc */ };
Bucket buckets_[NUM_BUCKETS];

// Synchronized writer
void removeNode(Node* prev, Node* target) {
  Node* next = curr->next_.load();
  next->acquire_link(); // acquire extra link to next
  prev->next_.store(next); // both prev and curr point to next
  curr->release_link(); // retire curr if unlinked
}
```

Hash Table Iterator Example 2/4

```
class Iterator {
  hazard_pointer hp_[2]; int idx_{0};  Node* node_{nullptr}; /* etc */
  // movable only

  void firstNode() {
    hp_[0] = make_hazard_pointer();
    hp_[1] = make_hazard_pointer();
    nextNode();
  }

  void nextNode() {
    while (!node_) {
      if (idx_ >= NUM_BUCKETS) break;
      node_ = hp_[0].protect(buckets_[idx_].head_);
      if (node_) break;
      ++idx_;
    }
  }
```

Hash Table Iterator Example 3/4

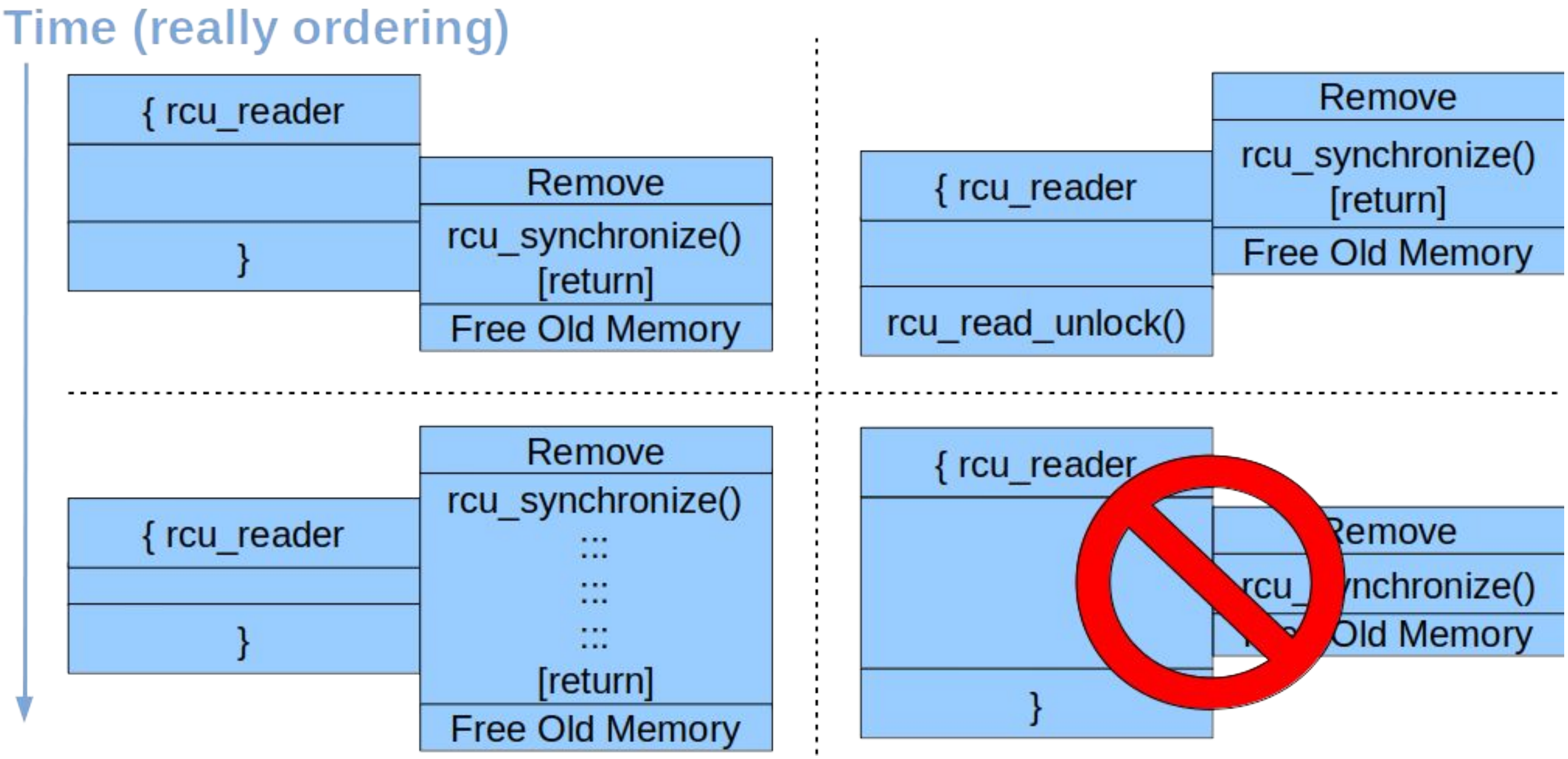
```
  const Iterator& operator++() {
    node_ = hp_[1].protect(node_->next_);
    hp_[0].swap(hp_[1]);
    if (!node_) {
      ++idx_;
      nextNode();
    }
    return *this;
  }
}; // Iterator

Iterator begin() { Iterator it; it.firstNode(); return it; }

Iterator end() { return Iterator(); }
```

Hash Table Iterator Example 4/4

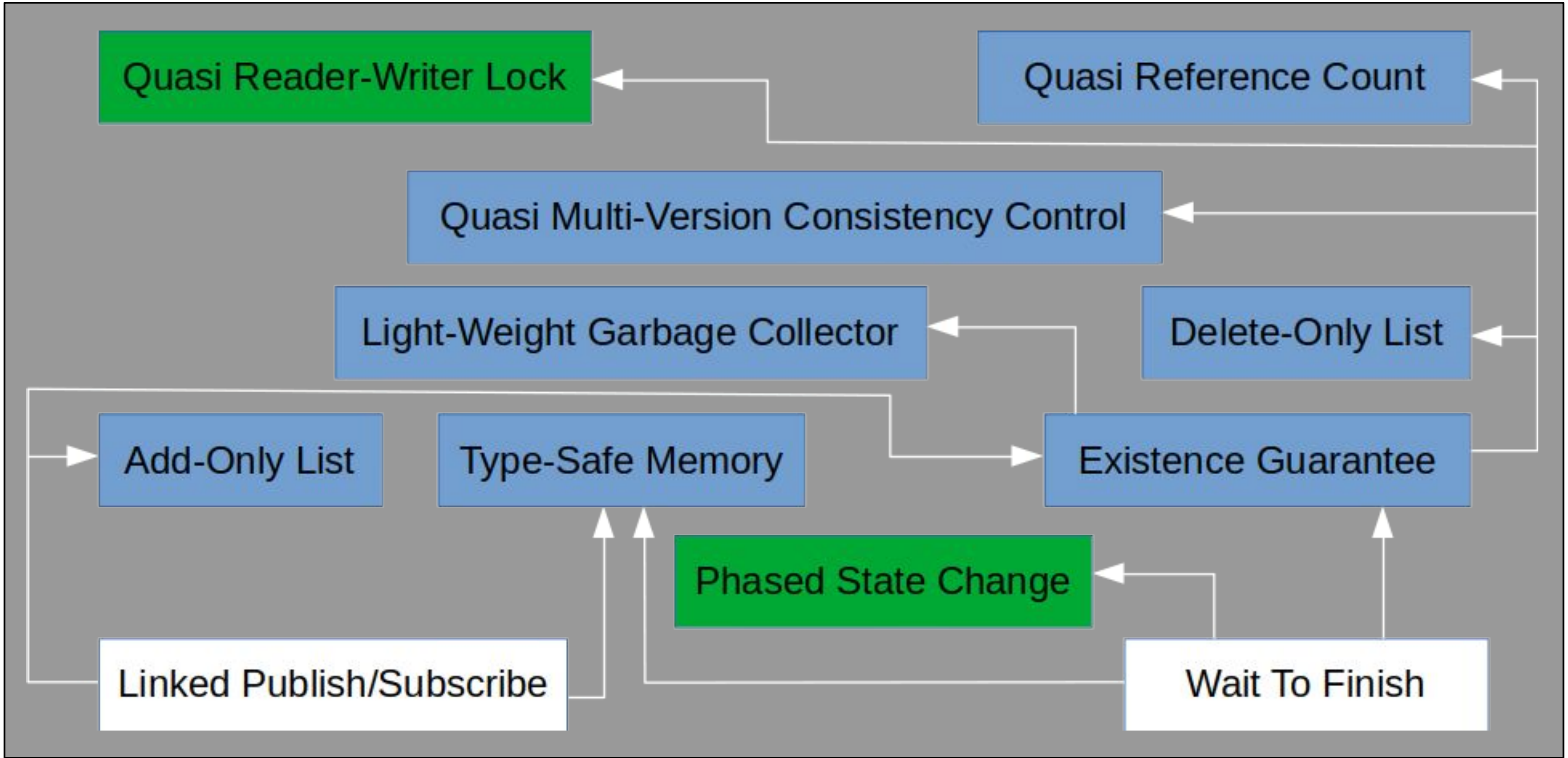
```
// User code

// Iteration can be concurrent with hashtable updates without interference
// Multiple concurrent iterations do not interfere with each other
// Protection duration is allowed to be arbitrarily long

for (Iterator it = ht.begin(); it != ht.end(); ++it)
  userOp(it);
```

A Stupid RCU Trick

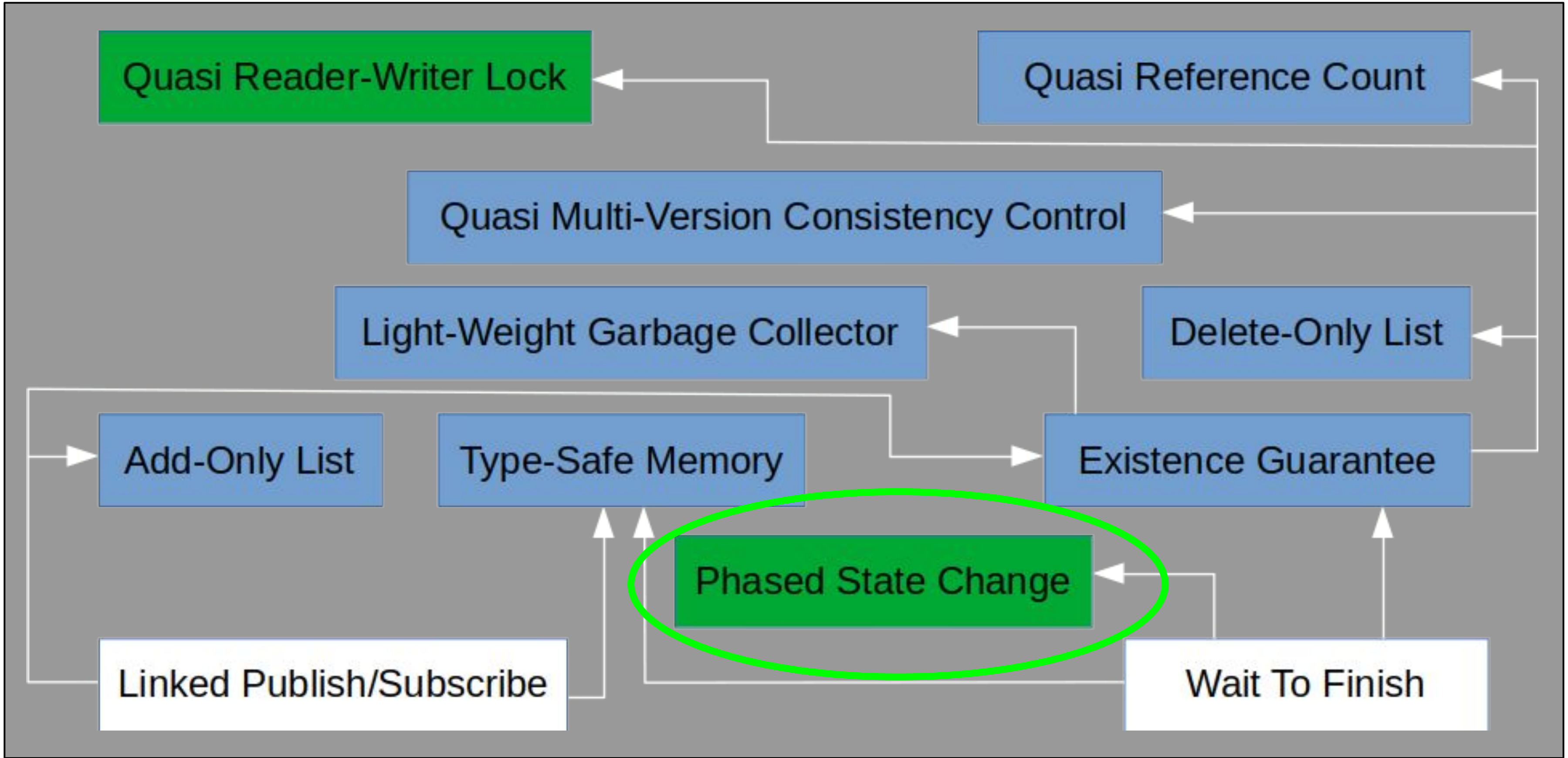
Graphical Introduction to RCU



One Trick of Many



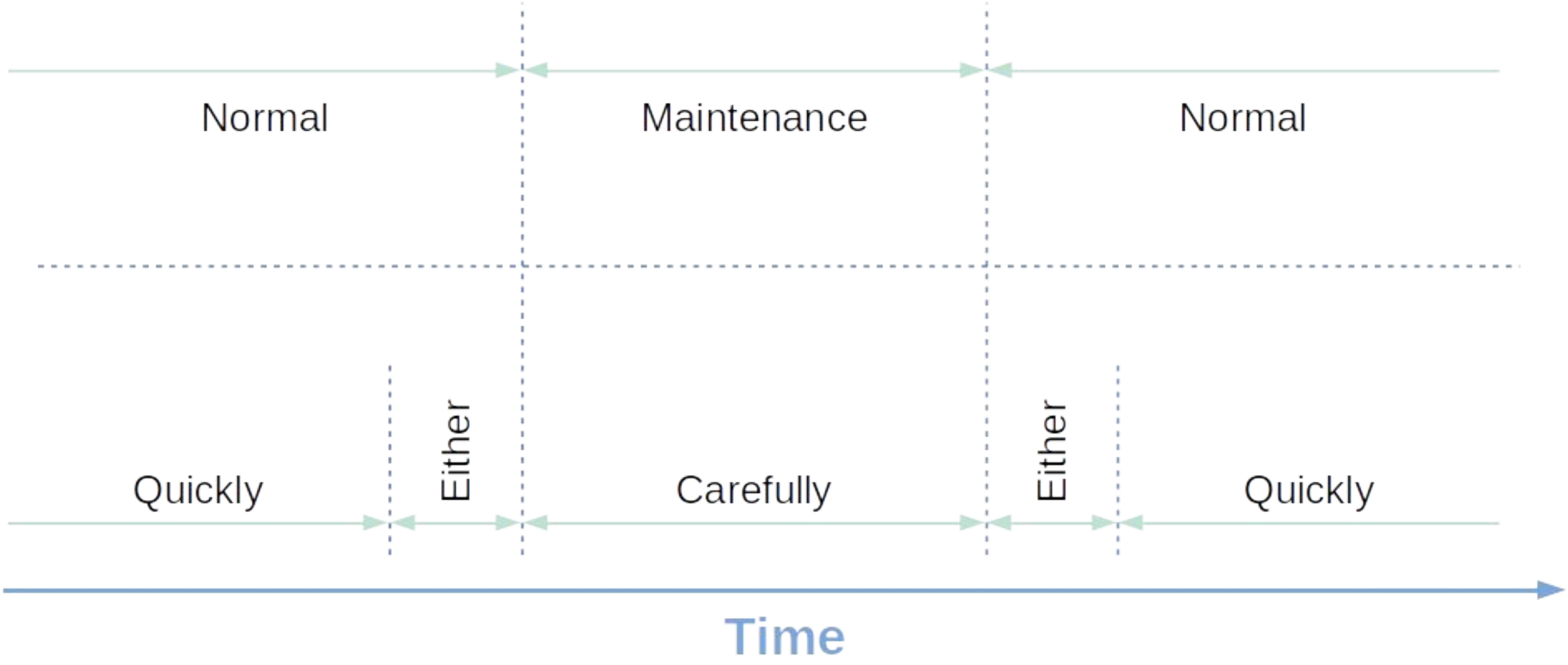
One Trick of Many: Phased State Change Today



RCU-Mediated Phased State Change

- The lowest-level and most primitive known RCU use case:
- Multithreaded application
    - Common-case operation must be fast
    - But care is required during maintenance
- Use flag to indicate that care is required
    - But how to reliably synchronize?
    - OK to be careful just before/after maintenance

RCU-Mediated Phased State Change (Graphical)



Common-Case Operation

```
atomic<Bool> be_careful;

void cco()
{
    std::scoped_lock l(std::rcu_default_domain());
    if (be_careful.load(memory_order_relaxed))
        cco_carefully();
    else
        cco_quickly();
} // RAII end of RCU reader
```

Maintenance Operation

```
void maint()
{
    be_careful.store(true, memory_order_relaxed);
    rcu_synchronize();
    do_maint();
    rcu_synchronize(); // Why is this needed?
    be_careful.store(false, memory_order_relaxed);
}
```

Problematic Maintenance Operation

```
void maint()
{
    be_careful.store(true, memory_order_relaxed);
    rcu_synchronize();
    do_maint();
    // rcu_synchronize();
    be_careful.store(false, memory_order_relaxed);
    // Because the above store can be reordered into
    // the call to do_maint(), which can in turn permit
    // a concurrent cco_quickly() access, which is BAD!!!
}
```

Alternative Maintenance Operation

```
void maint()
{
    be_careful.store(true, memory_order_relaxed);
    rcu_synchronize();
    do_maint();
    // No second rcu_synchronize()...
    be_careful.store(false, memory_order_release);
    // ...But this requires the change to cco() shown on
    // the next slide…
}
```

Alternative Common-Case Operation

```
atomic<Bool> be_careful;

void cco()
{
    std::scoped_lock l(std::rcu_default_domain());
    if (be_careful.load(memory_order_acquire))
        cco_carefully();
    else
        cco_quickly();
} // RAII end of RCU reader
```

Summary

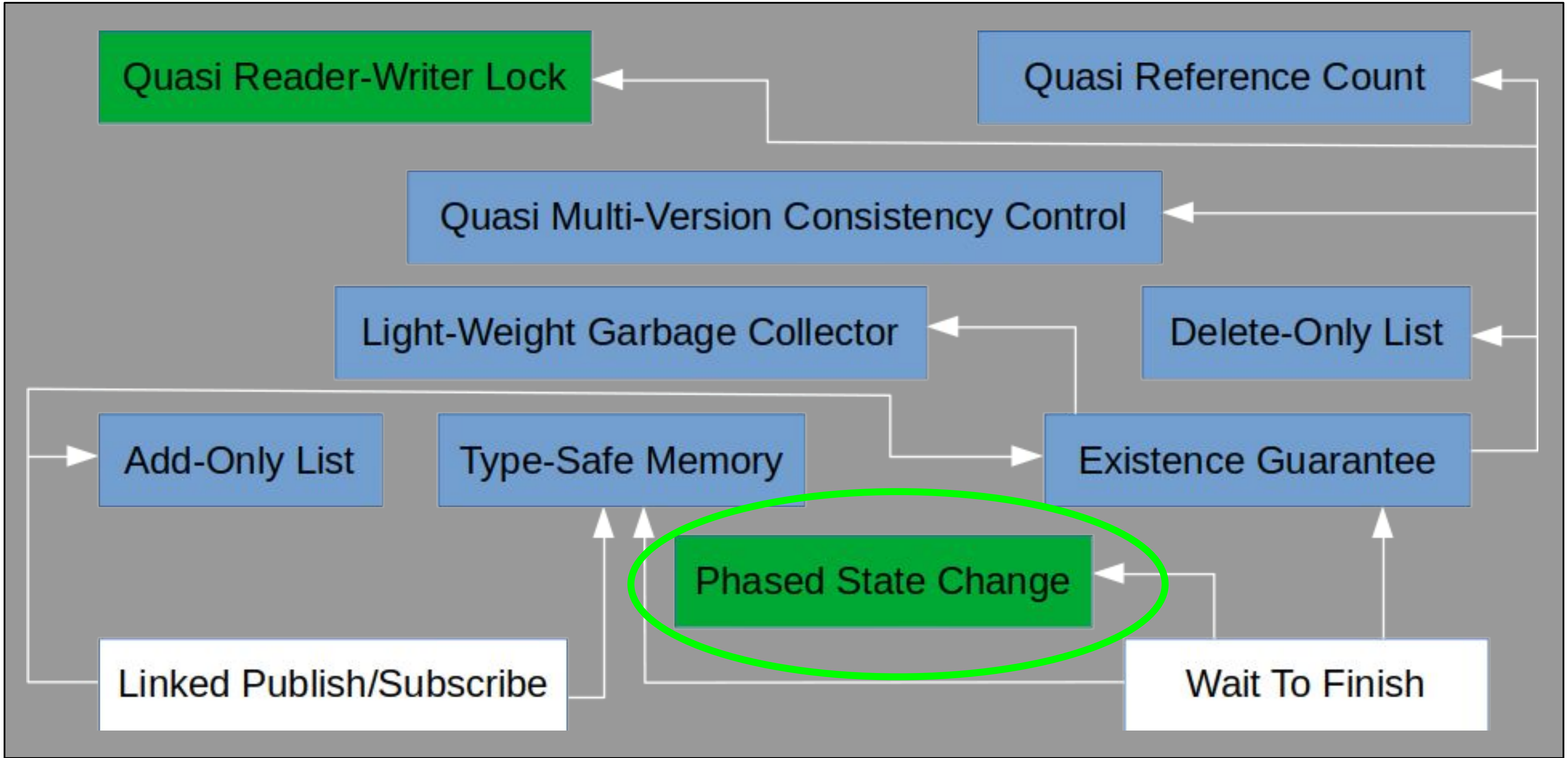
RCU is able to mediate a phased state change

Almost zero common-case read-side synchronization overhead

Addition of read-side acquire load removes update-side rcu_synchronize()

This pattern is used in the Linux kernel

One Trick of Many



Want More Stupid RCU Tricks?

1. Linux Foundation Mentorship Program Presentations:
    a. Unraveling RCU-Usage Mysteries (Fundamentals)
        i. Includes introductory overview of RCU
    b. Unraveling RCU-Usage Mysteries (Additional Use Cases)
2. Stupid RCU Tricks blog series
3. Is Parallel Programming Hard, And, If So, What Can You Do About It?
    a. Section 9.5.4 (“RCU Usage”)
    b. Chapter 13 (“Putting It All Together”)