How to Make Your Data Structures Wait-Free for Reads

CppCon 2015

Pedro Ramalhete

September 2015

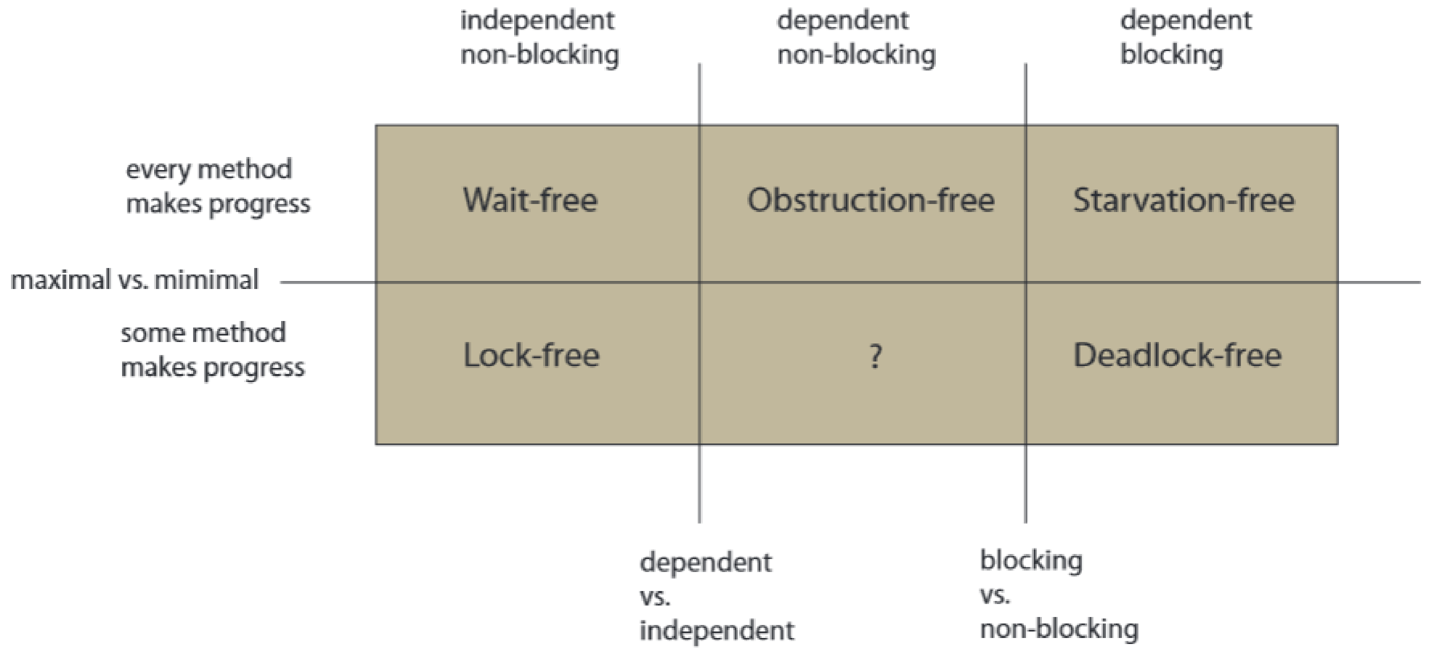
Content

Progress Conditions

Types of Progress Conditions

- Blocking
  - Deadlock-free
  - Starvation-free
- Obstruction-Free
- Lock-Free
- Wait-Free
  - Wait-Free Bounded
  - Wait-Free Population Oblivious

- Maurice Herlihy and Nir Shavit



On the Nature of Progress, Maurice Herlihy and Nir Shavit

Progress Conditions Definitions

Blocking: an unexpected delay by one thread can prevent others from making progress

Non-Blocking:

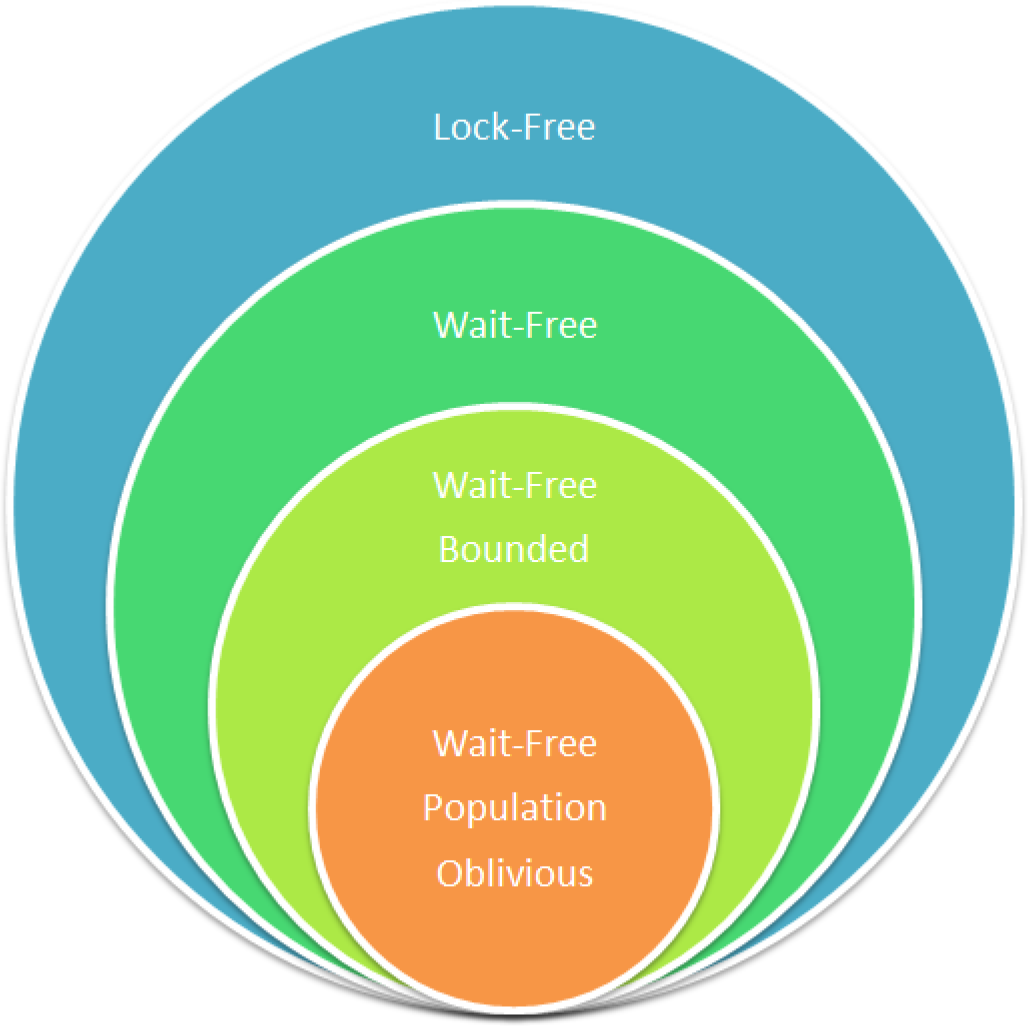
Obstruction-Free: A method is Obstruction-Free if, from any point after which it executes in isolation, it finishes in a finite number of steps.

Lock-Free: A method is Lock-Free if it guarantees that infinitely often some thread calling this method finishes in a finite number of steps.

Wait-Free: A method is Wait-Free if it guarantees that every call finishes its execution in a finite number of steps.

Wait-Free Bounded: A method is Wait-Free Bounded if it guarantees that every call finishes its execution in a finite and bounded number of steps. This bound may depend on the number of threads.

Wait-Free Population Oblivious: A Wait-Free method whose performance does not depend on the number of active threads is called Wait-Free Population Oblivious.



http://concurrencyfreaks.com/2013/05/lock-free-and-wait-free-definition-and.html

Progress Conditions and Time

- Notice that Progress Conditions are given in terms of *number of steps* and **not** in terms of *time*.
- This can be misleading due to one of the types of progress conditions being called *wait-free*, which can be incorrectly interpreted as if there is no waiting (time delay) associated with that particular method, but this guarantee can never be given on a multithreaded system, unless it is somehow provided by the thread scheduler itself.
- It could happen that a wait-free method takes an undetermined amount of time to complete if the thread scheduler decides not to give the calling thread an execution slice (for whatever reason/priority).

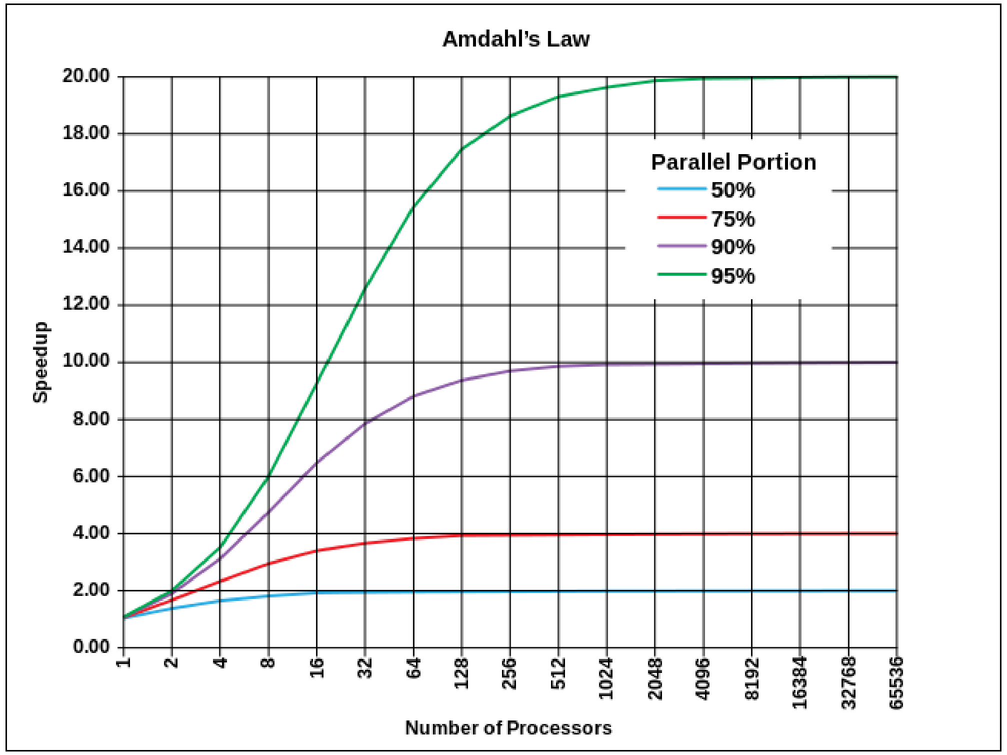
- Even so, different types of progress conditions have different implications on the Latency distribution, but this would be a topic for an entire presentation on itself.

Progress Conditions $\neq$

Progress Conditions
Blocking

Just because a method is Blocking, it doesn't mean it doesn't scale. Suppose you have the following method. How much more work could be done if we called it from 10 threads instead of a single thread?

```
void someMethod(auto& data) {
    auto result = aVeryComplexComputation(data); // Takes 10 µs to complete
    // The next two lines take 2 µs to complete
    std::lock_guard<std::mutex> lock(amutex);
    globalResult += result;
}
```



Amdahl's Law Wikipedia - https://en.wikipedia.org/wiki/Amdahl's_law

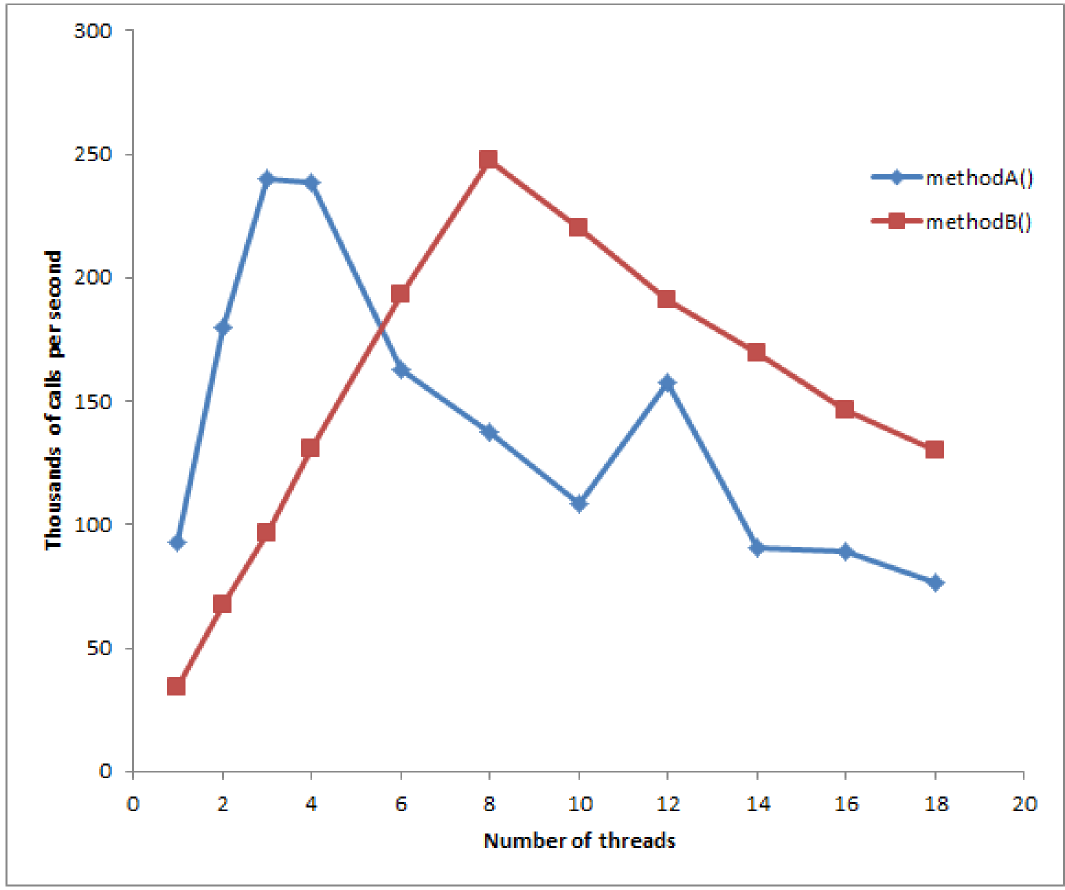
Quiz 1
Which one scales with more threads?

Suppose we have the two methods shown below. Assuming we have plenty of cores, which method scales to a larger number of threads ?

a) `methodA()`

b) `methodB()`

```
void methodA(auto& data) {
    auto result = aNotSoComplexComputation(data); // Takes ~10 µs
    std::lock_guard<std::mutex> lock(amutex);      // Takes ~2 µs
    globalResult += result;
}

void methodB(auto& data) {
    auto result = aVeryComplexComputation(data);  // Takes ~20 µs
    std::lock_guard<std::mutex> lock(amutex);      // Takes ~2 µs
    globalResult += result;
}
```



Progress Conditions
Starvation-Free

- When talking about locks, the property of **Freedom from Starvation** means that **every** thread that attempts to acquire a lock eventually succeeds. Every call to `lock()` eventually returns. This property is sometimes called *Lockout Freedom*.
- This is a highly desirable property for blocking methods because it has influence in their fairness and latency.
- A SpinLock is an example of a lock that is **not** starvation-free

```
#include <atomic>

class SpinLock {
    int UNLOCKED = 0;
    int LOCKED = 1;
    std::atomic<int> mlock = { 0 };

public:
    void lock() {
        int tmp = UNLOCKED;
        while (!mlock.compare_exchange_strong(tmp, LOCKED)) {
            std::this_thread::yield();
            tmp = UNLOCKED;
        }
    }

    void unlock() {
        mlock.store(UNLOCKED, std::memory_order_release);
    }
};
```

The Art of Multiprocessor Programming, Maurice Herlihy and Nir Shavit

Quiz 2
What is the Progress Condition of `lock()` ?

On x86, what is the progress condition of the method `lock()` ?

a) Blocking

b) Blocking but Starvation-Free

c) Non-Blocking

```
#include <atomic>

class TicketLock {
    std::atomic<long long> ticket = { 0 };
    std::atomic<long long> grant  = { 0 };

public:
    void lock() {
        long long lticket = ticket.fetch_add(1);
        while (lticket != grant.load()) {
            std::this_thread::yield();
        }
    }

    void unlock() {
        long long lgrant = grant.load(std::memory_order_relaxed);
        grant.store(lgrant+1, std::memory_order_release);
    }
};
```

Progress Conditions
Starvation-Free

- An example of a Lock that is starvation free is the Ticket Lock on x86, because the function `std::atomic::fetch_add()` is implemented as a single instruction (XADD) and not a (CAS) loop.

- A rule of thumb is that locking mechanisms that use at its core wait-free techniques are usually starvation-free (Ticket Lock, CLH Lock, Tidex Lock), while locking mechanisms that use lock-free techniques are usually non starvation-free (CAS Spinlock).

```
#include <atomic>

class TicketLock {
    std::atomic<long long> ticket = { 0 };
    std::atomic<long long> grant  = { 0 };

    void lock() {
        long long lticket = ticket.fetch_add(1);
        while (lticket != grant.atomic_load()) {
            std::this_thread::yield();
        }
    }

    void unlock() {
        long long lgrant = grant.load(std::memory_order_relaxed);
        grant.store(lgrant+1, std::memory_order_release);
    }
};
```

Progress Conditions
Lock-Free

- According to “The Art of Multiprocessor Programming”, the definition of Lock-free is: *A method is Lock-Free if it guarantees that infinitely often some thread calling this method finishes in a finite number of steps*.

- A simpler way to judge if a method is lock-free or not is: If we can show that at least one thread invoking a method of the Instance finishes in a finite number of steps, then this method has (at least) a lock-free progress condition.

- By definition, **a Lock-Free method is never free from starvation**, i.e. *Lock-Freeness* implies the possibility of starvation.

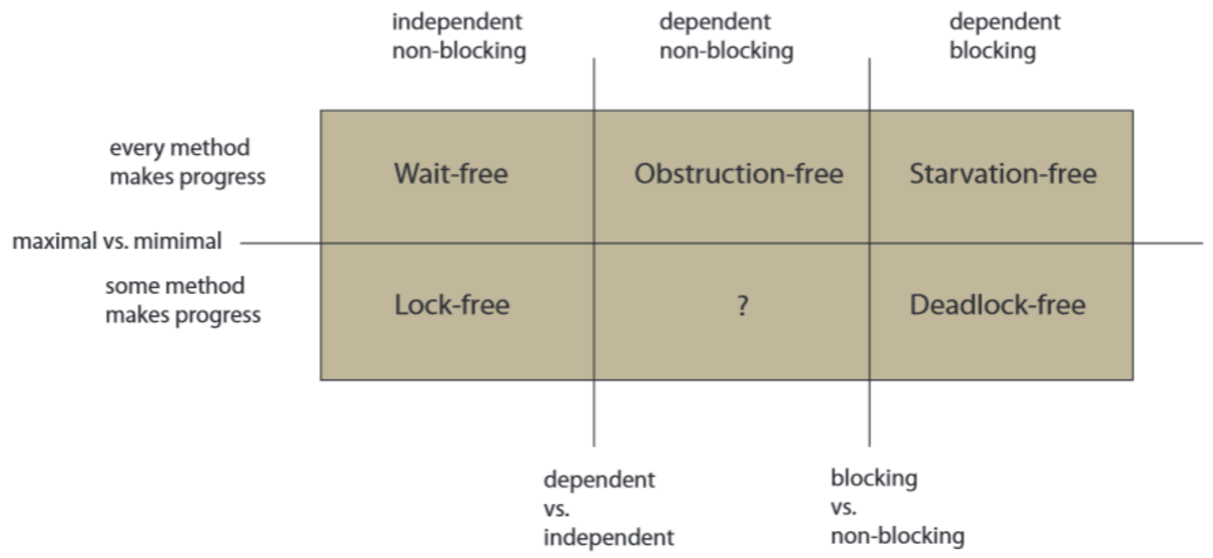
- An example of a Lock-Free method is the insertion of a node in a Lock-Free queue using Maged Michael & Michael Scott’s algorithm (seen on the right-side), in “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue algorithms” – PODC 1996

```
bool add(T *item) {
    Node *newNode = new Node(item);
    while (true) {
        Node *t = tail.load();
        Node *q = t->next.load();
        if (q == nullptr) {
            // It seems this is the last node, add the
            // newNode and try to move the tail to it
            if (t->casNext(nullptr, newNode)) {
                casTail(t, newNode); // Failure is OK
                return true;
            }
        } else {
            casTail(t, q);
        }
    }
}
```

Progress Conditions
Wait-Free

- A method is Wait-Free if it guarantees that every call finishes its execution in a finite number of steps.
- It has **not** been proven that a method that is Wait-Free always gives better performance than a Lock-Free method
- A thread calling a Wait-Free method will always make some progress, if given CPU time.
- Unlike Lock-Free, a Wait-Free method will never *starve*.

- Maurice Herlihy and Nir Shavit



Progress Conditions
Wait-Free – Myth busting

- Unlike what you may infer from the name, a Wait-Free method is NOT *free from waiting*.

- How long a wait-free method takes to complete, still depends on (how many and) how large is the time slice the thread scheduler gives the thread currently calling the wait-free method. If the slice is not enough to complete de method, then there may be a context switch which increases the absolute time it takes for the method to complete.

- If the thread scheduler is unfair, the thread may be preempted indefinitely, and thus make no progress, regardless of it attempting to execute a method which is wait-free

Progress Conditions
Wait-Free Population Oblivious (WFPO)

- A Wait-Free method whose performance does not depend on the number of active threads is called *Wait-Free Population Oblivious* (WFPO).
- Compared to other progress conditions, WFPO provides a stronger guarantee that the completion time is bounded.

- Examples of WFPO operations:

```
std::atomic<int>::load()
std::atomic<int>::store()
std::atomic<int>::exchange()                // WFPO for x86
std::atomic<int>::fetch_add()               // WFPO for x86
std::atomic<int>::compare_exchange_strong() // WFPO for x86, assuming no retries
```

Quiz 3
What are the progress conditions of each method?

Different methods of the same class (or data structure) can have different progress conditions.

What are the progress conditions of each of the three methods of the class `Cookie` ?

a) Lock-Free

b) Wait-Free Unbounded

c) Wait-Free Bounded

d) Wait-Free Population Oblivious

- `getCookieType()` is Wait-Free Population Oblivious
- `getTotalChocolateChips()` is Wait-Free Bounded
- `addChocolateChips()` is Lock-Free

```
class Cookie {
    std::atomic<enum cookie_type> type;
    std::atomic<int> chocolateChips[MAX_THREADS];

    cookie_type getCookieType() {
        return type.load();
    }

    int getTotalChocolateChips() {
        int sum = 0;
        for (int i = 0; i < MAX_THREADS; i++) {
            sum += chocolateChips[i].load();
        }
    }

    void addChocolateChips(int numChips) {
        while (true) {
            int tmp = chocolateChips[0].load();
            if (chocolateChips[0].compare_exchange_strong(tmp, tmp+1)) {
                return;
            }
        }
    }
};
```

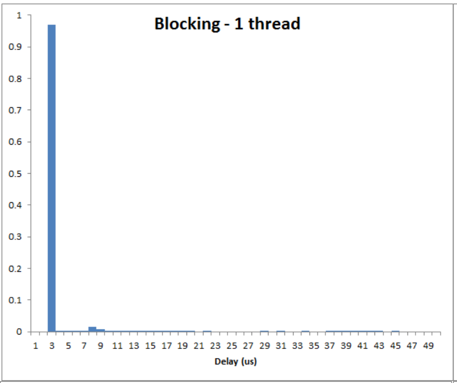
Progress Conditions

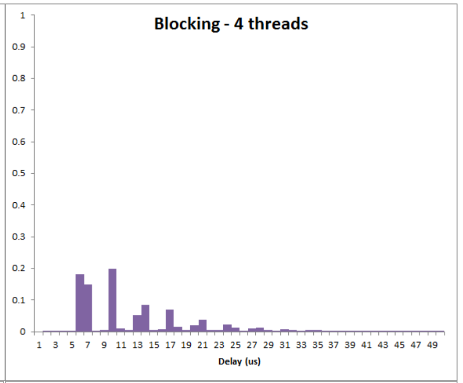
Latency Distribution

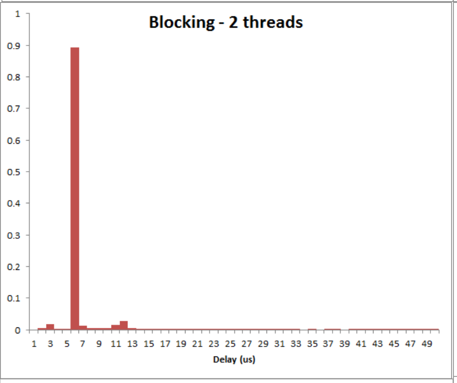
Progress Conditions
Latency

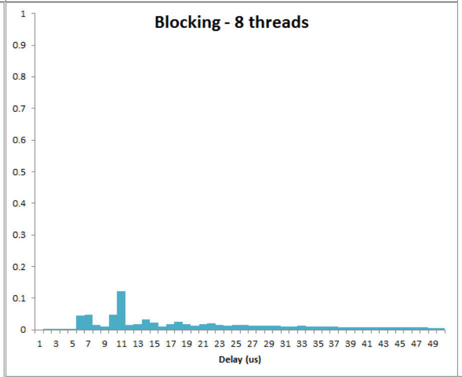
- Using algorithms with different progress conditions, we analyzed the latency distribution of each and how it evolves as the number of threads increase.
  - Blocking: a spinlock with a CAS loop
  - Starvation-Free: a Ticket Lock
  - Lock-Free: A CAS loop that takes some time to complete (length of operation has 20% jitter)
  - Wait-Free Bounded: Each thread increments all entries in an array of size NUM_THREADS (bounded by the number of threads)
  - Wait-Free Population Oblivious: Each thread increments just its own cache-line-padded entry in an array
- The plots on the next slides are just a toy experiment, so take them with a grain of salt. It's just to get an idea of the kind of effects that we can expect of each progress condition when under high contention.

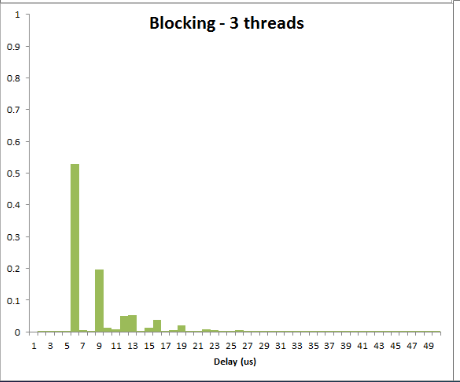
Latency
Blocking

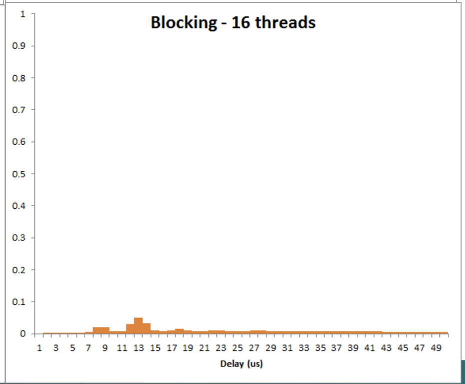


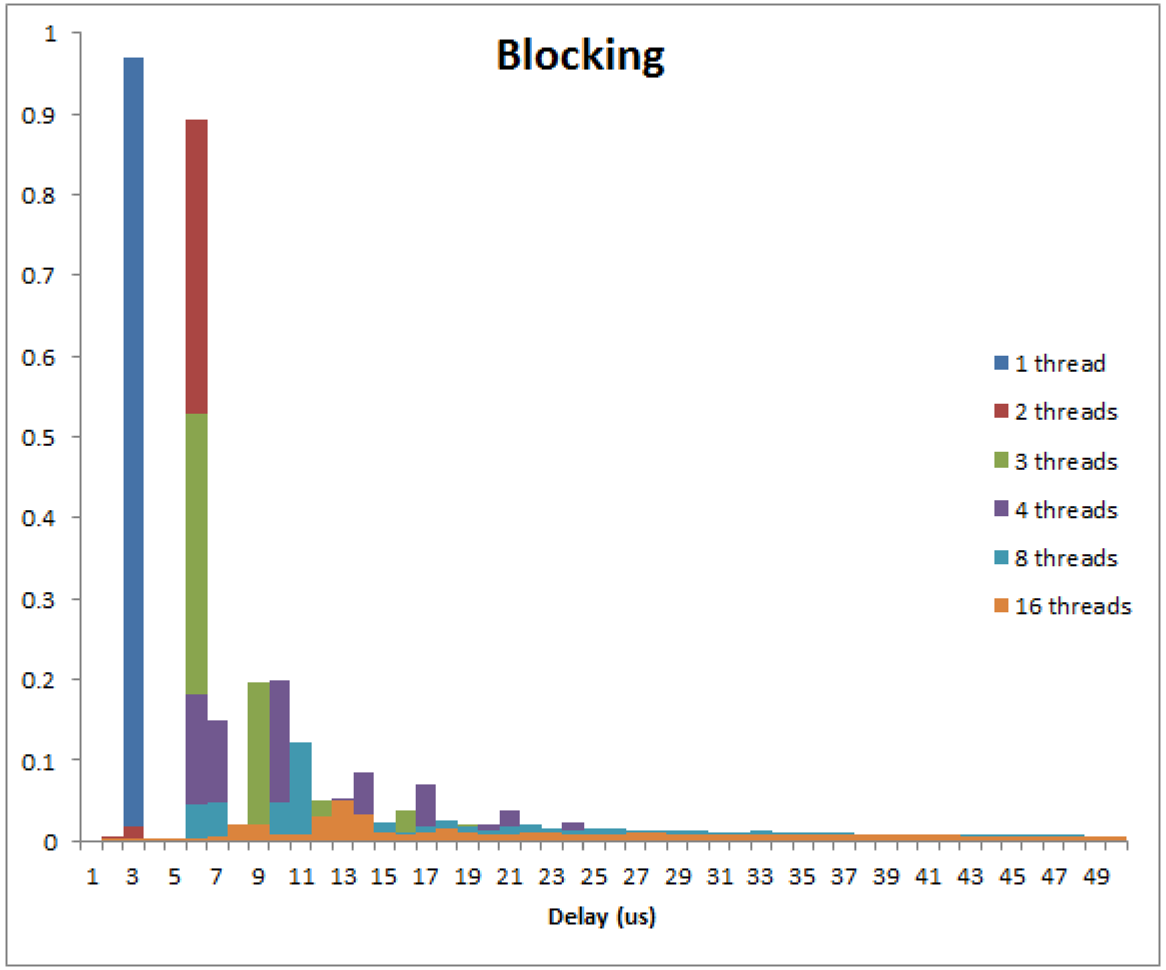




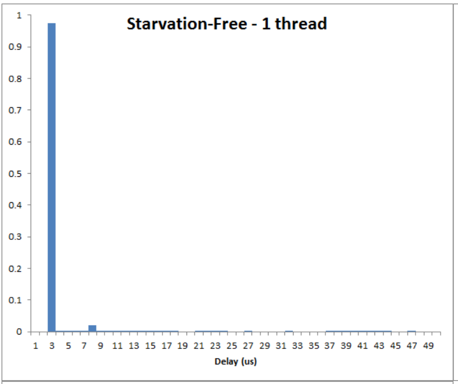


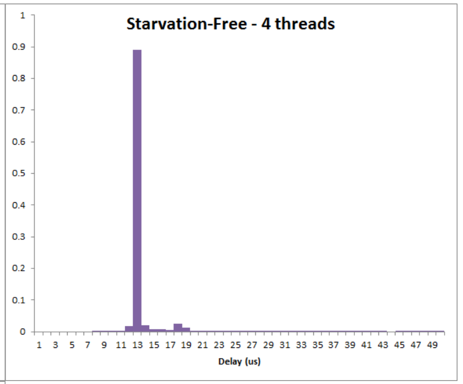


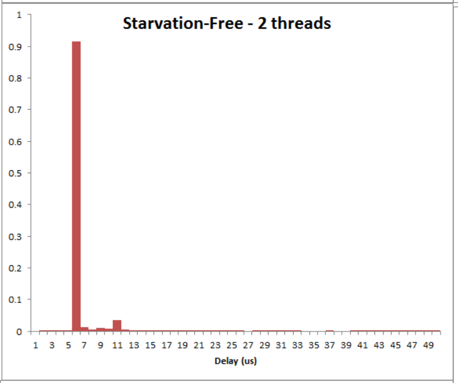


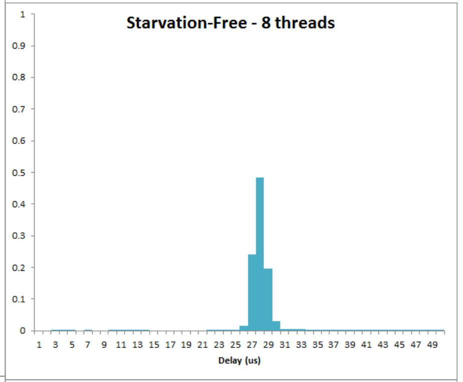


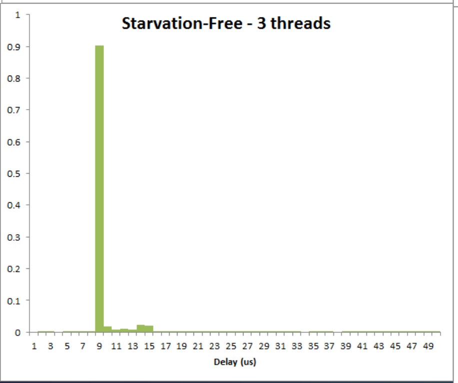
Latency
Starvation-Free (Blocking)

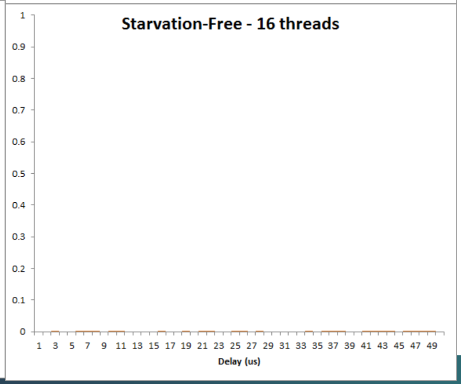


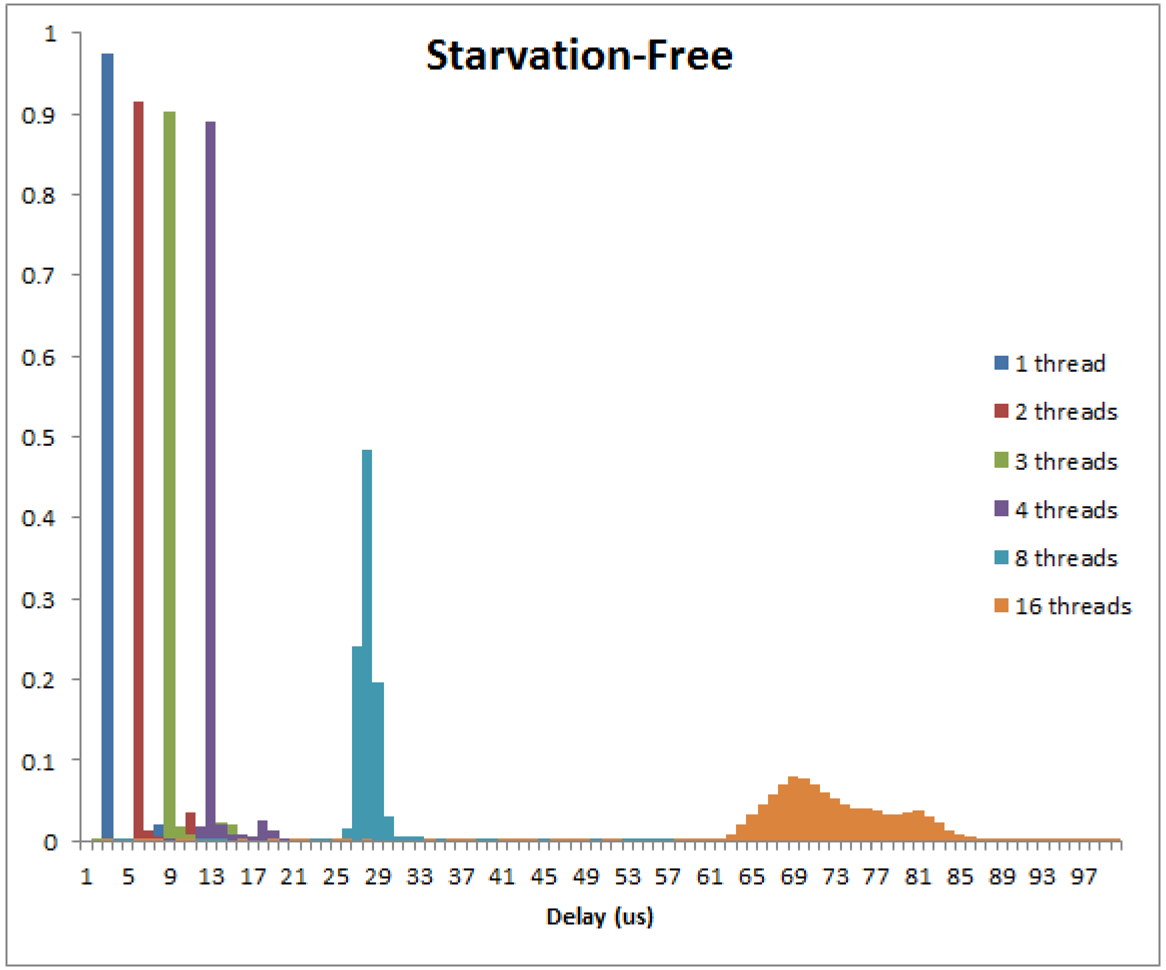




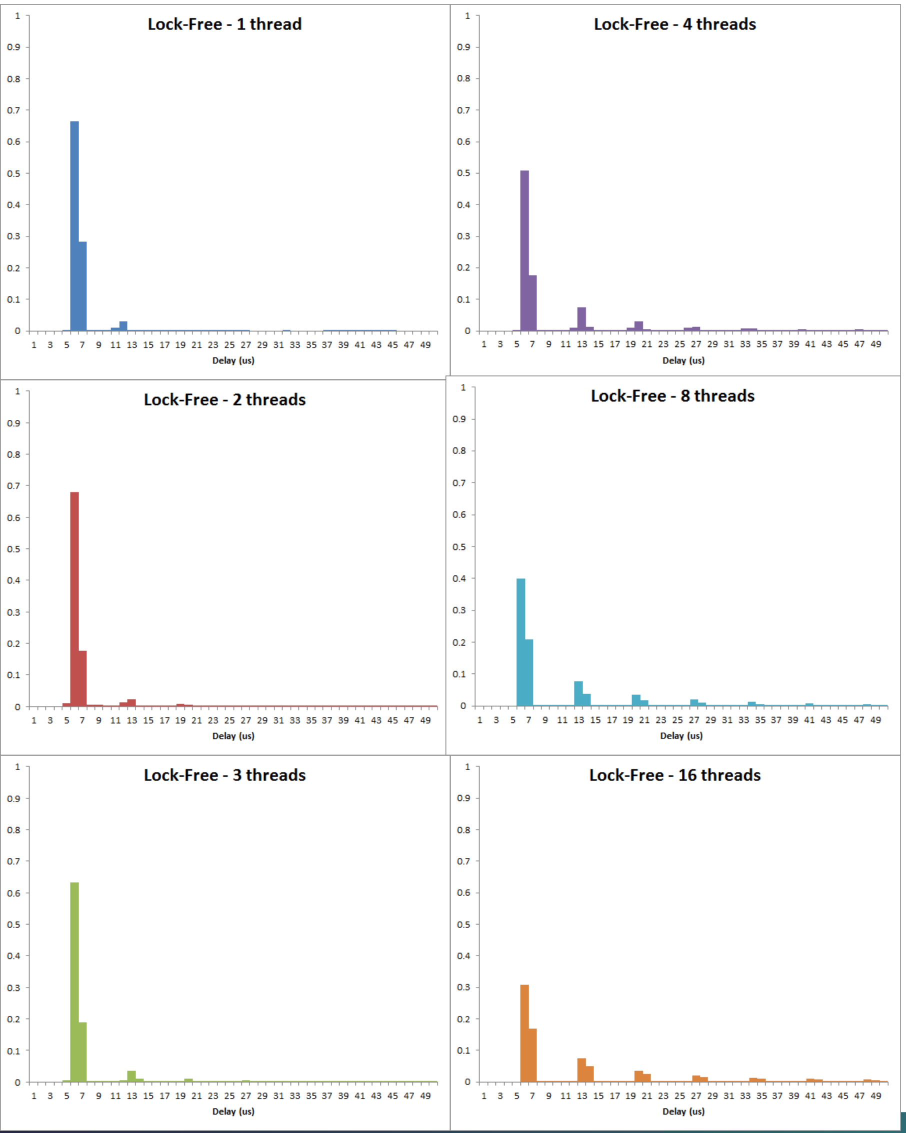


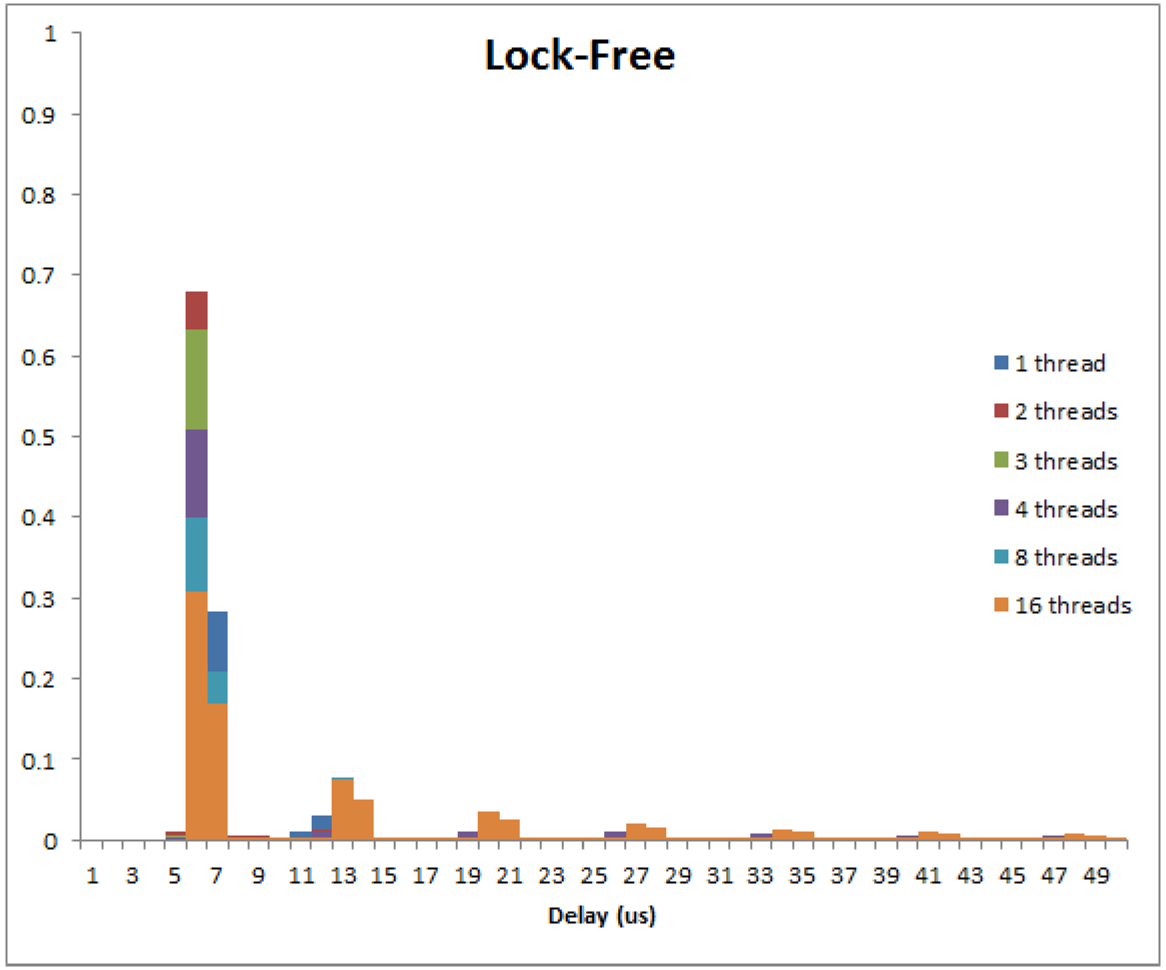




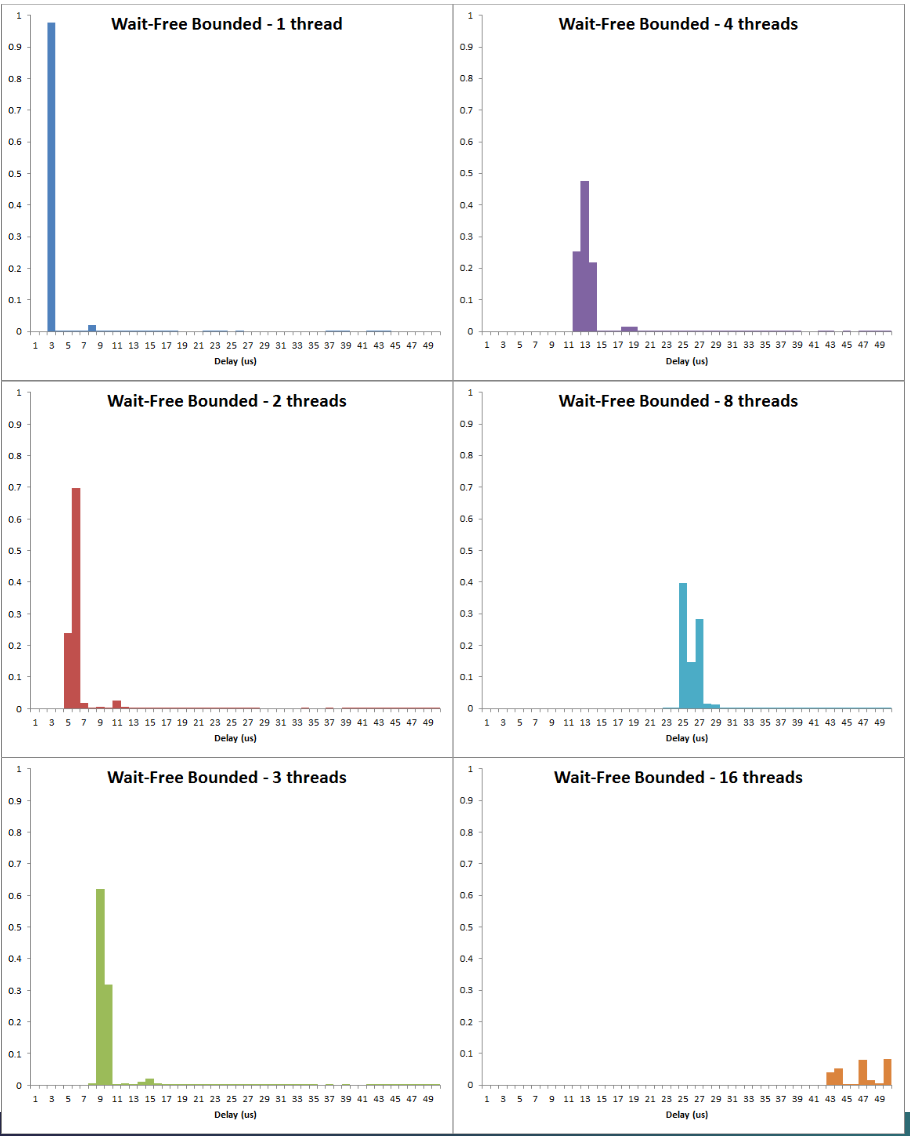


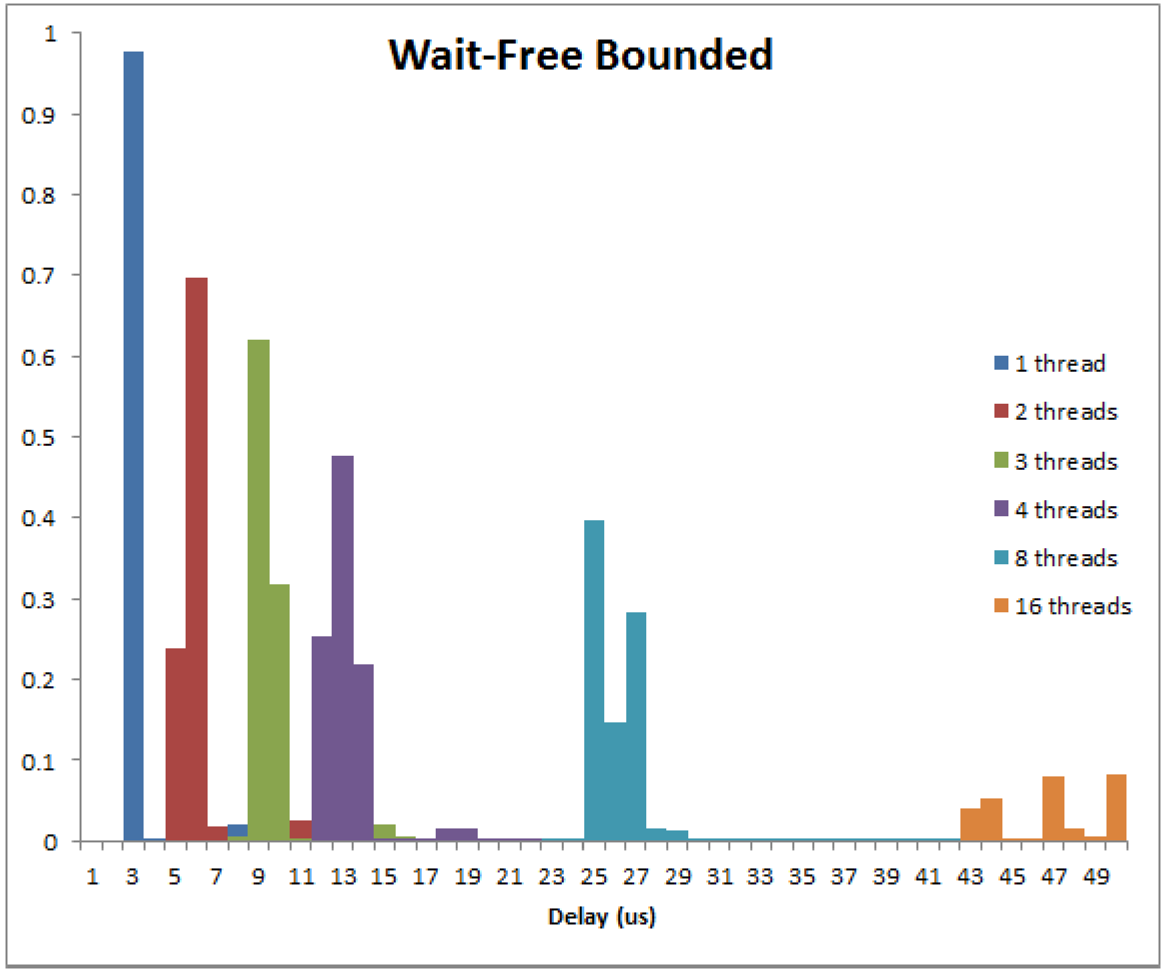
Latency
Lock-Free



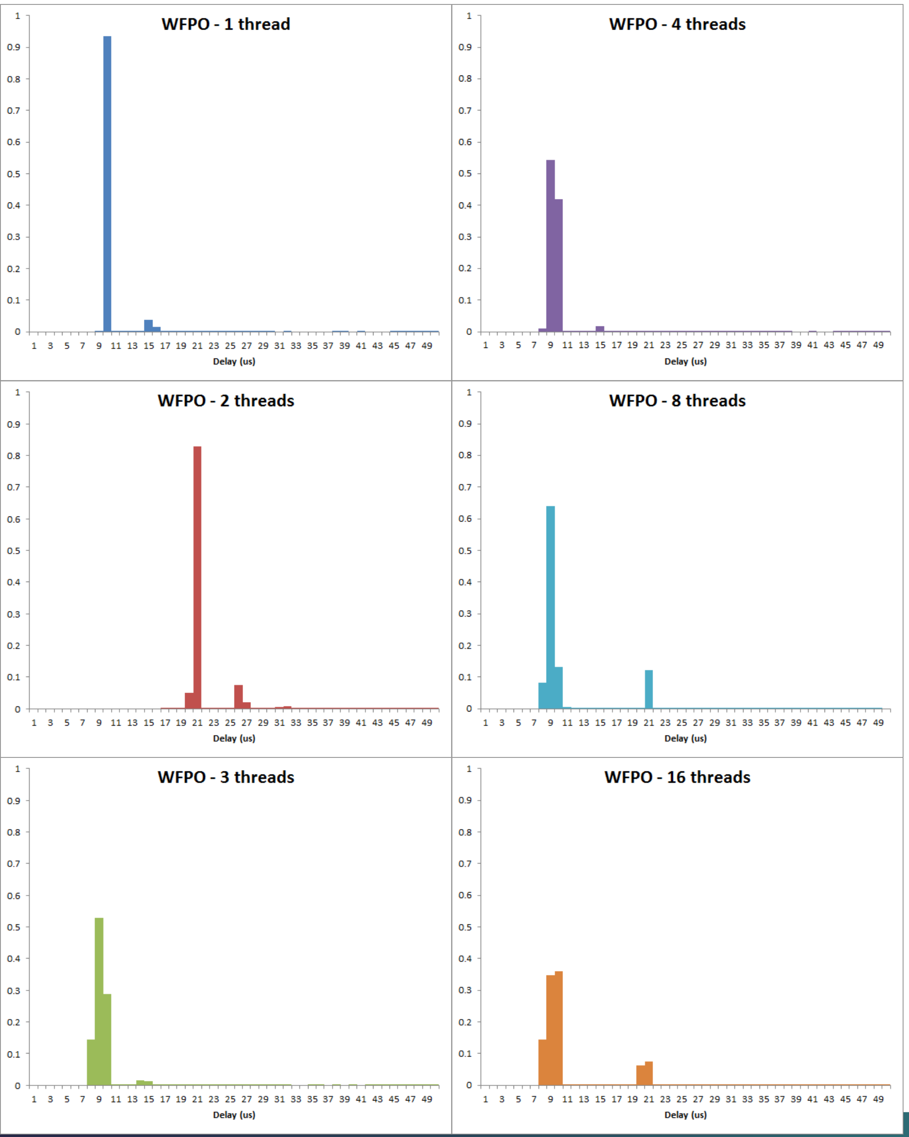


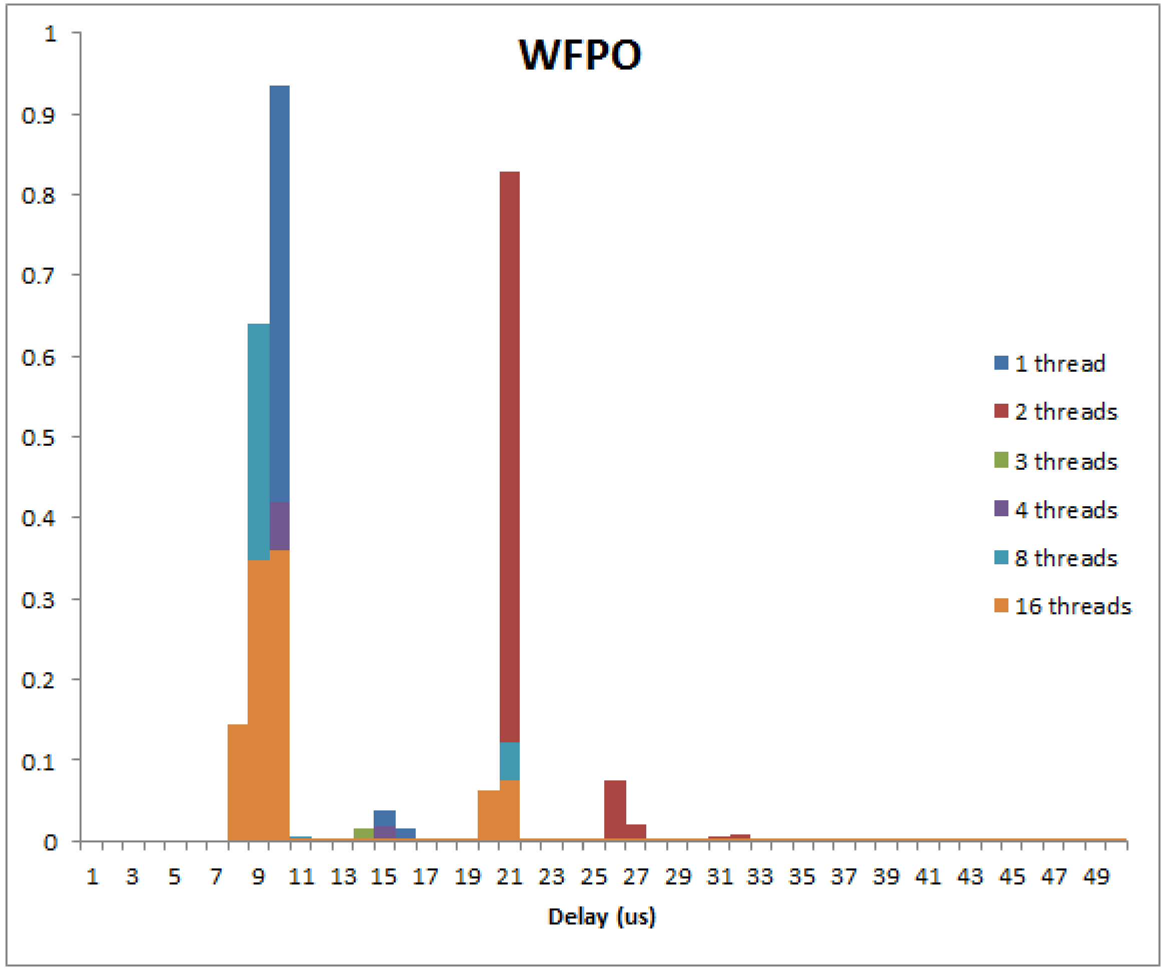
Latency
Wait-Free Bounded, by the number of threads with $O(N_{threads})$



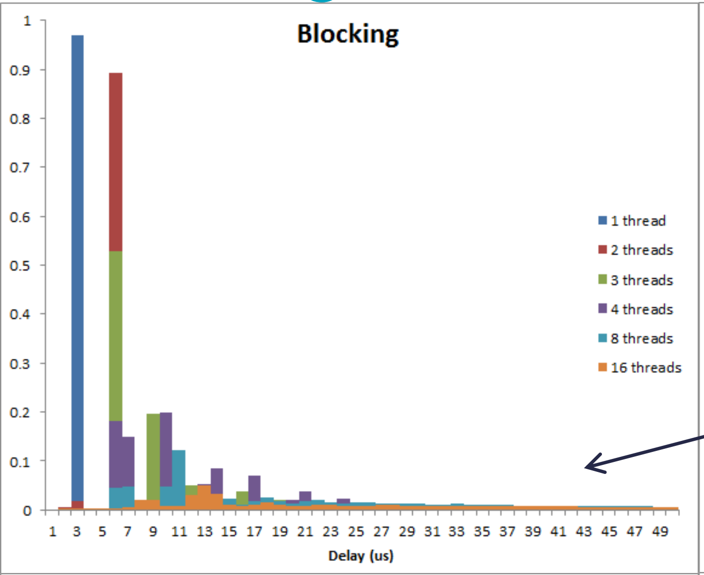


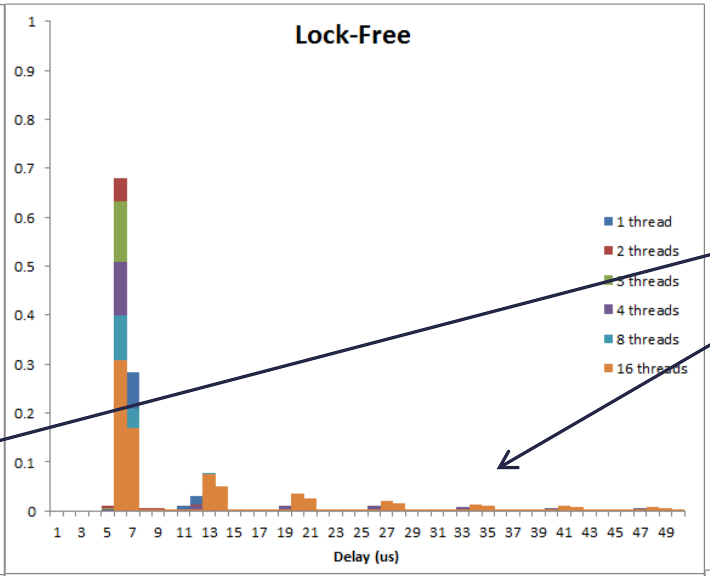
Latency
Wait-Free Population Oblivious



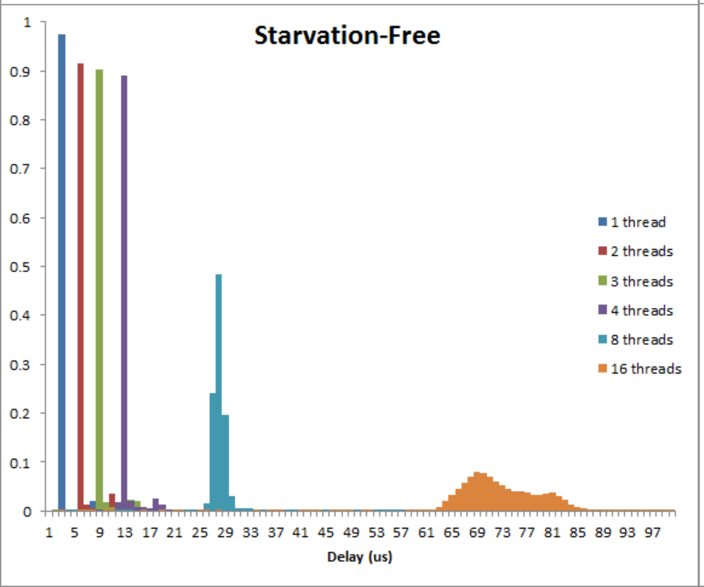


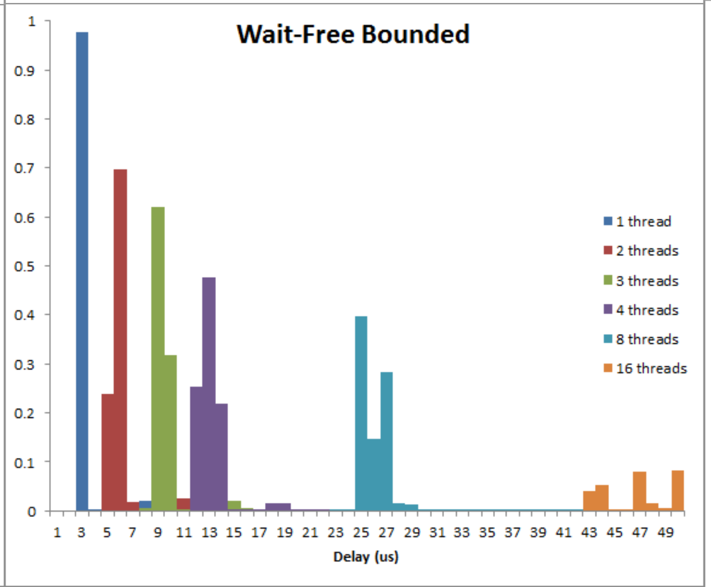
Latency
All together

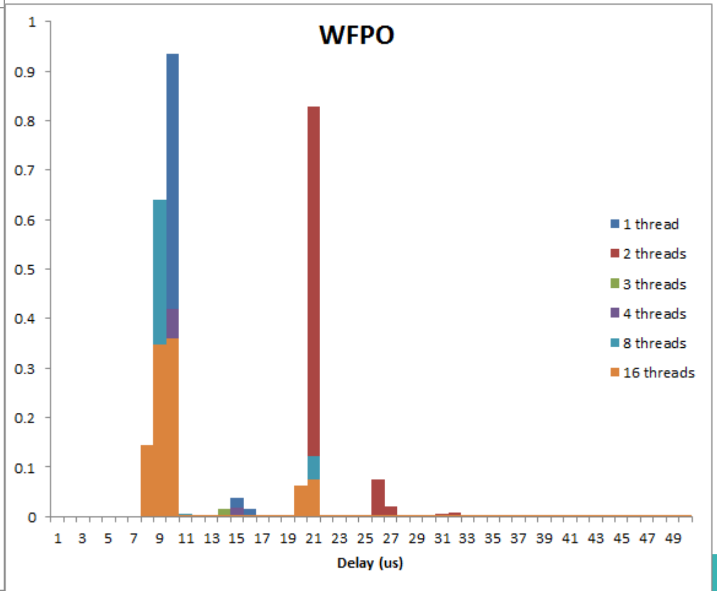




These have a *fat tail*

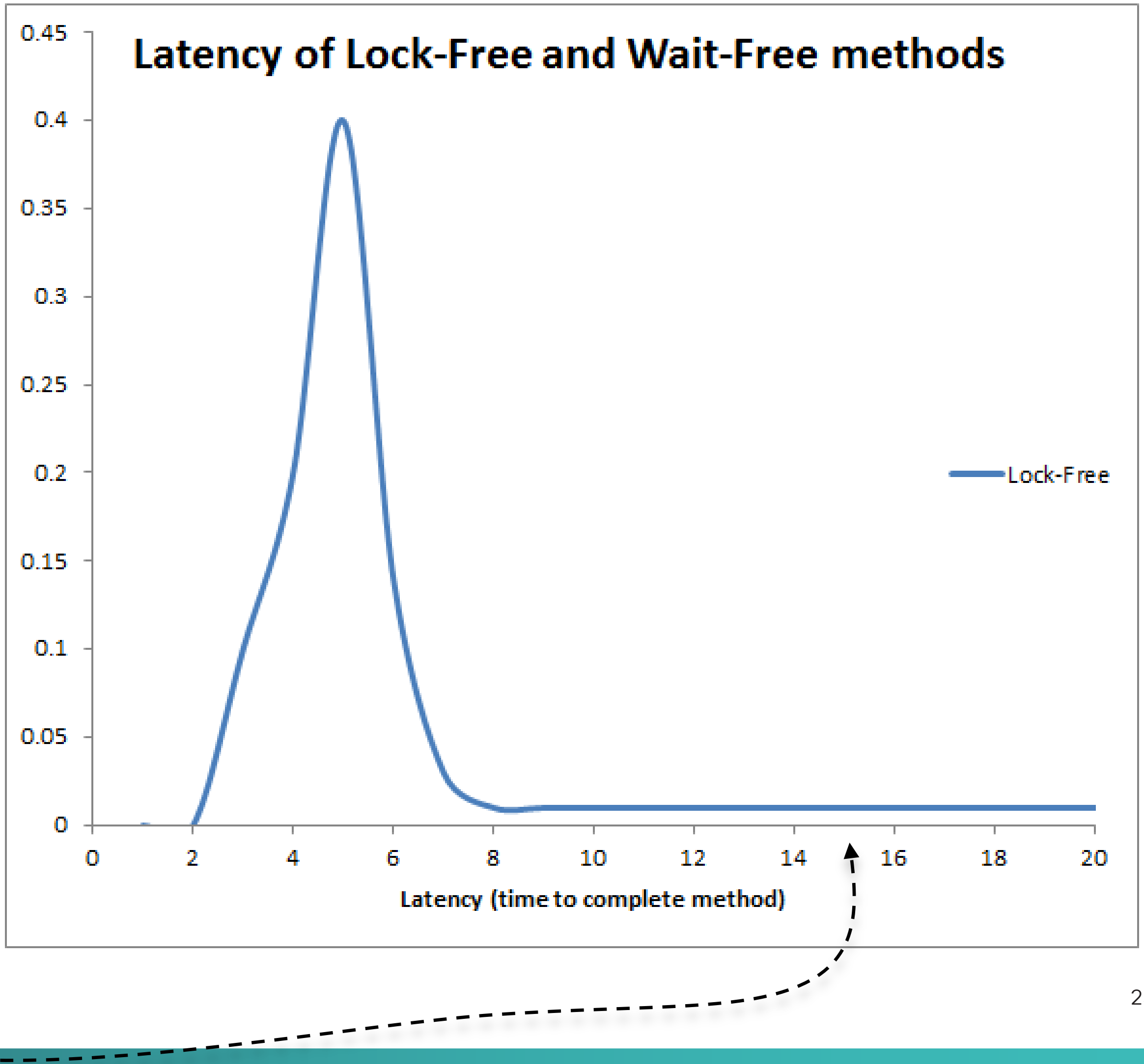






Progress Conditions
Wait-Free vs Lock-Free Latency

The possibility of starvation of Lock-Free methods implies that the Latency distribution of **a Lock-Free method has an infinite tail** (theoretically).



The same is not true for Wait-Free methods which have a cut-off, i.e. a maximum latency.

Progress Conditions Takeaway

- Progress Conditions are defined in terms of (the upper limit of) the number of operations it takes for a method call to complete.
- Progress conditions are defined in terms of number of steps, not in terms of time.
- Different methods of the same class or data structure can have different progress conditions.
- Memory allocation and deletion is usually Blocking.
- Lock-Free implies the possibility of *starvation*.
- The most *desirable* kind of Blocking is Starvation-Free
- The most *desirable* of all progress conditions is Wait-Free Population Oblivious
- Given a time slice, a thread calling a wait-free method is always making *progress*, thus not wasting resources

Reader-Writer Locks

Reader-Writer Locks

- Reader-Writer locks allow for shared ownership (read-only) as well as exclusive ownership (write-modify)
- Unlike Mutual Exclusion locks, Reader-Writer locks allow for simultaneous Readers, but they're still Blocking
- RW-Locks come in many flavors:
  - Some follow a strict FIFO ordering (MCS, CLH, Ticket Lock, Tidex Lock), others are Phase-Fair, and others are Unfair (a loop with CAS);
  - Some have Writer-Preference, others Reader-Preference, and others no particular preference;
  - Some have Starvation-Freedom properties, and others don't;
  - Some provide optimistic read access (StampedLock and Seqlock), and others don't;
- Known RW-Lock APIs:
  - C has `pthread_rwlock_t`
  - C++ has BOOST's `boost::shared_mutex`
  - C++17 has `std::shared_mutex` and `std::shared_timed_mutex`

Reader-Writer Locks
Recent work

- There have been some recent advancements in RW-Lock algorithms
  - Design, Verification and Applications of a New Read-Write Lock Algorithm by Jun Shirako and Nick Vrvilo
  - NUMA-Aware Reader-Writer Locks by Irina Calciu, Dave Dice, Yossi Lev, Victor Lunchangco, Virendra Marathe and Nir Shavit
- The second paper has three algorithms, one of which is named C-RW-WP, than can provide high scalability for reads, when there are very few writes.
- Although the title of the paper mentions "NUMA-Aware", this algorithm can be implemented without needing any *NUMA-Aware* API, and we've put several implementations available online, one of them uses an array of atomic counters and hashes the thread's id:
  - https://github.com/pramalhe/ConcurrencyFreaks/blob/master/CPP/locks/DCLCRWLock.h
  - https://github.com/pramalhe/ConcurrencyFreaks/blob/master/CPP/locks/DCLCRWLock.cpp
- The major drawbacks of C-RW-WP are:
  - its memory consumption per instance, at least one cache line per core, to be able to provide maximum scalability;
  - its writer-preference, which could theoretically starve Readers, but should rarely happen in practice;

Code
pthread_rwlock_t

- The simplest way to protect your data structure would be to wrap it with a pthread_rwlock_t. A naïve implementation might look like this:

```
bool add(T key) {
    pthread_rwlock_wrlock(&_rwlock);
    bool retValue = _set.add(key);
    pthread_rwlock_unlock(&_rwlock);
    return retValue;
}


bool contains(T key) {
    pthread_rwlock_rdlock(&_rwlock);
    bool retValue = _set.contains(key);
    pthread_rwlock_unlock(&_rwlock);
    return retValue;
}
```

```
bool remove(T key) {
    pthread_rwlock_wrlock(&_rwlock);
    bool retValue = _set.remove(key);
    pthread_rwlock_unlock(&_rwlock);
    return retValue;
}


template<typename T>
class RWLockLinkedListPT {

private:
    LinkedListSet<T> _set;
    pthread_rwlock_t _rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Reader-Writer Locks
Takeaway

- C++17 has a new Reader-Writer lock named std::shared_mutex

- Recently discovered Reader-Writer Locks allow for a good scalability when doing mostly read-only operations (C-RW-WP from “NUMA-Aware Reader-Writer Locks”), but it to have good scalability it must use at least one cache line per core.

Copy-On-Write

Copy-On-Write (COW)
COWLock

- Copy-On-Write (COW) is sometimes called Read-Copy-Update but should not to be confused with RCU.
- In C++ we can use COW+RCU to transform any single-threaded data structure into a concurrent data structure with wait-free reads and blocking writes.
- COW is a technique that through the use of (fake/logical) immutability reduces the sharing to a minimum, i.e. a single pointer is shared.
- When COW is coupled with immutable data structures it becomes very elegant, and can provide a performance that is not terrible.
- COW is fast and scalable for workloads that are mostly read-only

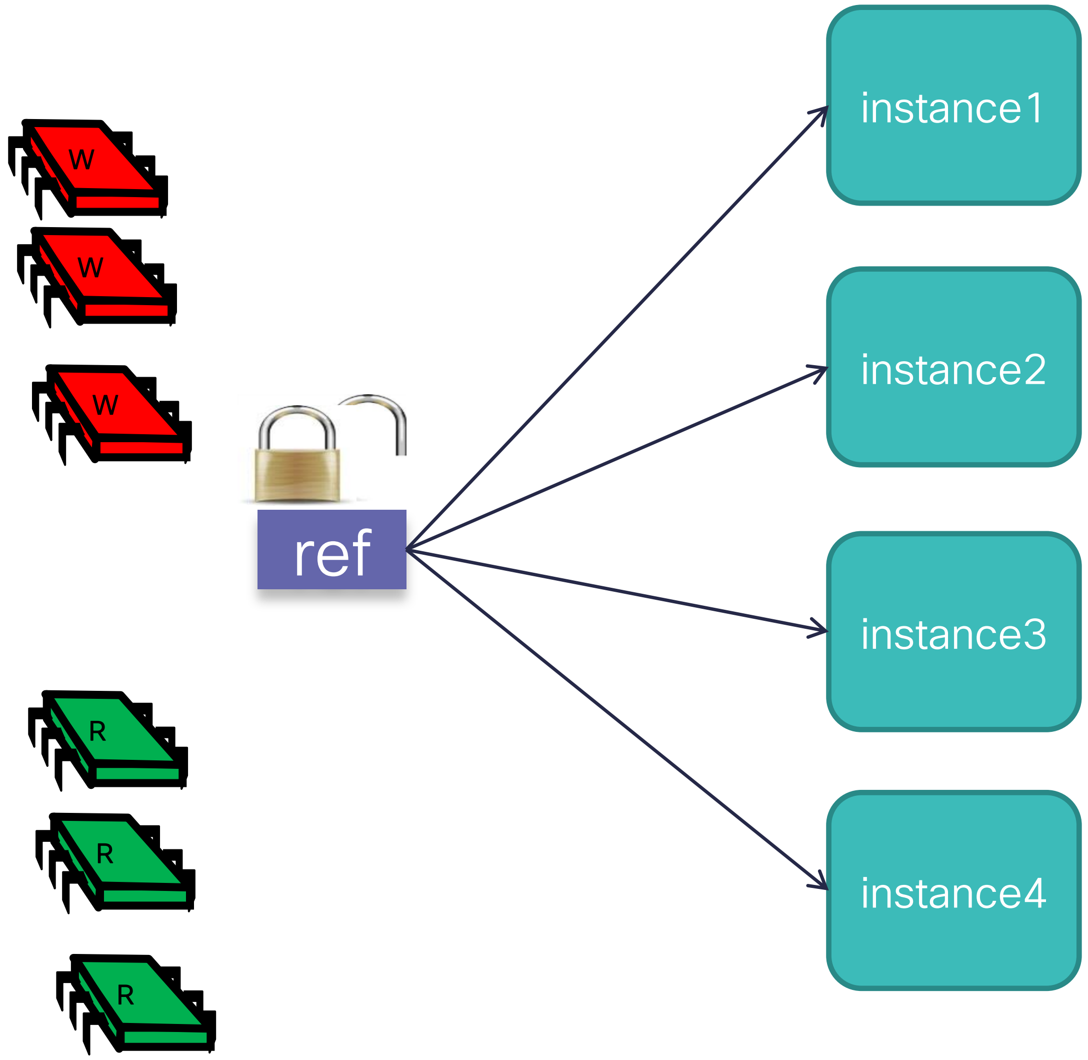
Copy-On-Write with Mutual exclusion lock COWLock

- On COWLock we use a lock to protect write access to the reference/pointer to the instance.
- Every time there is a mutation, a new instance is created (a copy of the current instance).
- At any given time, multiple instances may exist in memory. At most, $N_{Readers}$ of them can not be collected by the GC, and an unlimited number of them may exist that the GC has not yet collected.

```
template<typename Key, typename Value>
class COWLockMap {

private:
    std::atomic<std::map<Key,Value>*> mapRef;
    std::mutex mut;

    auto find( const Key& key ) {
        rcu_read_lock();
        auto ret = (mapRef.load())->find(key);
        rcu_read_unlock();
        return ret;
    }

    auto insert ( std::pair<Key,Value> val ) {
        mut.lock();
        std::map<Key,Value>* currMap = mapRef.load(std::memory_order_relaxed);
        std::map<Key,Value>* newMap = new std::map<Key,Value>(*currMap);
        auto ret = newMap->insert(val);
        _mapRef.store(newMap);
        // Wait for older Readers before deleting the old map
        synchronize_rcu();
        mut.unlock();
        currMap->clear(); // clear before deleting so we don't delete the keys/values
        delete currMap;
        return ret;
    }
}
```

Copy-On-Write with Mutual exclusion lock COWLock

- Writers (threads attempting a mutative operation) will have to wait for other Writers, but Readers (thread doing a read-only operation) can access `ref` directly to access the current instance.
- Read-only operations are non-blocking (wait-free population oblivious), but write/mutative operations are blocking.
- The instances still being referenced by a Reader can not be cleaned up and the Writer will block waiting for `synchronize_rcu()`.



Left-Right Pattern

What is the Left-Right pattern ?

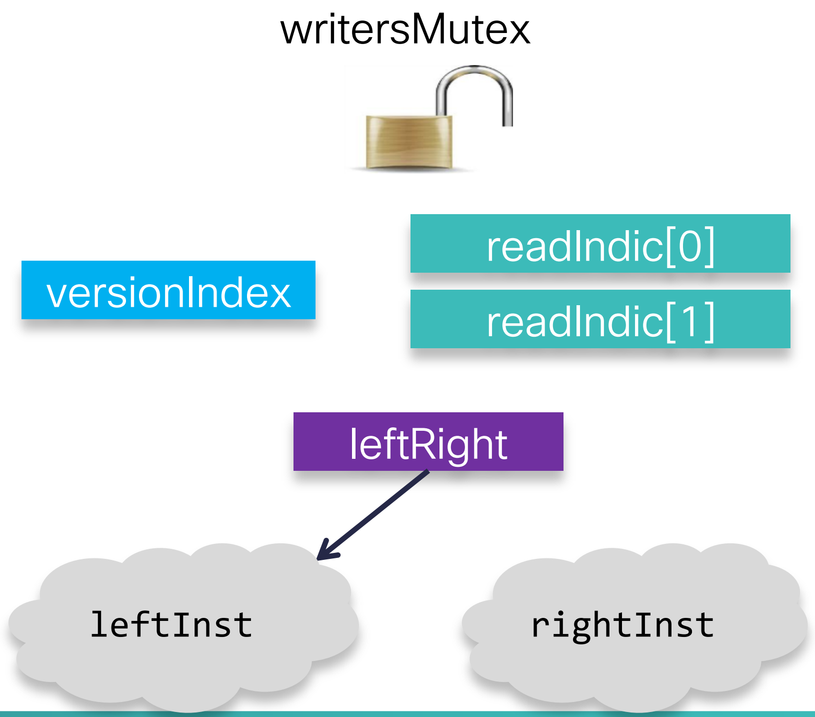
- Left-Right is a technique with some similarities with Double Instance Locking because it uses two instances, and has a mutex to serialize mutations. In this sense, it is reminiscent of the “Double Buffering” technique used in computer graphics
- However, Left-Right is **non-blocking** for read-only operations, and fast
- Left-Right **does its own memory reclamation**, which means we can safely use it in C++
- If so desired, Left-Right can support optimistic reads, similarly to SeqLocks in Linux, or StampedLock in Java
- Left-Right is a generic, linearizable, technique that can provide **mutual exclusivity for any object in memory or data structure**

… however, some data structures can not be replicated, for example, an array that uses more than half the available memory ☺

Classic Left-Right
What's it made of ?

- The Classic Left-Right variant is composed of the following components:
  - Two ReadIndicators, one per version;
  - A `versionIndex` atomic variable;
  - A `leftRight` atomic variable;
  - A mutual exclusion lock named `writersMutex`;
  - Two (constant) pointers to the instances to which we want to provide concurrent access;
- The advantage of using the Classic variant is that it allows for any kind of ReadIndicator, and therefore, it is very flexible in terms of the desired tradeoffs for memory, scalability, and latency.

```
ReadIndicator     readIndic[2];
std::atomic<int> versionIndex{ VERSION0 };
std::atomic<int> leftRight{ READS_ON_LEFT };
std::mutex        writersMutex;
T* const          leftInst;
T* const          rightInst;
```



Writer’s algorithm
Example with two `std::map` instances

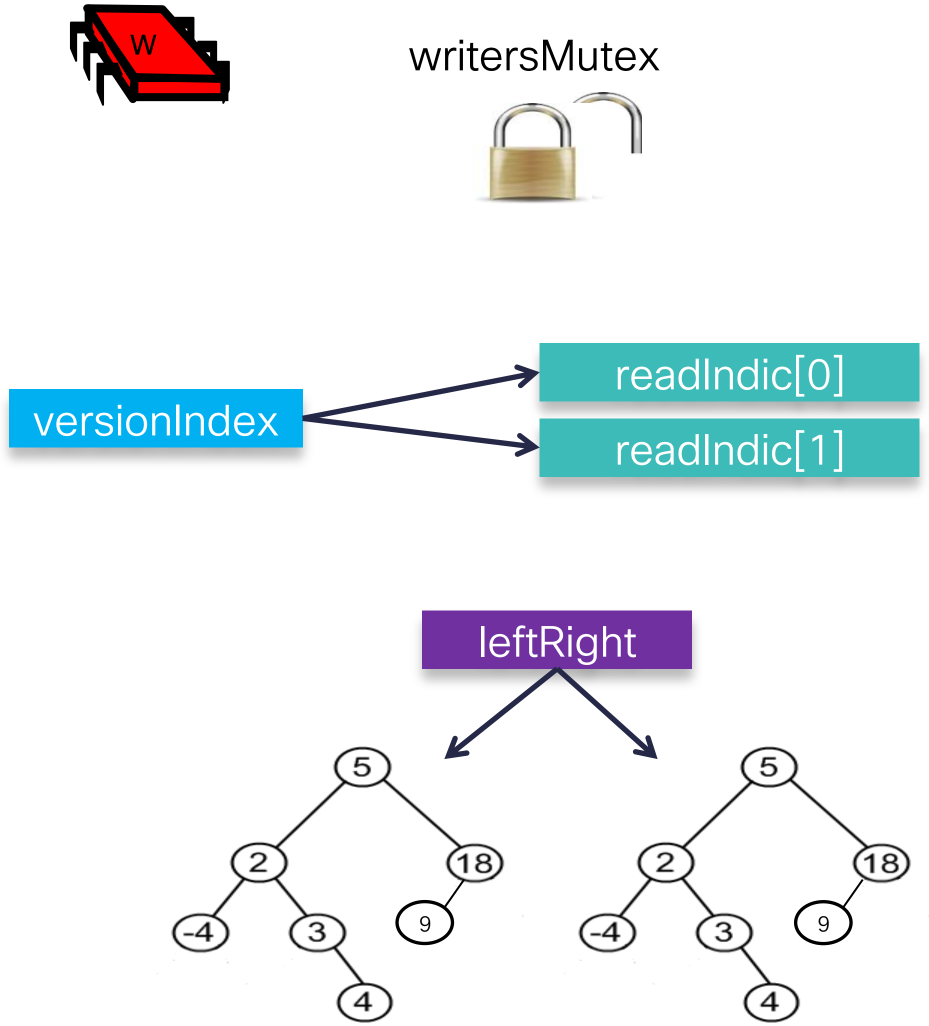
1. Acquire the `writersMutex` to guarantee there is a single Writer at a time;

2. Execute the mutation on the instance opposite to the one indicated by `leftRight`;

3. Toggle `leftRight` wait for the ReadIndicator opposite to `versionIndex` to be `isEmpty()` and then toggle `versionIndex`, and wait for the counters on the current opposite ReadIndicator to be `isEmpty()`;

4. Execute the mutation on the other instance;

5. Release the `writersMutex`;



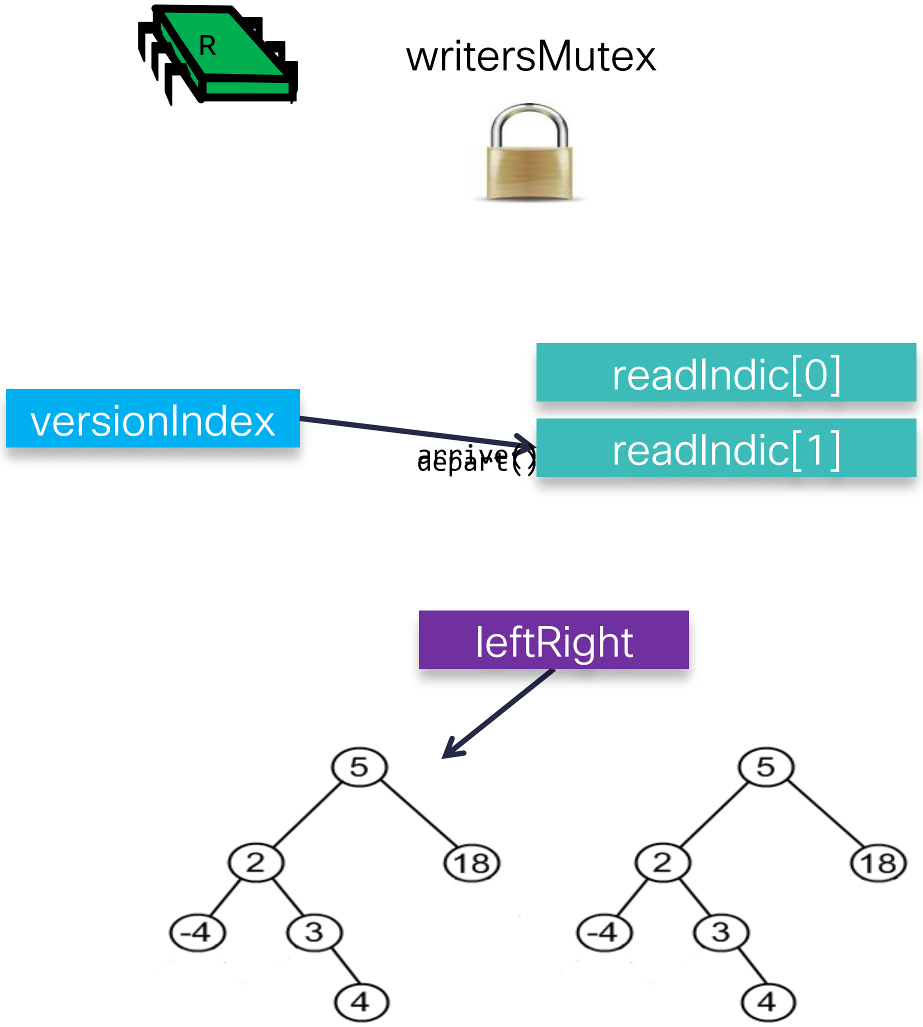
Reader's algorithm

Example with two `std::map` instances

1. Read the current `versionIndex` and do `arrive()` on the respective ReadIndicator;

2. Read the current `leftRight` and do the read-only operation on the corresponding instance;

3. Do `depart()` on the same ReadIndicator that was used for arrive()



Classic Left-Right
Show me some code!

- When implemented as method calls, the Left-Right has three APIs:
  - `arrive()`: Does arrive() on the ReadIndicator based on the versionIndex that was read;
  - `depart()`: does depart() on the previously arrived ReadIndicator;
  - `toggleVersionAndWait()`: Toggles the `versionIndex` variable and waits for old Readers to complete;

```
int arrive(void) {
    const int localVI = versionIndex.load();
    readIndic[localVI].arrive();
    return localVI;
}

void depart(int localVI) {
    readIndic[localVI].depart();
}
```

```
void toggleVersionAndWait(void) {
    const int localVI = versionIndex.load();
    const int prevVI = (int)(localVI & 0x1);
    const int nextVI = (int)((localVI+1) & 0x1);
    // Wait for Readers from next version
    while (!readIndic[nextVI].isEmpty()) {
        std::this_thread::yield();
    }
    // Toggle the versionIndex variable
    versionIndex.store(nextVI);
    // Wait for Readers from previous version
    while (!readIndic[prevVI].isEmpty()) {
        std::this_thread::yield();
    }
}
```

Classic Left-Right
Show me some code… with lambdas!

- When implemented with lambdas, the API consists of passing either a read-only method, or a mutative method. Here's what it looks like in C++1x
- Keep in mind that `applyRead()` is the one called by the *Arriver* threads, and that `applyMutation()` is the one called by the *Toggler* threads.

https://github.com/pramalhe/ConcurrencyFreaks/blob/master/CPP/leftright/LeftRightClassicLambda.h

```
template<typename R, typename A>
R applyRead(A& arg1, std::function<R(T*,A)>& readOnlyFunc) {
    const int lvi = lrc.arrive();
    T* inst = leftRight.load() == READS_LEFT ? leftInst : rightInst;
    R ret = readOnlyFunc(inst, arg1);
    lrc.depart(lvi);
    return ret;
}

template<typename R, typename A>
R applyMutation(A& arg1, std::function<R(T*,A)>& mutativeFunc) {
    std::lock_guard<std::mutex> lock(lrc.writersMutex);
    if (leftRight.load(std::memory_order_relaxed) == READS_LEFT) {
        mutativeFunc(rightInst, arg1);
        leftRight.store(READS_RIGHT);
        lrc.toggleVersionAndWait();
        return mutativeFunc(leftInst, arg1);
    } else {
        mutativeFunc(leftInst, arg1);
        leftRight.store(READS_LEFT);
        lrc.toggleVersionAndWait();
        return mutativeFunc(rightInst, arg1);
    }
}
```

Left-Right
Usage example with **`std::map`**

```
LeftRightClassicLambda<std::map<int,UserData>>   lrcLambda;

std::function<bool(std::map<int,UserData>*,int)> findLambda =
                [](auto _map, auto _key) { return _map->find(_key) != _map->end(); };

std::function<bool(std::map<int,UserData>*,int)> eraseLambda =
                [](auto _map, auto _key) { return _map->erase(_key); };

std::function<bool(std::map<int,UserData>*,std::pair<int,UserData>)> insertLambda =
                [](auto _map, auto _pair) { _map->insert(_pair); return true; };


lrcLambda.applyRead( i1, findLambda );
lrcLambda.applyMutation( i1, eraseLambda );
lrcLambda.applyMutation( i1pair, insertLambda );
```

Left-Right
Left-Right's algorithm variants

- There are many algorithms that can be used with the Left-Right Pattern
  - Classical http://concurrencyfreaks.blogspot.com/2013/12/left-right-classical-algorithm.html
  - Reader's Version http://concurrencyfreaks.blogspot.com/2014/02/left-right-readers-version-variant.html
  - No Version

    http://concurrencyfreaks.blogspot.com/2013/12/left-right-no-version-variant.html

    http://concurrencyfreaks.blogspot.com/2014/01/left-right-no-version-revisited.html
  - Atomic Long http://concurrencyfreaks.com/2013/12/left-right-concurrency-control.html
  - Atomic Long – No Version http://concurrencyfreaks.com/2014/12/left-right-atomic-no-version-variant.html
  - GT and ScalableGT http://concurrencyfreaks.com/2014/11/left-right-gt-variant.html
  - David Goldblatt's variant http://dgoldblatt.com/going-from-lock-free-to-wait-free.html
  - At least urcu-mb and urcu-bp algorithms https://github.com/urcu/userspace-rcu/blob/master/README.md
  - … and others

Left-Right

Read Indicators

Read Indicators

- **Read-Indicators** are some times called **Zero-Indicators** or even **Non-Zero-Indicators**
- They are used in many concurrency mechanisms and they have a very simple API:
  - **`arrive()`**: Mark the *arrival* (of a Reader)
  - **`depart()`**: Mark the *departure* (of a Reader)
  - **`isEmpty()`**: Returns false if there is a thread that has *arrived* but not yet *departed*
- Read Indicators are expected to provide (at least) sequential consistency among its three different method calls.
- Read Indicators are used in:
  - Different types of Reader-Writer Locks
  - Left-Right mechanism

Read Indicators
Single Counter

- A single atomic counter can be used as a ReadIndicator
- On x86 it is wait-free because atomic_fetch_add() is implemented as XADD (or LOCK ADD), but on other architectures it may be lock-free

```
class RIAtomicCounter : public ReadIndicator {

private:
    std::atomic<long long> counter { 0 };

public:
    void arrive(void) override {
        counter.fetch_add(1);
    }

    void depart(void) override {
        counter.fetch_add(-1);
    }

    bool isEmpty(void) override {
        return counter.load() == 0;
    }
};
```

- What are the progress conditions of each method?

Read Indicators
Array of Counters

- An array of counters with a few fence optimizations. This is sometimes called **Statistical Counters**, or **Split Counters**, or **Distributed Counters**.

```
class RIDistributedCacheLineCounter : public ReadIndicator {
private:
    int numCounters; // Size of the counters[] array
    std::atomic<long long> *counters;
    std::atomic<long long> acquireLoad { 0 };
    std::hash<std::thread::id> hashFunc;

    int thread_2_idx (void) {
        std::size_t idx = hashFunc(std::this_thread::get_id());
        return (int)((idx % numCounters)*CACHE_LINE_WORDS);
    }

public:
    RIDistributedCacheLineCounter(int numCounters) {
        numCounters = numCounters;
        counters = new[numCounters*CACHE_LINE_WORDS];
    }

    RIDistributedCacheLineCounter() {
        numCounters = std::thread::hardware_concurrency();
        if (numCounters == 0) numCounters = DEFAULT_NUM_COUNTERS;
        counters = new[_numCounters*CACHE_LINE_WORDS];
    }

    ~RIDistributedCacheLineCounter() {
        delete[] counters;
    }
```

```
  void arrive(void) override {
      counters[thread_2_idx()].fetch_add(1);
  }

  void depart(void) override {
      counters[thread_2_idx()].fetch_add(-1);
  }

  bool isEmpty(void) override {
     int idx = acquireLoad.load();
        for (; idx < numCounters*CACHE_LINE_WORDS; idx += CACHE_LINE_WORDS) {
            if (counters[idx].load(std::memory_order_relaxed) > 0) {
                atomic_thread_fence(std::memory_order_acquire);
                return false;
            }
        }
        atomic_thread_fence(std::memory_order_acquire);
        return true;
  }
};
```

Read Indicators Per Thread

- When the number of threads is known a priori and it is constant, we can create an array (with cache line padding) where each entry is assigned to a unique thread, based on the thread's id.
- Doing `arrive()/depart()` needs a single store-release, and is therefore, wait-free population oblivious in any architecture.
- `isEmpty()` does loads on each of the entries of the array and is therefore, wait-free bounded (but not population oblivious)

```
class RIStaticPerThread : public ReadIndicator {
private:
    enum State { NOT_READING=0, READING=1 };
    int numThreads; // Size of the _states[] array
    std::atomic<long> *states;
    std::atomic<long> acquireLoad { 0 };

public:
    RIStaticPerThread(int numThreads) {
        numThreads = numThreads;
        states = new[numThreads*CACHE_LINE_WORDS];
    }

    ~RIStaticPerThread() {
        delete[] states;
    }

    void arrive(void) override {
        states[(int)std::this_thread::get_id()].store(READING);
    }

    void depart(void) override {
        states[(int)std::this_thread::get_id()].store(NOT_READING);
    }

    bool isEmpty(void) override {
       int tid = acquireLoad.load();
        for (; tid < numThreads*CACHE_LINE_WORDS; tid += CACHE_LINE_WORDS) {
            if (states[tid].load(std::memory_order_relaxed) == READING) {
                atomic_thread_fence(std::memory_order_acquire);
                return false;
            }
        }
        atomic_thread_fence(std::memory_order_acquire);
        return true;
    }
};
```

Read Indicators
Comparison table

- The table below shows a comparison between some of the different Read Indicator implementations
- There is also SNZI which is Lock-Free for `arrive()` and `depart()`. This makes it less interesting for low-latency applications
- For generic usage, the “Array of Counters” provides a bounded memory usage, with high scalability

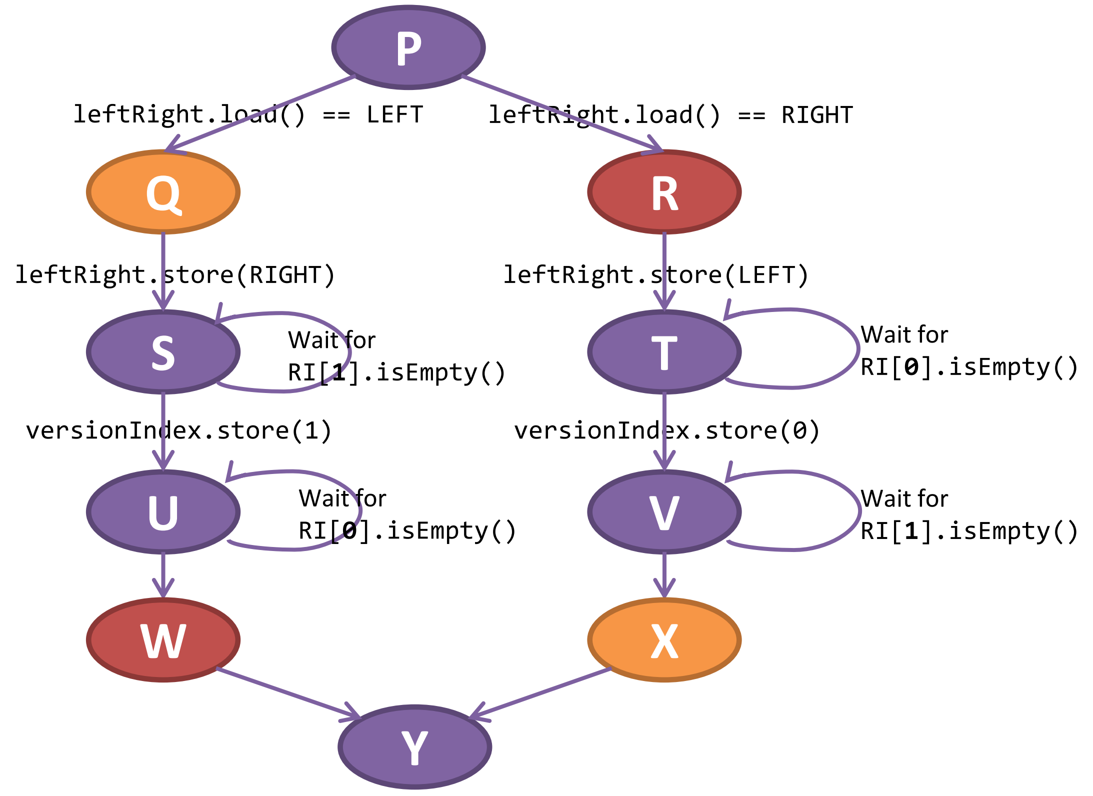
	Progress of arrive() / depart()	Progress of isEmpty()	Needs a GC ?	Scalability ?	Memory usage	Flexibility
Single Counter	WFPO on x86, lock-free on ppc/arm	WFPO	no	awful	O(1)	good
Array of Counters	WFPO on x86, lock-free on ppc/arm	Wait-free	no	good	O(cores)	good
One entry per thread	Wait-free	Wait-free	no	great	O(threads)	bad
SNZI	Lock-free	Wait-free	no	ok	O(cores)	good

Left-Right

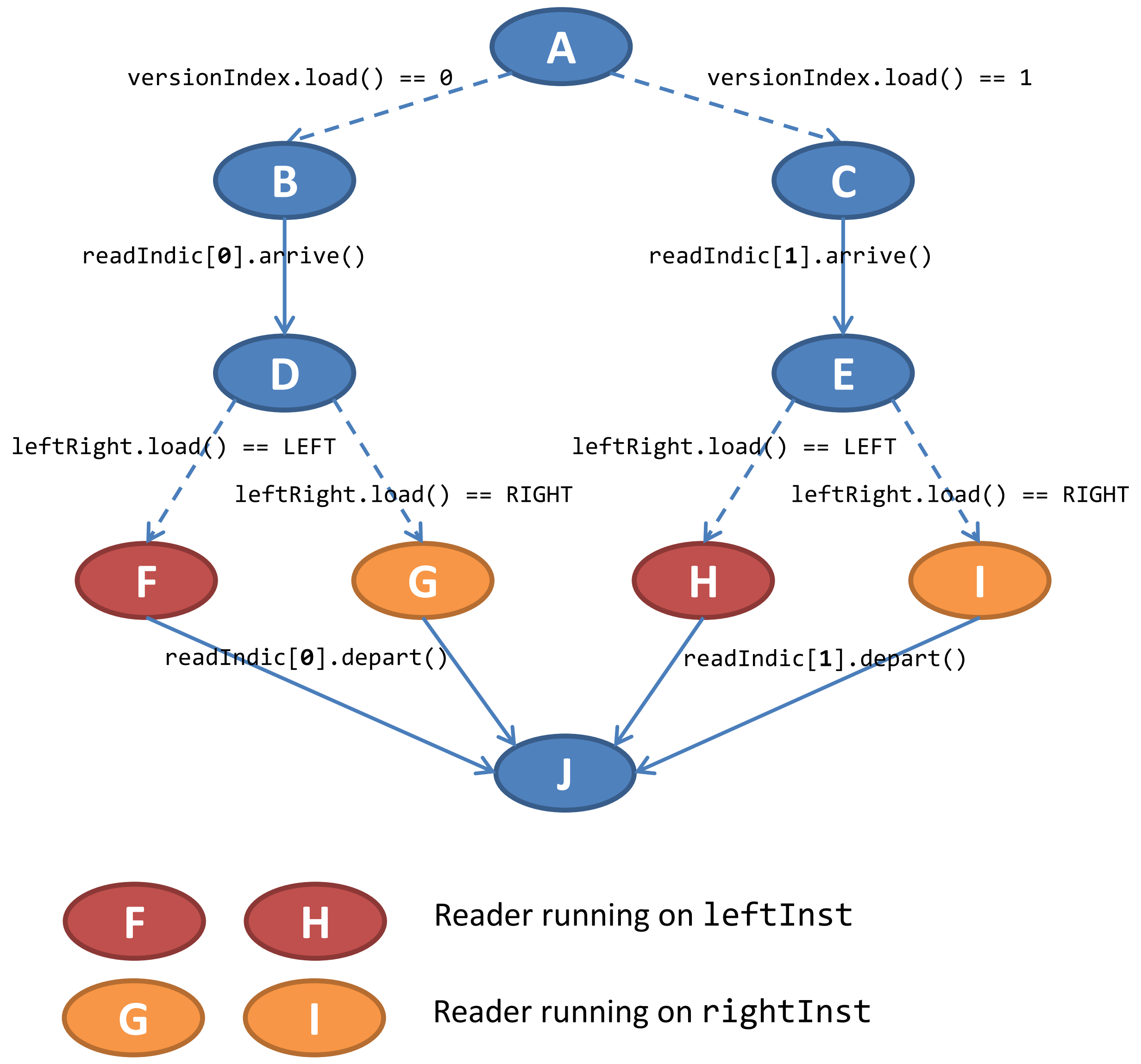
Correctness, Consistency, and Progress

Classic Left-Right
The Algorithm

Writer state diagram



Reader state diagram



Quiz 4

- What is the best Progress Condition for the write/**mutative** operations in the Left-Right technique?

a) Blocking

b) Blocking – Starvation-Free

c) Lock-Free

d) Wait-Free Unbounded

e) Wait-Free Bounded

f) Wait-Free Population Oblivious

```
template<typename R, typename A>
R applyMutation(A& arg1, std::function<R(T*,A)>& mutativeFunc) {
    std::lock_guard<std::mutex> lock(lrc.writersMutex);
    if (leftRight.load(std::memory_order_relaxed) == READS_LEFT) {
        leftRight.store(READS_RIGHT);
        mutativeFunc(rightInst, arg1);
        lrc.toggleVersionAndWait();
        return mutativeFunc(leftInst, arg1);
    } else {
        mutativeFunc(leftInst, arg1);
        leftRight.store(READS_LEFT);
        lrc.toggleVersionAndWait();
        return mutativeFunc(rightInst, arg1);
    }
}

void toggleVersionAndWait(void) {    // a kind of synchronize_rcu()
    const int localVI = versionIndex.load();
    const int prevVI = (int)(localVI & 0x1);
    const int nextVI = (int)((localVI+1) & 0x1);
    while (!readIndic[nextVI].isEmpty()) { // counter.load() == 0
        std::this_thread::yield();
    }
    versionIndex.store(nextVI);
    while (!readIndic[prevVI].isEmpty()) { // counter.load() == 0
        std::this_thread::yield();
    }
}
```

Quiz 5

- What is the best Progress Condition for the **read-only** operations in the Left-Right technique?

a) Blocking

b) Blocking – Starvation-Free

c) Lock-Free

d) Wait-Free Unbounded

e) Wait-Free Bounded

f) Wait-Free Population Oblivious

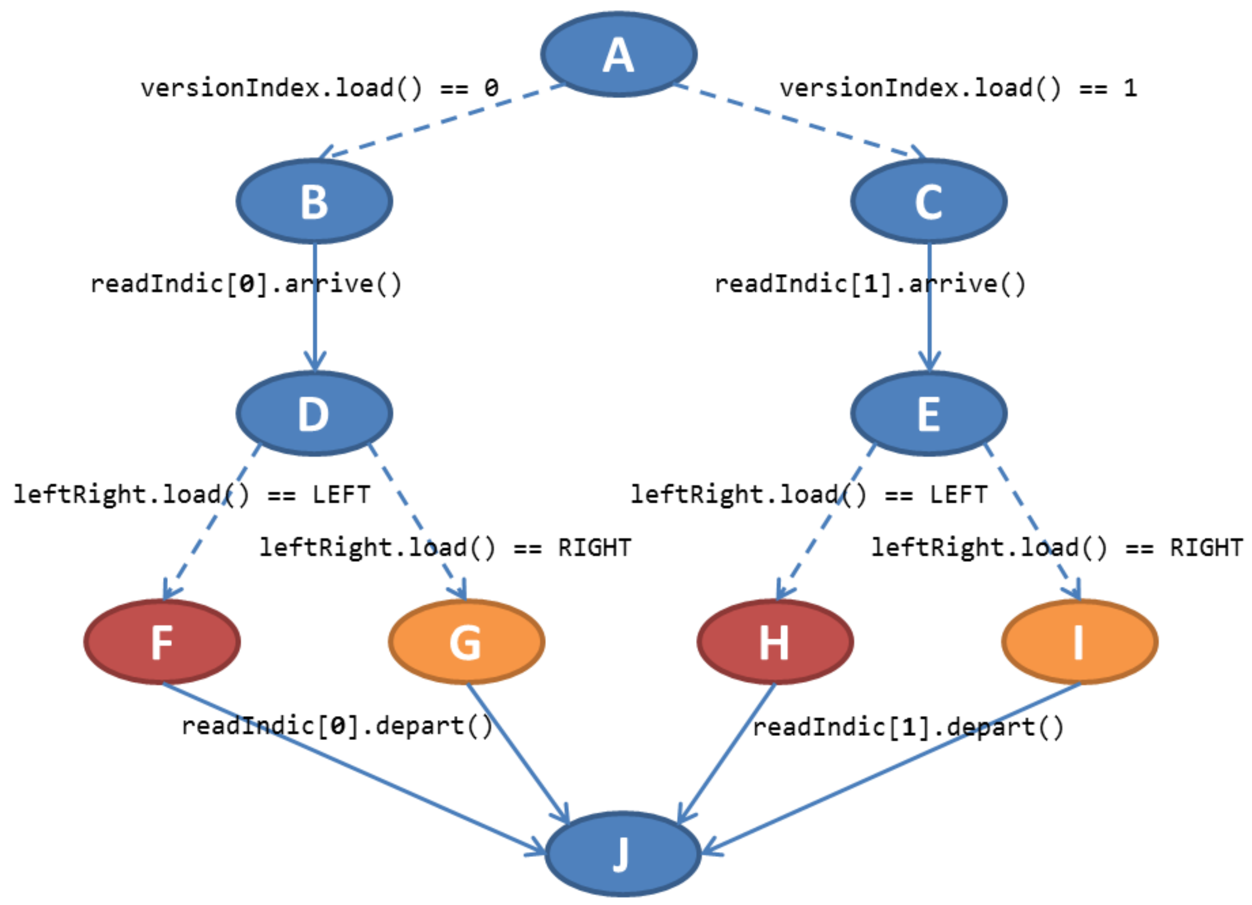
```
template<typename R, typename A>
R applyRead(A& arg1, std::function<R(T*,A)>& readOnlyFunc) {
    const int lvi = lrc.arrive();
    T* inst = leftRight.load() == READS_LEFT ? leftInst : rightInst;
    R ret = readOnlyFunc(inst, arg1);
    lrc.depart(lvi);
    return ret;
}

int arrive(void) {
    const int localVI = versionIndex.load();
    readIndic[localVI].arrive();    // counter.fetch_add(1)
    return localVI;
}

void depart(int localVI) {
    readIndic[localVI].depart();    // counter.fetch_add(-1)
}
```

Left-Right
Progress conditions

- As can be seen on the Reader's state-machine, there are no loops
- This absence of loops (transitions to the same state or other states) is enough to show that this technique is wait-free for Readers
- Because the number of steps is constant, it does not depend on the number of threads, and therefore, it is **Wait-Free Population Oblivious**

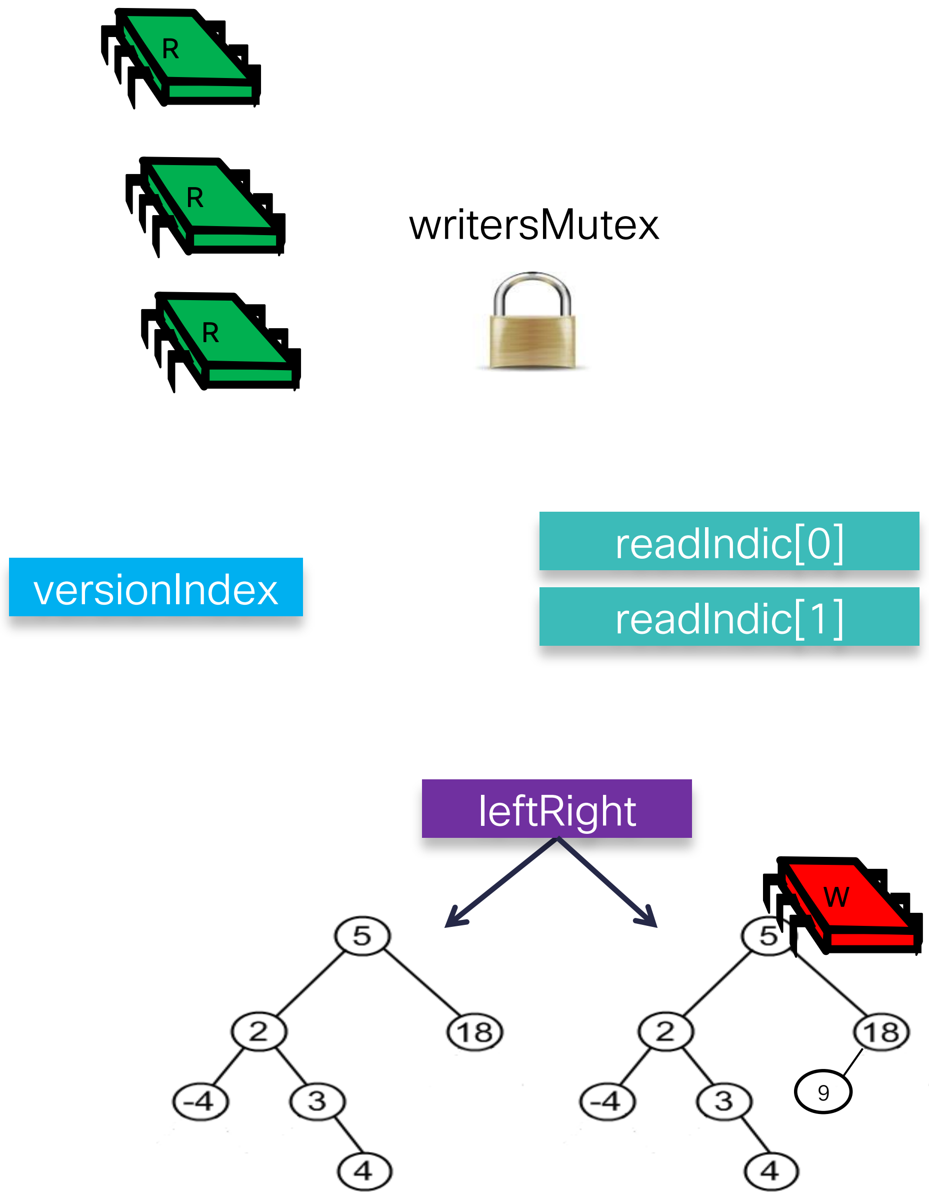
- As for Writers, after toggling the `versionIndex`, they have to wait only for *older* Readers, which means that as long as the Readers are making progress, the Writer will eventually make progress as well. This property is (Blocking) **Starvation-Free**

Reader state diagram



Classic Left-Right
What do you mean by *Linearizable*?

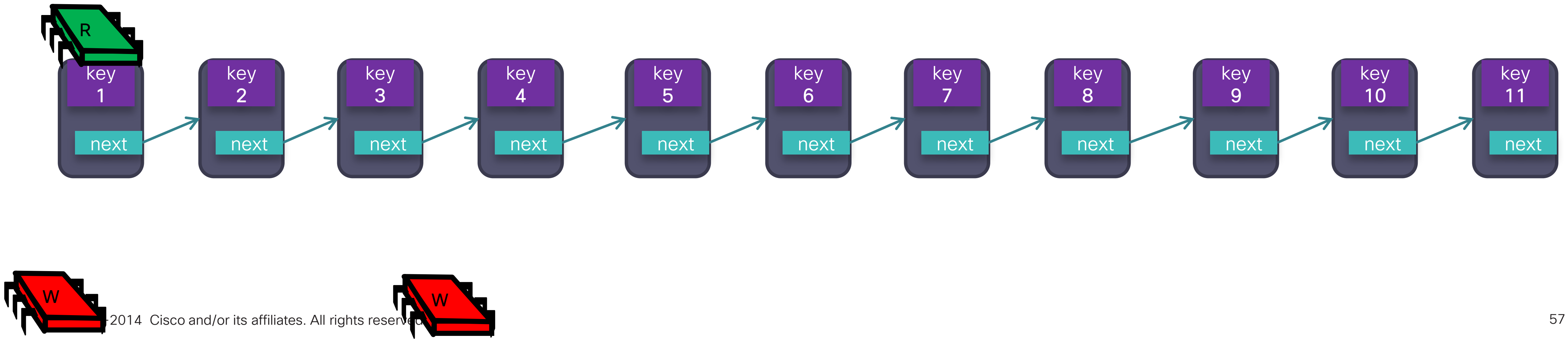
- Linearizability is a strong type of consistency (stronger even that **Sequential Consistency**), and it is the same kind of consistency that mutual exclusion locks (mutexes) and reader-writer locks provide.
- The `writersMutex` ensures there is a single Writer at a time and, therefore, any atomic step after holding this lock can be chosen as the linearization point for Writer to Writers.
- As soon as the Writer thread toggles the `leftRight` variable, the mutation that the Writer did becomes visible to new Readers.
- Readers who have already read `leftRight` with the previous value, are guaranteed to see the older data structure, without the most recent mutation.
- The Linearization Point for Readers is at the load of the `leftRight` variable.
- Notice that regardless of whether a Reader sees the *old* instance or the *new* instance, no Writer will modify it will the Reader is traversing it, i.e. no races occur.



Linearizable Iterators (1/2)

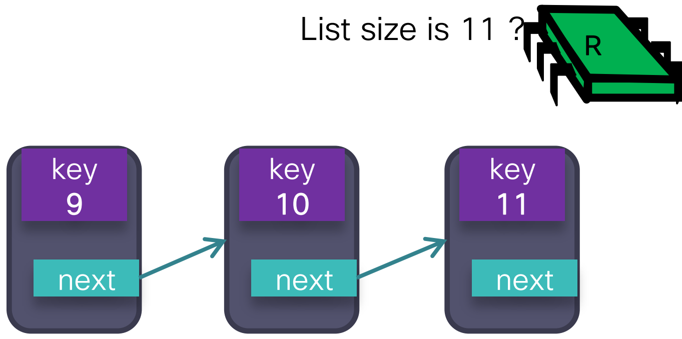
- One advantage of using Left-Right (or Copy-On-Write based techniques) is that ***any*** read-only operations will be wait-free ***and*** linearizable.
- This is something that lock-free and wait-free data structures do not support for iterators or operations requiring consistency with more than one item in the data structure
- For example, suppose we have a lock-free linked-list like the queue by Michael and Scott, where new items are added in the end (tail) and removed from the beginning (head). We start with a linked list containing only three items, whose "keys" are 1, 2, and 3
- Now, suppose that multiple (Writer) threads start adding and removing keys in sequence, such that there are never more than three items at any given instance in time in the list, and at the same time a (Reader) thread is traversing the list with the intuit of counting the number of items in the list

List size is 11 ?



Linearizable Iterators (2/2)

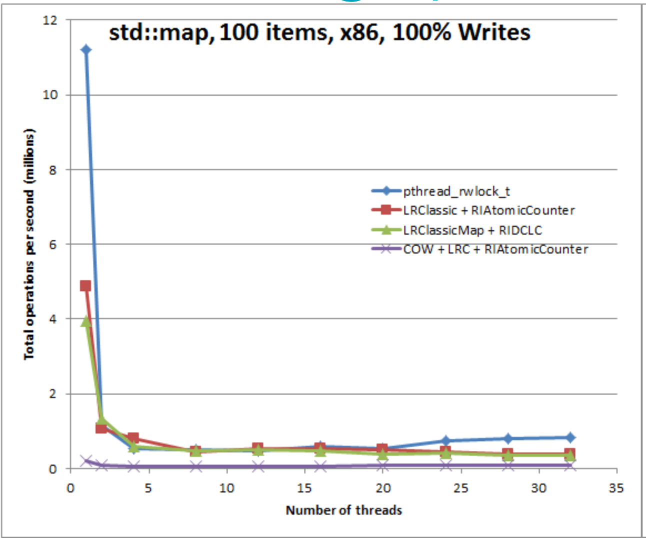
- **There wasn't any point in time** where there were more than three items in the list, however, the thread counting the number of items in the list saw that there were 11… how is this possible?
- Iterations on lock-free/wait-free data structures are not usually linearizable or even sequentially consistent, and in fact, they have very **weak consistency**.
- COW based data structures and Left-Right based data structures are exceptions to this behavior, because in the case of COW, **each Writer makes a new snapshot** of the data, and in the case of Left-Right, **each Writer must wait for every Reader to finish** before executing its mutation.
- Both Copy-On-Write and Left-Right would return either **2** or **3** as the answer to the number of items in the list.

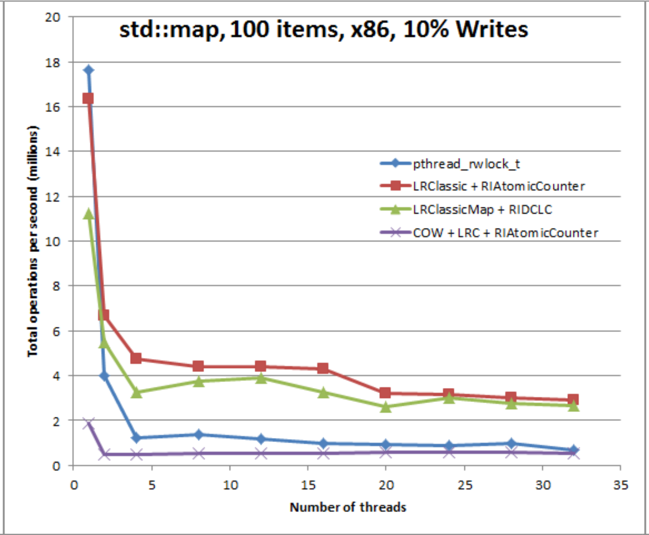


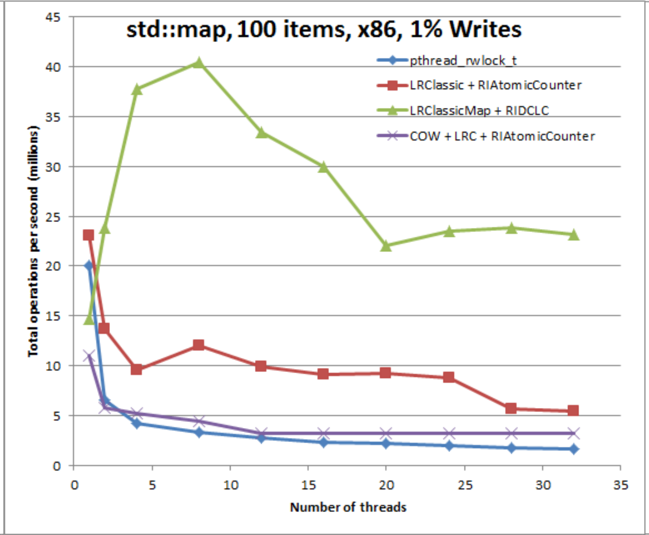
Left-Right

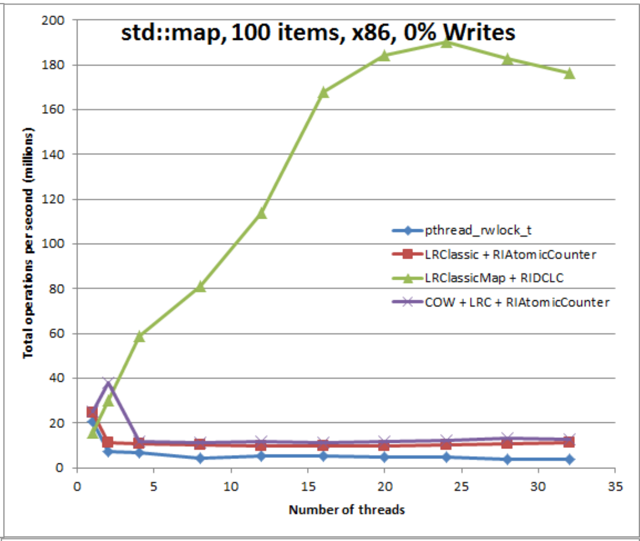
Microbenchmarks

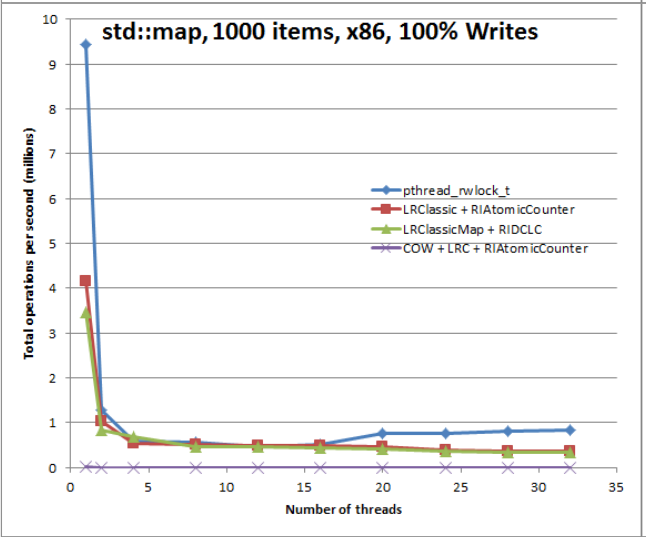
Benchmarks in C++1x – x86 Haswell with 18 cores (36 HT) Throughput

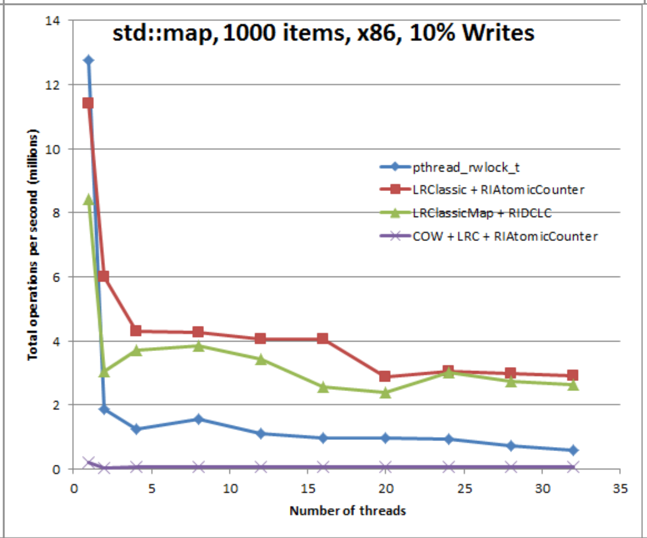


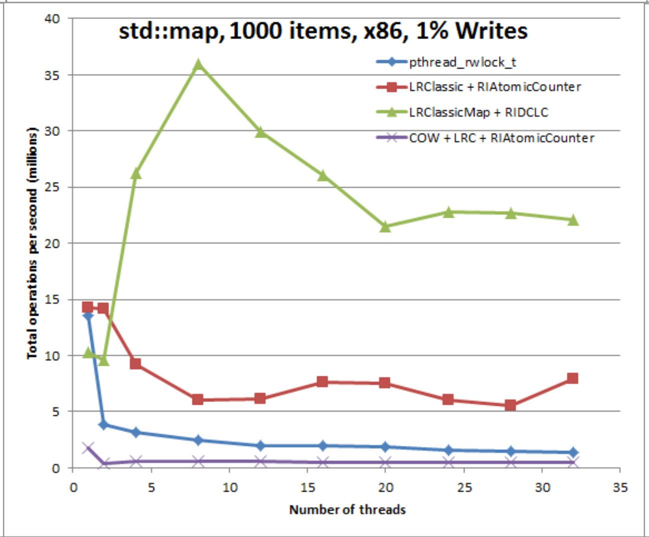


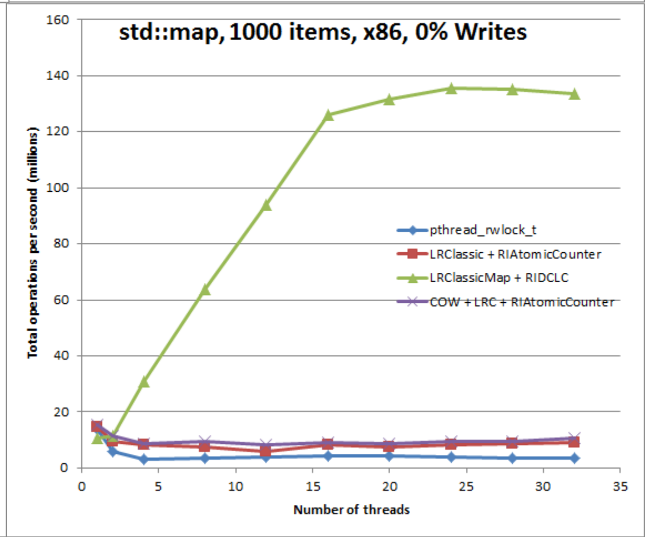






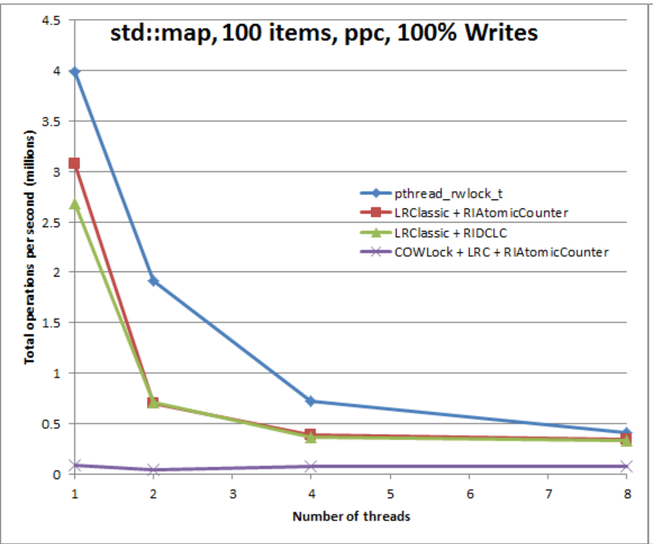


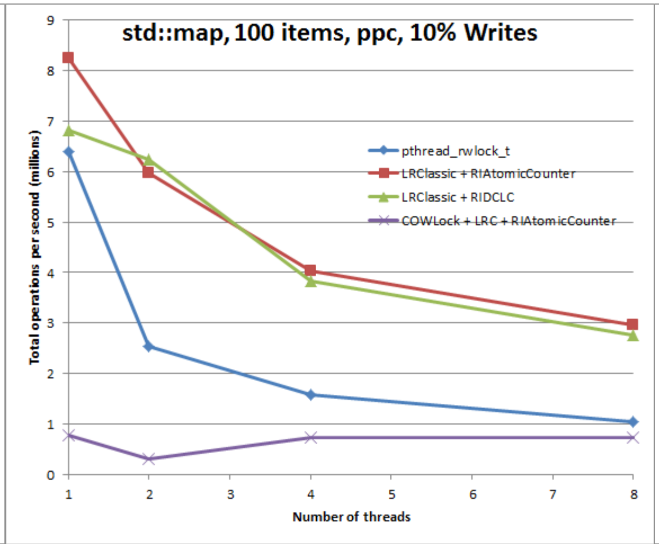


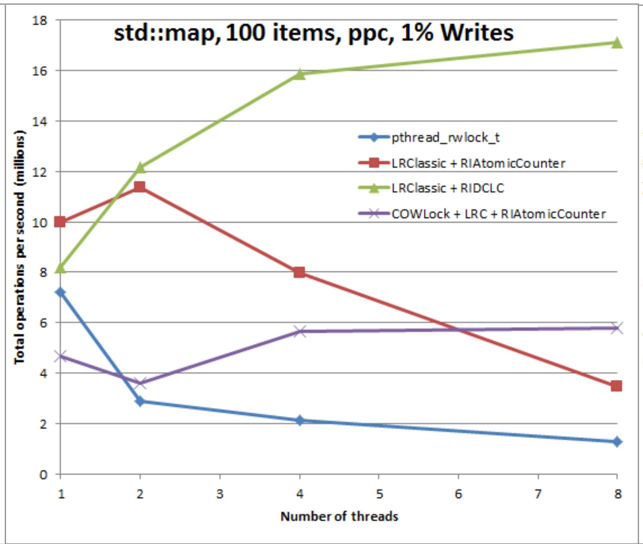


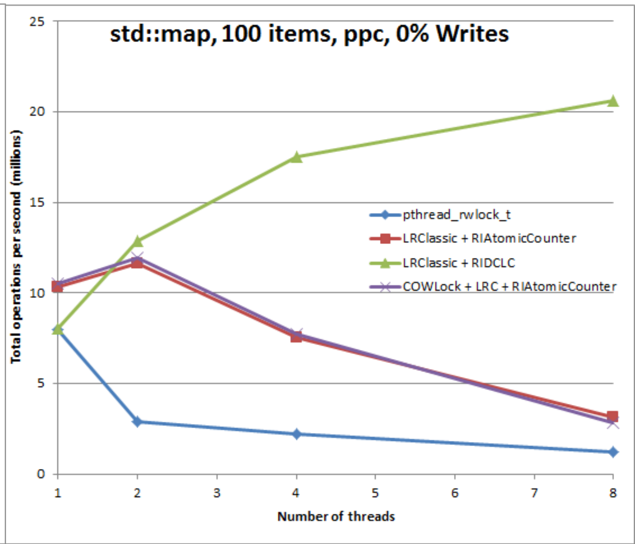
https://github.com/pramalhe/ConcurrencyFreaks/tree/master/CPP/trees

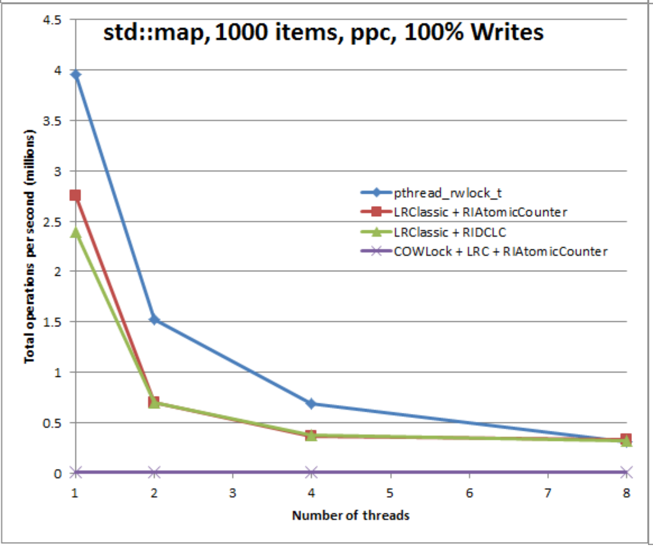
Benchmarks in C++1x – PowerPC with 8 cores Throughput

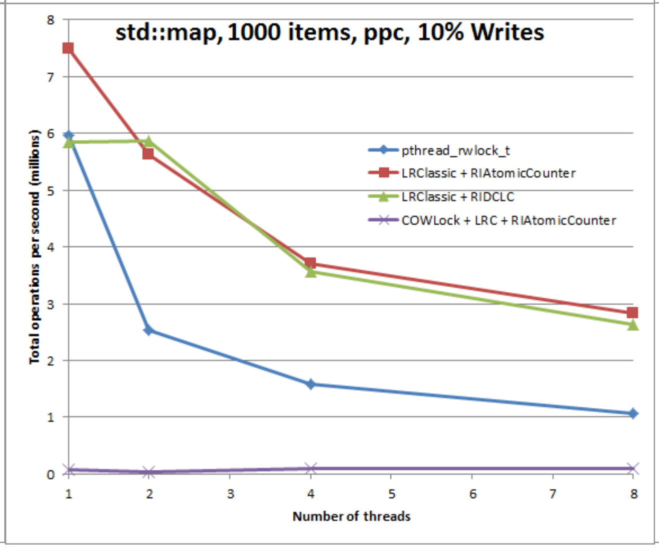


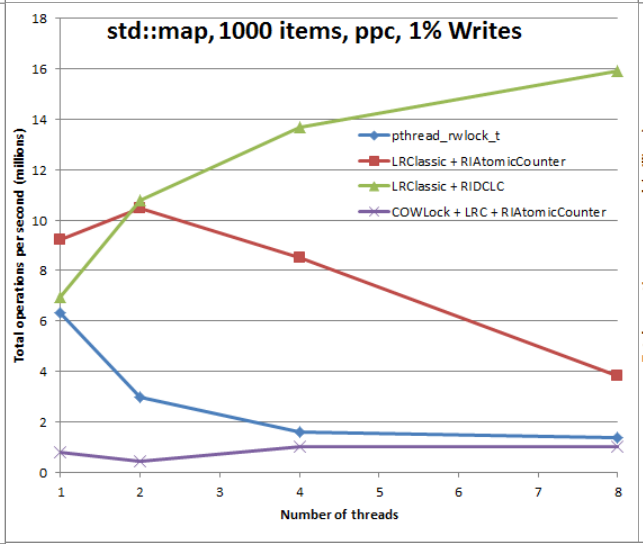


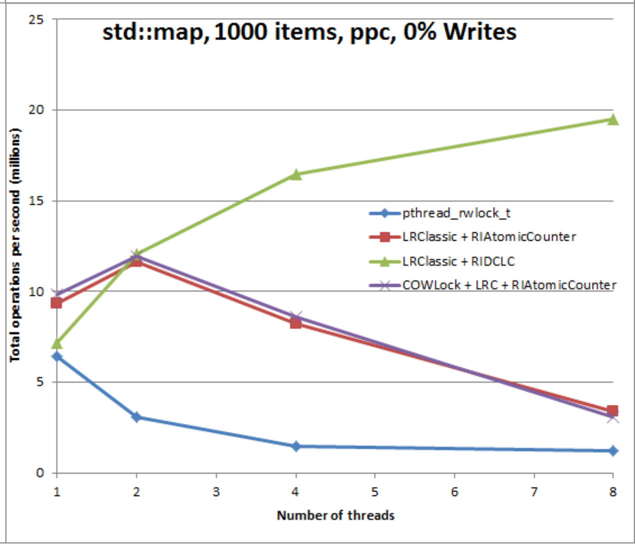












Benchmarks in C++1x Latency

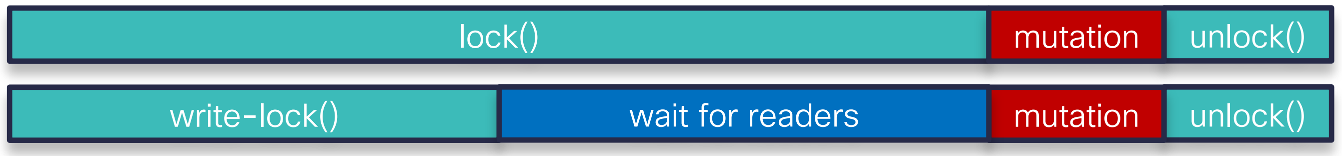
- Latency measurements using a std::map with 10000 (10k) items, measured over a period of 1000 seconds (16 minutes)
- Units are in microseconds (us)
- The table can be read as: *When using a Pthread rwlock, 99% of the calls to std::map::find() finish in less than 51 microseconds*
- The COW method for mutations is just too slow to measure because it takes such a long time copying the entire std::map (10000 nodes).
- Notice that although the Left-Right variants have to insert/erase on two instances, they finish quicker than a single instance protected with a pthread rwlock

Time (µs) of the *x*-th event	Pthread rwlock	Left-Right Atomic Counter	Left-Right Array of Counters	Copy-On-Write Userspace RCU
99% find()	51	2	2	2
99.9% find()	64	4	3	4
99.99% find()	74	8	7	7
99.999% find()	90	11	10	10
99.9999% find()	141	20	17	17
99% insert()	53	7	7	> 1000
99.9% insert()	77	15	17	> 1000
99.99% insert()	87	23	33	> 1000
99.999% insert()	104	35	75	> 1000
99.9999% insert()	167	99	120	> 1000
99% erase()	53	8	9	> 1000
99.9% erase()	70	16	19	> 1000
99.99% erase()	87	24	29	> 1000
99.999% erase()	104	36	86	> 1000
99.9999% erase()	179	85	129	> 1000

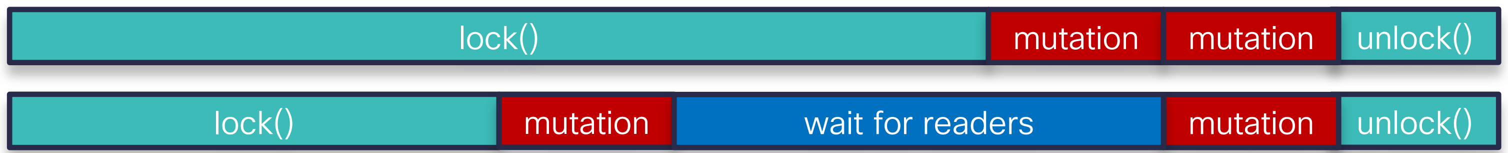
Benchmarks in C++1x
Non-obvious performance advantages

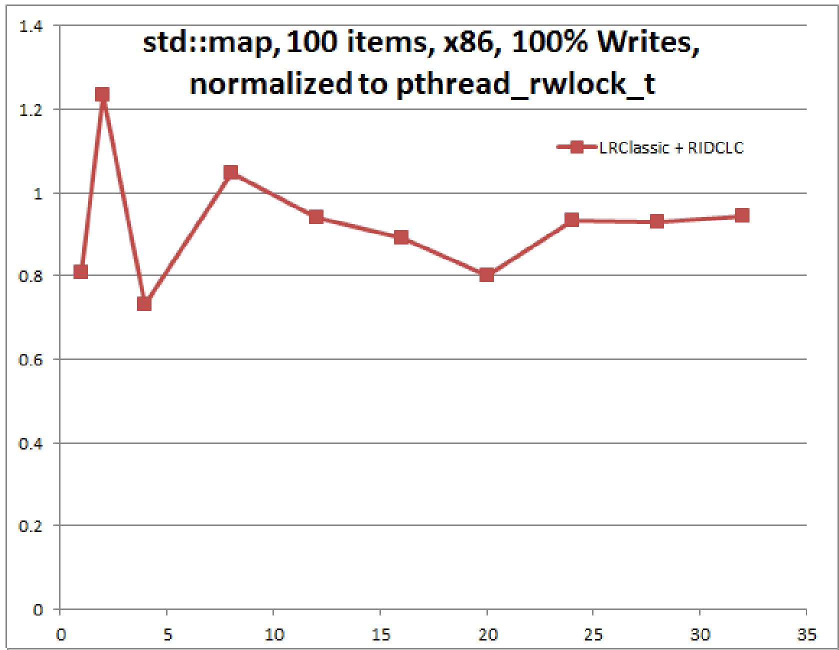
- An interesting fact to notice is that, when we're doing only mutations (100% Writes) **the throughput of Left-Right is not half of the Reader-Writer Lock**, even though Left-Right has to apply each mutation twice!
- This happens because the red-black tree used in std::map is not very big. Finding an item in the tree requires the traversal of about *log N* nodes.
- This traversal is so fast that doing it twice does not cause a significant impact, when compared with the time it takes to acquire/release the lock in write-mode.
- This can be true even with a larger tree, of 1 million nodes, which traverses 20 nodes. Same thing for using a `std::unordered_map` (an hash-table based data structure), which traverses O(1) nodes
- This effect will not occur if doing a mutative iteration over the entire data structure.

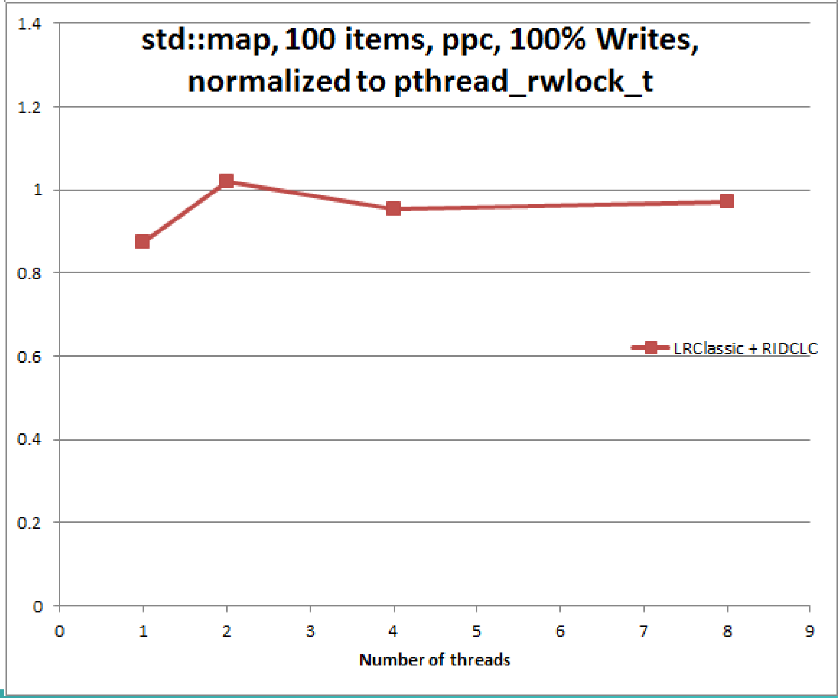
Mutative operation with **RW-Lock**



Mutative operation with **Left-Right**







Acquire-Loads
Comparison

- If we compare the number of fences done when **traversing** an Harris linked list (which has wait-free lookups), with the fences done by the Left-Right mechanism, we can see the immediate advantage.

- For Linked Lists like Harris's or Michael & Scott's, on ppc and arm architectures, we need to do a full fence per node. Even on a small list with just 100 nodes, that means doing **100** full fences, while for Left-Right we do **3** and only 3.

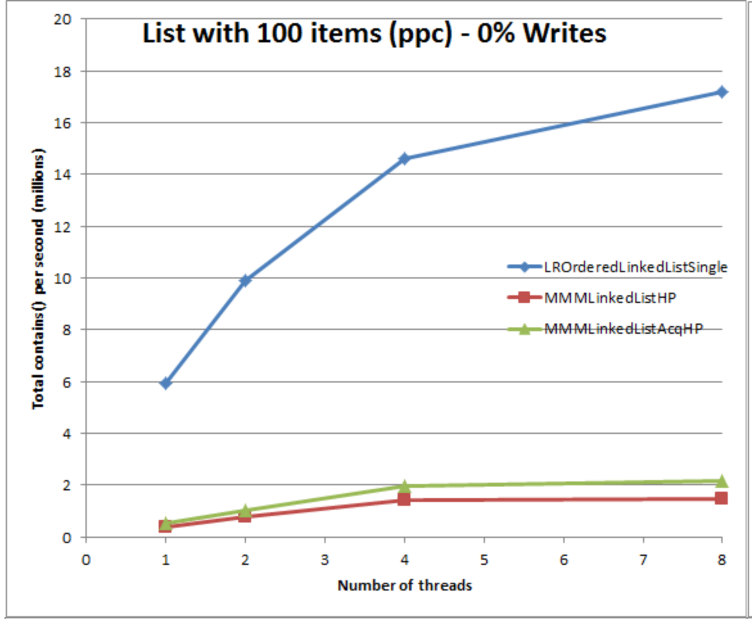
Linked List with N nodes	Fences on x86	Fences on PowerPC	Fences on ARMv7	Fences on ARMv8
Harris List	none	Nx isync	Nx isb	Nx LDAR
Michael & Scott List	none	Nx isync	Nx isb	Nx LDAR
List with Left-Right	2x MFENCE	3x hwsync	3x dmb	2x STLR + 1x LDAR
List with Copy-On-Write	none	1x hwsync	2x dmb	1x LDAR
List with RW-Lock	min 1x CAS	min 1x CAS + 1x hwsync	min 1x CAS + 1x dmb	min 1x CAS + 1x STLR

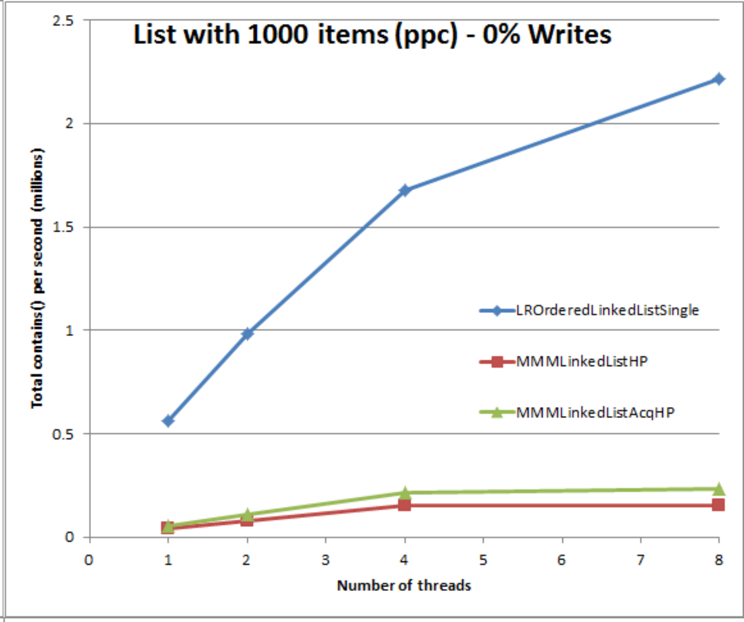
- On x86, if we use Hazard Pointers, then even for the Harris and Michael & Scott list we need to add an MFENCE per node. And if we use reference counting, we need to add two XADD per node.

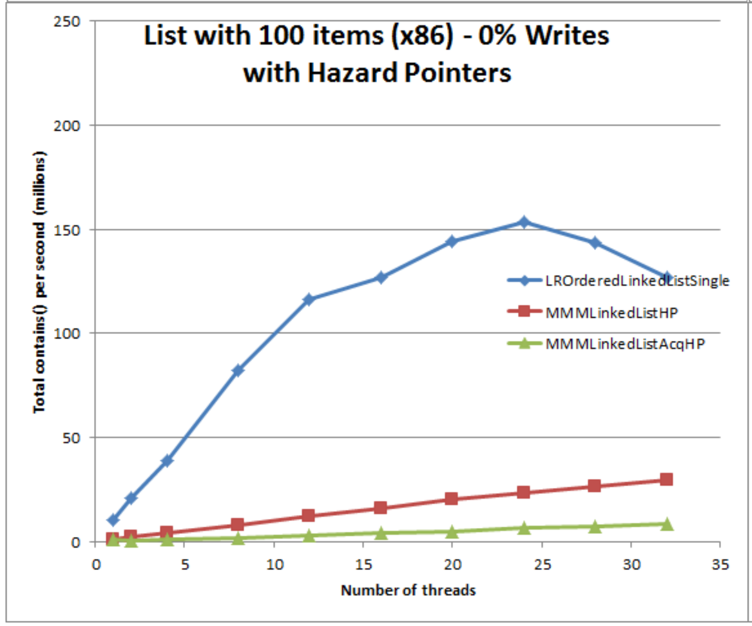
- Notice that the approaches with Harris, Michael & Scott, and COW, all need GC or a memory reclamation technique (Reference Counters, Hazard Pointers, RCU, etc), while Left-Right and RW-Lock do not.

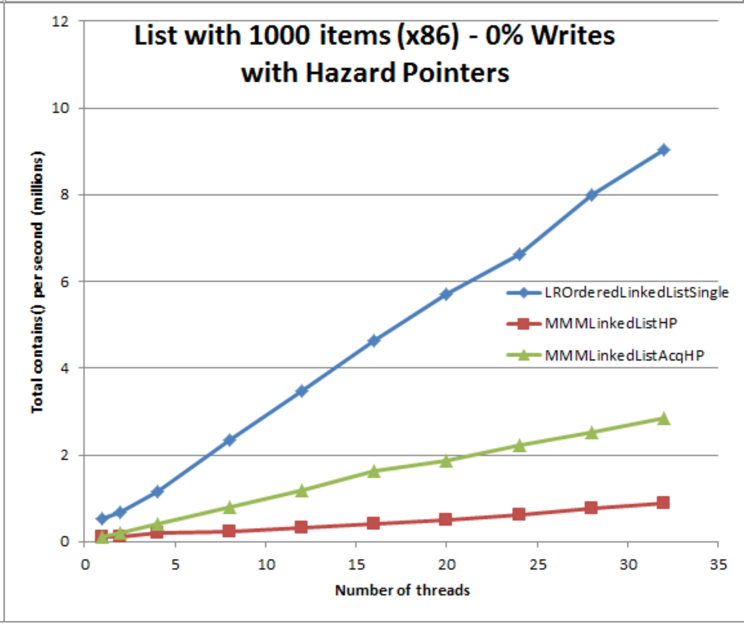
Acquire-Loads Benchmark

- In this plot we compare a linked list with Left-Right and Harris list doing only lookups (i.e. 0% writes) on x86 and on PowerPC, for lists with 100 and 1000.
- In this implementation we are not using any memory reclamation technique for Harris list. If we were to use such a technique, it would be slower on x86 (depending on which technique we use) and even slower on ppc
- The red line is from an implementation where the loads use `memory_order_seq_cst`, and the green line is for an implementation using `memory_order_acquire`.
- The plots on the right side show for PowerPC with a list of 100 and 1000 items respectively, how many lookups per second we can do on the list.









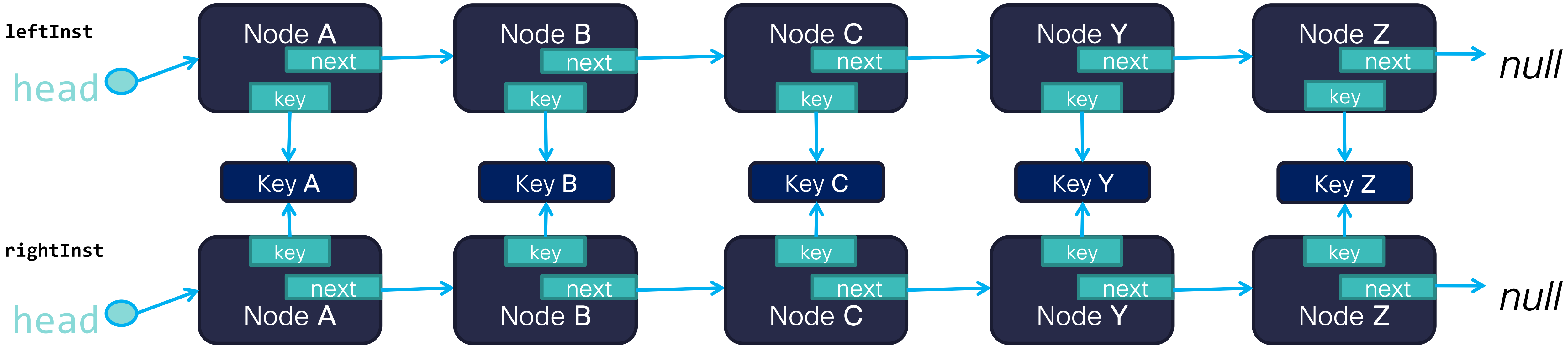
Why use Left-Right if we can use a Lock-Free linked list?

- There are a few Lock-Free linked list known, the most famous being Harris' linked list, which uses one bit of the reference to the next node to mark the current node as *logically removed*.
- To implement any Lock-Free data structure we need an automatic Garbage Collector, or a Lock-Free Memory Management technique
  - Even with GC, Harris' linked list requires some extensive modifications to be used
  - Hazard Pointers (HP) are easy to implement for a linked list, but can be really tricky (insanely hard) for more complex data structures
  - Hazard Pointers are not that fast, and they're patented by IBM
  - Reference counting is even slower than HP, and C++1x doesn't have support for atomic<unique_ptr>
- Linked lists aren't that fast anyways. Why not use an array instead? You can do it with the Left-Right pattern.
- The Left-Right pattern can be used for any data structure or user-defined class, and is easy to use because it has semantics close to a reader-writer lock

Left-Right Single instance Linked List

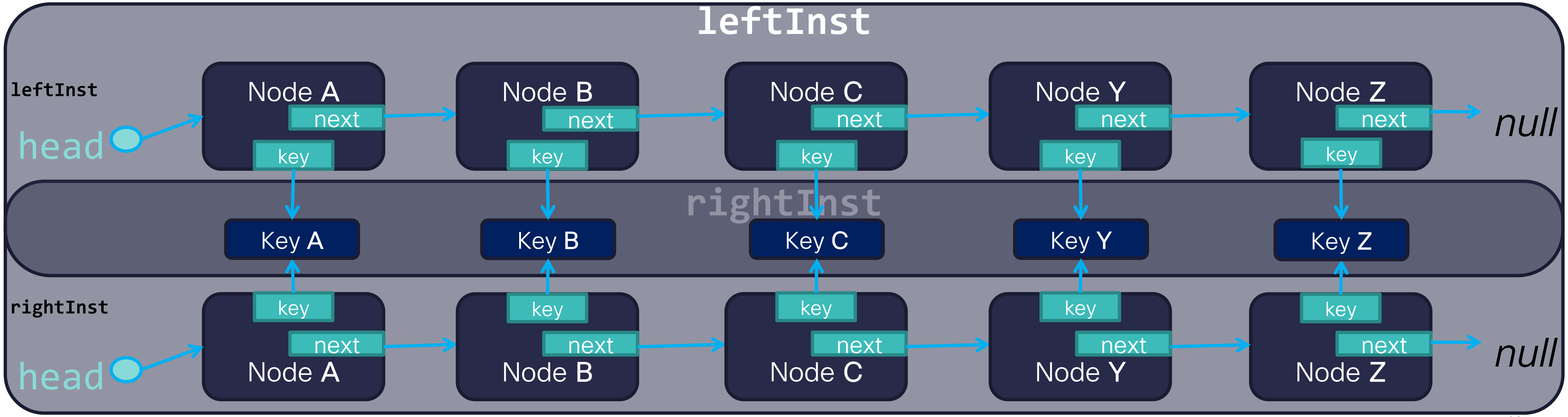
Left-Right Linked List

- The easiest way to make a linked list with wait-free lookups is to use a Left-Right pattern to protect two physical linked lists
- The basic idea is that read-only operations are going on one of the lists, while a single modifying thread is inserting/removing a node on the other list, and afterwards redirects new lookups to the recently modified list and waits for unfinished lookups before applying the same modification on the opposite list.
- Unlike Harris Linked List, there is no need for sentinel nodes at the `head` and `tail`. The `head` points to the first node on the list, and the last node on the list has a `next` that points to *null*.



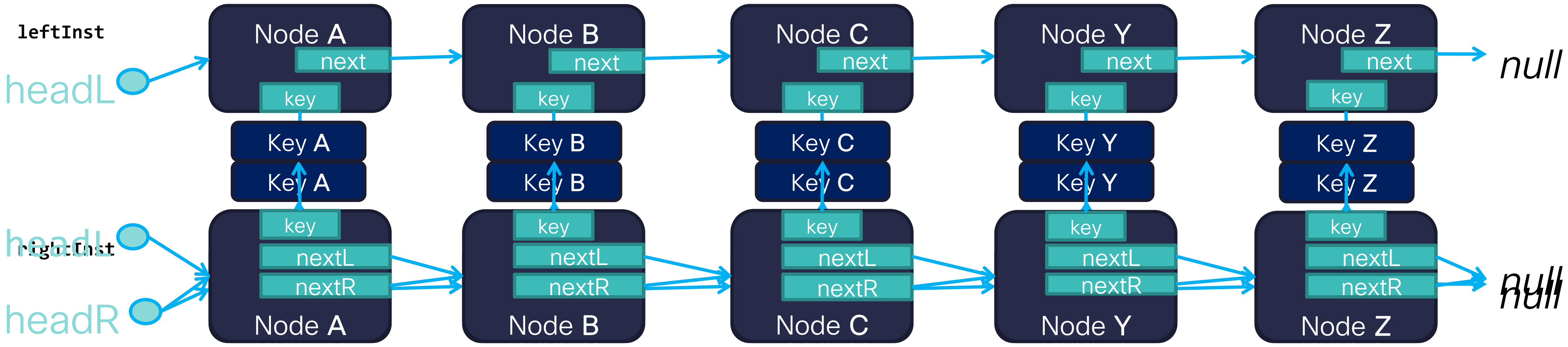
Left-Right Linked List
Two instances

- When applying the Left-Right pattern to a linked list (set) there are two physical linked lists but they point to the same keys
- Each node is composed of a pointer to the next node (`next`) and a pointer to the key (`key`)



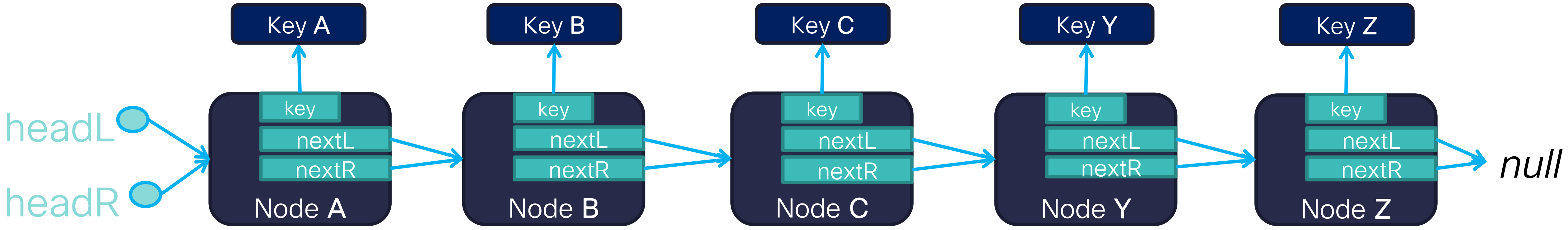
Left-Right Linked List Single

- Instead of having two physical lists, we can have a single physical list with two next pointers, one for the `leftInstance` logical list (named `nextL`) and another for the `rightInstance` logical list (named `nextR`)
- We will also need two heads, named `headL` and headR, one for each logical list



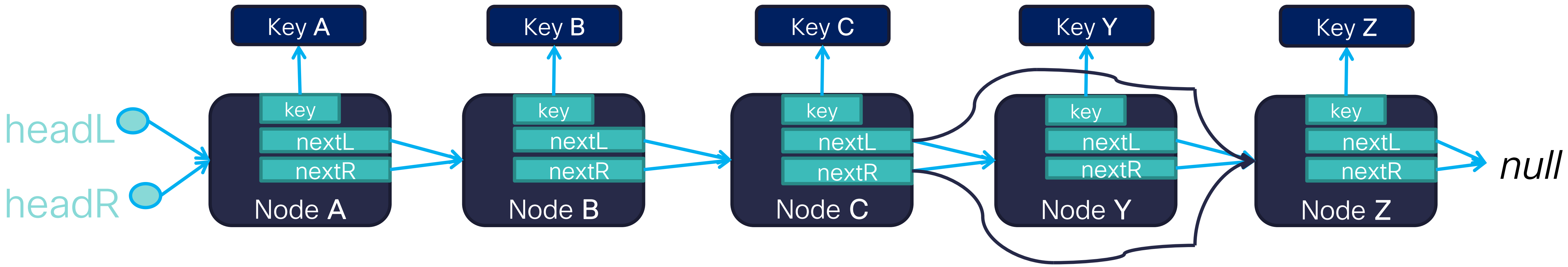
Left-Right Linked List Single

- Instead of having two physical lists, we can have a single physical list with two next pointers, one for the `leftInstance` logical list (named `nextL`) and another for the `rightInstance` logical list (named `nextR`)
- We will also need two heads, named `headL` and headR, one for each logical list



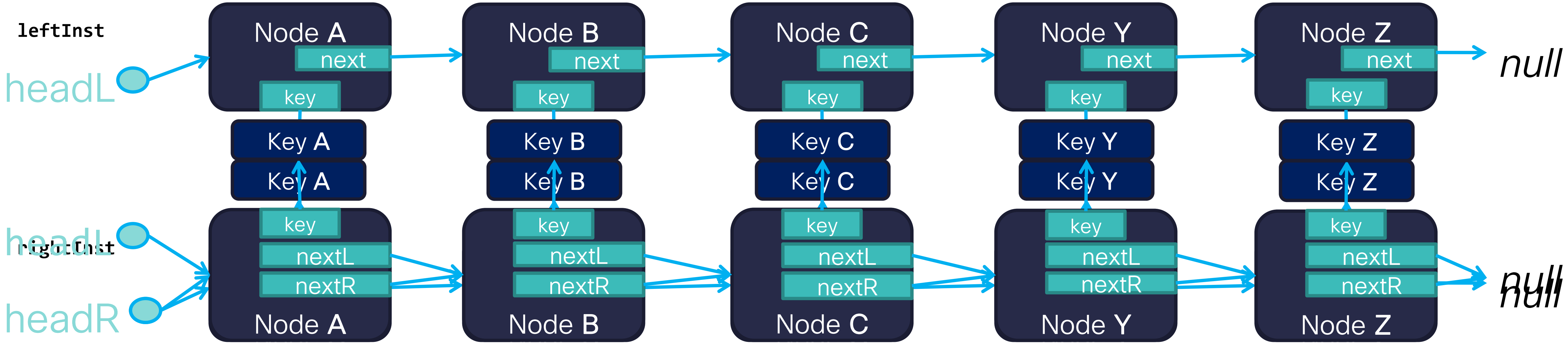
Left-Right Linked List Single `remove(Y)`

1. The Writer starts from the head of the logical list opposite to the one the Readers are using and unlinks the corresponding `next` field.
2. The Writer re-directs new Readers to the currently modified logical list, waits for older Readers to finish on the other logical list, and then unlinks the other `next` field.
3. The node with the key Y can now be deleted/free'ed



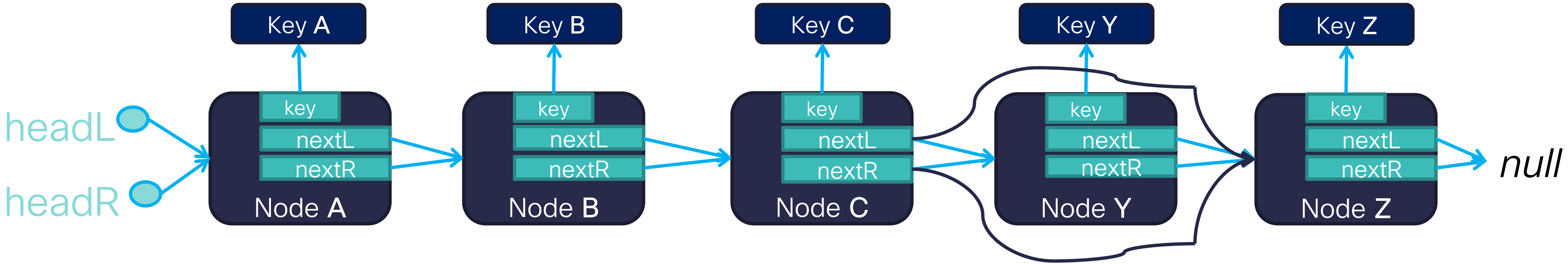
Left-Right Linked List Single

- Instead of having two physical lists, we can have a single physical list with two next pointers, one for the `leftInstance` logical list (named `nextL`) and another for the `rightInstance` logical list (named `nextR`)
- We will also need two heads, named `headL` and headR, one for each logical list



Left-Right Linked List Single `remove(Y)`

1. The Writer starts from the head of the logical list opposite to the one the Readers are using and unlinks the corresponding `next` field.
2. The Writer re-directs new Readers to the currently modified logical list, waits for older Readers to finish on the other logical list, and then unlinks the other `next` field.
3. The node with the key Y can now be deleted/free'ed



Left-Right vs RCU

and other techniques

What the Left-Right pattern is **not**

- Although they may seem similar, Left-Right and RCU are not the same thing

  https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt

  https://lwn.net/Articles/262464/

- RCU can work with the Copy-On-Write (COW) pattern, and hence the name RCU: Read-Copy-Update. RCU is (among many other things) a memory reclamation technique that can be used with COW.
- RCU is meant to be used with some other technique that provides mutual exclusivity (like COW) or as a memory reclamation mechanism for a data-structure that is already lock-free/wait-free, i.e. uses atomics.
- The Left-Right can be implemented in any language (preferably with a Memory Model), while RCU requires Kernel support. However, a few variants of Userspace-RCU can also be deployed generically, namely the General Purpose and Bullet-Proof variants.
- Left-Right uses two instances. RCU doesn't have *instances*.
- The data structures that you use with Left-Right are *single-threaded*, which means they don't use atomics.

- … However, we can replace the algorithm in the Left-Right pattern with one of the algorithms used by RCU, and vice-versa.

Left-Right vs RCU

- RCU has an API with **5 core methods** while Left-Right has an API with **3 methods**, but any of the algorithms used by RCU can be used as replacement algorithms for the Left-Right pattern.
- The problem is that most of the algorithms for RCU and Userspace RCU need some Kernel/OS support, so they aren't truly generic. The exception to this is "Userspace-RCU **Bullet Proof**", and "Userspace-RCU **Memory Barriers**" as well (as long as each thread can register/unregister).
- As far as the core API is concerned, the Left-Right and RCU algorithms are ***semantically interchangeable***.

RCU Algorithms

```
rcu_read_lock()
rcu_read_unlock()
synchronize_rcu()
rcu_assign_pointer()
rcu_dereference()
```

Left-Right Algorithms

```
arrive()
depart()
toggleVersionAndWait()
```

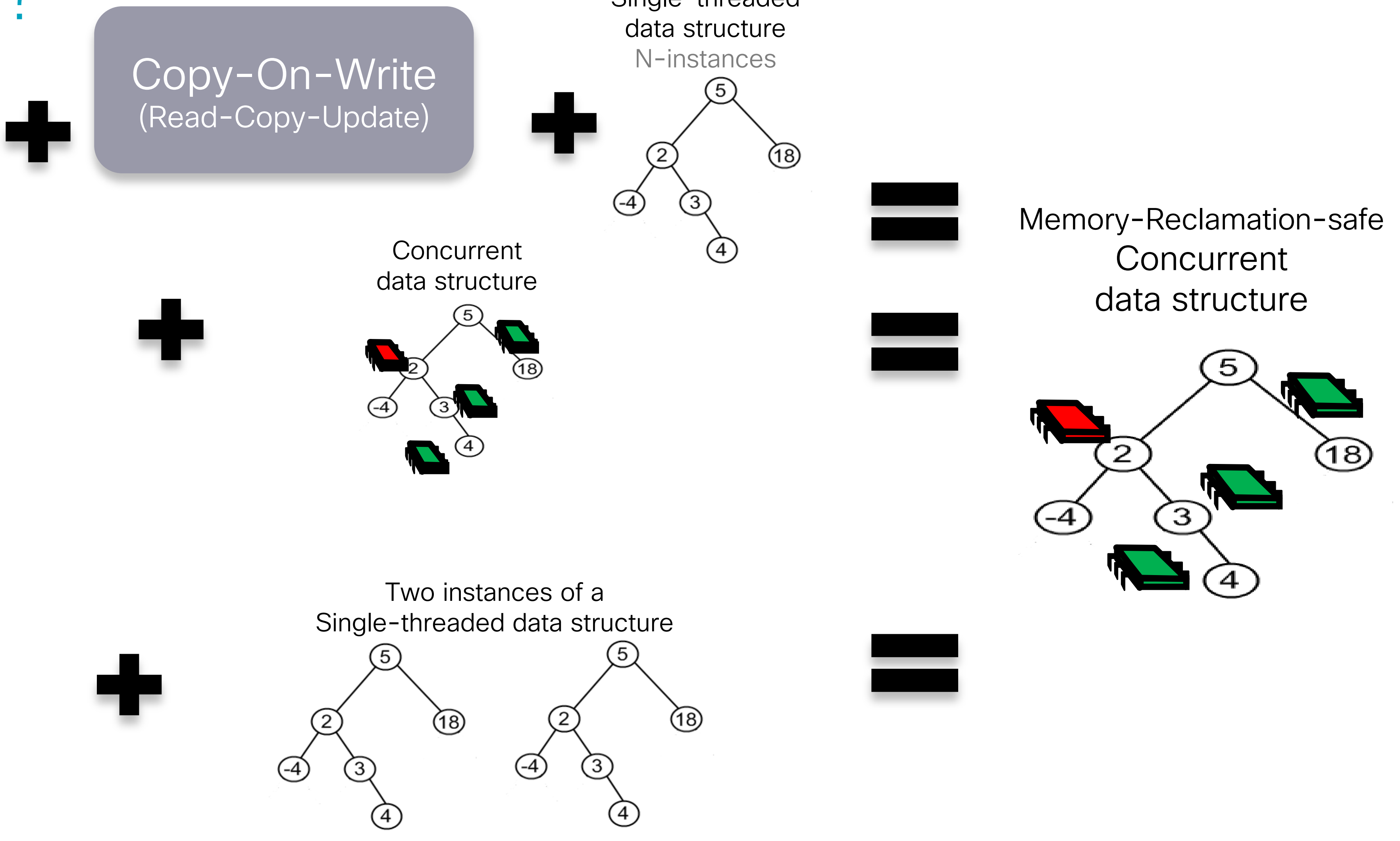
Note: to be equivalent to synchronize_rcu() we should call in sequence: `writersMutex.lock(); togglerVersionAndWait(); writersMutex.unlock();`

Algorithms for RCU and Left-Right
What are they for?

RCU +Left-Right Algorithms:

- RCU (requires kernel support)
- Userspace RCU Quiescent State
- Userspace RCU Signal
- Userspace RCU Memory Barriers
- Userspace RCU Bullet Proof
- LR Classical
- LR Readers Version
- LR No Version
- LR Atomic Long
- LR Atomic Long No Version
- LR GT
- LR Scalable GT
- LR David GoldBlatt's variant

- David GoldBlatt's variant



Keep in mind that of the RCU algorithms, not all can be implemented in portable C++1x code

Comparison table of some generic techniques

- Left-Right is the only *generic* linearizable technique to provide mutual exclusion that is easy to implement in C/C++ while providing high scalability and low latency for read-only operations.
- Keep in mind that there are other (somewhat) generic techniques, but the ones described below are the easiest ones to deploy and use

Technique	Progress for read-only operations	Progress for mutative operations	Memory usage	Obs
Mutual exclusion Lock	Blocking	Blocking	1 instance	No concurrency
Reader-Writer Lock	Blocking	Blocking	1 instance	Simultaneous Readers
Double Instance Locking	Lock-Free	Blocking	2 instances	2x the work of the mutation
Actors	Blocking	Blocking	1 instance	Can be made wait-free for readers, but loses seq-cst (not linearizable)
Copy-On-Write + RCU	WFPO	Blocking	1 or 2 instances for RCU, N or more instances for HPs	Churn on the memory allocation system and caches Have to (shallow) copy the entire data structure for a simple mutation
Left-Right	WFPO	Blocking	2 instances	2x the work of the mutation

Takeaways

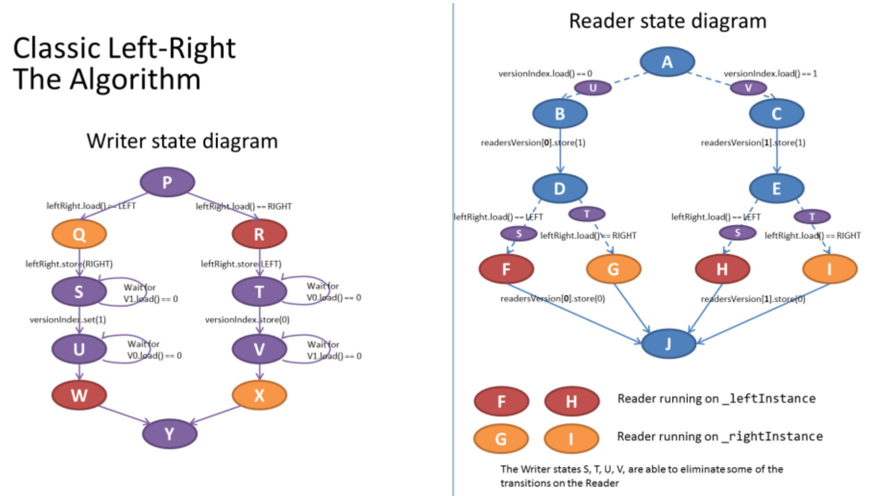
- The Left-Right Pattern provides mutual exclusivity for any object or data structure, with wait-free read-only access, at the cost of twice the memory usage

- Any of the algorithms used in the Left-Right Pattern is interchangeable with any of the RCU algorithms (with some limitations), and just like RCU, any of the Left-Right algorithms can be used for thread-safe memory management.

- With the Left-Right Pattern you can make any (single-threaded) data structure scale almost linearly for read-only operations, and have low-latency

- Less than 50 lines of code for a full implementation.

- … and did I say it is wait-free population oblivious ? ☺



Thank you.

Visit our blog at
http://www.concurrencyfreaks.com