WTF::Lock

Filip Pizlo
Apple

- WTF stands for Web Template Framework.
- You can pronounce it in whatever way feels most natural.

WTF::Lock

- Goals
- How Locks Work
- ParkingLot API
- WTF::Lock algorithm
- ParkingLot algorithm
- Conclusion

WTF::Lock

- Goals
- How Locks Work
- ParkingLot API
- WTF::Lock algorithm
- ParkingLot algorithm
- Conclusion

Goals

- Fit in two bits (or one byte)
- Single CAS fast paths
- Fast microcontention
- No unbounded spinning
- Stochastically fair

WTF::Lock

WTF::Condition

WTF::Lock

- Goals
- How Locks Work
- ParkingLot API
- WTF::Lock algorithm
- ParkingLot algorithm
- Conclusion

```
class SpinLock {
public:
    void lock()
    {
        while (!m_byte.compareExchangeWeak(0, 1)) { }
    }

    void unlock()
    {
        m_byte.store(0);
    }

private:
    Atomic<uint8_t> m_byte;
};
```

```
class SpinLock {
public:
    void lock()
    {
        while (!m_byte.compareExchangeWeak(0, 1))
            std::this_thread::yield();
    }

    void unlock()
    {
        m_byte.store(0);
    }

private:
    Atomic<uint8_t> m_byte;
};
```

```
class SpinLock {
public:
    void lock()
    {
        while (!m_byte.compareExchangeWeak(0, 1))
            std::this_thread::yield();
    }

    void unlock()
    {
        m_byte.store(0);
    }

private:
    Atomic<uint8_t> m_byte;
};
```

How Locks Work

at a really high level

- Locks have two parts:
  - Somewhere in memory to tell if the lock is held.
  - Some mechanism for *parking* threads that want to suspend instead of spinning.

```
class ThunderLock {
public:
    void lock()
    {
        while (!m_byte.compareExchangeWeak(0, 1))
            park();
    }

    void unlock()
    {
        m_byte.store(0);
        unparkAll();
    }

private:
    Atomic<uint8_t> m_byte;
};
```

```
class ThunderLock {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock()
    {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                park();
                continue;
            }
            if (m_byte.compareExchangeWeak(
                    oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock()
    {
        uint8_t oldValue = m_byte.exchange(0);
        if (oldValue & hasParked)
            unparkAll();
    }
private:
    Atomic<uint8_t> m_byte;
};
```

```
class ThunderLock {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock()
    {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                park();
                continue;
            }
            if (m_byte.compareExchangeWeak(
                    oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock()
    {
        uint8_t oldValue = m_byte.exchange(0);
        if (oldValue & hasParked)
            unparkAll();
    }
private:
    Atomic<uint8_t> m_byte;
};
```

```
class ThunderLock {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock()
    {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                park();
                continue;
            }
            if (m_byte.compareExchangeWeak(
                    oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock()
    {
        uint8_t oldValue = m_byte.exchange(0);
        if (oldValue & hasParked)
            unparkAll();
    }
private:
    Atomic<uint8_t> m_byte;
};
```

```
class ThunderLock {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock()
    {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                park();
                continue;
            }
            if (m_byte.compareExchangeWeak(
                    oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock()
    {
        uint8_t oldValue = m_byte.exchange(0);
        if (oldValue & hasParked)
            unparkAll();
    }
private:
    Atomic<uint8_t> m_byte;
};
```

```
class ThunderLock {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock()
    {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                park();
                continue;
            }
            if (m_byte.compareExchangeWeak(
                    oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock()
    {
        uint8_t oldValue = m_byte.exchange(0);
        if (oldValue & hasParked)
            unparkAll();
    }
private:
    Atomic<uint8_t> m_byte;
};
```

```
class ThunderLock {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock()
    {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                park();
                continue;
            }
            if (m_byte.compareExchangeWeak(
                    oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock()
    {
        uint8_t oldValue = m_byte.exchange(0);
        if (oldValue & hasParked)
            unparkAll();
    }
private:
    Atomic<uint8_t> m_byte;
};
```

```
class ThunderLock {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock()
    {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                park();
                continue;
            }
            if (m_byte.compareExchangeWeak(
                    oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock()
    {
        uint8_t oldValue = m_byte.exchange(0);
        if (oldValue & hasParked)
            unparkAll();
    }
private:
    Atomic<uint8_t> m_byte;
};
```

WTF::Lock

- Goals
- How Locks Work
- ParkingLot API
- WTF::Lock algorithm
- ParkingLot algorithm
- Conclusion

WTF::ParkingLot

- Heavily inspired by Linux's futexes:
  - Franke and Russel. "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux"
- Has since inspired Rust's parking_lot:
  - https://github.com/Amanieu/parking_lot

- Say you have a pointer to a byte.
- How much information is that?

- Say you have a pointer to a byte.
- How much information is that?

Is it 8 bits?

- Say you have a pointer to a byte.
- How much information is that?

~~*Is it 8 bits?*~~

- Say you have a pointer to a byte.
- How much information is that?

48 immutable bits + 8 mutable bits

- Say you have a pointer to a byte.
- How much information is that?

48 immutable bits + 8 mutable bits

- Say you have a pointer to a byte.
- How much information is that?

48 immutable bits + 8 mutable bits

Use the pointer as a key to a concurrent hashtable!

- Address is a key for a queue of threads.
- You can *park* and *unpark* threads on the queue.
- The queue has a lock.

```
class ParkingLot {
    static ParkResult parkConditionally(
        const void* address,
        const ScopedLambda<bool()>& validation,
        const ScopedLambda<void()>& beforeSleep,
        const TimeWithDynamicClockType& timeout);

    static void unparkOne(
        const void* address,
        const ScopedLambda<
            intptr_t(UnparkResult)>& callback);

    static unsigned unparkCount(
        const void* address,
        unsigned count);

    …
};
```

```
class ParkingLot {
    static ParkResult parkConditionally(
        const void* address,
        const ScopedLambda<bool()>& validation,
        const ScopedLambda<void()>& beforeSleep,
        const TimeWithDynamicClockType& timeout);

    static void unparkOne(
        const void* address,
        const ScopedLambda<
            intptr_t(UnparkResult)>& callback);

    static unsigned unparkCount(
        const void* address,
        unsigned count);

    …
};
```

```
class ParkingLot {
    static ParkResult parkConditionally(
        const void* address,
        const ScopedLambda<bool()>& validation,
        const ScopedLambda<void()>& beforeSleep,
        const TimeWithDynamicClockType& timeout);

    static void unparkOne(
        const void* address,
        const ScopedLambda<
            intptr_t(UnparkResult)>& callback);

    static unsigned unparkCount(
        const void* address,
        unsigned count);

    …
};
```

```
class ParkingLot {
    static ParkResult parkConditionally(
        const void* address,
        const ScopedLambda<bool()>& validation,
        const ScopedLambda<void()>& beforeSleep,
        const TimeWithDynamicClockType& timeout);

    static void unparkOne(
        const void* address,
        const ScopedLambda<
            intptr_t(UnparkResult)>& callback);

    static unsigned unparkCount(
        const void* address,
        unsigned count);

    …
};
```

```
class ParkingLot {
    static ParkResult parkConditionally(
        const void* address,
        const ScopedLambda<bool()>& validation,
        const ScopedLambda<void()>& beforeSleep,
        const TimeWithDynamicClockType& timeout);

    static void unparkOne(
        const void* address,
        const ScopedLambda<
            intptr_t(UnparkResult)>& callback);

    static unsigned unparkCount(
        const void* address,
        unsigned count);

    …
};
```

- Parking means running the *validation* lambda while the queue lock is held.
- Enqueue happens if validation returns true.

- Unparking means locking the queue lock, attempting dequeue a thread, and running running the *callback* lambda while the queue lock is held.

- The callback gets to know if a thread was dequeued as well as some queue stats.

- Unparking thread can send a *token* (a intptr_t) to the unparked thread.
- Unparked thread always knows if it was unparked for real (as opposed to timeout or validation failure).

What can you do with ParkingLot?

- WTF::Lock
- WTF::Condition
- Lots of other stuff
  - See ToyLocks.h in WebKit/Source/WTF/benchmarks
  - See Rust's parking_lot

WTF::Lock Algorithm

Goal	Technique
Fit in two bits in a byte	ParkingLot manages most of the state
Single CAS fast paths	Bits indicate if fast paths succeed
Fast microcontention	Spin 40 times while calling std::this_thread_yield()
No unbounded spinning	ParkingLot manages blocked threads
Stochastically fair	ParkingLot sends token from unparker to unparkee

WTF::Lock Algorithm

Goal	Technique
Fit in two bits in a byte	ParkingLot manages most of the state
Single CAS fast paths	Bits indicate if fast paths succeed
Fast microcontention	Spin 40 times while calling std::this_thread_yield()
No unbounded spinning	ParkingLot manages blocked threads
Stochastically fair	ParkingLot sends token from unparker to unparkee

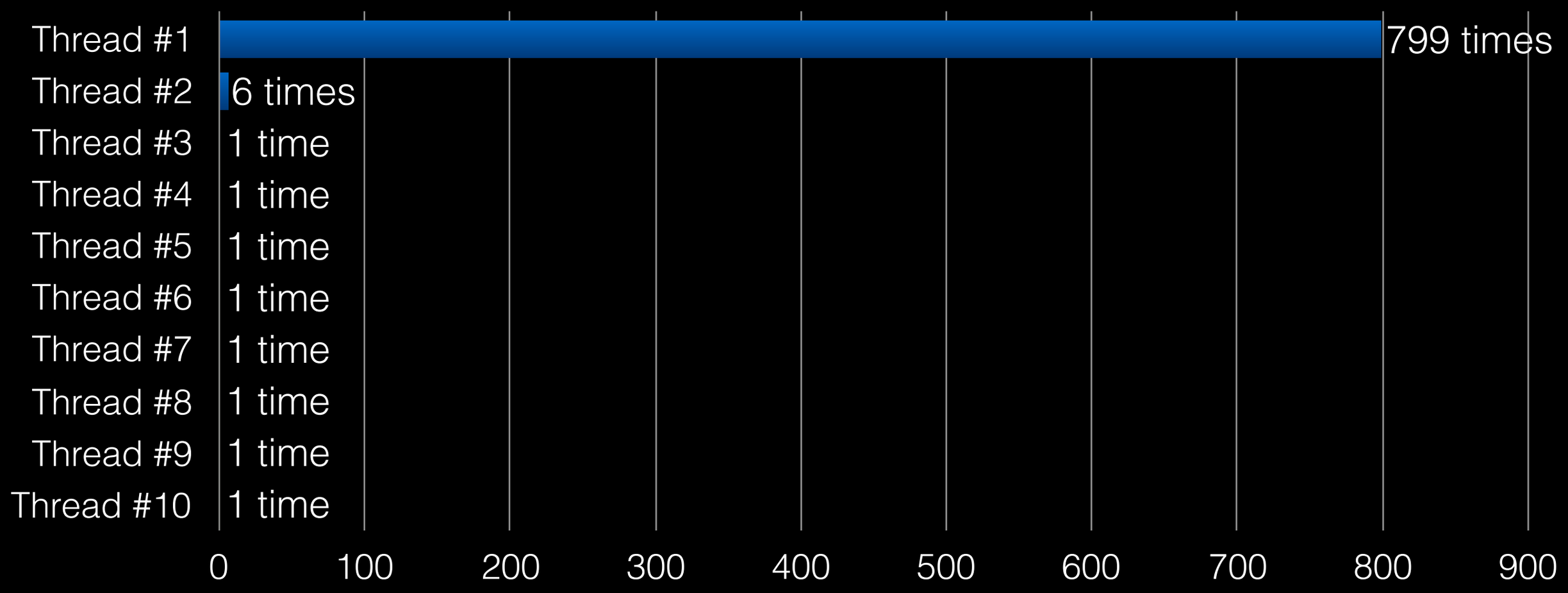
```
class WTF::Lock /* without fairness */ {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock() {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                parkConditionally(&m_byte, [&] () -> bool {
                    return m_byte.load() == isHeld | hasParked;
                });
                continue;
            }
            if (m_byte.compareExchangeWeak(oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock() {
        if (m_byte.compareExchangeWeak(oldValue, isHeld, 0)) return;
        ParkingLot::unparkOne(&m_byte, [&] (UnparkResult result) -> intptr_t {
            m_byte = result.mayHaveMoreThreads ? hasParked : 0;
            return 0;
        });
    }
private:
    Atomic<uint8_t> m_byte;
};
```

```
class WTF::Lock /* without fairness */ {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock() {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                parkConditionally(&m_byte, [&] () -> bool {
                    return m_byte.load() == isHeld | hasParked;
                });
                continue;
            }
            if (m_byte.compareExchangeWeak(oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock() {
        if (m_byte.compareExchangeWeak(oldValue, isHeld, 0)) return;
        ParkingLot::unparkOne(&m_byte, [&] (UnparkResult result) -> intptr_t {
            m_byte = result.mayHaveMoreThreads ? hasParked : 0;
            return 0;
        });
    }
private:
    Atomic<uint8_t> m_byte;
};
```

```
class WTF::Lock /* without fairness */ {
    static const uint8_t isHeld    = 1 << 0;
    static const uint8_t hasParked = 1 << 1;
public:
    void lock() {
        for (;;) {
            uint8_t oldValue = m_byte.loadRelaxed();
            if (oldValue & isHeld) {
                m_byte.exchangeOr(hasParked);
                parkConditionally(&m_byte, [&] () -> bool {
                    return m_byte.load() == isHeld | hasParked;
                });
                continue;
            }
            if (m_byte.compareExchangeWeak(oldValue, oldValue | isHeld))
                return;
        }
    }
    void unlock() {
        if (m_byte.compareExchangeWeak(oldValue, isHeld, 0)) return;
        ParkingLot::unparkOne(&m_byte, [&] (UnparkResult result) -> intptr_t {
            m_byte = result.mayHaveMoreThreads ? hasParked : 0;
            return 0;
        });
    }
private:
    Atomic<uint8_t> m_byte;
};
```

- This algorithm is enough to outperform system mutexes.
  - Always outperforms the system in space.
  - Depending on the system, may outperform in time.
- *But it's unfair!*
  - *Sometimes a thread will never get the lock.*

- Launch 10 threads.
- for (1 second) { lock(); sleep(1 ms); unlock(); }
- Number of times each thread got the lock:



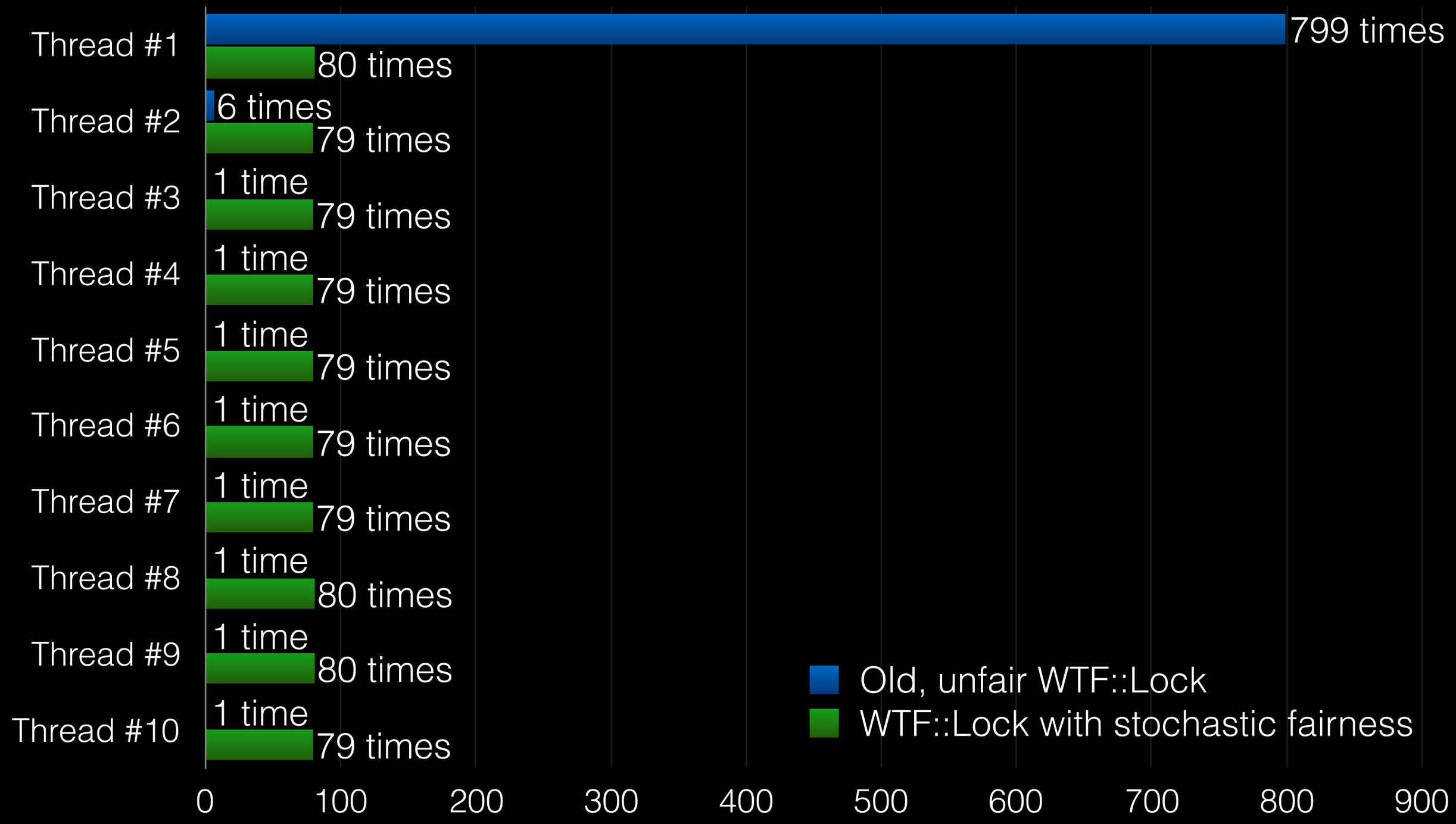
```
enum Token {
    BargingOpportunity,
    DirectHandoff
};
```

```
ParkingLot::ParkResult parkResult =
    parkConditionally(&m_byte, [&] () -> bool {
        return m_byte.load() == isHeld | hasParked;
    });
if (parkResult.wasUnparked) {
    switch (static_cast<Token>(parkResult.token)) {
    case DirectHandoff:
        RELEASE_ASSERT(isLocked(lock));
        return; // We're done!
    case BargingOpportunity:
        break; // Contend for the lock again
    }
} // else contend for the lock again
```

```
ParkingLot::unparkOne(
    &m_byte,
    [&] (ParkingLot::UnparkResult result) -> intptr_t {
        if (result.didUnparkThread && result.timeToBeFair)
            return DirectHandoff;
        m_byte = result.mayHaveMoreThreads ? hasParked : 0;
        return BargingOpportunity;
    });
```

```
ParkingLot::unparkOne(
    &m_byte,
    [&] (ParkingLot::UnparkResult result) -> intptr_t {
        if (result.didUnparkThread && result.timeToBeFair)
            return DirectHandoff;
        m_byte = result.mayHaveMoreThreads ? hasParked : 0;
        return BargingOpportunity;
    });
```

- Launch 10 threads.
- for (1 second) { lock(); sleep(1 ms); unlock(); }
- Number of times each thread got the lock:



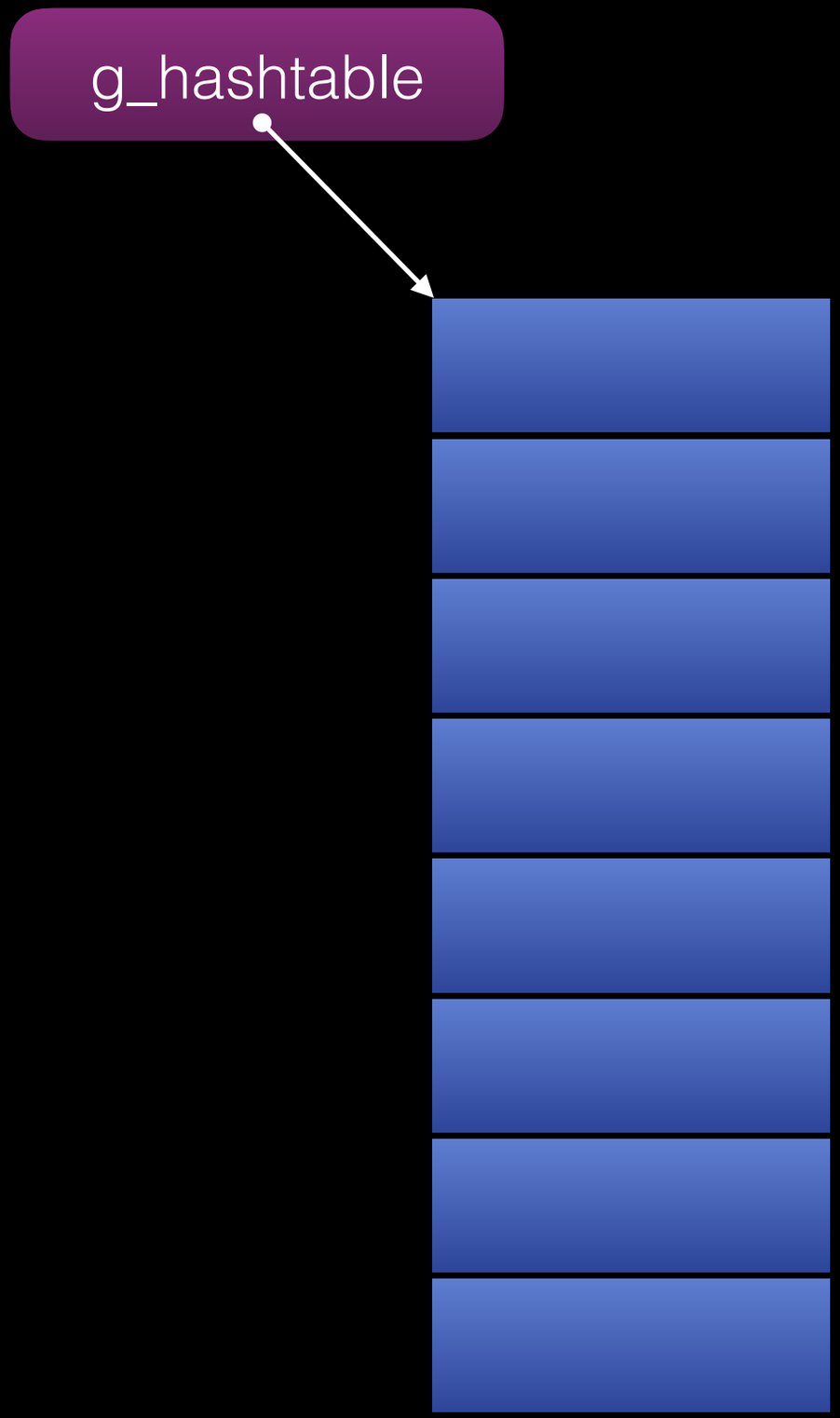
WTF::Lock Algorithm

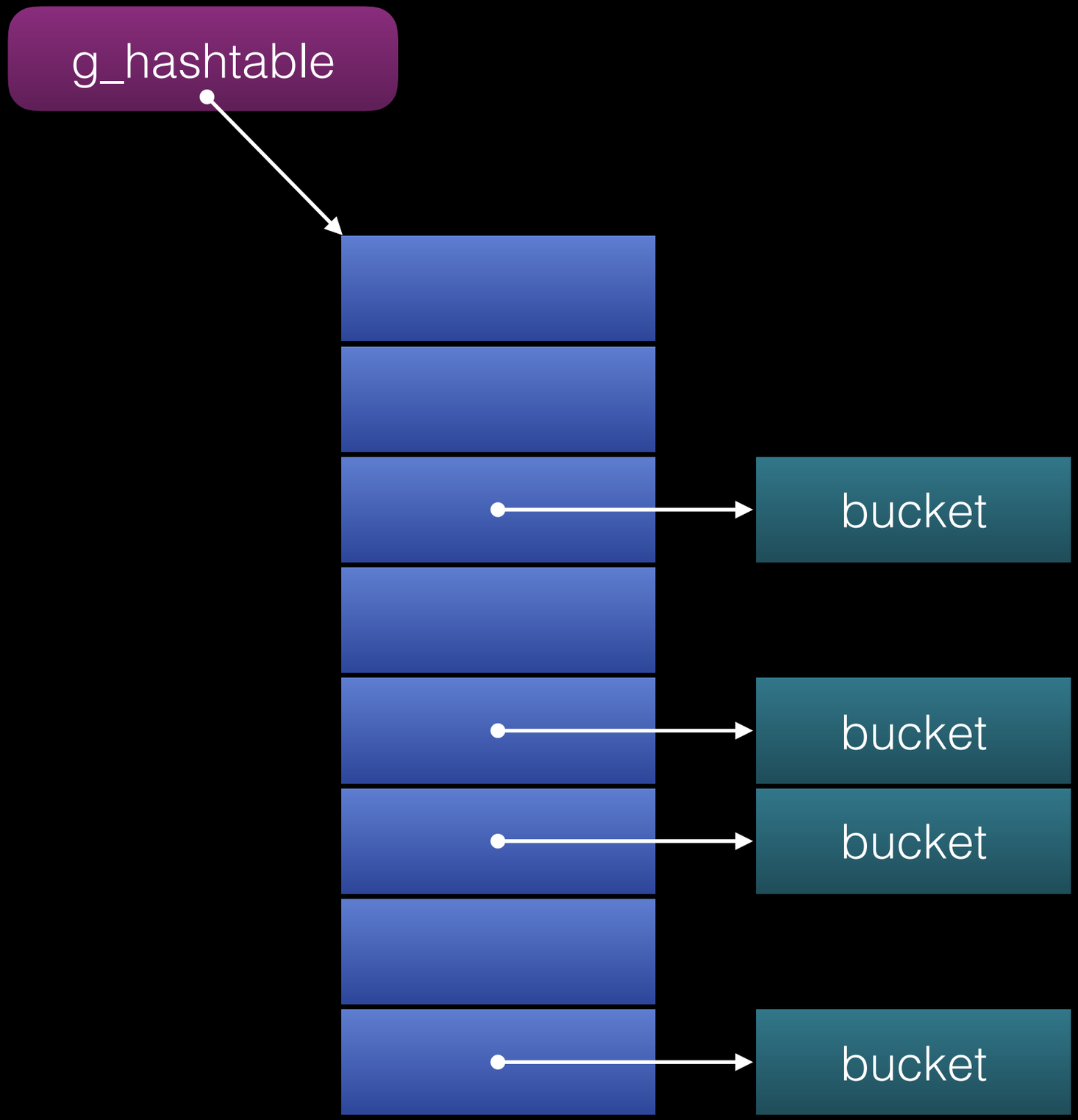
Goal	Technique
Fit in two bits in a byte	ParkingLot manages most of the state
Single CAS fast paths	Bits indicate if fast paths succeed
Fast microcontention	Spin 40 times while calling std::this_thread_yield()
No unbounded spinning	ParkingLot manages blocked threads
Stochastically fair	ParkingLot sends token from unparker to unparkee

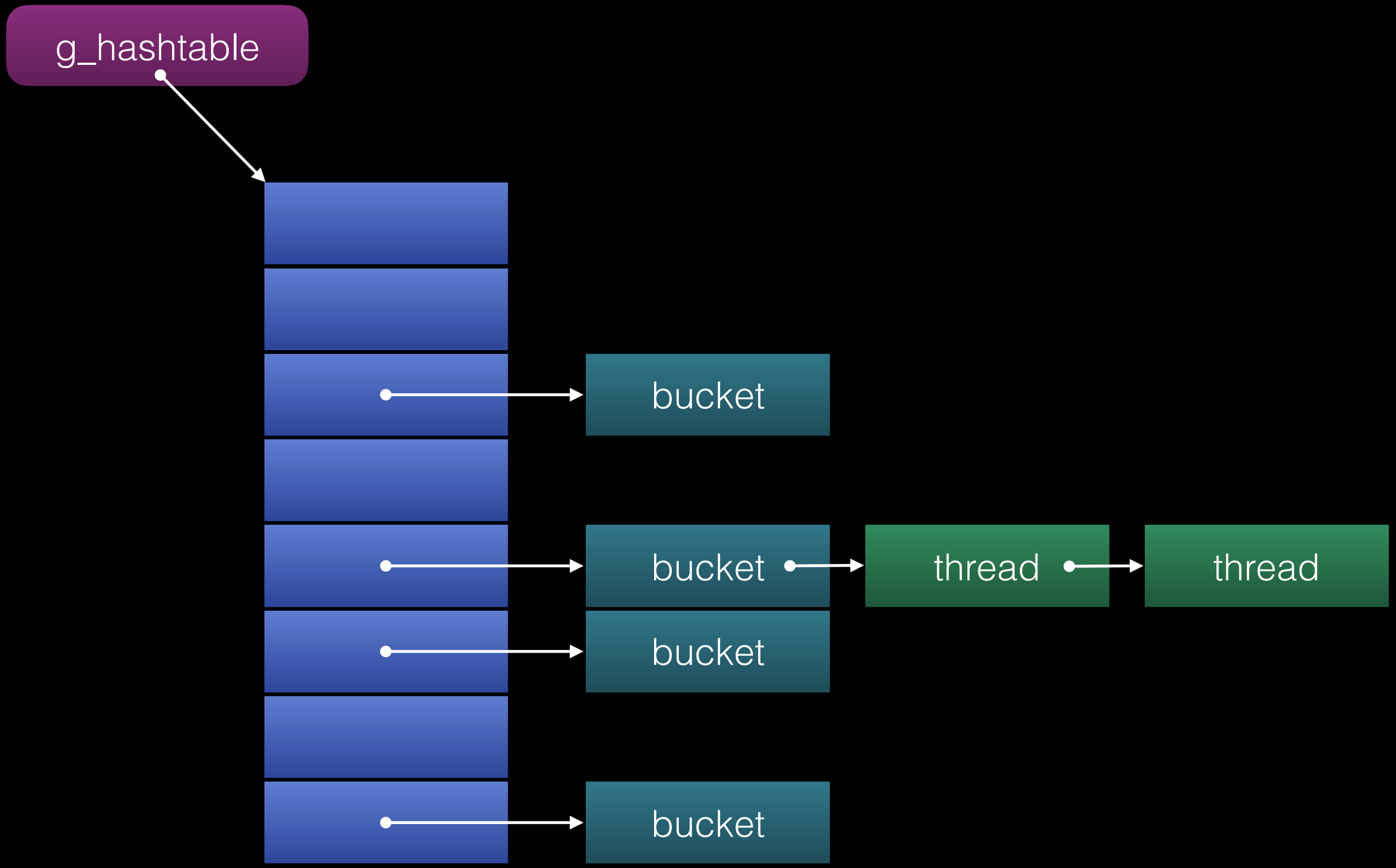
WTF::Lock

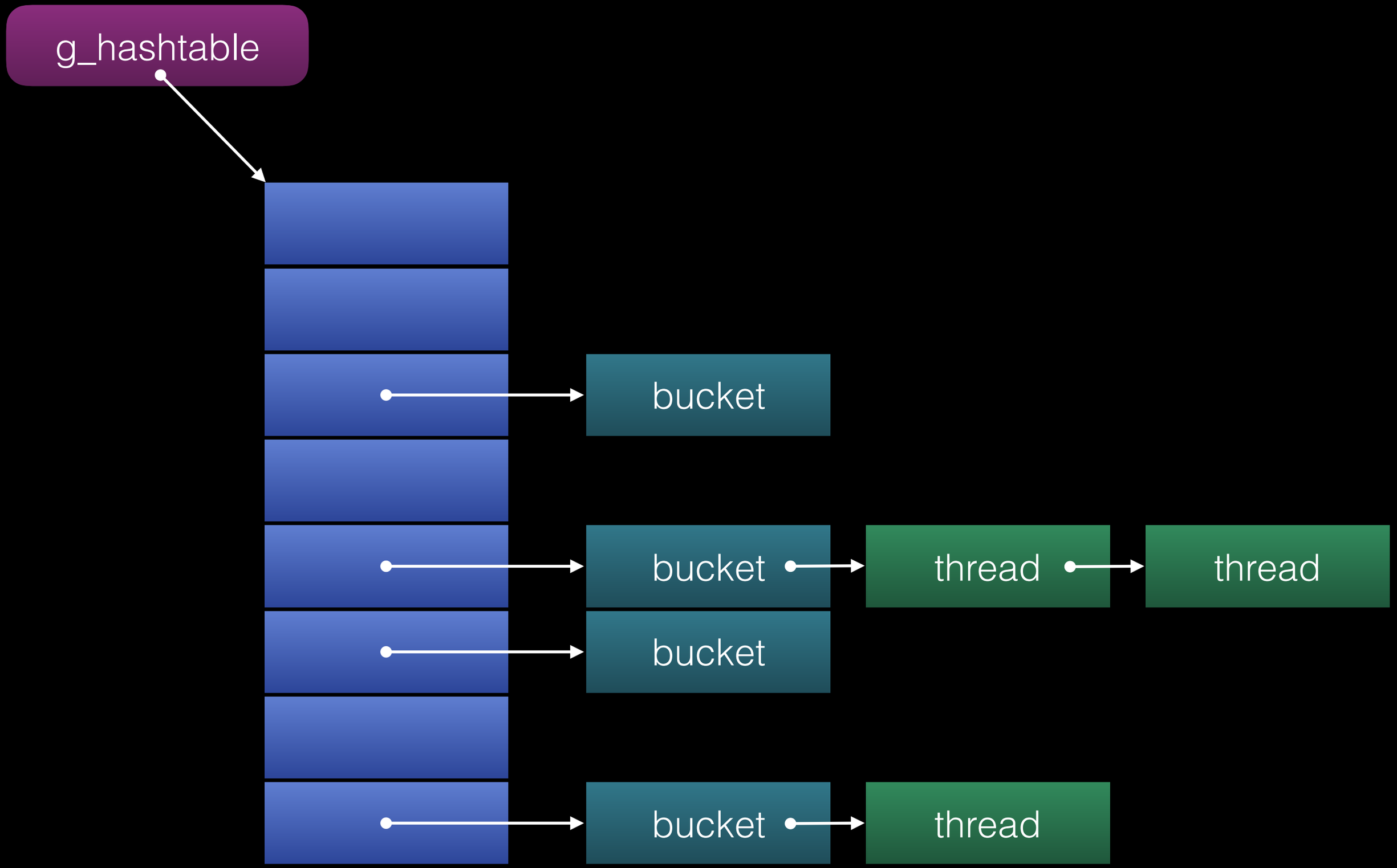
- Goals
- How Locks Work
- ParkingLot API
- WTF::Lock algorithm
- ParkingLot algorithm
- Conclusion

g_hashtable


g_hashtable


g_hashtable
bucket
bucket
bucket
bucket


g_hashtable
bucket
bucket
thread
thread
bucket
bucket


g_hashtable
bucket
bucket
thread
thread
bucket
bucket
thread

g_hashtable

Each bucket has a lock

bucket

bucket

bucket

thread

thread

bucket

thread

g_hashtable

Each bucket has a lock

bucket

Per-thread objects

bucket

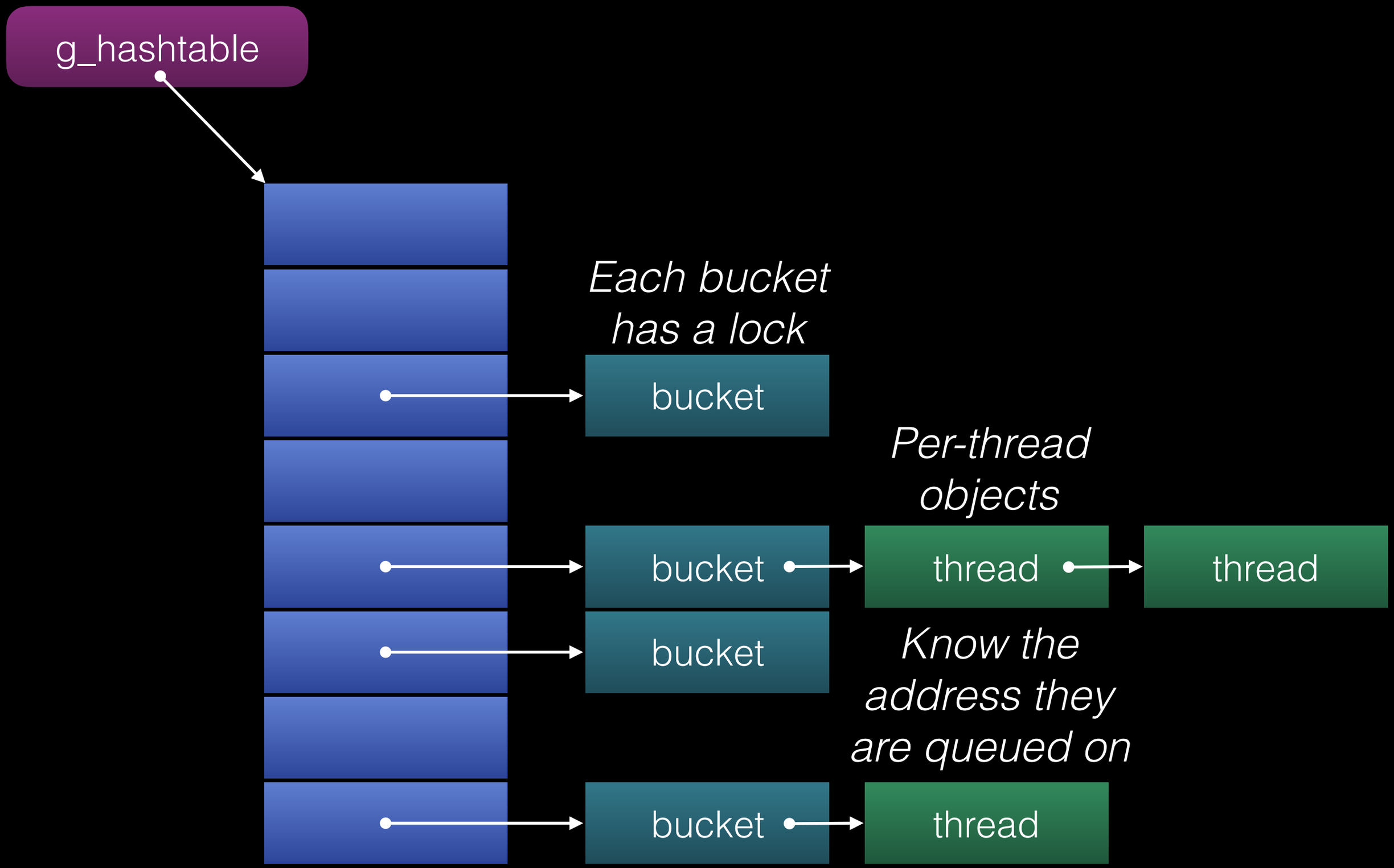
thread

thread

bucket

bucket

thread


g_hashtable
Each bucket has a lock
bucket
Per-thread objects
bucket
thread
thread
bucket
Know the address they are queued on
bucket
thread

Tricks

- Use bucket for first time: *CAS the bucket pointer from null to the new bucket.*

- Resize the hashtable: *lock all of the buckets in address order.*

Tricks

- Use bucket for first time: *CAS the bucket pointer from null to the new bucket.*
- Resize the hashtable: *lock all of the buckets in address order.*

ParkingLot is “just” a concurrent hashtable of queues, and each queue is guarded by a lock.

WTF::Lock

- Goals
- How Locks Work
- ParkingLot API
- WTF::Lock algorithm
- ParkingLot algorithm
- Conclusion

Conclusion

- Locks fit in two bits.
  - Can sneak a lock into the JS object header.
  - WTF::Lock is 1 byte.
- Condition variables fit in one bit.
  - WTF::Condition is 1 byte.
  - Super fast.
- Tons of other possibilities!

Agenda

- ~~Introduction~~
- ~~JavaScriptCore~~
- ~~Efficient Mark-Sweep~~

  - *30 minute break*

- ~~Concurrent GC~~
- ~~bmalloc~~
- ~~WTF::Lock~~