Учим Кукушку летать или Concurrent Hash Table with SeqLocks

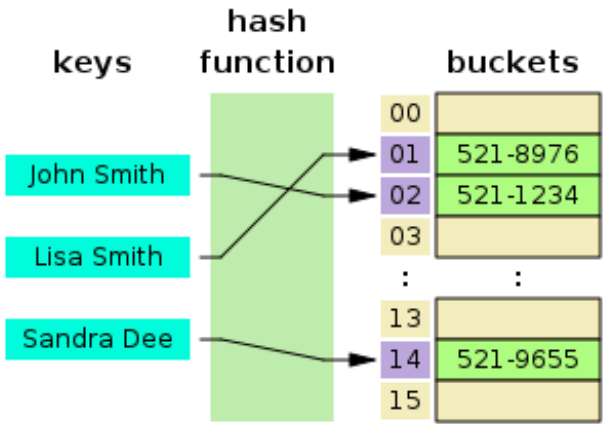
Антон Малахов и Антон Потапов, C++ Russia 2023

Parallel Runtimes Team @ Computing / Huawei / Н.Новгород

www.huawei.com

HUAWEI

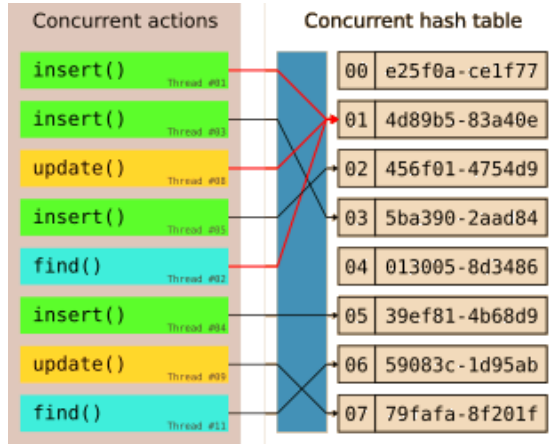
Concurrent Hash Table - что это, зачем?

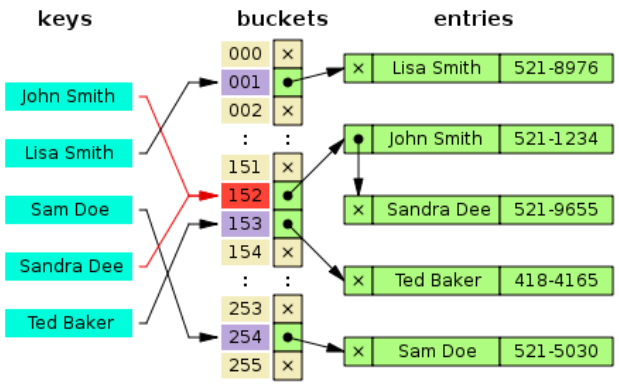


Concurrent *containers* (C++) или *collections* (Java) – это про масштабируемость на потоках

← Hash Table – Хэш-Таблица →
ассоциативный массив, оперирующий с парами ключа и значения

$Bucket_{index} = Hash(Key) \bmod Capacity$



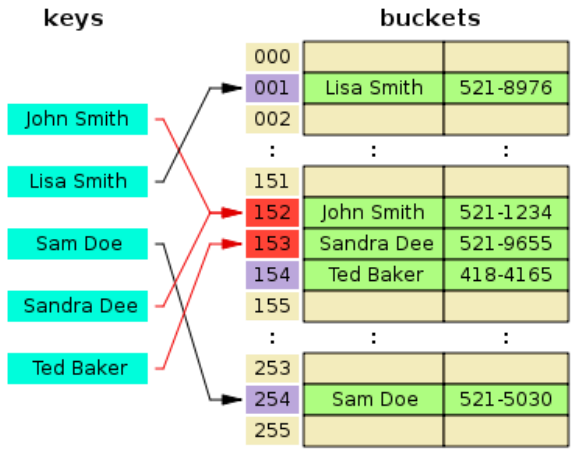


Два ключевых следствия:

- **Рехэшинг**
- **Коллизии**

Методы разешения коллизий:

- **← Списком**
- **Открытая адресация →**

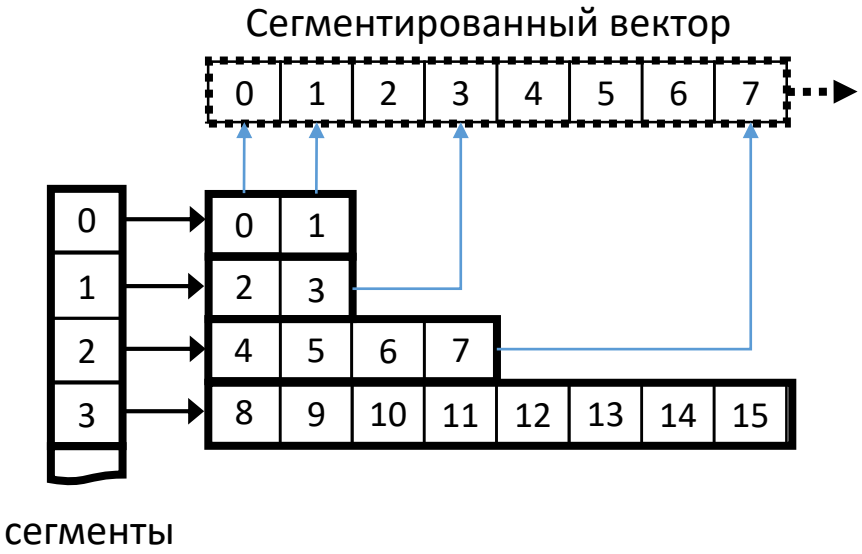


* Картинки из Wikipedia

HUAWEI

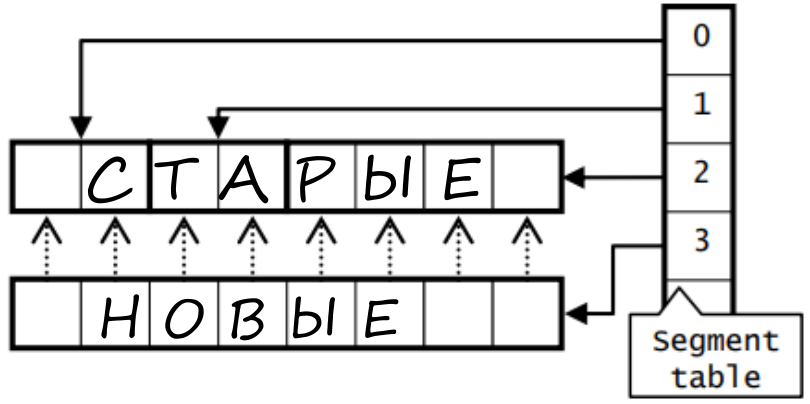
Пролог...

- ...давным давно, в далёком 2006-2007 мне поручили развивать tbb::concurrent_vector и tbb::concurrent_hash_map
- tbb::concurrent_hash_map (<TBB 2.2) был похож на ConcurrentHashMap из Java
  - Сегментированная (шардированная), растущая хэш-таблица на списках
  - Сегментация уменьшает борьбу за локи (contention) во время изменения размера таблицы
- На критическом пути 3 лока
  - На сегмент
  - На бакет
  - На пару K/V
- Слишком много локов!
  - Уберём локи на сегменты? Редко нужны
  - Но изменение размера приведёт к инвалидации указателей...
- tbb::concurrent_vector позволяет увеличивать размер без инвалидации итераторов и указателей
  - Добавляя таблицу сегментов
  - Каждый сегмент = сумме размеров предыдущих



HUAWEI

Ленивое рехэширование



- Эврика!
  - Убираем шардирование и локи на сегменты
  - Меняем сегменты как в concurrent_vector
  - Изобретаем ленивое рехэширование на основе синхронизации между бакетами

https://arxiv.org/abs/1509.02235

- Меньше локов, больше профит!
  - (в финальной версии ускорение до 2x)
  - Зачем же ещё нам нужен лок на сегмент?

- Налетаем на гонку данных!
  - Синхронизации элементов не достаточно для защиты глобального состояния

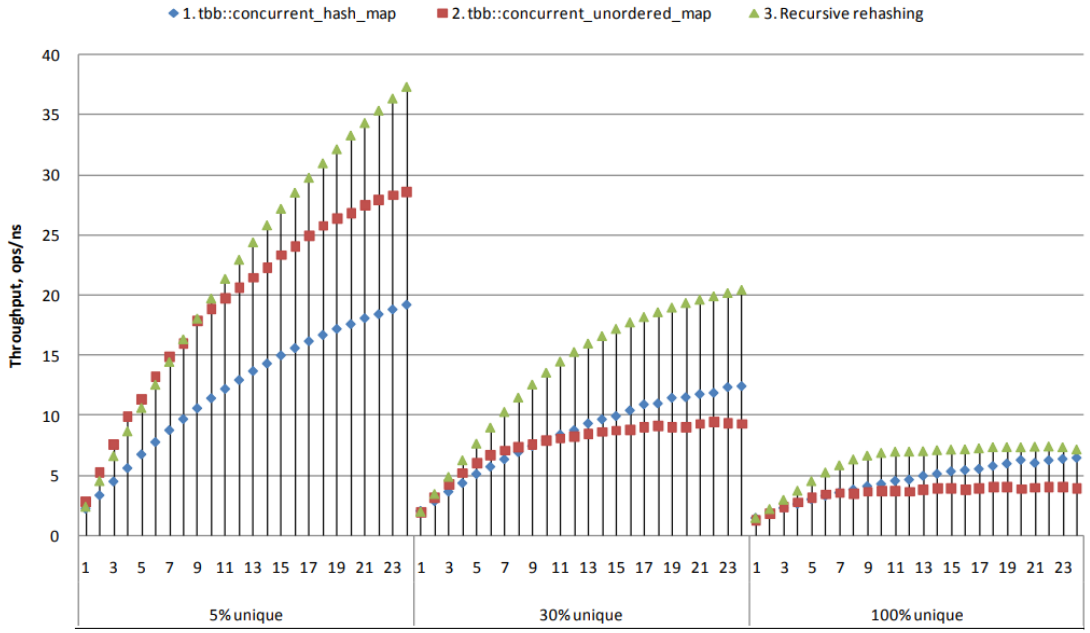


Figure 6. Throughput scalability of filling the table up to 2M pairs on various input rates

HUAWEI

Ловись датарэйс большой, но реденький

- Можно не найти ключ, который переехал в новый бакет пока его искали

- Обидно..
  - Такое редкое событие...
  - Но так дорого платить в каждой операции

- А что если..
  - Обнаружим гонку и перезапустим поиск!
    - Если ключ не найден, читаем Capacity заново
    - Если новый Capacity != старый, возможна гонка

Thread 1	Threads 2,3
B = H(Key) mod Capacity	
	Capacity *= 2 RehashBucket(B) // Key вылетает из B
Lookup(Key, B)	

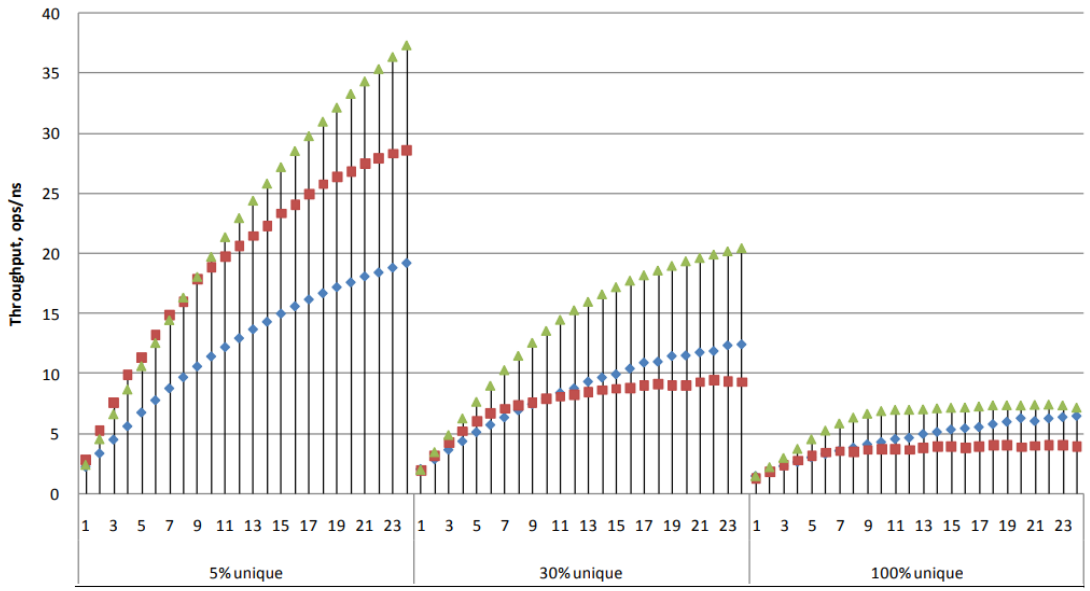
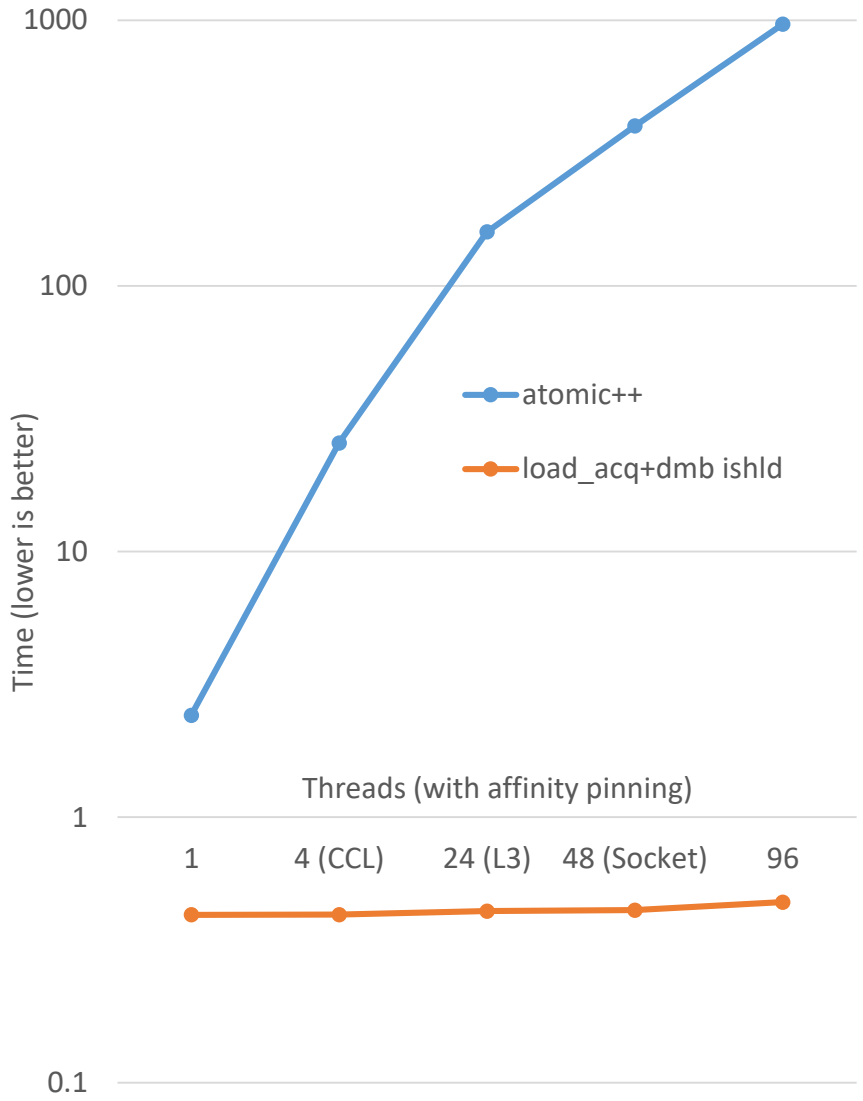


Figure 6. Throughput scalability of filling the table up to 2M pairs on various input rates

HUAWEI

Эпохальные гонки и SeqLock

- Итак, гонка данных это ещё не конец
  - Можно корректно выйти из неё, если правильно понимать последствия
  - Есть серая зона между деструктивным UB и тотальной засинхронизованностью
  - Синхронизация – это всегда дорого
- Спекулятивные транзакции на основе эпох
  - Capacity в tbb::concurrent_hash_map – частный случай
  - Эпоха отражает версию данных, только ростёт
  - Эпоху безопасно читать без лока
- Sequence Lock (SeqLock) – пример спекулятивной синхронизации на основе эпох
  - Функционально похож на RW-лок
  - Идея – позволить читателям читать параллельно
  - Чтение масштабируется, синхронизация - нет



HUAWEI

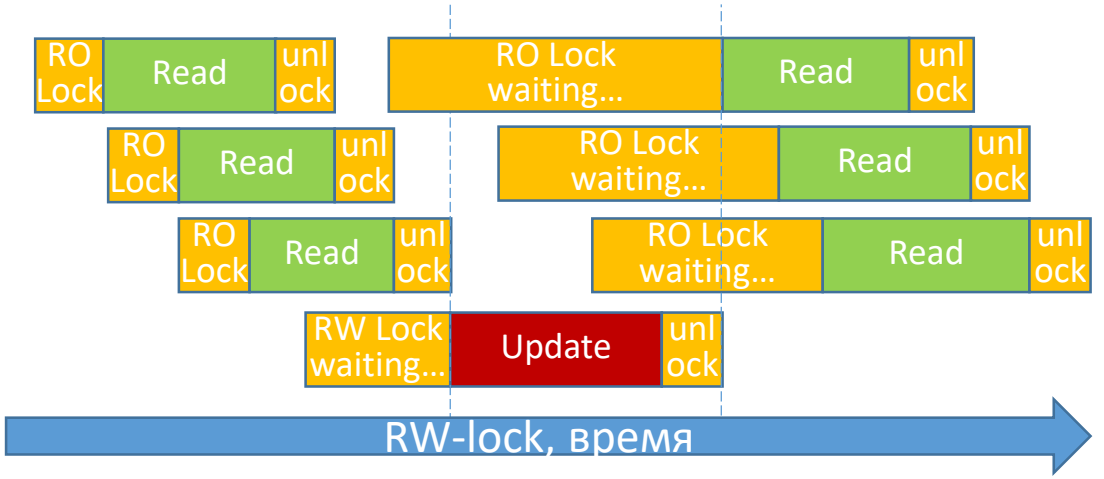
Как работают SeqLock и RW-лок

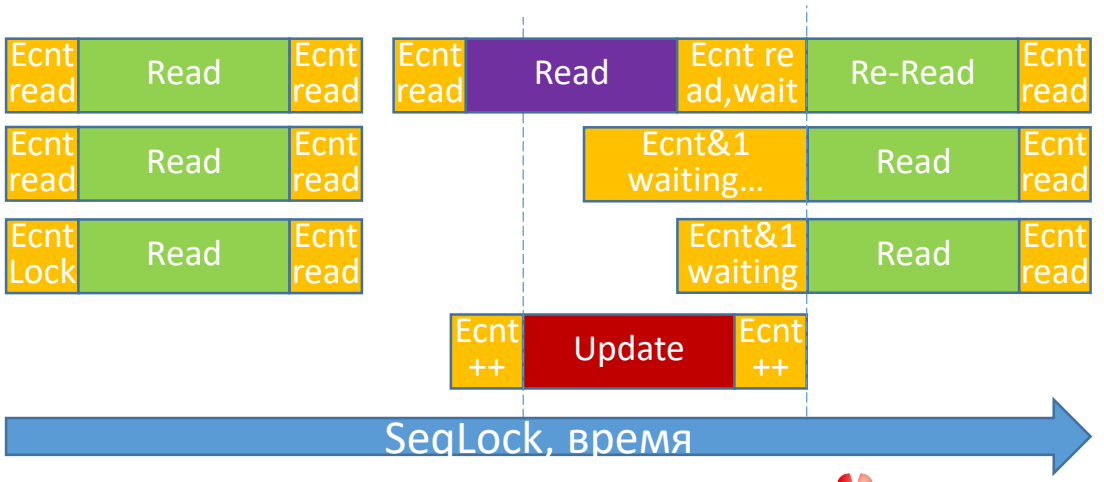
RW-лок

- Писатель исключает любые другие параллельные операции
- Читатели увеличивают счётчик читателей, если нет писателя и ждут его если есть

SeqLock

- Перед и после критической секции писателя увеличим счётчик Эпохи
- Читатели читают Эпоху, читают данные, перечитывают Эпоху
- Если Эпохи не совпадают, значит прочитали мусор и надо повторить





HUAWEI

++ SeqLock --

За: Читателям не надо модифицировать атомик чтобы что-то прочитать

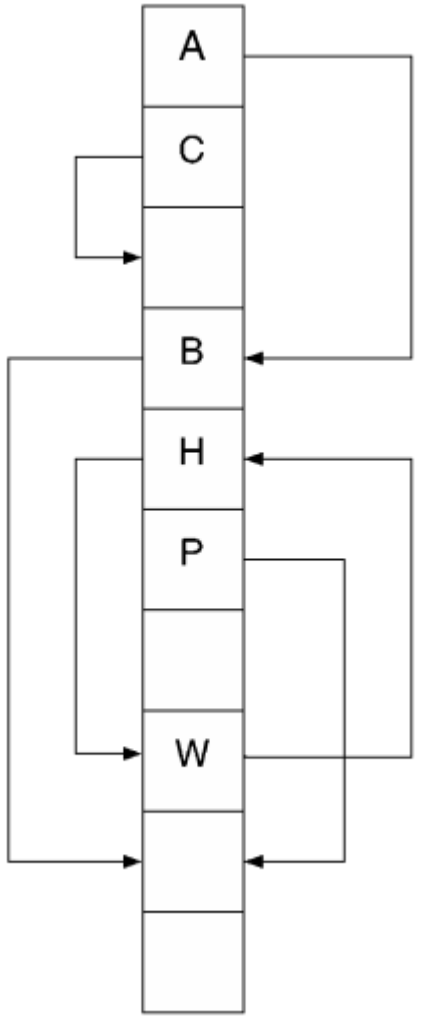
- **Быстрее**: Чтение + load fence просто быстрее RMW атомика в RW локе
- **Масштабируется**: Чтение из общей кэш-линии оседает в L1 и масштабируется. Запись в общую кэш-линию выстраивается в очередь благодаря протоколу когерентности кэшей

Против: Читатели невидимы, писатели не ждут читателей

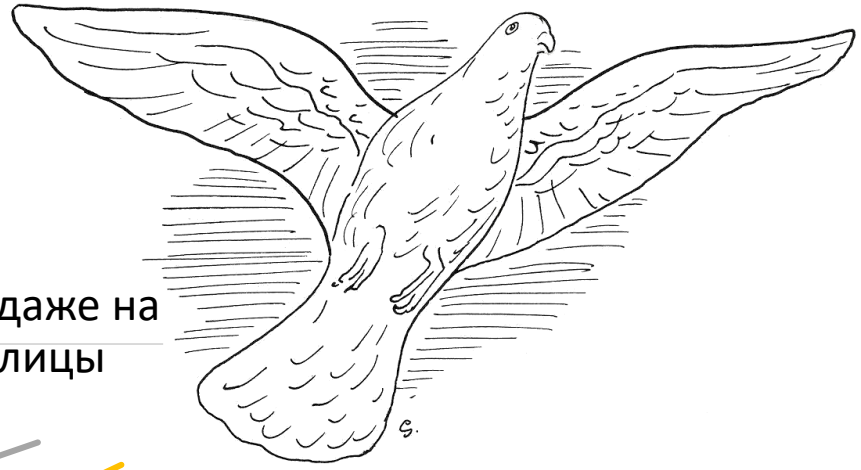
- Можно прочитать мусор
  - Безопасно только с простыми типами (PODs, memcpy)
  - Сильно ограничивает применимость в C++
- Чтение не защищает время жизни объекта
  - SeqLock не видит сколько читателей читают
  - Нельзя освобождать память пока есть читатели
- Коллизии тратят время
  - Если записей много, читатели могут зависнуть
  - Применять при редких обновлениях

Применим SeqLock к хэш-таблице

- Заменим RW-локи на СекЛоки? Проверим наши ограничения:
  - ✓Коллизии: Хэш-таблица используется больше для чтения
  - ✓Мусор: Механизм можно скрыть за copy-by-value интерфейсом
    - ..кроме сложных пользовательских типов – но можно обойти трейтами
  - ✓Время жизни и размещение объектов – под контролем контейнера
    - Изменение размера можно сделать без освобождения бакетов
    - Можно использовать открытую адресацию – без освобождения пар
      - Кстати, она всё-равно быстрее списков
- Начнём с алгоритма Кукушки (Cuckoo)
  - Открытая адресация, один из лучших в индустрии
  - Отличие: коллизия при вставке вытесняет элемент в альтернативный бакет
    - Кукушке для поиска ключа требуется взять локи сразу на два бакета
    - А значит, применение SeqLock ещё более релевантно здесь

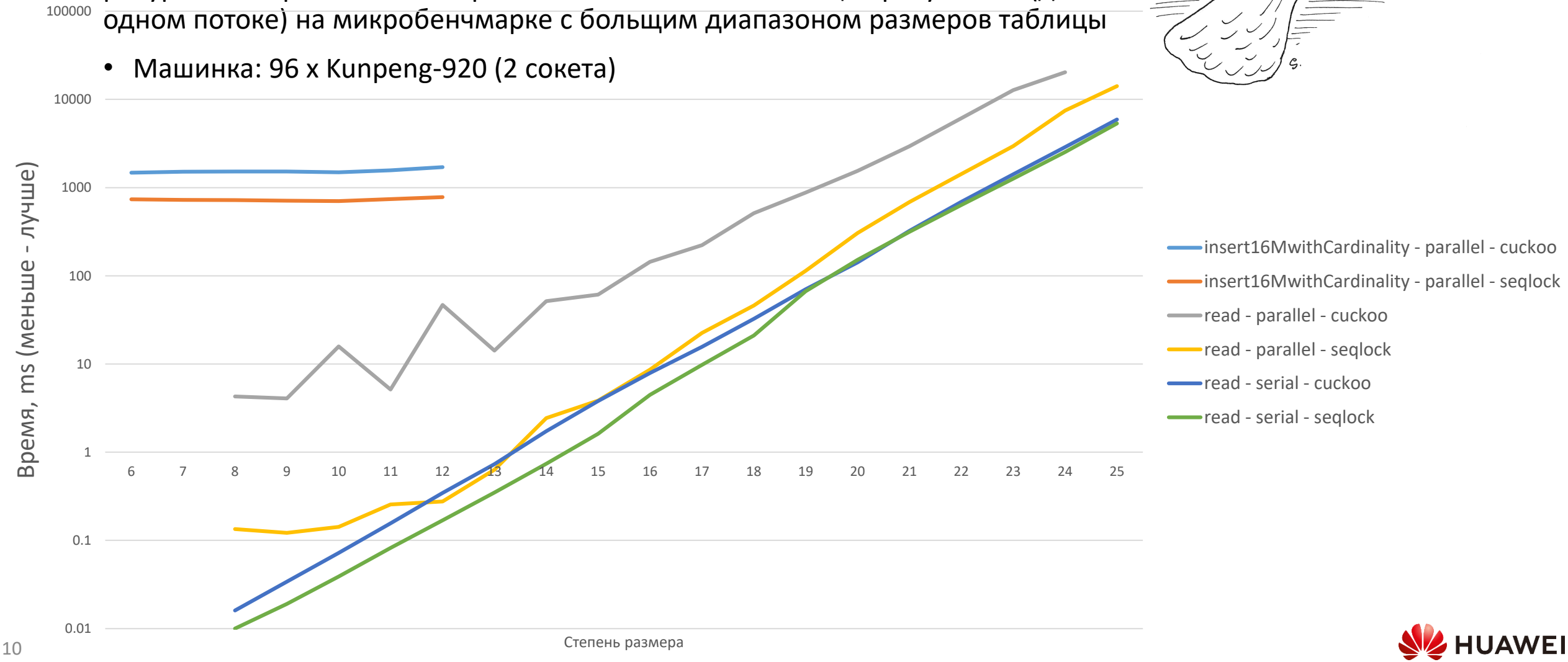


Кукушка просто летает!



Наша реализация cuckoo hashing с SeqLock в каждом бакете и ленивым рекурсивным рехэшингом. Прототип показал впечатляющие результаты (даже на одном потоке) на микробенчмарке с большим диапазоном размеров таблицы

- Машинка: 96 x Kunpeng-920 (2 сокета)



HUAWEI

И ещё!

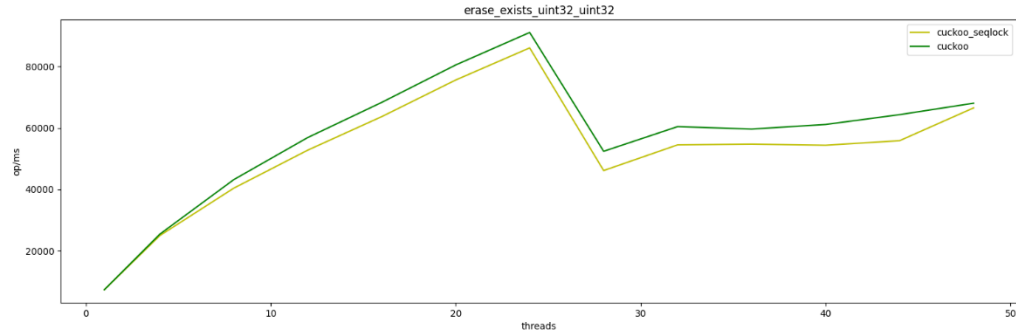
Разные комбинации операций, больше=лучше

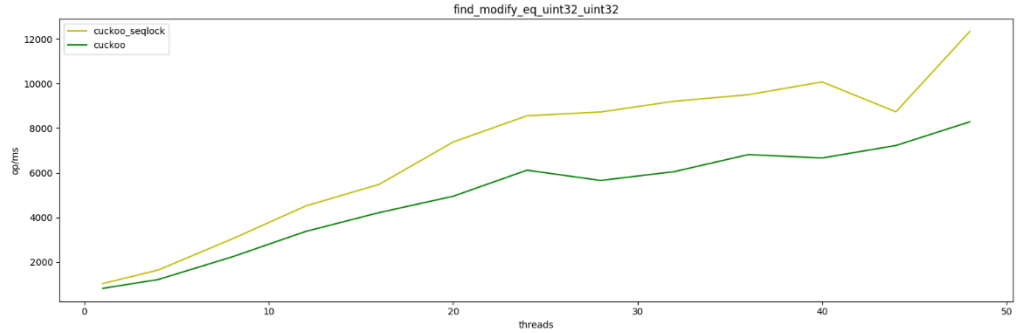
Реализация с выделенной страницей СекЛоков

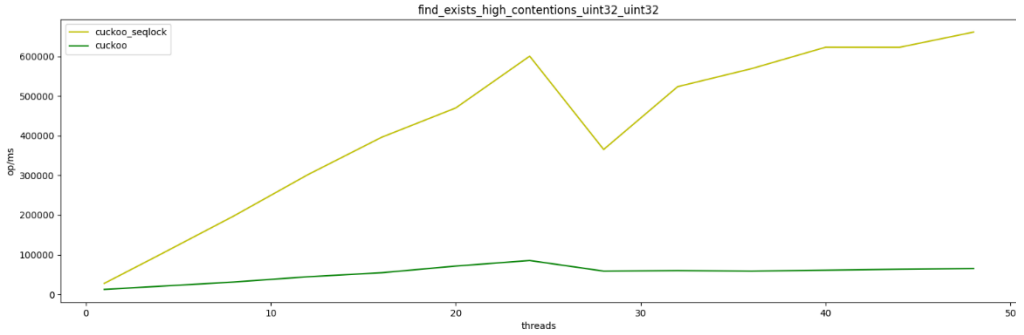
- Машинка:

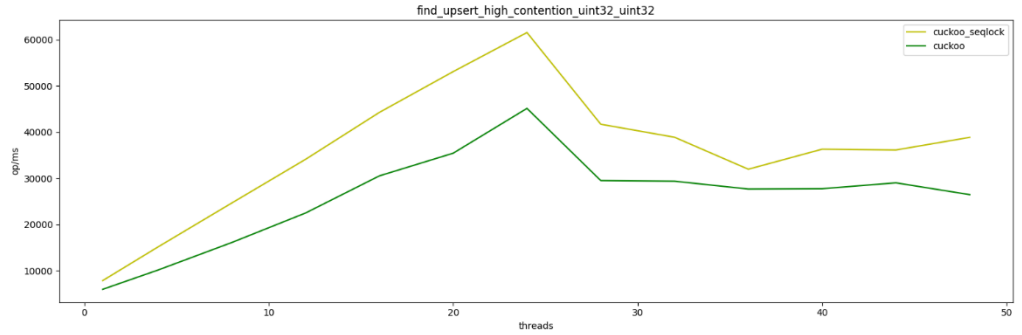
48 x Kunpeng-920 (2 сокета)

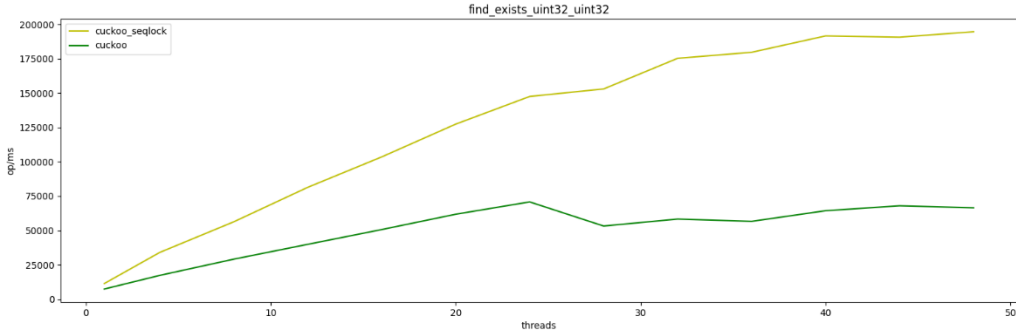
Из дипломной работы (ITMO) Артёма Давыдова

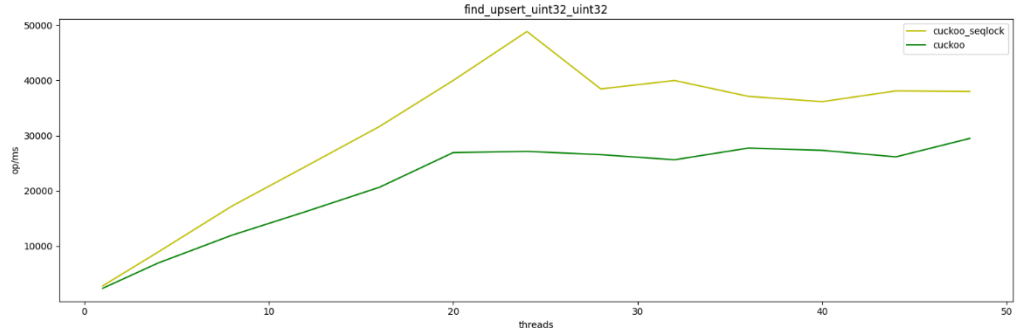


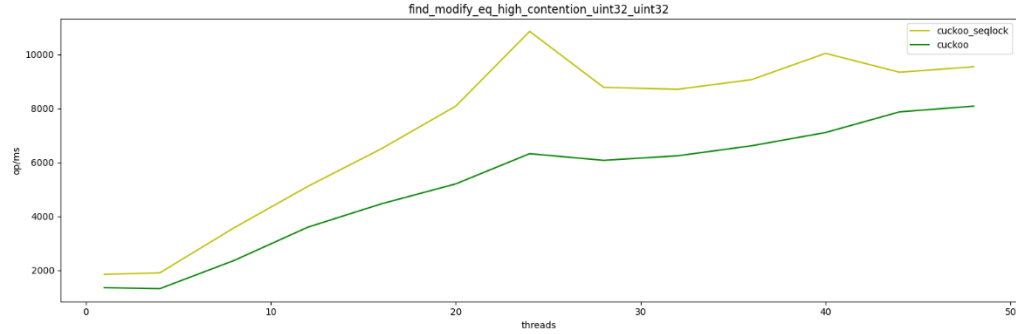


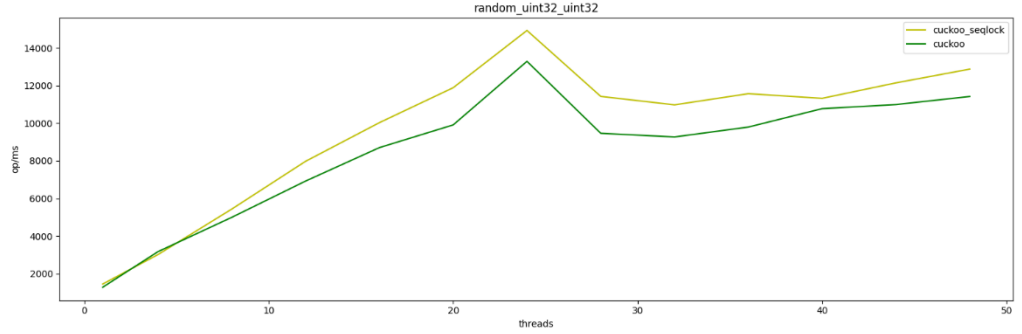












HUAWEI

Промежуточные итоги

- Любая модифицирующая атомарная операция не масштабируется так как чтение, особенно если оно повторяется
- SeqLock – важная техника оптимизации
  - Особенно на многоядерных процессорах, таких как Kunpeng
  - Ускорение в сотни раз против любых даже лучших локов
- Concurrent containers могут скрыть особенности SeqLock
  - Можно построить многопоточную хэш-таблицу с СекЛоком если
    - Избежать освобождения памяти
    - Копировать по значению
    - Испорченные копии не проблема
- В C++ всё это не тривиально…

SeqLock ?

```
int data1 = 0;
int data2 = 0;

std::atomic<std::size_t> counter {0};

void read(){
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(acquire);
        r1 = data1;
        r2 = data2;
        seq1 = counter.load(release);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

```
bool is_locked(std::size_t v)  { return v & 1UL; }
```

«А как правильно написать load() чтобы он не передвигался относительно чтения r1 и r2 ?”

Примерный вопрос на Stack Overflow….

SeqLock ?

```
int data1 = 0;
int data2 = 0;

std::atomic<std::size_t> counter {0};

void read(){
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(acquire);
        r1 = data1;
        r2 = data2;
        seq1 = counter.load(release);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

```
bool is_locked(std::size_t v)  { return v & 1UL; }
```

«А как правильно написать load() чтобы он не передвигался относительно чтения r1 и r2 ?”

Примерный вопрос на Stack Overflow….

Undefined Behavior

Гонки Данных

- **[intro.races].21.2**
  "The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other. ... Any such **data race** results in **undefined behavior**."

[intro.races].11

•An evaluation *A* *happens before* an evaluation *B* if either

- *A* is sequenced before *B*, or
- *A* synchronizes with *B*, or
- *A* happens before *X* and *X* happens before *B*.

[intro.execution].8

Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread ([intro.multithread]), which induces a partial order among those evaluations.

Given any two evaluations *A* and *B*, if *A* is sequenced before *B* (or, equivalently, *B* is *sequenced after* *A*), then the execution of *A* shall precede the execution of *B*.

[intro.execution].9

Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.[48]

[intro.races].6

Certain library calls *synchronize with* other library calls performed by another thread.

For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store ([atomics.order]).

Spin Mutex

```
#include <atomic>

constexpr auto acquire = std::memory_order_acquire;
constexpr auto release = std::memory_order_release;
constexpr auto relaxed = std::memory_order_relaxed;

struct spin_mutex {
    std::atomic<bool> locked {false};

    void lock() {
       bool expected = false;
       while(!locked.compare_exchange_weak(expected, true, acquire, relaxed)){
                pause();
                expected = false;
        }
    }
    void unlock() {   locked.store(false, release); }
};
```

```
spin_mutex m; int data1; int data2;

void write_thread() {
    m.lock();
    data1 = 42;
    data2 = 43;
    m.unlock();
}

void read_thread() {
    m.lock();
    auto r1 = data1;
    auto r2 = data2;
    m.unlock();
}
```

Sequenced Before

Sequenced Before

```
spin_mutex m; int data1; int data2;

void write_thread() {
  locked.store(false, release);
}
```

store-release

```
void read_thread() {
        bool expected = false;
        while(!locked.compare_exchange_weak(
                expected, true, acquire, relaxed))
        {
                expected = false;
        }

}
```

```
spin_mutex m; int data1; int data2;

void write_thread() {
  locked.store(false, release);
}

                    void read_thread() {
                          bool expected = false;
                          while(!locked.compare_exchange_weak(
                                expected, true, acquire, relaxed))
                          {
                                expected = false;
                          }

                    }
```

store-release

load-acquire

```
spin_mutex m; int data1; int data2;
```

```
void write_thread() {
  locked.store(false, release);
}
```

takes its value from the store
(успешный compare_exchange читает false записанный store-ом)

store-release

load-acquire

```
void read_thread() {
        bool expected = false;
        while(!locked.compare_exchange_weak(
                expected, true, acquire, relaxed))
        {
                expected = false;
        }

}
```

```
spin_mutex m; int data1; int data2;
```

```
void write_thread() {
  locked.store(false, release);
}
```

takes its value from the store
(успешный compare_exchange читает false записанный store-ом)

store-release

load-acquire

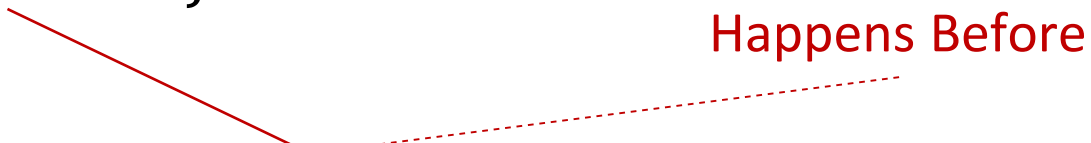
```
void read_thread() {
        bool expected = false;
        while(!locked.compare_exchange_weak(
                expected, true, acquire, relaxed))
        {
                expected = false;
        }

}
```

Synchronize With

```
spin_mutex m; int data1; int data2;

void write_thread() {
    m.lock();
    data1 = 42;
    data2 = 43;
    locked.store(false, std::memory_order_release);
    m.unlock();
}

locked.compare_exchange_weak(false, true,
std::memory_order_acquire, )

void read_thread() {
    m.lock();
    auto r1 = data1;
    auto r2 = data2;
    m.unlock();
}
```

Synchronize With

```
spin_mutex m; int data1; int data2;

void write_thread() {
    m.lock();
    data1 = 42;
    data2 = 43;
    m.unlock();
}

void read_thread() {
    m.lock();
    auto r1 = data1;
    auto r2 = data2;
    m.unlock();
}
```



SeqLock

```
int data1 = 0;
int data2 = 0;

std::atomic<std::size_t> counter {0};

std::size_t lock(){
    std::size_t seq = 0;
    do {
        // wait for other writer to complete
        while (is_locked(seq = counter.load(relaxed))) { pause(); }
    } while (counter.compare_exchange_weak(seq, seq + 1, seq_cst, relaxed));
    return seq;
}

void unlock(std::size_t seq0){
    counter.store(seq0 + 2, seq_cst);
}
```

```
bool is_locked(std::size_t v)  { return v & 1UL; }
```

SeqLock

```
int data1 = 0;
int data2 = 0;

std::atomic<std::size_t> counter {0};

void write(int d1, int d2) {
    std::size_t seq = lock();

    data1 = d1;
    data2 = d2;

    unlock(seq);
}
```

Data race

```
bool is_locked(std::size_t v)  { return v & 1UL; }

void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter;
        r1 = data1;
        r2 = data2;
        seq1 = counter;
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

SeqLock – Гонка

- Почему это гонка данных?
- “The execution of a program … contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other”
- **Нельзя проигнорировать (т.к. это UB в C++)**

SeqLock

```
std::atomic<int> data1 = 0; bool is_locked(std::size_t v)  { return v & 1UL; }
std::atomic<int> data2 = 0;

std::atomic<std::size_t> counter {0};
```

```
void write(int d1, int d2) {
    std::size_t seq = lock();

    data1.store(d1, seq_cst);
    data2.store(d2, seq_cst);

    unlock(seq);
}
```

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(seq_cst);
        r1 = data1.load(seq_cst);
        r2 = data2.load(seq_cst);
        seq1 = counter.load(seq_cst);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

SeqLock, sec_cst версия

- Стандарт гарантирует, что все seq_cst операции (над всеми переменными) упорядоченны
- переупорядочивания невозможны (ненаблюдаемы)
- «Неоптимальная» производительность

SeqLock, оптимизированный?

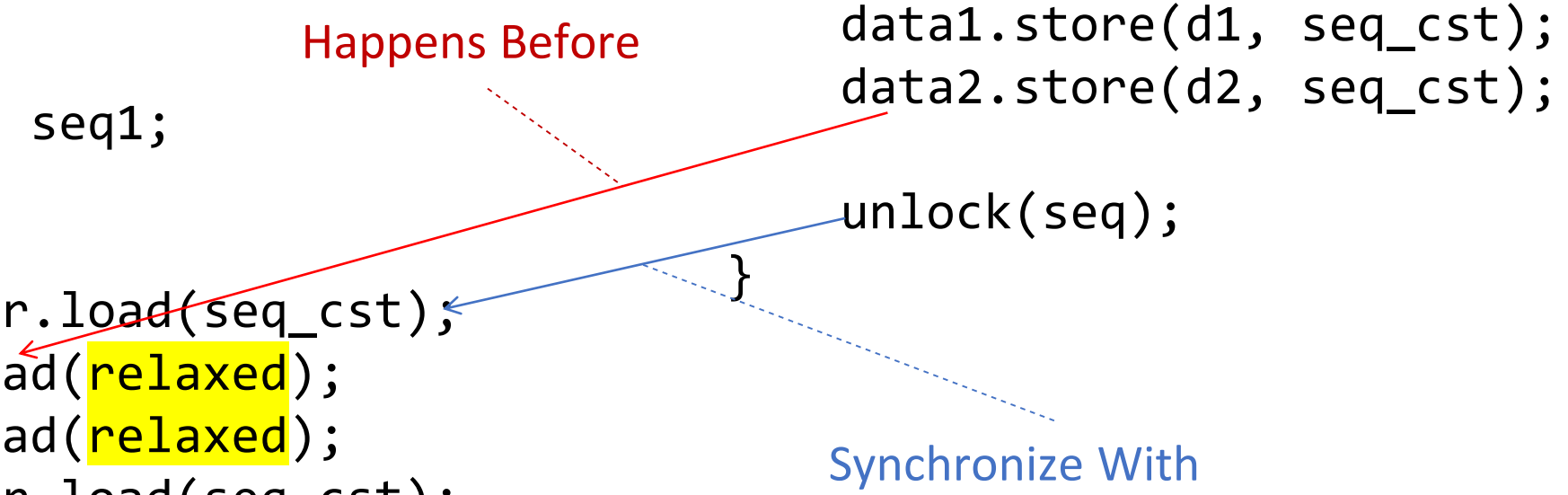
```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(seq_cst);
        r1 = data1.load(relaxed);
        r2 = data2.load(relaxed);
        seq1 = counter.load(seq_cst);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

SeqLock, оптимизированный?

сеции чтения и записи не перекрываются:

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(seq_cst);
        r1 = data1.load(relaxed);
        r2 = data2.load(relaxed);
        seq1 = counter.load(seq_cst);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

```
void write(int d1, int d2) {
    std::size_t seq = lock();

    data1.store(d1, seq_cst);
    data2.store(d2, seq_cst);

    unlock(seq);
}
```



HUAWEI

SeqLock, оптимизируем?

- Для перекрывающихся во времени секций чтения/записи необходимо гарантировать:
- Если читателю (частично) видны изменения данных, до должны быть видны и модификации счетчика (как минимум первая)
- Давайте попробуем их найти:

SeqLock, оптимизированный?

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(seq_cst);

        r1 = data1.load(relaxed);
        r2 = data2.load(relaxed);

        seq1 = counter.load(seq_cst);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

```
void write(int d1, int d2) {
    std::size_t seq = lock();

    data1.store(d1, seq_cst);
    data2.store(d2, seq_cst);

    unlock(seq);
}
```

Код не дает таких гарантий

Reads value

SeqLock, оптимизированный?

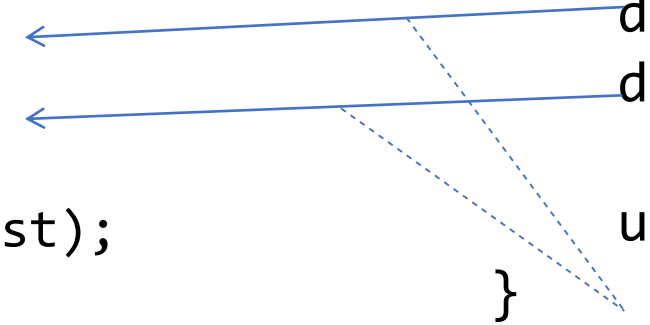
Идея (не моя ☹):

> Использовать защищаемые данные для синхронизации (т.е. получения необходимых гарантий)

SeqLock, оптимизированный?

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(seq_cst);

        r1 = data1.load(acquire);
        r2 = data2.load(acquire);

        seq1 = counter.load(seq_cst);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

```
void write(int d1, int d2) {
    std::size_t seq = lock();

    data1.store(d1, release);
    data2.store(d2, release);

    unlock(seq);
}
```



SeqLock, оптимизированный?

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(seq_cst);

        r1 = data1.load(acquire);
        r2 = data2.load(acquire);

        seq1 = counter.load(seq_cst);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

```
void write(int d1, int d2) {
    std::size_t seq = lock();

    data1.store(d1, release);
    data2.store(d2, release);

    unlock(seq);
}
```

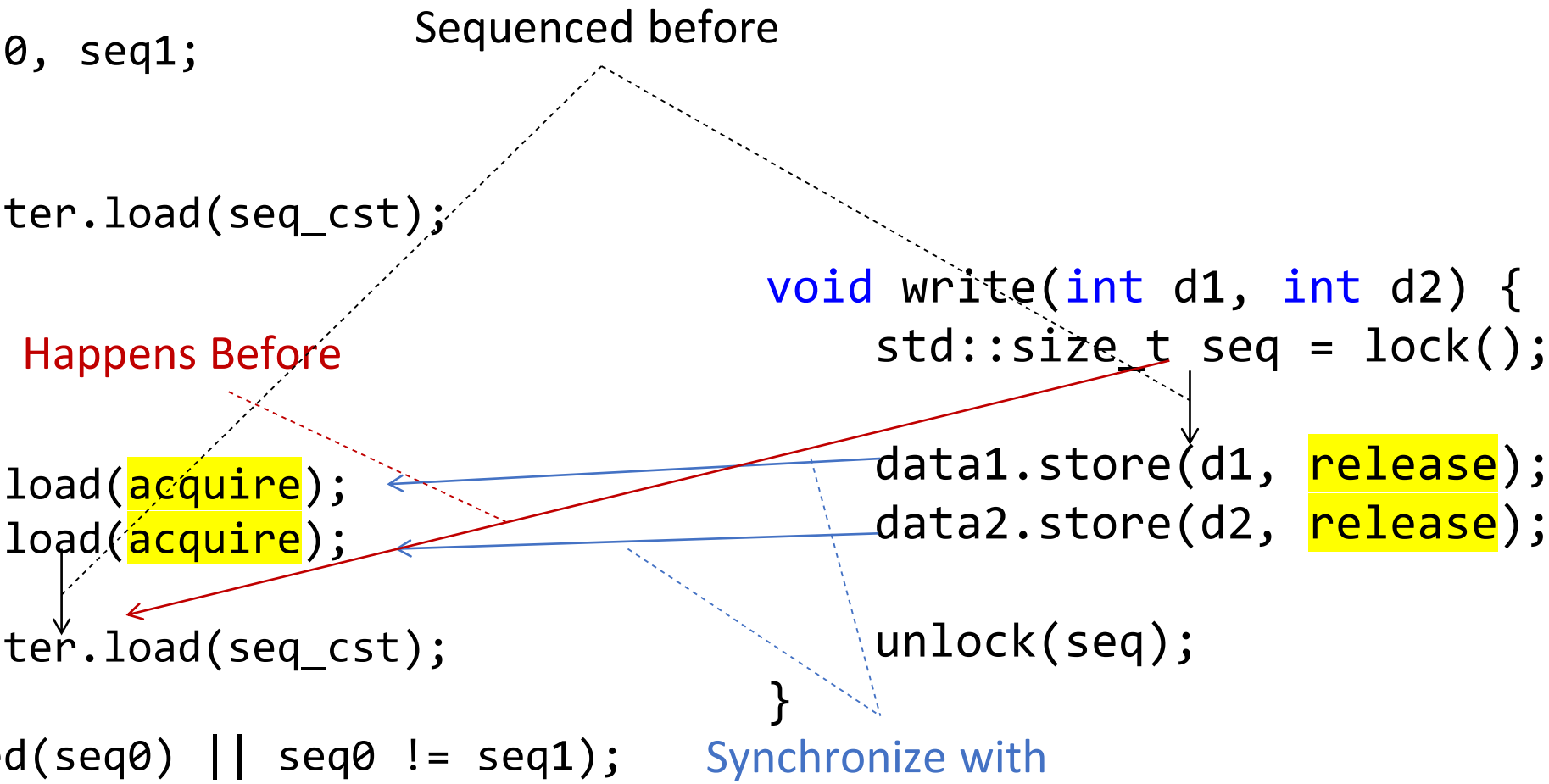
Sequenced before

Synchronize with

SeqLock, оптимизированный!

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(seq_cst);

        r1 = data1.load(acquire);
        r2 = data2.load(acquire);

        seq1 = counter.load(seq_cst);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

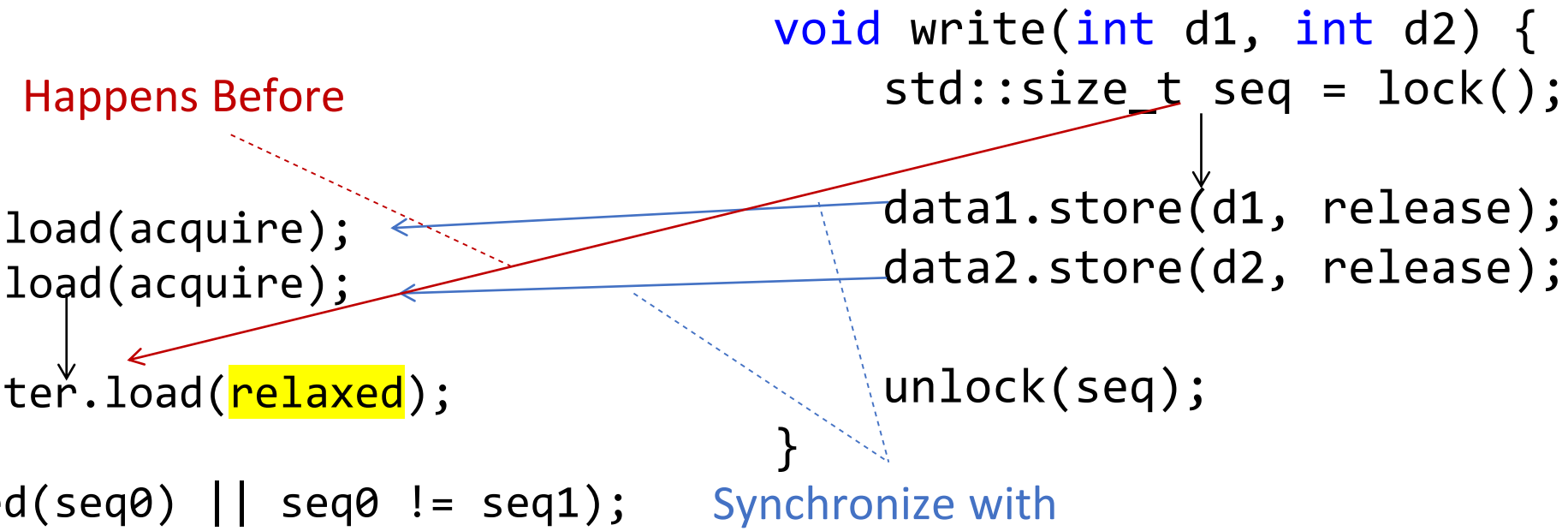
```
void write(int d1, int d2) {
    std::size_t seq = lock();

    data1.store(d1, release);
    data2.store(d2, release);

    unlock(seq);
}
```



SeqLock, оптимизированный!

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(acquire);

        r1 = data1.load(acquire);
        r2 = data2.load(acquire);

        seq1 = counter.load(relaxed);
    }
    while (is_locked(seq0) || seq0 != seq1);
}
```

```
void write(int d1, int d2) {
    std::size_t seq = lock();

    data1.store(d1, release);
    data2.store(d2, release);

    unlock(seq);
}
```



SeqLock, оптимизированный, ещё!

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(acquire);

        r1 = data1.load(relaxed);
        r2 = data2.load(relaxed);

        atomic_thread_fence(acquire);

        seq1 = counter.load(relaxed);
    }

    while (is_locked(seq0) || seq0 != seq1);
}
```

```
void write(int d1, int d2) {
    std::size_t seq = lock();

    atomic_thread_fence(release);

    data1.store(d1, relaxed);
    data2.store(d2, relaxed);

    unlock(seq);
}
```

SeqLock, заборы ☺

- [atomics.fences].2
- A release fence *A* synchronizes with an acquire fence *B* if there exist atomic operations *X* and *Y*, both operating on some atomic object *M*, such that *A* is sequenced before *X*, *X* modifies *M*, *Y* is sequenced before *B*, and *Y* reads the value written by *X* ....

```
void thread_a(){
    atomic_thread_fence(release); // A
    M.store(0,relaxed);           // X
}
```

Sequenced before

Synchronize with

```
void thread_b(){
    auto r = M.load(relaxed);       //Y
    atomic_thread_fence(acquire);   //B
}
```

SeqLock, оптимизированный, ещё!

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(acquire);

        r1 = data1.load(relaxed);
        r2 = data2.load(relaxed);

        atomic_thread_fence(acquire);

        seq1 = counter.load(relaxed);
    }

    while (is_locked(seq0) || seq0 != seq1);
}
```

Sequenced before

```
void write(int d1, int d2) {
    std::size_t seq = lock();

    atomic_thread_fence(release);

    data1.store(d1, relaxed);
    data2.store(d2, relaxed);

    unlock(seq);
}
```

SeqLock, оптимизированный, ещё!

```
void read() {
    std::size_t seq0, seq1;
    int r1, r2;
    do {
        seq0 = counter.load(acquire);

        r1 = data1.load(relaxed);
        r2 = data2.load(relaxed);

        atomic_thread_fence(acquire);

        seq1 = counter.load(relaxed);
    }

    while (is_locked(seq0) || seq0 != seq1);
}
```

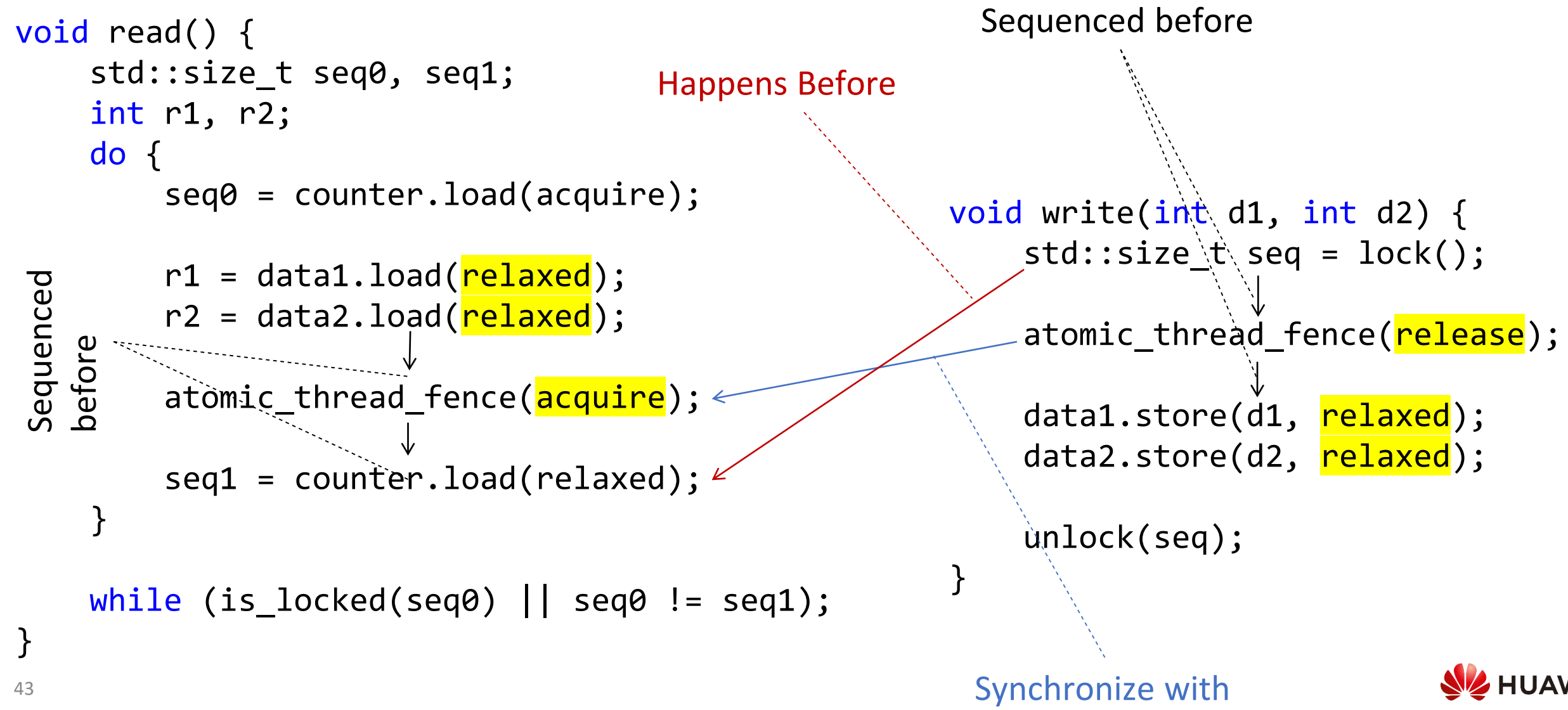
Sequenced before

```
void write(int d1, int d2) {
    std::size_t seq = lock();

    atomic_thread_fence(release);

    data1.store(d1, relaxed);
    data2.store(d2, relaxed);

    unlock(seq);
}
```

Sequenced before

Synchronize with

SeqLock, оптимизированный, ещё!



SeqLock, (полу-)финальная версия

```
class seq_lock {
    using counter_t = unsigned long int;
    std::atomic<counter_t> seq_ {0};
public:

    counter_t lock() ;

    void unlock(counter_t seq0);

    template <typename T>
    friend T load(seq_lock const& l, const T& data);
   template <typename T>
    friend void store(seq_lock& lck, T& dst, const T& src);
};
```

SeqLock, (полу-)финальная версия

```
counter_t seq_lock::lock() {
    counter_t seq0 = seq_.load(std::memory_order_relaxed);

    do {
        while (is_locked(seq0)) {
            // some waiting/backoff code like:
            //   std::this_thread::yield();
            seq0 = seq_.load(relaxed);
        }
    } while (!seq_.compare_exchange_weak(seq0, seq0 + 1, acquire, relaxed));
    return seq0;
}

void seq_lock::unlock(counter_t seq0){ seq_.store( seq0 + 2, release); }
```

SeqLock, (полу-)финальная версия

```
template <typename T>
T load(seq_lock const& l, const T& data){
    static_assert(std::is_trivially_copyable_v<T>);

    typename decltype(l.seq_)::value_type seq0, seq1;
    T loaded;
    do {
        seq0 = l.seq_.load(std::memory_order_acquire);

        atomic_load_memcpy(&loaded, &data, sizeof(loaded));
        std::atomic_thread_fence(acquire);

        seq1 = l.seq_.load(std::memory_order_relaxed);
    } while (is_locked(seq0) || (seq0 != seq1));

    return loaded;
}
```

SeqLock, (полу-)финальная версия

```
inline void* atomic_load_memcpy(void* dest, const void* source, size_t count){
    size_t uintmax_count = count / sizeof(uintmax_t);
    size_t uintmax_bytes = uintmax_count * sizeof(uintmax_t);

    for (size_t i = 0; i < uintmax_count; ++i){
        auto& dst = as<uintmax_t*>(dest)[i];
        auto& src = as<const uintmax_t*>(source)[i];

        dst = as_atomic(src).load(relaxed);
    }
    for (size_t i = uintmax_bytes; i < count; ++i) {
        auto& dst = as<char*>(dest)[i];
        auto& src = as<const char*>(source)[i];

        dst = as_atomic(src).load(relaxed);
    }
    return dest;
}
```

SeqLock, (полу-)финальная версия

```
template <typename T>
void store(seq_lock& lck, T& dst, const T& src){
    static_assert(std::is_trivially_copyable_v<T>);

    auto lck_tckn = lck.lock();

    std::atomic_thread_fence(release);
    detail::atomic_store_memcpy(&dst, &src, sizeof(dst));

    lck.unlock(lck_tckn);
}
```

SeqLock, (полу-)финальная версия

```
inline void* atomic_store_memcpy(void* dest, const void* source, size_t count){
    size_t uintmax_count = count / sizeof(uintmax_t);
    size_t uintmax_bytes = uintmax_count * sizeof(uintmax_t);

    for (size_t i = 0; i < uintmax_count; ++i) {
        auto& dst = as<uintmax_t*>(dest)[i];
        auto& src = as<const uintmax_t*>(source)[i];
        as_atomic(dst).store(src, relaxed);
    }

    for (size_t i = uintmax_bytes; i < count; ++i) {
        auto& dst = as<char*>(dest)[i];
        auto& src = as<const char*>(source)[i];
        as_atomic(dst).store(src, relaxed);
    }
    return dest;
}
```

Промежуточные итоги++ (ещё одни)

- SeqLock реализуем на C++
  - Корректно и
  - Эффективно
- Модель памяти C++
  - не описывается в терминах переупорядочивания инструкций
  - не эквивалентна модели памяти в аппаратных архитектурах
  - доступна для понимания простым смертным (ну или почти...☺)
- Стандарт C++ можно и нужно читать (+рисовать)
- C++ - неумолимая сила! ☺

Полезные ссылки

- “Can Seqlocks Get Along With Programming Language Memory Models” , Hans-J. Boehm https://www.hpl.hp.com/techreports/2012/HPL-2012-68.pdf
- «Атомарный» memcpy "P1478: Byte-wise atomic memcpy“
- Оригинальная Кукушка https://github.com/efficient/libcuckoo
- Статья про ленивый рехэшинг https://arxiv.org/abs/1509.02235
- «Литекс» - ещё более продвинутый алгоритм синхронизации на основе эпох с управлением циклом жизни (достоин отдельной публикации) https://github.com/oneapi-src/oneTBB/pull/131
- А что, вы что-то ещё хотели? ☺

g_Итоги

- Синхронизация: не локом единым, есть разные методы
  - Модифицирующие атомарные операции – дорогие
  - Только чтение по настоящему масштабируется, изучайте MESI
  - Счётчик Эпох и SeqLock – важные техники оптимизации
- Библиотеки, их интерфейсы могут скрыть особенности SeqLock
  - Можно построить многопоточную хэш-таблицу с СекЛоком
  - «Песен ещё не написаных сколько, скажи Кукушка, пропой!»
- (Стандарт) C++ позволяет писать эффективные алгоритмы синхронизации и структур данных
  - Но это не тривиально… ☺
  - И не для всех случаев
- Поделитесь со мной пожалуйста если смогли это применить на практике!