

# Многоядерные структуры и поиск

Первунецких Еремей

19 ноября 2025 г.

Целью данного мероприятия является возможность заинтересовать людей в изучении многоядерных структур данных. Я постарался собрать всё, что мне было бы нужно услышать в первую очередь, прежде чем писать свои первые проекты, а также добавил немного того, что выходит за рамки обычного курса многопоточного программирования и затрагивает недавние идеи, созданные в этой области.

## Итоговая задача

В конечном итоге мы хотели бы написать WAIT-FREE самобалансирующееся дерево, которое будет способно быстро обрабатывать запросы, не ломаться в случаях аварийных завершений и разумно работать с оперативной памятью.

## Лекция первая

### 1 Многопоточное исполнение

*Процессом исполнения* называется последовательный запуск инструкций, который в любой момент может быть прерван планировщиком системы и возобновлён через некоторое время. За данный промежуток состояние системы может кардинально измениться в пределах используемых функций. Единственным способом сделать какое-либо действие *атомарно* - использовать специальные инструкции процессора, которые в большинстве языков программирования сведены в отдельный тип и называются *atomic*.

Так или иначе ядро ОС предоставляет системные вызовы, которые предоставляют *блокирующий* механизм синхронизации - *mutex*. Он универсален и достаточно быстр и оптимален, если речь идёт об использовании других системных вызовов в этом же блоке кода. Но если заглянуть глубже в те механизмы, которые мы используем для обработки запросов, то оказывается, что его скорости мягко говоря недостаточно. В зависимости от того, какие механизмы синхронизации мы используем, появляется 4 основных вида гарантий работы приложения:

- Блокирующие

- LOCKED - мы верим, что программа завершится когда-нибудь, но мы не можем гарантировать, что система будет выполнять какие-либо действия, если каким-либо потокам будет дано управление

```
1  int memory = 0;
2  std::mutex lock_mutex;
3  int fetch_add() {
4      std::lock_guard<std::mutex> lock(lock_mutex); // mutex.lock();
5      int cur_memory = memory++;
6      // ~lock_guard<std::mutex>(lock)                // mutex.unlock();
7      return cur_memory;
8  }
```

- Неблокирующие

- OBSTRUCTION-FREE - мы гарантируем, что система будет продвигаться, если на пути одного потока не будет встречено никаких других. Иначе гарантии нет

```

1  int memory = 0;
2  std::atomic<bool> spin_lock{false};
3  int fetch_add() {
4      bool expected = false;
5      while (spin_lock.compare_exchange_weak(expected, true))
6          std::this_thread::yield();
7      int cur_memory = memory++;
8      spin_lock.store(false);
9      return cur_memory;
10 }

```

Данный пример НЕ является OBSTRUCTION-FREE, потому что система не будет продвигаться в случае выключения потоков в процессе работы. OBSTRUCTION-FREE являются методы, которые, например, сначала используют несколько CAS, и в случае отказа хотя бы одного выполняют операцию снова. Такая гарантия нужна, чтобы показать что процесс LOCK-FREE только в специфичных случаях.

- LOCK-FREE - мы гарантируем, что в целом система будет продвигаться в каждый момент времени, но нет гарантии, что та или иная операция будет завершена за конечное число шагов

```

1  std::atomic<int> memory{0};
2  int fetch_add() {
3      int cur_memory = memory.load();
4      while (!memory.compare_exchange_weak(cur_memory, cur_memory + 1)) {
5          cur_memory = memory.load();
6      }
7      return cur_memory;
8  }

```

- WAIT-FREE - самая сильная гарантия, мы уверены в том, что каждая операция будет завершена за конечное число шагов

```

1  std::atomic<int> memory{0};
2  int fetch_add() {
3      return memory.fetch_add(1);
4  }

```

В дальнейшем мы будем прибегать в общим приемам, которые эти гарантии предоставляют.

## 2 Особенности работы с памятью

У каждого потока есть своя локальная память, которую он может менять без опасений изменения другими потоками. Также у него есть доступ к общей памяти. Чтобы понять какие проблемы могут возникнуть при её изменении давайте представим следующие программы:

```

1  Object *ref;
2  void first() {
3      ref = new Object(A);
4      // the thread has stopped
5      if (ref != nullptr) {
6          // do smth with A
7      }
8  }

1  void second() {
2      if (ref != nullptr) delete ref;
3      ref = new Object(B);
4  }

```

Решением подобного рода проблем, связанных с удалением данных, является применение SMR(safe memory reclamation) алгоритмов (либо сборщика мусора для некоторых языков). Суть их очень проста - они сначала ждут завершения всех потоков, имеющих ссылку на удаляемый объект, и лишь затем осуществляют освобождение. Мы рассмотрим два из них, для начала поговорим о счётчике ссылок.

### 3 Счётчик ссылок

Первое решение, приходящее в голову - аккуратно подсчитывать количество ссылающихся на объект потоков, чтобы при достижении нуля начать освобождение. Давайте реализуем MUTEX-BASED подход

```
1 using ull = unsigned long long;
2 class ReferenceCounter {
3     ull counter = 1;
4     std::mutex lock_mutex;
5 public:
6     bool increment() {
7         std::lock_guard<std::mutex> lock(lock_mutex);
8         if (counter != 0) {
9             ++counter;
10            return true;
11        }
12        return false;
13    }
14    bool decrement() {
15        std::lock_guard<std::mutex> lock(lock_mutex);
16        if (counter == 0) return false;
17        if (--counter == 0) {
18            return true;
19        }
20        return false;
21    }
22    ull get_counter() {
23        std::lock_guard<std::mutex> lock(lock_mutex);
24        return counter;
25    }
26 };
```

Давайте его немного ускорим и превратим в LOCK-FREE

```
1 using ull = unsigned long long;
2 class ReferenceCounter {
3     std::atomic<ull> counter = 1;
4 public:
5     bool increment() {
6         ull cur_counter = counter.load();
7         while (cur_counter != 0 &&
8             !counter.compare_exchange_weak(cur_counter, cur_counter + 1)) {
9             cur_counter = counter.load();
10        }
11        return cur_counter != 0;
12    }
13    bool decrement() {
14        return counter.fetch_sub(1) == 1;
15    }
16    ull get_counter() {
17        return counter.load();
18    }
19 };
```

Заметим, что наше состояние всегда определено и на написание каждой операции мы не меняем функции. Но что здесь

```

1  using ull = unsigned long long;
2  class ReferenceCounter {
3      std::atomic<ull> counter = 1;
4      const ull zero_flag = (1ll << 63);
5  public:
6      bool increment() {
7          return ((counter.fetch_add(1) & zero_flag) == 0);
8      }
9      bool decrement() {
10         if (counter.fetch_sub(1) == 1) {
11             ull expected = 0;
12             if (counter.compare_exchange_strong(expected, zero_flag)) {
13                 return true;
14             }
15         }
16         return false;
17     }
18     ull get_counter() { // weak - the cake is a lie
19         ull cur_counter = counter.load();
20         if (cur_counter & zero_flag) return 0;
21         if (cur_counter == 0) return 1;
22         return cur_counter;
23         // we will change this function to explain help approach
24     }
25 };

```

Мы можем представить, что операция *increment* будет вызвана после *fetch\_sub(1)*, но до *compare\_exchange*. Тогда состояние счетчика будет 0 и операция прибавления 1, будучи реально вызванной после *decrement*, изменит состояние до неё, то есть по сути будет исполнена раньше. Нам необходимо будет определить что вообще значит порядок операций в терминах параллельного программирования, но для начала поговорим о простейшем применении данного счётчика.

### 3.1 Механизм COW для строк

Допустим мы хотим научиться сжимать создаваемую при копировании строк информацию, чего с помощью COW делать не стоит (более того его из-за некоторых причин удалили для строк в 11-ом стандарте C++). Тогда вместо обычного *basic\_string* вида

```
1 class basic_string {
2     size_t size;
3     char *c_str;
4 public:
5     basic_string(const basic_string &other) {
6         size = other.size;
7         c_str = new char[ size ];
8         memcpy(c_str, other.c_str, size);
9     }
10 };
```

Будем при копировании лишь ссылаться на тот же *c\_str*, не переписывая заново его содержимое. Если при изменении мы являемся исключительным владельцем данной строки, то тогда мы действительно поменяем данные. Иначе мы скопируем строку, как мы и хотели до этого.

```
1 class basic_string {
2     size_t size;
3     char *c_str;
4     ReferenceCounter *counter;
5 public:
6     basic_string(const basic_string &other) {
7         if (!other.counter->increment())
8             throw std::invalid_argument("");
9         size = other.size;
10        c_str = other.c_str;
11        counter = other.counter;
12    }
13    char& operator[](size_t index) {
14        if (counter->get_counter() > 1) {
15            char *new_c_str = new char[ size ];
16            memcpy(new_c_str, c_str, size);
17            if (counter->decrement()) {
18                delete c_str;
19            }
20            c_str = new_c_str;
21            counter = new ReferenceCounter();
22        }
23        return c_str[index];
24    }
25    ~basic_string() {
26        if (counter->decrement()) {
27            delete c_str;
28            delete counter;
29        }
30    }
31 };
```

## 3.2 Help подход и взаимопомощь

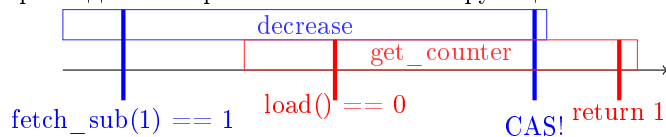
Чтобы добавить метод *get\_counter* в *ReferenceCounter* без трюков с переупорядочиванием давайте разберёмся с механизмом помощи. Пусть тот поток, который получил значение счётчика 1 при вызове *get\_counter* поможет установить его в *zero\_flag* и даст понять это потоку с вызванным *decrement*. В исходном материале, откуда было это всё взято есть дополнительная проверка на то, что только один поток из *decrement* смог бы поменять счётчик из состояния когда уже помогли и он ровно *zero\_flag*. Эта проверка - *counter.exchange(zero\_flag) & help\_flag*. Она не необходима, но её труднее убрать чем добавить.

```
1 using ull = unsigned long long;
2 class ReferenceCounter {
3     std::atomic<ull> counter = 1;
4     const ull zero_flag = (1ll << 63);
5     const ull help_flag = (1ll << 62);
6 public:
7     bool increment() {
8         return ((counter.fetch_add(1) & zero_flag) == 0);
9     }
10    bool decrement() {
11        if (counter.fetch_sub(1) == 1) {
12            ull expected = 0;
13            if (counter.compare_exchange_strong(expected, zero_flag)) {
14                return true;
15            } else if ((expected & help_flag) &&
16                (counter.exchange(zero_flag) & help_flag)) {
17                return true;
18            }
19        }
20        return false;
21    }
22    ull get_counter() { // strong - the cake is real
23        ull cur_counter = counter.load();
24        if (cur_counter == 0 &&
25            counter.compare_exchange_strong(cur_counter, zero_flag | help_flag)) {
26            return 0;
27        }
28        if (cur_counter & zero_flag) {
29            return 0;
30        }
31        return cur_counter;
32    }
33 };
```

Заметьте как получается использовать время другого потока, потраченное на помощь, чтобы развернуть бесконечный цикл в конечное исполнение на всей системе. Мы будем рассматривать это дальше в LOCK-FREE очереди Майкла и Скотта, которая использует эту технику для поддержания одного полезного свойства системы. Помимо прочего такой подход в принципе нужен по причине стабильности документации. Даже если мы будем аккуратно врать, то трудоемкость поддержки последней резко возрастает, нежели чем работать с ситуацией, где соблюдается инвариант структуры.

## 4 Линеаризуемость

Вернёмся к проблеме, возникшей при написании WAIT-FREE счётчика, связанной с `get_counter`. Давайте рассмотрим один из вариантов исполнения функции.

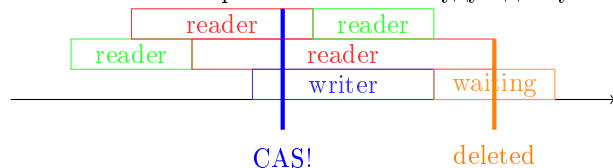


Объект называется *линеаризуемым*, если любая операция над ним выполняется *атомарно* в какой-то момент времени. И более того, любое чтение возвращает результат всех выполненных на момент его прихода записей. То есть нашу операцию можно воспринимать как *атомарную*, а всю систему - как набор последовательных инструкций, и именно это и позволяет нам писать идущие друг за другом операции в *линеаризуемой* системе, использующей эти *линеаризуемые* объекты. Все примеры счетчиков являются линеаризуемыми, поскольку мы можем *оправдать* их поведение соответствующим атомарным, так чтобы результат остался корректным. Является ли предыдущая реализация COW строк линеаризуемой? Почему?

Если бы наша согласованность операций была бы *последовательная*, чему есть термин *Sequential consistency*, то судя по ней "результат любого выполнения такой же, как в случае если бы **операции всех процессоров были выполнены в некотором последовательном порядке**, и операции каждого отдельного процессора появлялись в этой последовательности в порядке определённом его программой" (то есть в том самом порядке, какой дан из определения процесса). Является ли реализация COW строк последовательно согласованной? Почему?

## 5 RCU как несимметричный механизм SMR

Чтобы научиться выполнять операцию записи в многопоточную структуру данных необходимо учесть то, что в текущий момент есть потоки, которые выполняют оттуда чтение. Одним из решений, в основном применяемых в *kernel-space* (но также существующих в *user-space*) является RCU. Мы будем менять данные атомарно, затем дожидаться завершения всех читающих потоков, и только после этого изменять их. То есть при работе с ссылками каждый читающий поток входит в критическую секцию и запрещает прерывать себя. А пишущий поток пытается исполнить себя на каждом ядре, таким образом у него это выйдет только после того, как данные безопасно обработаются и не будут доступны.



Заметим также, что просто входить в критическую секцию для обеспечения WAIT-FREE подхода возможно только если система работает на одном ядре, в нашем же случае мы лишь используем этот механизм для бесплатного отслеживания состояния потоков. Также запись будет не то чтобы LOCK-FREE, скорее всего проще будет реализовать LOCKED алгоритм, зато чтение будет WAIT-FREE и практически не потребовать ничего для его реализации.

## 6 Lock-free очередь майкла-скотта

Представим себе задачу - нам необходимо написать LOCK-FREE очередь, так что у нас есть *head* и *tail* - ссылки на голову и хвост, соответственно. Изначально в ней лежит одна пустая вершина, при взятии которой с головы мы будем делать стоящую после неё вершину такой же пустой. Надо придумать инвариант, который потоки будут сохранять при работе с этой очередью, который сильно поможет в результате.

Пусть конец очереди никогда не будет отставать от настоящего конца более чем на 1 элемент. Если мы можем помочь сдвинуть *tail* на следующий за ним, то мы *сделаем* это. То есть мы для *enqueue* в бесконечном цикле сначала будем менять конец на следующий после него элемент, и если такого нет, то поменяем его на наш, добавляемый. Аналогично для *dequeue*, поскольку очередь может находиться в состоянии *head == tail* и мы не сможем забрать элемент. Затем мы просто меняем голову очереди и используем наш вынутый элемент. Чтобы его удалить нам понадобится какой-либо механизм SMR, но суть от этого не меняется.

## Лекция вторая

### 7 Односвязный список как последовательная структура данных

Для того, чтобы решить итоговую задачу нужно сначала научиться поддерживать структуру данных, в которой может происходить 3 операции: поиск элемента, вставка элемента в определенное место, удаление элемента. Давайте хранить односвязный список, состоящий из вершин с полями *T data* и *Node \*next*. Поиск будет происходить итерационно, вставка и удаление будут содержать в себе предварительный поиск, и в случае успеха, требуемую операцию.

1. Coarse-grained - обычным образом написанная структура, мы будем блокировать весь список, производить нашу операцию и затем разблокировать всё. В реализации можно заметить тип *recursive\_mutex*, который, как и обычный *mutex*, работает за счёт средств ядра ОС, вызывая системный вызов на блокировку. Примечателен он тем, что при повторном запросе от того же потока блокировку он не вызывает.
2. Fine-grained - давайте идти по вершинам, блокируя их только при входе и разблокируя при выходе. Так мы можем позволить немного разделить обход, но такое поведение не сильно улучшает ситуацию, поскольку мы всё ещё используем системные вызовы. Также заметим что у нас не может произойти никакого плохого случая, даже при том, что мы всё-таки используем параллельную обработку списка в нескольких местах одновременно.
3. Optimistic synchronization - позволим себе сначала пройти до вершины, и только затем заблокировать её. Если мы не смогли её снова найти, то есть её кто-то удалил, то повторим весь проход. И это релевантно только для удаления, поскольку вставка осуществляется сразу после элемента.
4. Lazy synchronization - не будем заново всё обходить, попробуем просто хранить флаг логической удаленности вершины. Мы будем его игнорировать при проходе, поскольку с ним ничего уже не сделать. При вставке после блокировки уже никто не сможет изменить его, поэтому его состояние актуально. При удалении аналогичная ситуация, только в случае удаленности предыдущей вершины необходимо заново её найти.
5. Non-blocking synchronization - попробуем использовать тот факт, что количество действий реальной обработки при удалении/добавлении мало. Будем использовать CAS, который работает только с 64-битными ссылками: заменим первый бит указателя на следующий элемент на логическую удаленность текущего. Мы сначала терпим небольшую неудачу при применении физического удаления после логического. Но, если сделать поиск больше не WAIT-FREE, а LOCK-FREE, добавив в него удаление логически вычеркнутых вершин, то алгоритм получается реализовать. Также можно заметить, что нельзя игнорировать удаленные вершины в этом случае, поскольку иначе они могут быть никогда не удалены.

И мы в итоге получаем наш LOCK-FREE односвязный список. Причем наша структура всегда линейаризуема, поскольку каждая операция выполняется за ровно одну инструкцию. Заметим также, что с помощью такого подхода можно реализовать и skip-list, позволяющий хранить данные точно также, но реализующий возможность быстрого поиска с асимптотикой  $O(\log(N))$ .

В такой структуре данных к каждому элементу добавляется сверху случайное (и грамотно подобранное) количество ссылок быстрого доступа, которые должны быть связаны между своим уровнем также, как и нижние. То есть ссылки от всех уровней первого элемента (который имеет наибольшую высоту) должны вести последовательно по списку в том же порядке до всех элементов того же уровня, причем если уровень выше, то и количество связей там меньше. Тогда для добавления элемента мы будем поочередно снизу вверх добавлять ссылки. Нижний слой skip-list'a закрепляет элементы и дает гарантию на их существование и порядок, все верхние слои служат лишь для ускоренного поиска. Удаление работает аналогичным образом - мы пытаемся сначала пометить все элементы сверху вниз, и только потом приступаем к физическому удалению в том же направлении.



## 7.1 Coarse-grained

```
1  template <typename T>
2  class List;
3
4  template <typename T>
5  class Node {
6      T      data;
7      Node<T> *next;
8
9      friend class List<t>;
10 public:
11     Node(T val) : data(val), next(nullptr) {}
12     T get_data() { return data; }
13 };
14
15 template <typename T>
16 class List {
17     Node<T> *head; // = dummy node
18     std::recursive_mutex lock_mutex;
19 public:
20     Node<T>* search_anc(T elem) {
21         std::lock_guard<std::recursive_mutex> lock(lock_mutex);
22         Node<T> *cur = head;
23         while (cur->next != nullptr) {
24             if (cur->next->T == elem) {
25                 return cur;
26             }
27             cur = cur->next;
28         }
29         return nullptr;
30     }
31     bool insert(T after, T elem) {
32         Node<T> *node = new Node<T>{elem};
33         std::lock_guard<std::recursive_mutex> lock(lock_mutex);
34         Node<T> *found_anc = this->search_anc(after);
35         if (found_anc != nullptr) {
36             Node<T> *curr = found_anc->next;
37             node->next = curr->next; curr->next = node;
38             return true;
39         }
40         return false;
41     }
42     bool delete(T elem) {
43         std::lock_guard<std::recursive_mutex> lock(lock_mutex);
44         Node<T> *found_anc = this->search_anc(elem);
45         if (found_anc != nullptr) {
46             found_anc->next = found_anc->next->next;
47             return true;
48         }
49         return false;
50     }
51 };
```

## 7.2 Fine-grained

```
1  template <typename T>
2  class Node {
3      T      data;
4      Node<T> *next;
5      std::mutex lock_mutex;
6
7      friend class List<t>;
8  public:
9      Node(T val) : data(val), next(nullptr) {}
10     T get_data() { return data; }
11 };
12
13 template <typename T>
14 class List {
15     Node<T> *head; // = dummy node
16 public:
17     // after search operation need to unlock Node<T> *result and result->next
18     Node<T>* search_anc(T elem) {
19         Node<T> *cur = head;
20         cur->lock_mutex.lock();
21         while (cur->next != nullptr) {
22             cur->next->lock_mutex.lock();
23             if (cur->next->T == elem) {
24                 return cur;
25             }
26             cur->lock_mutex.unlock();
27             cur = cur->next;
28         }
29         cur->lock_mutex.unlock();
30         return nullptr;
31     }
32     bool insert(T after, T elem) {
33         Node<T> *node = new Node<T>{elem};
34         Node<T> *found_anc = this->search_anc(after);
35         if (found_anc != nullptr) {
36             Node<T> *curr = found_anc->next;
37             node->next = curr->next; curr->next = node;
38             found_anc->lock_mutex.unlock(); curr->lock_mutex.unlock();
39             return true;
40         }
41         return false;
42     }
43     bool delete(T elem) {
44         Node<T> *found_anc = this->search_anc(elem);
45         if (found_anc != nullptr) {
46             found_anc->next = found_anc->next->next;
47             found_anc->lock_mutex.unlock(); curr->lock_mutex.unlock();
48             return true;
49         }
50         return false;
51     }
52 };
```

### 7.3 Optimistic synchronization

```
1  template <typename T>
2  class List {
3      Node<T> *head; // = dummy node
4  public:
5      pair<Node<T>*, Node<T>*> search_anc(T elem) {
6          Node<T> *cur = head;
7          Node<T> *next_cur = cur->next;
8          while (next_cur != nullptr) {
9              if (next_cur->T == elem) {
10                 return {cur, next_cur};
11             }
12             cur = next_cur;
13             next_cur = next_cur->next;
14         }
15         return {nullptr, nullptr};
16     }
17     bool insert(T after, T elem) {
18         Node<T> *node = new Node<T>{elem};
19         while (true) {
20             auto [found_anc, curr] = this->search_anc(after);
21             if (found_anc == nullptr) return false;
22
23             std::lock_guard<std::mutex> lock_next(curr->lock_mutex);
24             if (this->search_anc(after).second != curr) return false;
25
26             node->next = curr->next; curr->next = node;
27             return true;
28         }
29         return false;
30     }
31     bool delete(T elem) {
32         while (true) {
33             auto [found_anc, curr] = this->search_anc(elem);
34             if (found_anc == nullptr) return false;
35
36             std::lock_guard<std::mutex> lock(found_anc->lock_mutex);
37             std::lock_guard<std::mutex> lock_next(curr->lock_mutex);
38             if (found_anc->next != curr) continue;
39             if (this->search_anc(elem).first != found_anc) continue;
40
41             found_anc->next = curr->next;
42             return true;
43         }
44         return false;
45     }
46 };
```

## 7.4 Lazy synchronization

```
1  template <typename T>
2  class Node {
3      T      data;
4      Node<T> *next;
5      volatile bool    alive;
6  };
7
8  template <typename T>
9  class List {
10     Node<T> *head; // = dummy node
11 public:
12     pair<Node<T>*, Node<T>*> search_anc(T elem) {
13         Node<T> *cur = head;
14         Node<T> *next_cur = cur->next;
15         while (next_cur != nullptr) {
16             if (next_cur->T == elem) {
17                 return {cur, next_cur};
18             }
19             cur = next_cur;
20             next_cur = next_cur->next;
21         }
22         return {nullptr, nullptr};
23     }
24     bool insert(T after, T elem) {
25         Node<T> *node = new Node<T>{elem};
26         while (true) {
27             auto [found_anc, curr] = this->search_anc(after);
28             if (found_anc == nullptr) return false;
29
30             std::lock_guard<std::mutex> lock_next(curr->lock_mutex);
31             if (!curr->alive) return false;
32
33             node->next = curr->next; curr->next = node;
34             return true;
35         }
36         return false;
37     }
38     bool delete(T elem) {
39         while (true) {
40             auto [found_anc, curr] = this->search_anc(elem);
41             if (found_anc == nullptr) return false;
42
43             std::lock_guard<std::mutex> lock(found_anc->lock_mutex);
44             std::lock_guard<std::mutex> lock_next(curr->lock_mutex);
45             if (!curr->alive) return false;
46             if (found_anc->next != curr) continue;
47             if (!found_anc->alive) continue;
48
49             curr->alive = false; found_anc->next = curr->next;
50             return true;
51         }
52         return false;
53     }
54 };
```

## 7.5 Non-blocking synchronization

```
1 // Node<T>.next is atomic<Node<T>*>
2 const size_t alive_flag = (1ll << 63); // Node<T>.next has an alive_flag
3 const size_t alive_mask = ~alive_flag; // (ref & alive_mask) to use the pointer
4
5 template <typename T>
6 class List {
7     pair<Node<T>*, Node<T>*> search_anc(T elem) {
8         Node<T> *cur = head;
9         Node<T> *next_cur = cur->next; // like atomic.load()
10        while (next_cur != nullptr) {
11            while (!(next_cur & alive_flag)) { // helping delete
12                Node<T> *expected = next_cur;
13                if (!cur->next.compare_exchange_strong
14                    (expected, next_cur->next)) {
15                    next_cur = head->next;
16                    // otherwise haven't even lock-free
17                    // we need to do full traversal
18                }
19            }
20            if (next_cur->T == elem) { // like using alive_mask
21                return {cur, next_cur};
22            }
23            cur = next_cur;
24            next_cur = next_cur->next; // like using alive_mask
25        }
26        return {nullptr, nullptr};
27    }
28    bool insert(T after, T elem) {
29        Node<T> *node = new Node<T>{elem};
30        while (true) {
31            auto [found_anc, curr] = this->search_anc(after);
32            if (found_anc == nullptr) return false;
33            Node<T> *expected = curr->next.load(); //preparing for CAS
34            if (!(expected & alive_flag)) return false;
35            node->next.store(expected);
36            if (!curr->next.compare_exchange_weak(expected, node)) continue;
37            return true;
38        }
39        return false;
40    }
41    bool delete(T elem) {
42        while (true) {
43            auto [found_anc, curr] = this->search_anc(elem);
44            if (found_anc == nullptr) return false;
45            Node<T> *expected = curr->next.load(); //preparing for CAS
46            if (!(expected & alive_flag)) return false;
47            if (!curr->next.compare_exchange_weak
48                (expected, expected & alive_flag)) continue;
49            found_anc->next.compare_exchange_strong(curr, expected);
50            // if last CAS failed then other threads will help!
51            return true;
52        }
53        return false;
54    }
55 };
```

## 8 Универсальные WAIT-FREE конструкции

### 8.1 Грамотная постановка задачи

Гарантии многопоточного программирования нужны не столько по причине скорости их работы (которая является следствием), сколько по их надежности и способности обеспечивать максимальное использование ресурсов вычислительных устройств. Но в некоторых задачах физически невозможно реализовать параллельное исполнение. То есть если в системе  $P$  потоков, а время выполнения одной операции  $T$ , то как ни крути, суммарное время будет  $O(PT)$ . Чем же *lock-free* себя выделяет по сравнению с подходом без гарантий? Тем, что на самом деле при остановке исполнения одного из потоков он может обеспечить работу остальных, и хотя бы один выполнит нужную ему операцию. Давайте попытаемся придумать способ, как сделать именно это - обеспечить возможность каждому из потоков выполняться, несмотря на состояние всех остальных

Задача: В системе есть  $P$  потоков исполнения. Каждому на выполнение своей операции над структурой данных  $Rx$  требуется  $T$  времени. Необходимо написать способ распределения задач, чтобы при засыпании/просыпании любого из потоков каждый выполнялся бы за конечное число шагов.

### 8.2 Тривиальное решение

При грамотной постановке задачи сразу приходит решение, хоть и медленное. Оно было описано ещё очень давно Морисом Херлихи, и заключается в следующем: мы пытаемся выполнить старые операции всех остальных потоков, пока у нас это не получится. Если у нас так и не вышло, то значит, что кто-то другой уже сделал это за нас.

```
1 Operation announce[P];
2 Result do_operation(Operation x) {
3     x.time = operation_counter->increment();
4     announce[this_thread_id] = x;
5     while (!announce[i].is_done()) {
6         Operation announce' = announce.copy();
7         Rx' = Rx.copy();
8         newRx' = Rx';
9         for (int i : {0..P-1}) {
10             if (!announce'[i].is_done()) {
11                 // next rule is required to achieve wait-freedom
12                 if (announce'[i].time <= x.time) {
13                     newRx'->announce'[i](); // do an operation
14                     announce'[i].done();
15                 }
16             }
17         }
18         if (!CAS(Rx, Rx', newRx')) continue;
19
20         for (int i : {0..P-1}) {
21             if (announce'[i].is_done()) {
22                 CAS(announce[i], announce'[i].like_not_done, announce'[i]);
23             }
24         }
25     }
26     return done;
27 }
```

Можно заметить что такое решение работает за  $O(P^2T)$  сложность на каждом из потоков, хоть и является WAIT-FREE. Необходимо придумать что-то похожее, но оптимальнее. Также можно заметить, что невозможно отменить операцию, даже если у неё был сделан анонс - по крайней мере здесь это является достаточно сложной задачей. Попробуем применить подход с нумерацией операций.

### 8.3 Решение с помощью LOCK-FREE очереди

Следующим шагом к созданию алгоритма является применения понятия очереди. Пусть все новые процессы будут заносить анонс своей операции в очередь, и потом любой, кто захочет сможет оттуда взять последнюю. Как только кто-то её выполнит мы удалим этот элемент и будем пытаться выполнить следующую операцию. Так или иначе каждому потоку повезет быть исполненным.

```
1 Queue<Operation> announces;
2 Result do_operation(Operation x) {
3     x.time = operation_counter->increment();
4     announces.push(x);
5     while (!x.is_done()) {
6         Operation announce' = announces.front();
7         Rx' = Rx.copy();
8         newRx' = Rx';
9         if (announce'.time <= x.time) { // should be always true
10             newRx'->announce'(); // do an operation
11             announce'.done();
12         }
13         CAS(Rx, Rx', newRx');
14         announce'->done(); // for !x.is_done() rule
15
16         // if announce.front() is announce' then pop
17         // otherwise someone else removed this
18         announce.pop(announce');
19         free(announce');
20     }
21     return done;
22 }
```

Наша асимптотика стала  $O(PT)$ , чего мы и хотели добиться. Заметим, что для начала нам нужна WAIT-FREE очередь, а также механизм освобождения памяти, выделенной под анонс, чтобы реализовать данный подход. Написание WAIT-FREE аллокации памяти не является тривиальной задачей, попробуем использовать преобразование очереди.

Нам по факту не важно будет ли поток, который первым добавился в очередь, исполнен первым, нам нужно лишь чтобы все операции рано или поздно были исполнены. Тогда пусть наша очередь будет расположена на массиве, и у нас будет единый счетчик текущей исполняемой операции. Назовем его *gate*, также в нём же будем хранить количество последних исполненных операций, т.е. наш *operation\_counter*.

## 8.4 Добавление gate

```
1 struct Gate {
2     size_t cur; // pointer - current active operation
3     size_t seq; // counter - number of done operations
4 };
5 Operation announce[P];
6 Gate gate = {nullptr, 0};
7
8 Result do_operation(Operation x) {
9     i = this_thread::get_id();
10    x.time = gate.seq;
11    announce[i] = x;
12    Gate gate';
13    while (!x.is_done()) {
14        gate' = gate.load();
15        if (gate'.cur == nullptr) {
16            Operation announce' = announce[gate'.seq % P];
17            if (announce'.is_done()) {
18                announce' = x;
19            }
20            if (announce'.is_done()) break;
21            CAS(gate, {gate', {announce', gate'.seq}});
22            gate' = gate.load();
23        }
24        if (gate'.cur != nullptr) {
25            Operation announce' = announce[gate'.cur];
26            // gate' is like a pointer to head of the queue
27            // that's why we really need to check
28            if (announce'.time <= gate'.seq) { // gate' isn't old
29                Rx' = Rx.copy();
30                newRx' = Rx';
31                newRx'->announce'(); // do an operation
32                CAS(Rx, Rx', newRx');
33            }
34            CAS(announce[gate'.cur], announce', announce'.done());
35            CAS(gate, {gate', {nullptr, gate'.seq + 1}});
36        }
37    }
38    if (gate'.cur == i) {
39        CAS(gate, {gate', {nullptr, gate'.seq + 1}});
40    }
41    return done;
42 }
```



## 8.5 Доведение до кондиции

Во всех случаях мы упустили один момент: нам нужно обеспечить возможность доступа к старым значениям структуры данных, даже если их только что обновили. Поскольку потоки могут "заснуть" ровно перед моментом помощи, нужно сделать две версии структуры, которые будут чередоваться по использованию. Теоретически их количество может быть неограниченным, но больше чем две нам обеспечивать не нужно, ведь после копирования и до выполнения операции мы можем проверить временную корректность данных.

Также необходимо учесть ситуацию, когда один из потоков уже завершил выполнение операции полностью, переключил *toggle* значение в структуре данных, но ещё не выполнил CAS, который фиксирует завершенность *announce[gate'.cur].done()*. И даже после этого стоило бы добавить логику обработки отмены операций, которая должна быть описана примерно в каждой части проверок, из-за чего псевдокод очень сильно увеличится в размере. Поэтому эту часть я описывать не буду.

```
1 struct Version {
2     bool toggle; // version — bit, which resolved current problem
3     size_t seq; // counter — number of done operations
4
5 };
6 Version version{0};
7
8 Result do_operation(Operation x) {
9     ...
10         if (gate'.cur != nullptr) {
11             Operation announce' = announce[gate'.cur];
12             if (announce'.time <= gate'.seq) { // gate' isn't old
13                 Rx' = Rx.copy();
14                 version' = version.load();
15                 if (version'.seq > gate'.seq) break;
16
17                 before = &Rx'[version'.toggle];
18                 after = &Rx'[version'.toggle ^ 1];
19                 if (version'.seq == gate'.seq) { // because CAS(announce)
20                     swap(*before, *after);
21                 }
22
23                 after = &(*before)→announce'(); // do an operation
24                 CAS(Rx, Rx', {*before, *after});
25
26                 if (version'.seq < gate'.seq)
27                     CAS(version, version'.toggle, version'.toggle ^ 1);
28             }
29             CAS(announce[gate'.cur], announce', announce'.done());
30             CAS(gate, {gate', {nullptr, gate'.seq + 1}});
31         }
32     }
33     ...
34 }
```

Если рассуждать грамотно, то проверка на соответствие версий структур должна быть полная, с учетом предыдущего состояния структуры. Для этого стоит не всей структуре дать версию, а каждому элементу в ней. Также в оригинальной статье про универсальные конструкции также рассмотрен способ встроенной и поэтому удобной аллокации. Если есть желание, то предлагаю её на чтение, но данное усложнение лишь мешает пониманию общих принципов, поэтому я его не привожу.

## 9 WAIT-FREE красно-черное дерево

Теперь мы изучили достаточно и готовы решить итоговую задачу. Для этого см. оригинальную статью Аравинда Натараджана, эта часть должна быть в точности по ней.

## 10 Литература

### Лекция 1

1. The art of Multiprocessor Programming <https://www.cs.sfu.ca/~ashriram/Courses/CS431/assets/distrib/AMP.pdf>
2. Курс по параллельному программированию <https://youtu.be/fhcyQ2wU7Hk?si=QI1Qx9BlBxH4VP18>
3. Подход COW и его описание <https://en.wikipedia.org/wiki/Copy-on-write>
4. Документация по типам *atomic* в C++ <https://en.cppreference.com/w/cpp/atomic/atomic.html>
5. Введение в WAIT-FREE программирование <https://www.youtube.com/watch?v=kPh8pod0-gk>

### Лекция 2

1. Статьи Максима Хижинского по LOCK-FREE <https://habr.com/ru/users/khizmax/articles/>
2. Библиотека *libcds* того же автора <https://github.com/khizmax/libcds>
3. Представление необходимых принципов для структур <https://oasis.library.unlv.edu/cgi/viewcontent.cgi?article=3425&context=thesesdissertations>
4. Универсальные конструкции WAIT-FREE <https://scispace.com/pdf/a-universal-construction-for-wait-free-transaction-friendly-38as61nsj3.pdf>
5. WAIT-FREE красно-черное дерево поиска [https://link.springer.com/chapter/10.1007/978-3-319-03089-0\\_4](https://link.springer.com/chapter/10.1007/978-3-319-03089-0_4)