

Многоядерные структуры и поиск

Первунецких Еремей

10 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 много больше того, что выходит за рамки обычного курса многопоточного программирования и затрагивает недавние идеи, созданные в этой области

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

```
int memory = 0;
std::mutex lock_mutex;
int fetch_add() {
    std::lock_guard<std::mutex> lock(lock_mutex); // mutex.lock();
    int cur_memory = memory++;
    // ~lock_guard<std::mutex>(lock)                // mutex.unlock();
    return cur_memory;
}
```

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

```
int memory = 0;
std::atomic<bool> spin_lock{false};
int fetch_add() {
    bool expected = false;
    while (spin_lock.compare_exchange_weak(expected, true))
        std::this_thread::yield();
    int cur_memory = memory++;
    spin_lock.store(false);
    return cur_memory;
}
```

- Неблокирующие

- 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

```
std::atomic<int> memory{0};
int fetch_add() {
    int cur_memory = memory.load();
    while (!memory.compare_exchange_weak(cur_memory, cur_memory + 1)) {
        cur_memory = memory.load();
    }
    return cur_memory;
}
```

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

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

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:

```
Object *ref;
void first() {
    ref = new Object(A);
    // the thread has stopped
    if (ref != nullptr) {
        // do smth with A
    }
}

void second() {
    if (ref != nullptr) delete ref;
    ref = new Object(B);
}
```

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

```
using ull = unsigned long long;
class ReferenceCounter {
    ull counter = 1;
    std::mutex lock_mutex;
public:
    bool increment() {
        std::lock_guard<std::mutex> lock(lock_mutex);
        if (counter != 0) {
            ++counter;
            return true;
        }
        return false;
    }
    bool decrement() {
        std::lock_guard<std::mutex> lock(lock_mutex);
        if (counter == 0) return false;
        if (--counter == 0) {
            return true;
        }
        return false;
    }
    ull get_counter() {
        std::lock_guard<std::mutex> lock(lock_mutex);
        return counter;
    }
};
```

Давайте его немного ускорим и превратим в LOCK-FREE

```
using ull = unsigned long long;
class ReferenceCounter {
    std::atomic<ull> counter = 1;
public:
    bool increment() {
        ull cur_counter = counter.load();
        while (cur_counter != 0 &&
            !counter.compare_exchange_weak(cur_counter, cur_counter + 1)) {
            cur_counter = counter.load();
        }
        return cur_counter != 0;
    }
    bool decrement() {
        return counter.fetch_sub(1) == 1;
    }
    ull get_counter() {
        return counter.load();
    }
};
```

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

```
using ull = unsigned long long;
class ReferenceCounter {
    std::atomic<ull> counter = 1;
```

```

    const ull zero_flag = (1ll << 63);
public:
    bool increment() {
        return ((counter.fetch_add(1) & zero_flag) == 0);
    }
    bool decrement() {
        if (counter.fetch_sub(1) == 1) {
            ull expected = 0;
            if (counter.compare_exchange_strong(expected, zero_flag)) {
                return true;
            }
        }
        return false;
    }
    ull get_counter() {
        // why not just:
        ull cur_counter = counter.load();
        if (cur_counter & zero_flag) return 0;
        if (cur_counter == 0) return 1;
        return cur_counter;
        // give an example of situation when it doesn't work
    }
};

```

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

```
class basic_string {
    size_t size;
    char *c_str;
public:
    basic_string(const basic_string &other) {
        size = other.size;
        c_str = new char[size];
        memcpy(c_str, other.c_str, size);
    }
};
```

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

```
class basic_string {
    size_t size;
    char *c_str;
    ReferenceCounter *counter;
public:
    basic_string(const basic_string &other) {
        if (!other.counter->increment())
            throw std::invalid_argument("");
        size = other.size;
        c_str = other.c_str;
        counter = other.counter;
    }
    char& operator[](size_t index) {
        if (counter->get_counter() > 1) {
            char *new_c_str = new char[size];
            memcpy(new_c_str, c_str, size);
            if (counter->decrement()) {
                delete c_str;
            }
            c_str = new_c_str;
            counter = new ReferenceCounter();
        }
        return c_str[index];
    }
    ~basic_string() {
        if (counter->decrement()) {
            delete c_str;
            delete counter;
        }
    }
};
```

Но здесь всё равно есть одна ошибка, которую мы вообще не можем никак устранить (из-за принципов, по которым идёт присваивание по номеру элемента в C++). Попробуйте сказать, какая

3.2 Help подход и взаимопомощь

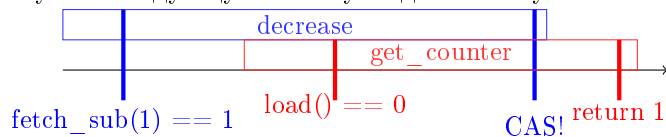
Чтобы добавить метод *get_counter* в *ReferenceCounter* давайте разберёмся с механизмом помощи. Пусть тот поток, который получил значение счётчика 1 при вызове *get_counter* поможет установить его в *zero_flag* и даст понять это потоку с вызванным *decrement*. В исходном материале, откуда было это всё 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

```
using ull = unsigned long long;
class ReferenceCounter {
    std::atomic<ull> counter = 1;
    const ull zero_flag = (1ll << 63);
    const ull help_flag = (1ll << 62);
public:
    bool increment() {
        return ((counter.fetch_add(1) & zero_flag) == 0);
    }
    bool decrement() {
        if (counter.fetch_sub(1) == 1) {
            ull expected = 0;
            if (counter.compare_exchange_strong(expected, zero_flag)) {
                return true;
            } else if ((expected & help_flag) &&
                (counter.exchange(zero_flag) & help_flag)) {
                return true;
            }
        }
        return false;
    }
    ull get_counter() {
        ull cur_counter = counter.load();
        if (cur_counter == 0 &&
            counter.compare_exchange_strong(cur_counter, zero_flag | help_flag)) {
            return 0;
        }
        if (cur_counter & zero_flag) {
            return 0;
        }
        return cur_counter;
    }
};
```

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

4 Линеаризуемость

Вернёмся к проблеме, возникшей при написании WAIT-FREE счётчика. Нам можно было бы просто вернуть 1. И получить следующую ошибку в одном из случаев исполнения



Но в том то и соль, что исполнение у нас было бы верным, если бы было другое требование к порядку применения операций. 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). Нам такой критерий не подходит, он рушится на *if (get_count() > 0) decrement()*; будет выдавать неверный результат на данном примере

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

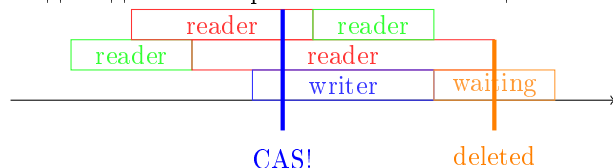
5 Lock-free очередь майкла-скотта

Чтобы создать самую полезную структуру данных из всех - очередь, нам необходимо понять одну простую мысль - если мы можем помочь сдвинуть конец очереди на следующий за ним элемент, то мы *сделаем* это. И тогда ссылка на него никогда не будет отставать от настоящего конца более чем на один элемент ни в какой момент времени.

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 либо счётчик ссылок (который невозможно так просто применить без hazard pointers, по причине, отчасти появившейся в COW подходе), либо RCU

6 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 удалять их



7 Заметка на будущее: задача про крыс

На следующей лекции мы, используя всё то, что сегодня изучили, попытаемся решить следующую задачу через построение быстрого lock-free (а также wait-free) дерева поиска:

У нас есть сто миллионов крыс и большой суперкомпьютер. У каждой твари есть своё имя. Нам приходят по 20 миллионов запросов в секунду о том, что крыса под именем *Name* съела *Count* головок сыра. Также иногда, раз в несколько секунд, крысы умирают и рождаются новые. Требуется в каждый момент времени называть имя любой крысы, которая съела больше всего сыра