Мьютексы

Мьютекс (англ. mutex, от mutual exclusion — «взаимное исключение») — это базовый механизм синхронизации. Он предназначен для организации взаимоисключающего доступа к общим данным для нескольких потоков с использованием барьеров памяти (для простоты можно считать мьютекс дверью, ведущей к общим данным).

Синтаксис

- Заголовочный файл | #include <mutex>
- Объявление | std::mutex mutex_name;
- Захват мьютекса | mutex_name.lock();

Поток запрашивает монопольное использование общих данных, защищаемых мьютексом. Дальше два варианта развития событий: происходит захват мьютекса этим потоком (и в этом случае ни один другой поток не сможет получить доступ к этим данным) или поток блокируется (если мьютекс уже захвачен другим потоком).

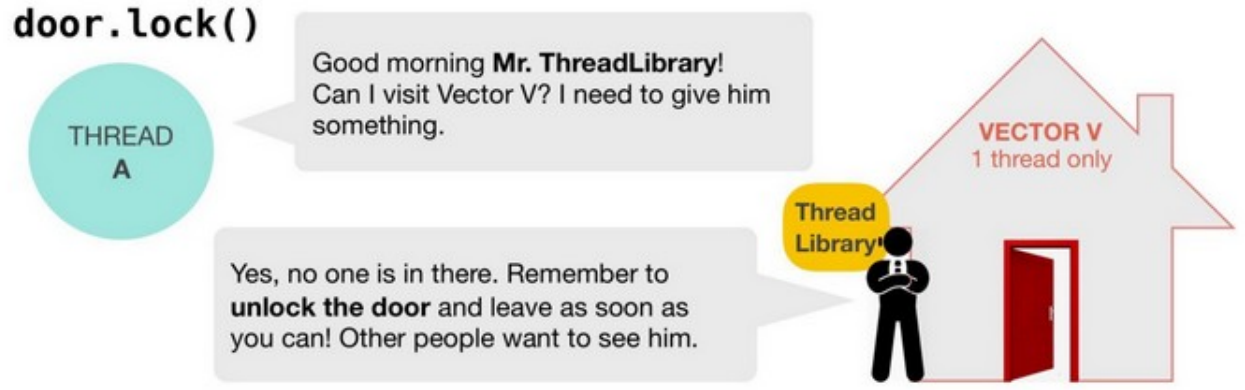
Освобождение мьютекса | mutex_name.unlock();

Когда ресурс больше не нужен, текущий владелец должен вызвать функцию разблокирования unlock, чтобы и другие потоки могли получить доступ к этому ресурсу. Когда мьютекс освобождается, доступ предоставляется одному из ожидающих потоков.

Мьютексы

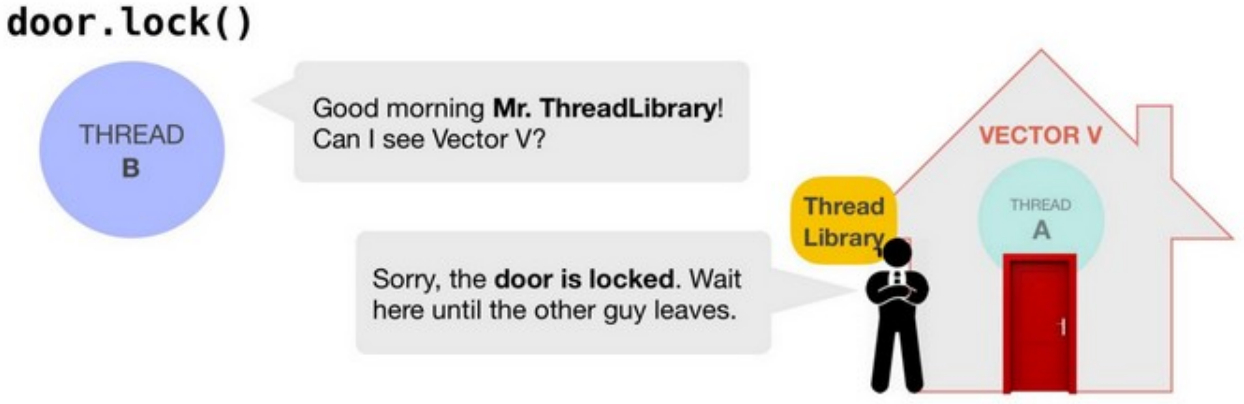
```cpp
#include <mutex>
#include <vector>

int main()
{
    std::mutex door;    // объявление мьютекса
    std::vector<int> v; // общие данные
    door.lock();
    /*------------------------*/
    /* Это потокобезопасная зона: допускается только один поток за раз
     *
     * Гарантируется монопольное использование вектора v
     */
     /*------------------------*/
    door.unlock();
}
```

Мьютексы



after some nanoseconds . . .

@valentina.codes



Пример работы с блокировками: потокобезопасная очередь

Разберёмся, как реализовать простейшие потокобезопасные очереди, то есть очереди с безопасным доступом для потоков.

В библиотеке стандартных шаблонов уже есть готовая очередь (**queue**). Наша реализация будет предполагать: а) **извлечение и удаление целочисленного значения из начала очереди** и б) **добавление нового в конец очереди**. И всё это при обеспечении потокобезопасности.

Сначала выясним, почему и как эти две операции могут создавать проблемы для многопоточности.

Извлечение и удаление

Для извлечения и удаления значения из начала очереди необходимо выполнить три операции:

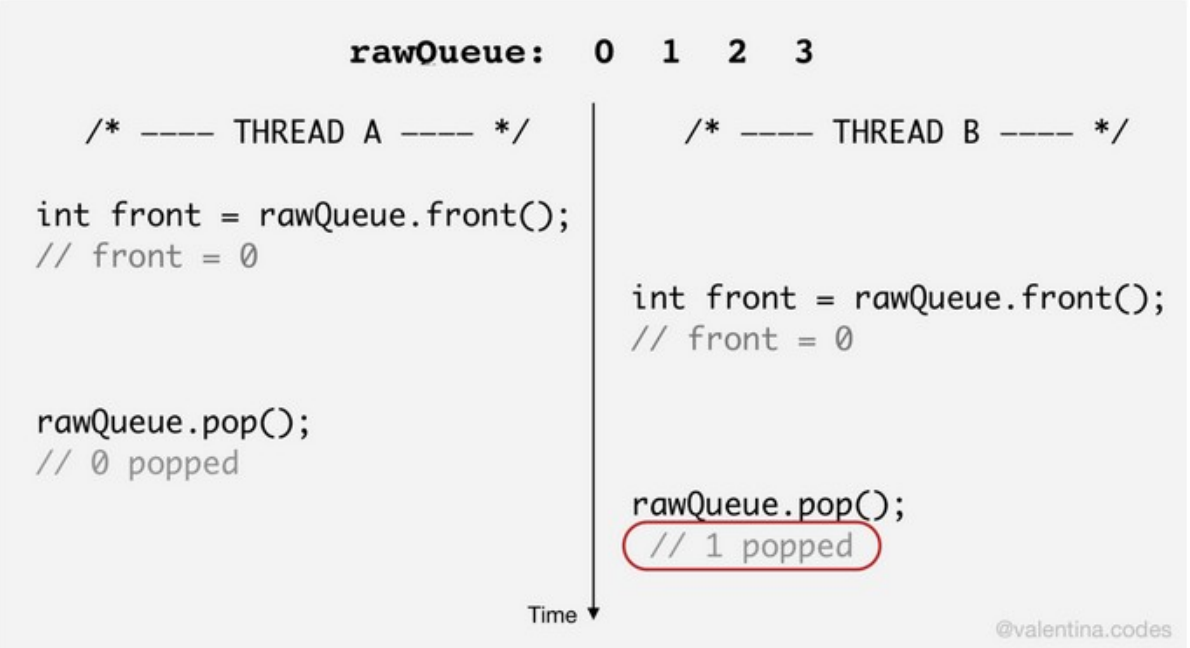
1. Проверить, не пуста ли очередь.

2. Если нет, получается ссылка на начало очереди (rawQueue.front()).

3. Удаляется начало очереди (rawQueue.pop()).

В промежутках между этими этапами к очереди могут получать доступ и другие потоки с целью внесения изменений или чтения. Попробуйте сами.

Пример работы с блокировками: потокобезопасная очередь



Проблемы для многопоточности:

Удалили “1”, хотя до его извлечения даже не дошли, потому что поток B извлекает 0 и удаляет 1.

Дальше — больше: если rawQueue состоит из одного элемента, поток B видит непустую очередь, и тут же поток A удаляет последнее значение. Теперь поток B пытается удалить первое значение из пустой очереди, приводя к неопределённому поведению.

Пример работы с блокировками: потокобезопасная очередь

Добавление.
Рассмотрим теперь добавление нового значения с помощью rawQueue.push(): новый элемент добавляется в конец контейнера и становится следующим за последним на данный момент элементом.
Дальше на единицу увеличивается размер.

Заметили здесь проблему? А что, если два потока одновременно добавят новое значение, увидев этот последний элемент? И что может произойти в интервале между добавлением нового элемента и увеличением размера? Кто-нибудь возьмёт да и прочитает **неправильный размер**.

Пример работы с блокировками: потокобезопасная очередь

```
class threadSafe_queue {

    std::queue<int> rawQueue; // структура, общая для всех потоков
    std::mutex m; // красная дверь rawQueue

public:

    int& retrieve_and_delete() {
        int front_value = 0; // если пустая, возвращает 0
        m.lock();
        // Текущий поток единственный, который имеет доступ к rawQueue
        if( !rawQueue.empty() ) {
            front_value = rawQueue.front();
            rawQueue.pop();
        }
        m.unlock();
        // теперь другие потоки могут захватить мьютекс
        return front_value;
    };

    void push(int val) {
        m.lock();
        rawQueue.push(val);
        m.unlock();
    };

};
```

Пример работы с блокировками: потокобезопасная очередь lock_guard

Недостатки предыдущего способа реализации блокировки:

- Что произойдёт, если мы забудем вызвать **unlock()**? Ресурс будет недоступен в течение всего времени существования мьютекса, и уничтожение последнего в неразблокированном состоянии приведёт к неопределённому поведению.

- Что произойдёт, если до вызова unlock() будет выброшено **исключение**? unlock() так и не будет исполнен, а у нас будут все перечисленные выше проблемы.

Пример работы с блокировками: потокобезопасная очередь lock_guard

К счастью, проблемы можно решить с помощью класса **std::lock_guard**.

Он всегда гарантированно освобождает мьютекс, используя парадигму **RAII** (Resource Acquisition Is Initialization, что означает «получение ресурса есть инициализация»).

Вот как это происходит: мьютекс инкапсулируется внутри lock_guard, который вызывает lock() в его конструкторе и unlock() в деструкторе при выходе из области видимости. Это безопасно даже при исключениях: раскрутка стека уничтожит lock_guard, вызывая деструктор и таким образом освобождая мьютекс.

lock_guard

```cpp
#include <mutex>
#include <vector>

std::mutex door;    // объявление мьютекса
std::vector<int> v;
{
     std::lock_guard<std::mutex> lg(door);
     /* Вызывается конструктор lg, эквивалентный door.lock();
      * lg, размещается в стеке */
     /*-----------------------*/

        /* Гарантируется монопольное использование вектора v */

     /*-----------------------*/
} /* lg выходит из области видимости. Вызывается деструктор, эквивалентный door.unlock(); */
```

Пример работы с блокировками: потокобезопасная очередь lock_guard

```cpp
#include <mutex>
#include <queue>

class threadSafe_queue {

    std::queue<int> rawQueue; // структура, общая для всех потоков
    std::mutex m; // красная дверь rawQueue

public:

    int& retrieve_and_delete() {
        int front_value = 0; // если пустая, return будет 0
        std::lock_guard<std::mutex> lg(m);
        // Отныне текущий поток единственный, который имеет доступ к rawQueue
        if( !rawQueue.empty() ) {
            front_value = rawQueue.front();
            rawQueue.pop();
        }
        return front_value;
    };  // теперь другие потоки могут захватить мьютекс

    void push(int val) {
        std::lock_guard<std::mutex> lg(m);
        // отныне текущий поток единственный, который имеет доступ к rawQueue
        rawQueue.push(val);
    }; // теперь другие потоки могут захватить мьютекс
};
```

unique_lock

Как только владение мьютексом получено (благодаря std::lock_guard), он может быть освобождён. **std::unique_lock** действует в схожей манере плюс делает возможным **многократный захват и освобождение** (всегда в таком порядке) мьютекса, используя те же преимущества безопасности парадигмы RAII.

```
#include <mutex>
#include <vector>

std::mutex door; //объявление мьютекса
std::vector<int> v;
{
     std::unique_lock<std::mutex> ul(door);
     // Вызывается конструктор ul, эквивалентный door.lock();
     // ul, размещённый в стеке
     // гарантируется монопольное использование вектора

     door.unlock();

     // выполнение операций, не связанных с вектором
     // ....
     // теперь мне снова нужен доступ к вектору

     door.lock();
     //Снова гарантируется монопольное использование вектора
} /* unique_lock выходит из области видимости. Вызывается
деструктор, эквивалентный door.unlock(); */
```

shared_mutex

std::mutex — это мьютекс, которым одномоментно может владеть только один поток. Однако это ограничение не всегда обязательно. Например, потоки могут одновременно и безопасно читать одни и те же общие данные. Просто читать, не производя с ними никаких изменений. Но в случае с доступом к записи только записывающий поток может иметь доступ к данным.

Начиная с C++17, **std::shared_mutex** формирует доступ двух типов:

- Общий доступ: потоки вместе могут владеть мьютексом и иметь доступ к одному и тому же ресурсу. Доступ такого типа можно запросить с помощью std::shared_lock (lock guard для общего мьютекса). При таком доступе любой эксклюзивный доступ блокируется.

- Эксклюзивный доступ: доступ к ресурсу есть только у одного потока. Запрос этого типа осуществляется с помощью класса unique lock.

shared_mutex

```cpp
#include <shared_mutex>
#include <vector>

std::shared_mutex door; //объявление мьютекса
std::vector<int> v;
int readVectorSize() {
    /* потоки могут вызывать эту функцию одновременно
     * доступ на запись запрещена, когда получен sl */

    std::shared_lock<std::shared_mutex> sl(door);
    return v.size();
}
void pushElement(int new_element) {
    /* гарантирован эксклюзивный доступ к вектору */

    std::unique_lock<std::shared_mutex> ul(door);
    v.push_back(new_element);
}
```

Взаимоблокировка

Оба потока для выполнения некоторой операции должны захватить два мьютекса, но сложилось так, что каждый поток захватил только один мьютекс и ждет другого. Ни один поток не может продолжить, так как каждый ждет, пока другой освободит нужный ему мьютекс. Такая ситуация называется взаимоблокировкой; это самая трудная из проблем, возникающих, когда для выполнения операции требуется захватить более одного мьютекса.

Общая рекомендация, как избежать взаимоблокировок, заключается в том, чтобы всегда захватывать мьютексы в одном и том же порядке, – если мьютекс A всегда захватывается раньше мьютекса B, то взаимоблокировка не возникнет. Иногда это просто, потому что мьютексы служат разным целям, а иногда совсем не просто, например, если каждый мьютекс защищает отдельный объект одного и того же класса. Рассмотрим, к примеру, операцию сравнения двух объектов одного класса. Чтобы сравнению не мешала одновременная модификация, необходимо захватить мьютексы для обоих объектов. Однако, если выбрать какой-то определенный порядок (например, сначала захватывается мьютекс для объекта, переданного в первом параметре, а потом – для объекта, переданного во втором параметре), то легко можно получить результат, обратный желаемому: стоит двум потокам вызвать функцию сравнения, передав ей одни и те же объекты в разном порядке, как мы получим взаимоблокировку!

Взаимоблокировка

Функция **std::lock** - захватывает сразу два и более мьютексов без риска получить взаимоблокировку.
Параметр **std::adopt_lock** сообщает объектам std::lock_guard, что мьютексы уже захвачены, и им нужно лишь принять владение существующей блокировкой, а не пытаться еще раз захватить мьютекс.

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);

class X
{
private:
    some_big_object some_detail;
    std::mutex m;
public:
    X(some_big_object const& sd):some_detail(sd){}
    friend void swap(X& lhs, X& rhs)
    {
        if(&lhs==&rhs)
        return;
        std::lock(lhs.m,rhs.m); // захватываем оба мьютекса
        std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);
        std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock);
        swap(lhs.some_detail,rhs.some_detail);
    }
};
```

Условные переменные

Представьте, что вы в кухне ресторана. Повар ждёт, когда принесут новый заказ. Наконец, официант принимает заказ, и повар начинает готовить. Их коммуникация осуществляется благодаря общей переменной orderStatus, защищаемой мьютексом. Эта переменная может иметь следующие значения:

idle // значение по умолчанию;
newOrder // есть новый заказ для повара;
ready // официант может принести заказ.

Теперь представьте, что было бы, если бы официант через каждые две минуты заходил на кухню и спрашивал повара, не готов ли заказ. Это называется поллингом: поток-официант всё время захватывает мьютекс и проверяет условие в цикле, что приводит к необязательным потерям цикла и батареи.

Было бы лучше, если бы официант ставил будильник на 10 минут (используя std::this_thread::sleep_for()), прежде чем в очередной раз спрашивать у повара о готовности заказа.

Условные переменные

```
std::unique_lock<std::mutex> ul(order);
while ( orderStatus != ready ) {
  // повар всё ещё готовит
  ul.unlock();
  // ставим будильник
  std::this_thread::sleep_for(std::chrono::milliseconds(300));
  // просыпаемся и идём проверять, не готов ли заказ
  ul.lock();
}
// заказ готов
```

Но в случае, если заказ будет готов на 11 минуте, пройдут ещё девять минут, прежде чем будильник зазвонит в следующий раз.

Идеальной была бы такая ситуация: повар нажимает на кнопку вызова официанта, как только заказ будет готов, а официант дремлет в ожидании сигнала от повара. В роли кнопки вызова в C++ выступает примитив синхронизации **condition variable**, позволяющий блокировать один или несколько потоков до поступления сигнала.

Как работает механизм синхронизации?

Синхронизация нужна, когда есть как минимум два потока, один из которых ожидает от другого уведомления о наступлении какого-то события, как то: запись значения, добавление элемента в очередь и др.

Список ожидающих потоков

std::condition_variable по сути представляет собой список потоков (изначально пустой), которые в настоящий момент используют его для ожидания уведомления.

Потоки должны иметь общий доступ к:

1) **переменной, связанной с событием** (например, the orderStatus);

2) **мьютексу**, защищающему такую совместно используемую переменную | std::mutex mutex_name;

3) **объекту condition variable**, который моделирует механизм синхронизации | std::condition_variable condition;.

Механизм синхронизации. Уведомление

Поток может уведомить:

- только **один** случайный поток из числа ожидающих на condition variable, вызвав с помощью **notify_one()** объект на condition variable;
- **все** ожидающие потоки с помощью **notify_all()**.

Перед уведомлением поток должен захватить мьютекс. Уведомление поступает только после освобождения мьютекса.

```
std::lock_guard<std::mutex> lg(mutex_name);
   /* Здесь поток может безопасно выполнить
    * нужные ему действия над общей переменной*/

condition.notify_one();
// или condition.notify_all();
```

Механизм синхронизации. Ожидание

```
std::unique_lock<std::mutex> ul(mutex_name);     // этап 1
condition.wait(ul, <optional_condition_check>); // этап 2
    /* Здесь поток получил уведомление
     * и может безопасно выполнять действия
     * над общей переменной */
ul.unlock();                                     // этап 3
```

1. Захват мьютекса с помощью unique lock

→ std::unique_lock<std::mutex> ul(mutex_name)

Прежде чем переходить к следующему этапу, нужно **захватить** мьютекс. Если сделать это не удаётся, происходит блокировка, которая продолжается до тех пор, пока мьютекс не будет захвачен. Мьютекс захватывается во избежание критических состязаний при пробуждении. Но мьютекс всего один, а уведомление получают несколько потоков. Поэтому пробуждаться они будут последовательно (один за другим), так же последовательно разблокируясь и захватывая мьютекс. Пропуск этого этапа может привести к неопределённому поведению, чего бы очень не хотелось.

Из-за необходимости проводить многократный захват и освобождение мьютекса приходится использовать unique lock вместо lock guard (подробнее об этом далее).

Механизм синхронизации. Ожидание

```
std::unique_lock<std::mutex> ul(mutex_name);     // этап 1
condition.wait(ul, <optional_condition_check>); // этап 2
    /* Здесь поток получил уведомление
     * и может безопасно выполнять действия
     * над общей переменной */
ul.unlock();                                     // этап 3
```

2. Ожидание на condition variable

→ **condition.wait(ul, <condition_check>);**
wait() вызывается у condition variable, передавая **unique_lock** и опционально вызываемый объект (как правило, это **лямбда-функция**), который оценивает, **удовлетворяется ли ожидаемое условие** и возвращает true или false в соответствии с этой оценкой. Проверка условия необязательна, но она нужна для того, чтобы не возникло ложных уведомлений (как если бы повар нажимал на кнопку вызова официанта по ошибке).

Эта функция ожидания выполняет следующие действия:
1) вносит текущий поток в список ожидающих потоков;
2) проверяет, удовлетворяется ли ожидаемое условие, задействуя вызываемый объект;
3) в случае неудовлетворения, освобождает мьютекс и отмечает поток как невыполняемый.

Механизм синхронизации. Ожидание

```
std::unique_lock<std::mutex> ul(mutex_name);     // этап 1
condition.wait(ul, <optional_condition_check>); // этап 2
    /* Здесь поток получил уведомление
     * и может безопасно выполнять действия
     * над общей переменной */
ul.unlock();                                     // этап 3
```

Уведомление — ложное или нет — должно разбудить поток:

4) происходит повторная попытка захвата мьютекса (или блокировка, продолжающаяся до тех пор, пока он не будет захвачен);
5) проводится проверка того, удовлетворяется ли условие (чтобы не было ложных уведомлений);
6) в случае удовлетворения, поток удаляется из списка ожидающих потоков и отмечается как выполняемый. Если условие не удовлетворяется, выполняется действие №3.

Механизм синхронизации. Ожидание

```
std::unique_lock<std::mutex> ul(mutex_name);     // этап 1
condition.wait(ul, <optional_condition_check>); // этап 2
    /* Здесь поток получил уведомление
     * и может безопасно выполнять действия
     * над общей переменной */
ul.unlock();                                     // этап 3
```

3. Освобождение мьютекса после использования

→ **ul.unlock();**

После пробуждения потока выполняется код, в соответствии с приведённой выше инструкцией ожидания, с уже захваченным мьютексом. Теперь, когда стало известно, что ожидаемое событие произошло, выполняются действия над общей переменной.

По завершении работы с общей переменной важно вызвать функцию разблокирования unlock, чтобы и другие потоки могли захватить мьютекс.

Механизм синхронизации. Пример

Действия официанта

Уведомить повара о новом заказе.
Дремать в ожидании сигнала от повара.
Принести заказ.

Действия повара

Ждать, когда принесут новый заказ.
Приготовить блюда из заказа.
Нажать на кнопку вызова официанта.

Повар и официант должны иметь общий доступ к:

переменной: **orderStatus**;
мьютексу, защищающему заказ: **std::mutex order**;
condition variable: **std::condition_variable orderBell**;

Механизм синхронизации. Пример

```cpp
#include <mutex>
#include <condition_variable>
#include <iostream>

class Restaurant {
    enum class Status { idle, newOrder, ready };
    Status orderStatus = Status::idle;
    std::mutex order;
    std::condition_variable orderBell;
public:
    void chef() {
        std::unique_lock<std::mutex> ul(order);
        orderBell.wait(ul, [=]() { return orderStatus == Status::newOrder; });
        //приготовление блюд из заказа
        orderStatus = Status::ready;
        orderBell.notify_one();
    }
    void waiter() {
        {
            std::lock_guard<std::mutex> lg(order);
            orderStatus = Status::newOrder;
            orderBell.notify_one();
        } // lg вне области видимости = order.unlock()
        std::unique_lock<std::mutex> ul(order);
        orderBell.wait(ul, [=]() { return orderStatus == Status::ready; });
        orderStatus = Status::idle;
        ul.unlock();
        //приносят заказ
    }
};
```

Атомарные типы данных

Атомарный объект – это такой объект операции над которым можно считать неделимыми, т.е. такими, которые не могут быть прерваны или результат которых не может быть получен, до окончания операции. Таким образом исключается сама возможность получения некорректных данных в результате наблюдения половины записанных данных, к примеру.

std::atomic_flag — простейший атомарный объект, это булев флаг.
atomic_flag содержит две операции:
test_and_set — устанавливает для хранимого флага значение true и возвращает начальное значение флага.
clear - устанавливает для хранимого флага значение false.

std::memory_order_acquire - операция загрузки с этим упорядочением памяти выполняет операцию захвата (acquire) над задействованной областью памяти: предыдущие записи, сделанные в зависимую от данных область памяти потоком, который выполнил освобождение (release), становятся видимыми в данном потоке.

std::memory_order_release - операция сохранения с этим упорядочением памяти выполняет операцию освобождения (release): предыдущие записи в другие области памяти, становятся видимыми для потоков, которые выполняют операцию поглощения (consume) или захвата (acquire) над той же областью памяти.

Атомарные типы данных

С помощью **std::atomic_flag** можно реализовать **мьютекс-спинлок**. Первоначально флаг сброшен и мьютекс свободен. Чтобы захватить мьютекс, нужно в цикле вызывать функцию **test_and_set()**, пока она не вернет прежнее значение false, означающее, что теперь в этом потоке установлено значение флага true. Для освобождения мьютекса нужно просто сбросить флаг.

```
class spinlock_mutex
{
    std::atomic_flag flag;
public:
    spinlock_mutex(): flag(ATOMIC_FLAG_INIT) {}
    void lock()
    {
        while(flag.test_and_set(std::memory_order_acquire));
    }
    void unlock()
    {
        flag.clear(std::memory_order_release);
    }
};
```

Атомарные типы данных

std::atomic<T>

std::atomic<T>, являясь базовым шаблоном для других атомарных типов, предоставляет нам следующие базовые операции:

- **is_lock_free** – предоставляет нам информации о свободности данного типа от блокировок.
- **store** – кладет новое значение в объект.
- **load** – извлекает значение из объекта.
- **exchange** – заменяет значение в объекте на новое и возвращает старое.