# КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

## Лекция № 24.4 — Семафоры и мьютексы

**Преподаватель: Поденок Леонид Петрович, 505а-5**

**+375 17 293 8039 (505а-5)**
**+375 17 320 7402 (ОИПИ НАНБ)**
**prep@lsi.bas-net.by**

**ftp://student:2ok*uK2@Rwox@lsi.bas-net.by**

**Кафедра ЭВМ, 2021**

# Оглавление

## Алгоритмы синхронизации

Внешние проблемы кооперации процессов связаны с организацией их взаимодействия.

Допустим, что надежная связь процессов организована, и они умеют обмениваться информацией. Какие еще действия необходимо предпринять для организации правильного решения задачи взаимодействующими процессами? Нужно ли как-то изменять их внутреннее поведение? Если да, то как?

**Активность (сущ.) —** последовательное выполнение некоторых действий, направленных на достижение определенной цели (программа, набор операторов и/или вызовов процедур).

Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных.

Активности можно разбить на некоторые **неделимые** или **атомарные** операции.

Например, активность «**`c = a + b`**» можно разбить на следующие атомарные операции:

1) прочитать ячейку памяти **`a`** в регистр **`A`**; **`// load  a, A`**
2) прочитать ячейку памяти **`b`** в регистр **`B`**; **`// load  b, B`**
3) добавить содержимое регистра **`A`** к регистру **`B`**; **`// add   A, B`**
4) записать содержимое регистра **`B`** в ячейку памяти **`c`**. **`// store B, c`**

Активность «**`b += a`**» можно также разбить на следующие атомарные операции:

1) прочитать ячейку памяти **`a`** в регистр **`A`**; **`// load  a, A`**
2) прочитать ячейку памяти **`b`** в регистр **`B`**; **`// load  b, B`**
3) добавить содержимое регистра **`A`** к регистру **`B`**; **`// add   B, A`**
4) записать содержимое регистра **`B`** в ячейку памяти **`b`**. **`// store B, b`**

**`B`** — r-value значение выражения **`b += a`**

Неделимые операции могут иметь некоторые внутренние невидимые действия (**xchg**, **<atomic>exchange()**). Мы называем их **неделимыми** потому, что считаем их одним целым, выполняемыми за один раз, без прерывания действий.

Пусть имеется две активности

```
P: (a b c)
Q: (d e f)
```

где **a**, **b**, **c**, **d**, **e**, **f** — атомарные операции. При последовательном выполнении активностей **P Q** мы получаем следующую последовательность атомарных действий:

```
PQ: a b c d e f
```

При исполнении этих активностей псевдопараллельно в режиме разделения времени они могут расслоиться на неделимые операции с различным их *чередованием*[1].

Возможные варианты чередования:

```
a b c d e f
a b d c e f
a b d e c f
a b d e f c
a d b c e f
......
d e f a b c
```

То есть атомарные операции активностей могут чередоваться всевозможными способами с сохранением своего порядка расположения внутри активностей, в результате чего результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения.

---

1 interleaving

Говорят, что набор активностей (например, программ) **детерминирован**, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные.

В противном случае он **недетерминирован**.

Детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени.

Для недетерминированного набора такое исполнение нежелательно.

## Взаимное исключение (mutual exclusion)

Если не важна ***очередность доступа*** к совместно используемым данным, задачу упорядоченного к ним доступа (устранение ***состояния гонок***[2]) можно решить, если обеспечить каждому процессу ***исключительное***[3] право доступа к этим данным.

Каждый процесс, обращающийся к совместно используемым ресурсам, исключает для всех других процессов возможность общения с этими ресурсами одновременно с ним, если это может привести к недетерминированному поведению набора процессов.

Такой прием называется ***взаимоисключением***[4].

Если же для получения правильных результатов очередность доступа к совместно используемым ресурсам важна, то одними взаимоисключениями уже не обойтись.

## Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие ***критической секции***[5] программы.

**Критическая секция** — это часть программы, исполнение которой может привести к возникновению состояния гонок.

Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом.

---

2 race condition

3 exclusive

Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ.

Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция.

Для решения этой задачи необходимо, чтобы в том случае, когда процесс находится в своей критической секции, другие процессы не могли войти в свои критические секции.

Критический участок кода должен сопровождаться ***прологом***[6] и ***эпилогом***[7].

Во время выполнения пролога процесс должен, в частности, получить разрешение на вход в критический участок, а во время выполнения эпилога — сообщить другим процессам, что он покинул критическую секцию.

Устранение условий гонки возможно при ограничении допустимых вариантов чередований атомарных операций с помощью синхронизации поведения активностей.

Необходимым условием для устранения гонок является организация взаимоисключения на критических участках — внутри соответствующих критических участков не может одновременно находиться более одной активности.

---

4 mutual exclusion

5 critical section

## Требования, предъявляемые к алгоритмам

Организация взаимоисключения для критических участков кода позволяет избежать возникновения состояния гонок, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Есть ***пять условий***, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут проходить их в произвольном порядке:

1) Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (load, store, test) являются атомарными операциями.

2) Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.

3) Если процесс исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в своих соответствующих критических секциях.

Это условие получило название условия взаимоисключения (mutual exclusion).

4) Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки.

Если нет процессов в критических секциях, и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в регулярной секции (regular section), должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress).

---

6 entry section

7 exit section

5) Не должно возникать бесконечного ожидания для входа процесса в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз.

Это условие получило название условия ограниченного ожидания (bound waiting).

Описание соответствующего алгоритма означает описание способа организации пролога и эпилога для критической секции.

Надо отметить, что у алгоритмов, удовлетворяющих вышеприведенным пяти требованиям и построенных средствами обычных языков программирования на пользовательском уровне, существуют серьезные недостатки. Например, если планировщик устроен таким образом, что при готовности высокоприоритетного процесса H низкоприоритетный L вытесняется на все время работы H, то может случиться так, что L будет вытеснен во время выполнения им критической секции, а H не может войти в нее, и будет ожидать выхода L, что никогда не случится — ***тупик***[8].

Чтобы устранить возникновение подобных проблем были разработаны различные механизмы синхронизации более высокого уровня:

- семафоры;
- мониторы;
- сообщения.

---

8 dedlock

## Семафоры

Семафор (semaphore) — объект, ограничивающий количество процессов/потоков, которые могут войти в заданный участок кода. Определение введено Эдсгером Дейкстрой (1930 — 2002).

Семафоры используются для синхронизации и защиты передачи данных через совместно используемую память, а также для синхронизации работы процессов и потоков.

Семафор представляет собой целочисленную переменную-счетчик, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции:

**P** (от датского слова proberen — проверять);

**V** (от verhogen — увеличивать).

Сегодня эти операции называют **down** и **up**.

Операция **P(S)** или **down(s)** выясняет, отличается ли значение семафора **S** от **0**.

Если отличается, она уменьшает это значение на **1** и выполнение продолжается.

Если значение равно **0**, процесс приостанавливается, не завершая операцию **down**.

Все операции — проверка значения семафора, его изменение, и, возможно, приостановка процесса осуществляются как единое и неделимое атомарное действие. Тем самым гарантируется, что с началом семафорной операции никакой другой процесс не может получить доступ к семафору до тех пор, пока операция **down** не будет завершена (**S >= 0**) или заблокирована (**S == 0**).

Операция **V(S)** или **up(S)** увеличивает значение переменной семафора, на **1**. Если с этим семафором связаны один или более приостановленных процессов, способных завершить ранее начатые операции **down**, система выбирает один из них и позволяет ему завершить его операцию **down**.

Таким образом, после применения операции **up** в отношении семафора, с которым были связаны приостановленные процессы, значение семафора так и останется нулевым, но количество приостановленных процессов уменьшится на **1**.

Операция увеличения значения семафора на 1 и активизации одного из процессов является атомарной. При этом, если процесс выполняет операцию **up**, он не может быть заблокирован.

Если при выполнении операции **P** заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным.

Соответствующая целочисленная переменная-счетчик располагается внутри адресного пространства ядра ОС, которая также обеспечивается атомарность операций **down** и **up** (это можно сделать используя привилегированную инструкцию запрета прерываний на время выполнения соответствующих системных вызовов).

Если семафор инициализируется с максимальным значением счетчмка N, он называется N-местным или N-валентным семафором.

Частный случай — одноместный (бинарный) семафор.

## Мьютексы

Мьютекс (mutex, mutual exclusion — «взаимное исключение») — аналог одноместного семафора, служащий для синхронизации одновременно выполняющихся потоков.

**Мьютекс отличается от семафора тем, что только владеющий им поток может его освободить**.

Мьютексы — это один из вариантов семафорных механизмов для организации взаимного исключения. Они реализованы во многих ОС, их основное назначение — организация взаимного исключения для потоков из одного и того же или из разных процессов.

Мьютексы могут находиться в одном из двух состояний — открытом или закрытом (отмечен/неотмечен).

Есть две операции — захватить (get_lock, acquire, take, get) и освободить (release, give).

Когда поток, принадлежащий любому процессу, захватывает мьютекс, мьютекс переводится в закрытое состояние. Если задача освобождает мьютекс, его состояние становится открытым.

Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, который защищён мьютексом. Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток засыпает до тех пор, пока мьютекс не будет освобождён.

Цель использования мьютексов — защита данных от повреждения в результате асинхронных изменений (состояние гонки), однако могут порождаться другие проблемы — такие, как взаимная блокировка (клинч).

**Спинлок** — мьютекс, не освобождающий процессор в ожидании.

Мьютексы всегда используют следующую последовательность:

```
Acquire (захватить)
    Критическая секция  // монопольный доступ к ресурсу
Release (освободить)
```

**Мьютекс** — это как ключ от кабинки туалета. Человек, у которого есть ключ, занимает кабинку. Когда заканчивает, передает ключ следующему в очереди.

Официально[9]: «Мьютексы обычно используются для сериализации доступа к разделу реентерабельного кода, который не может выполняться одновременно более чем одним потоком. Объект мьютекса позволяет только одному потоку войти в контролируемую данным мьютексом секцию, вынуждая другие потоки, которые пытаются получить доступ к ней, дожидаться, пока первый поток не выйдет из этой секции".

**Семафор** — это некоторое количество одинаковых бесплатных ключей от туалета. Например, у нас есть четыре туалета с одинаковыми замками и ключами. Счетчик семафора — счетчик ключей - вначале установлен на 4 (все четыре туалета свободны), затем значение счетчика уменьшается по мере того, как люди входят. Если все туалеты заполнены, т.е. свободных ключей не осталось, счетчик семафоров равен 0. Теперь, когда кто-то выходит из туалета, семафор увеличивается до 1 (один свободный ключ), а ключ передается следующему человеку в очереди (счетчик -> 0).

Официально: «Семафор ограничивает количество одновременных пользователей общего ресурса некоторым максимальным числом. Потоки могут запрашивать доступ к ресурсу (уменьшая семафор) и могут сигнализировать, что они закончили использование ресурса (увеличивая семафор). »

---

9 Руководство разработчика Symbian

## Mutex — <mutex>

Заголовок предоставляет средства, позволяющие реализовать взаимное исключение (мьютекс) с целью предотвращения одновременного выполнения критических участков кода, что позволяет явно избежать гонки данных.

Он содержит несколько типов мьютексов, типов блокировок и специальные функции:

**Типы мьютексов** — это блокируемые типы, используемые для защиты доступа к критическому участку кода — захват мьютекса предотвращает его захват другими потоками (монопольный доступ) до тех пор, пока он не будет разблокирован:

```
mutex
recursive_mutex
timed_mutex
recursive_timed_mutex
```

**Блокировки** — это объекты, которые управляют мьютексом, связывая его доступ со своим собственным временем жизни:

```
lock_guard
unique_lock
```

**Функции для одновременной блокировки нескольких мьютексов:**

try_lock — шаблон функции для попытки заблокировать несколько мьютексов

lock — шаблон функции для блокировки несколько мьютексов

**Функции для предотвращения одновременного выполнения определенной функции:**

call_once — шаблон функции, которая не будет вызывана одновременно несколькими вызовами.

Помимо этого заголовок содержит ряд флагов:

```
once_flag      – тип аргумента флага для call_once (class )
adopt_lock_t   – тип adopt_lock
defer_lock_t   – тип defer_lock
try_to_lock_t – тип try_to_lock
```

## std::mutex — класс мьютекса

Мьютекс — это блокируемый объект, который предназначен для указания, когда критическим участкам кода требуется монопольный доступ с предотвращением одновременного выполнения других потоков с такой же защитой и доступом к тем же ячейкам памяти.

Объекты **mutex** обеспечивают исключительное владение и не поддерживают рекурсивность, т.е. поток не может блокировать мьютекс, которым он уже владеет.

## std::mutex::mutex — конструкторы

(1) по умолчанию

```
constexpr mutex() noexcept;
```

(2) копирования [deleted]

```
mutex (const mutex&) = delete;
```

Объект находится в разблокированном состоянии.

Объекты типа **mutex** нельзя копировать/перемещать (для этого типа удаляются и конструктор копирования, и оператор присваивания).

Само создание объекта **mutex** не является атомарным — доступ к объекту во время его создания может инициировать гонку данных.

## std::mutex::lock — блокировать мьютекс

```
void lock()
```

The calling thread locks the mutex, blocking if necessary:

Вызывающий поток блокирует мьютекс, при необходимости блокируясь до его захвата.

Если мьютекс в настоящее время не заблокирован каким-либо потоком, вызывающий поток блокирует его. С этого момента и до тех пор, пока не будет вызвана функция-член **`unlock()`**, поток будет владеть мьютексом.

Если мьютекс в данное время заблокирован другим потоком, выполнение вызывающего потока приостанавливается, пока мьютекс не будет разблокирован другим потоком, который его удерживает. Другие незаблокированные потоки продолжают свое выполнение.

Если мьютекс в настоящее время заблокирован тем же потоком, который вызывает эту функцию, он создает взаимоблокировку с неопределенным поведением[10].

Все операции блокировки и разблокировки на мьютексе следуют единому общему порядку, при этом все видимые эффекты синхронизируются между операциями блокировки и предыдущими операциями разблокировки для одного и того же объекта.

---

10 recursive_mutex — мьютекс, который позволяет несколько блокировок из одного и того же потока.

Функция блокировки, не являющаяся членом, позволяет блокировать более одного объекта мьютекса одновременно, избегая потенциальных взаимоблокировок, которые могут возникнуть, когда несколько потоков блокируют/разблокируют отдельные объекты мьютекса в разном порядке.

**Важно!** Порядок, в котором запланировано возвращение различных одновременных блокировок, не определен и не обязательно связан с порядком, в котором они были заблокированы (все зависит от системы и реализации библиотеки).

## std::mutex::unlock — освободить мьютекс

```
void unlock();
```

Разблокирует мьютекс, освобождая право владения им.

Если другие потоки в настоящее время заблокированы при попытке заблокировать этот же мьютекс, один из них получает право вледения им и продолжает свое выполнение.

Все операции блокировки и разблокировки на мьютексе следуют единому общему порядку, при этом все видимые эффекты синхронизируются между операциями блокировки и предыдущими операциями разблокировки для одного и того же объекта.

Если мьютекс в настоящее время не заблокирован вызывающим потоком, это вызывает неопределенное поведение.

**Пример `mutex::lock/unlock`**

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex

std::mutex mtx;           // mutex for critical section

void print_thread_id(int id) { // печатаем ID потока
  // критическая секция (исключ. доступ к std::cout указывается захватом mtx):
  mtx.lock();
  std::cout << "thread #" << id << '\n';
  mtx.unlock();
}

int main () {
  std::thread threads[10];
  for (int i = 0; i < 10; ++i) threads[i] = std::thread(print_thread_id, i+1);
  for (auto& th : threads) th.join();
  return 0;
}
```

**Вывод** (порядок строк может быть разным, но они никогда не пересекаются)

```
thread #1
thread #2
...
thread #10
```

## std::mutex::try_lock — заблокировать, если не заблокирован

```
bool try_lock();
```

Попытка захватить мьютекс без блокировки потока.

Если мьютекс в настоящее время не заблокирован каким-либо потоком, вызывающий поток блокирует его. С этого момента и до тех пор, пока не будет вызвана функция-член **`unlock()`**, поток будет владеть мьютексом.

Если мьютекс в настоящее время заблокирован другим потоком, функция завершается ошибкой и возвращает **`false`** без ожидания, а вызывающий поток продолжает свое выполнение.

Если мьютекс в настоящее время заблокирован тем же потоком, что вызывает эту функцию, он создает взаимоблокировку (с неопределенным поведением).

Эта функция может ошибочно завершиться ошибкой даже в том случае, если никакой другой поток блокирует мьютекс, поэтому имеет смысл делать повторные вызовы — в какой-то момент они будут успешными.

**Возвращаемое значение**

**`true`** — функции удалось захватить мьютекс для потока.

**`false`** — увы, не удалось

**Пример — `mutex::try_lock`**

```
#include <iostream>      // std::cout
#include <thread>        // std::thread
#include <mutex>         // std::mutex

volatile int counter(0); // non-atomic counter
std::mutex mtx;          // locks access to counter

void attempt_10k_increases () {
  for (int i = 0; i < 10000; ++i) {
    if (mtx.try_lock()) {
      ++counter;          // инкремент счетчика если тот не заблокирован
      mtx.unlock();
} } }

int main () {
  std::thread threads[10];
  for (int i=0; i<10; ++i)                          // порождаем 10 потоков
    threads[i] = std::thread(attempt_10k_increases);
  for (auto& th : threads) th.join();
  std::cout << counter << " успешных инкрементов счетчика.\n";
  return 0;
}
```

**Вывод**

```
81462 успешных инкрементов счетчика.
```

**Пример**

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex

std::mutex mtx;           // mutex for critical section

void print_block (int n, char c) {
  mtx.lock();
  for (int i = 0; i < n; ++i) { std::cout << c; } // просто выводим символы
  std::cout << '\n';
  mtx.unlock();
}

int main () {
  std::thread th1(print_block, 50, '*');
  std::thread th2(print_block, 50, '$');
  th1.join();
  th2.join();
}
```

**Вывод**

```
**************************************************
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Без мьютекса
*********$$***$$$$$**$**$*******************$$$$**$**$$**$$***$$**$$**$$*$**
$$$$$$$$$$$$$$$$$$$$
```

## std::recursive_mutex — класс рекурсивного мьютекса

Рекурсивный мьютекс является блокируемым объектом, как и мьютекс, но позволяет одному и тому же потоку получить несколько уровней владения таким мьютексом.

Это позволяет заблокировать или попробовать заблокировать (**lock** или **try_lock**) объект мьютекса в потоке, который уже его блокирует, получая в результате новый уровень владения над этим мьютексом — мьютекса фактически останется заблокированным потоком-владельцем, пока не будет вызван **unlock** столько раз, каков уровень владения у этого мьютекса.

## std::recursive_mutex::recursive_mutex — конструкторы

(1) по умолчанию

```
constexpr recursive_mutex( ) noexcept;
```

(2) копирования [deleted]

```
recursive_mutex (const recursive_mutex&) = delete;
```

Объект находится в разблокированном состоянии.

Объекты типа **recursive_mutex** нельзя копировать/перемещать (для этого типа удаляются и конструктор копирования, и оператор присваивания).

Само создание объекта **recursive_mutex** не является атомарным — доступ к объекту во время его создания может инициировать гонку данных.

## std::recursive_mutex::lock — захват рекурсивного мьютекса

```
void lock();
```

Вызывающий поток блокирует мьютекс, при необходимости блокируясь до его захвата.

Если мьютекс в настоящее время не заблокирован каким-либо потоком, вызывающий поток блокирует его. С этого момента и до тех пор, пока не будет вызвана функция-член **`unlock()`**, поток будет владеть мьютексом.

Если мьютекс в данное время заблокирован другим потоком, выполнение вызывающего потока приостанавливается, пока мьютекс не будет разблокирован другим потоком, который его удерживает. Другие незаблокированные потоки продолжают свое выполнение.

Если мьютекс в настоящее время заблокирован тем же потоком, который вызывает эту функцию, поток получает новый уровень владения мьютексом. Для полной разблокировки рекурсивно удерживаемого мьютекса потребуется дополнительный вызов функции-члена **`unlock()`**.

**Важно!** Порядок, в котором запланировано возвращение различных одновременных блокировок, не определен и не обязательно связан с порядком, в котором они были заблокированы (все зависит от системы и реализации библиотеки).

## std::recursive_mutex::unlock — разблокирует мьютекс

```
void unlock();
```

Разблокирует мьютекс, освобождая один уровень права владения им.

Если у вызывающего потока был единственный уровень владения рекурсивным мьютексом, он полностью разблокируется и, если другие потоки в настоящее время заблокированы в попытке завладеть этим же рекурсивным мьютексом, один из них получает право владения им и продолжает свое выполнение.

Если мьютекс в настоящее время не заблокирован вызывающим потоком, это вызывает неопределенное поведение.

## std::recursive_mutex::try_lock — заблокировать, если не заблокирован другими

```
bool try_lock() noexcept;
```

Пытается захватить **recursive_mutex** без блокировки:

- если **recursive_mutex** в настоящее время не заблокирован каким-либо потоком, вызывающий поток блокирует его. С этого момента и до тех пор, пока не будет вызвана функция-член **unlock()**, поток будет владеть рекурсивным мьютексом.

- если **recursive_mutex** в настоящее время заблокирован другим потоком, функция завершается ошибкой и возвращает **false** без ожидания, а вызывающий поток продолжает выполнение.

- если **recursive_mutex** в настоящее время заблокирован тем же потоком, который вызывает эту функцию, поток получает новый уровень владения рекурсивным мьютексом. Для полной его разблокировки потребуется дополнительные вызовы **unlock()**.

Эта функция может ошибочно завершиться ошибкой даже в том случае, если никакой другой поток блокирует мьютекс, поэтому имеет смысл делать повторные вызовы — в какой-то момент они будут успешными.

**Возвращаемое значение**

**true** — функции удалось захватить рекурсивный мьютекс.

**false** — увы, не удалось

## std::timed_mutex — временный мьютекс

```
class timed_mutex;
```

Класс временного мьютекса

Временный мьютекс — это объект с временной блокировкой, который предназначен для сигнализации, когда критическим участкам кода требуется монопольный доступ, как и в случае обычного мьютекса, но дополнительно поддерживаются запросы на временную блокировку.

Соответственно, **timed_mutex** имеет два дополнительных члена:

**try_lock_for()** — попытка захвата в течение некоторого времени

**try_lock_until()** — попытка захвата до некоторого момента времени

**lock()**, **try_lock()**, **unlock()** — ведут себя аналогично функциям класса **mutex**

### Конструкторы

(1) по умолчанию

```
constexpr timed_mutex() noexcept;
```

(2) копирования [deleted]

```
timed_mutex (const timed_mutex&) = delete;
```

Объект после создания находится в разблокированном состоянии.

Объекты **timed_mutex** нельзя копировать/перемещать (для этого типа удаляются и конструктор копирования, и оператор присваивания).

### std::timed_mutex::try_lock_for — захватить на некоторое время

```
template <class Rep, class Period>
  bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

Пытается заблокировать **timed_mutex**, не более чем на **rel_time**:

- если **timed_mutex** в настоящее время не заблокирован каким-либо потоком, вызывающий поток блокирует его (с этого момента и до тех пор, пока не будет вызвана **unlock( )**, поток владеет **timed_mutex**).

- если **timed_mutex** в настоящее время заблокирован другим потоком, выполнение вызывающего потока приостаналивается до тех пор, пока мьютекс не будет разблокирован или по истечении **rel_time**, в зависимости от того, что произойдет раньше (тем временем другие незаблокированные потоки продолжают свое выполнение).

- если **timed_mutex** в настоящее время заблокирован тем же потоком, который вызывает эту функцию, он создает взаимоблокировку (с неопределенным поведением)[11].

**rel_time** — максимальный промежуток времени, в течение которого поток будет блокироваться, ожидая захвата мьютекса.

**duration** — это объект, который представляет конкретное относительное время.

**Возвращаемое значение**

**true** — если функции удается захватить **timed_mutex** для потока.

**false** — в противном случае

11 recursive_timed_mutex допускает множественные блокировки из одного и того же потока

**Пример `timed_mutex::try_lock_for`**

```
#include <iostream>       // std::cout
#include <chrono>         // std::chrono::milliseconds
#include <thread>         // std::thread
#include <mutex>          // std::timed_mutex

std::timed_mutex mtx;

void fireworks() {
  // ждем получения мьютекса: каждый поток печатает "-" каждые 200 мс:
  while (!mtx.try_lock_for(std::chrono::milliseconds(200))) {
    std::cout << "-";
  }
  // мьютекс получен! - ждем 1с, после чего печатаем "*"
  std::this_thread::sleep_for(std::chrono::milliseconds(1000));
  std::cout << "*\n";
  mtx.unlock();
}

int main () {
  std::thread threads[10]; // порождаем 10 потоков и заносим в массив
  for (int i = 0; i < 10; ++i)
    threads[i] = std::thread(fireworks);
  for (auto& th : threads) th.join(); // дожидаемся завершения всех потоков

  return 0;
}
```

**Возможный вывод (примерно через 10 секунд длина строк может незначительно меняться):**

```
_____________________________________*
_________________________________________*
_____________________________________*
_____________________________*
_________________________*
____________________*
_______________*
__________*
_____*
*
```

## std::timed_mutex::try_lock_until — ожидать захвата до указанного момента времени

```
template <class Clock, class Duration>
  bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Попытка заблокировать **timed_mutex**, ожидая максимум до **abs_time**:

- если **timed_mutex** в настоящее время не заблокирован каким-либо потоком, вызывающий поток блокирует его (с этого момента и до тех пор, пока не будет вызвана **unlock( )**, поток владеет timed_mutex).

- если **timed_mutex** в данное время заблокирован другим потоком, выполнение вызывающего потока блокируется до тех пор, пока не будет разблокировано или до **abs_time**, в зависимости от того, что произойдет раньше (другие же незаблокированные потоки продолжают выполнение).

- если **timed_mutex** в настоящее время заблокирован тем же потоком, который вызвал эту функцию, он создает взаимоблокировку (с неопределенным поведением)[12].

**abs_time** — момент времени, когда поток разблокируется, отказавшись от попытки захвата.
**time_point** — объект, представляющий конкретное абсолютное время.

**Возвращаемое значение**
**true** — если функции удается захватить **timed_mutex** для потока.
**false** — в противном случае

12 recursive_timed_mutex допускает множественные блокировки из одного и того же потока.

**Пример timed_mutex::try_lock_until**

```
#include <iostream>       // std::cout
#include <chrono>         // std::chrono::system_clock
#include <thread>         // std::thread
#include <mutex>          // std::timed_mutex
#include <ctime>          // std::time_t, std::tm, std::localtime, std::mktime

std::timed_mutex cinderella;              // золушка

// получим момент времени, следующий за следующей полуночью
std::chrono::time_point<std::chrono::system_clock> midnight() {
  using std::chrono::system_clock;
  std::time_t tt      = system_clock::to_time_t(system_clock::now());
  struct std::tm *ptm = std::localtime(&tt); // дата и время -> в компоненты
  ++ptm->tm_mday;   // следующий день
  ptm->tm_hour = 0; ptm->tm_min  = 0; ptm->tm_sec  = 0; // 0 часов 0 минут
  return system_clock::from_time_t(mktime(ptm));
}

void carriage() { // карета
  if (cinderella.try_lock_until(midnight())) {
    std::cout << "возвращаемся домой в карете\n";
    cinderella.unlock();
  }  else {
    std::cout << "карета превратилась в тыкву\n";
} }
```

```
void ball() {
  cinderella.lock();
  std::cout << "танцуют все ...\n";
  cinderella.unlock();
}

int main () {

  std::thread th1(ball);
  std::thread th2(carriage);
  th1.join();
  th2.join();
}
```

**Вывод**

```
танцуют все ...
возвращаемся домой в карете
```

## std::recursive_timed_mutex — рекурсивный временный мьютекс

```
class recursive_timed_mutex;
```

Класс рекурсивного временного мьютекса

Рекурсивный временный мьютекс сочетает в себе функции **recursive_mutex** и функции **timed_mutex** в одном классе — он поддерживает как получение нескольких уровней блокировки одним потоком, так и запросы временной блокировки **try-lock**.

Функции **lock()**, **try_lock()**, **unlock()** — ведут себя аналогично функциям **recursive_mutex.**

```
template <class Rep, class Period>
  bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);

template <class Clock, class Duration>
  bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

**rel_time** — максимальный промежуток времени, в течение которого поток будет блокироваться, ожидая захвата мьютекса.

**duration** — это объект, который представляет конкретное относительное время.

**abs_time** — момент времени, когда поток разблокируется, отказавшись от попытки захвата.

**time_point** — объект, представляющий конкретное абсолютное время.

**Возвращаемое значение**

**true** — если функции удается захватить **timed_mutex** для потока.

**false** — в противном случае

# Блокировки