Многопоточность ч.1

Практическое объектно-ориентированное программирование

06.11.2024

План

- Потоки и синхронизация
- Проблемы проектирования
- События и оповещения
- Различные блокировки

Потоки

- Что такое поток исполнения (thread of execution)?
- По определению стандарта:

A thread of execution (also known as a thread) is a single flow of control within a program [intro.multithread.general]

- При этом:

The execution of the entire program consists of an execution of all of its threads (C++17, 4.7)

- Логическая многопоточность (concurrency) внутри программы не имеет отношения к аппаратной параллельности (parallelism).

Создание потока

- Поток создаётся начиная с C++11 в конструкторе класса std::thread.

```
int main() {
  std::thread t([]{
    std::cout < < “Hello, world!“ < < std::endl;
  });
  // тут что угодно пока поток исполняется
  t.join();
}
```

- Поток стартует с переданной в конструктор функции или функтора.
- Обязательно сделать join() или detach().

Поток это всегда поток

- Создаваемый поток в принципе ничем не отличается от родителя.

```
std::cout << "Main: " << std::this_thread::get_id() << std::endl;
std::thread t([]{
  std::cout << "Spawned: " << std::this_thread::get_id() << std::endl;
});
```

- Теперь очевидно, что в функции два потока: на экране два разных id.
- На уровне языка нас мало волнует кто и как управляет потоками.
- Кстати, видите ли вы некую опасность в таком коде?

Области памяти

• Любая программа работает с объектами (objects). Объект это что угодно, требующее места (storage) для хранения значения.
int x = 42; // x это объект, 42 нет
foo(&x); // теперь x требует сохранения в память
• Область памяти (memory location) определена в стандарте так:
A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having nonzero width (C++17, 4.4)
• Основная часть этой лекции будет посвящена тому как передача управления в программе сочетается с состоянием областей памяти в ней же.

Гонка за область памяти

• Следующая программа весьма иллюстративна.

```
int x; // область памяти
void race() {
  for(int i = 0; i < 100; ++i)  x += 1; use(x);
  for(int i = 0; i < 100; ++i)  x -= 1; use(x);
}
```

• Для одного потока всё хорошо.

```
std::thread trace; t.join();
assert(x == 0);
```

• Что здесь произойдёт если в эту функцию зайдут два потока?

Data race

• Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location. [intro.races]
• The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior. [intro.races]

Правила гонки

• Гонка происходит если сколько угодно потоков читают область памяти и хотя бы один одновременно пишет в неё же.

```
unsigned x = 0, i = 0, j = 0; // области памяти
void readerf() { while (i++ < 'g') x += 0x1; }
void writerf() { while (j++ < 'g') x += 0x10000; }
std::thread t1{readerf}, t2{writerf};
t1.join(); t2.join();
```

• Тут всё равно не определено что будет на экране, несмотря на то, что запись и чтение в разных потоках трогают разные байты объекта.

Убираем гонку

• Область памяти это скалярный объект

```
char x[2] = {0, 0}, i = 0, j = 0; // области памяти
void readerf() { while (i++ < 'g') x[0] += 0x1; }
void writerf() { while (j++ < 'g') x[1] += 0x1; }
thread t1{readerf}, t2{writerf};
t1.join(); t2.join();
```

• Теперь мы работаем с разными скалярными объектами и data race нет.

Сомнительный код

- Вернемся к примеру выше

```
std::thread t([]
  std::cout < < “Spawned: “ « std::this_thread::get_id() « std::endl;
);
std::cout < < “Main: “ < < std::this_thread::get_id() < < std::endl;
t.join();
```

- Основное сомнение нет ли тут UB?
- Такое чувство, что мы пишем в одну область памяти.

Объект никогда не область памяти

• Мы можем подозревать здесь гонку.

```
cout < < “Parent id: “ < < std::this_thread::get_id() < < endl;
thread t([]{
  cout < < “Spawned id: “
< < std::this_thread::get_id() < < endl;
});
```

• Сложный объект с поведением (напр. std::cout) это никогда не область памяти. Тот же std::cout по стандарту (C++17 30.4.2.5) безопасен.

• Несмотря на то, что с объектом потока ничего не случится, порядок вывода на экран символов в этой программе не определён.

Простая синхронизация

• Для синхронизации доступов между потоками управления, служат мьютексы.

```
int x; // область памяти
std::mutex mforx; // мьютекс для синхронизации x
void race() {
for(int i = 0; i < 100; ++i) {
  mforx.lock(); x += 1; mforx.unlock();
}
// и то же самое для второго цикла
}
```

• Мьютексы вводят отношение happens-before между доступами.

Интерфейс мьютекса

метод	сюрприз
`lock()`	Попытка вызвать повторно в том же потоке это UB. Также может кинуть исключение `std::system_error`
`try_lock()`	То же что и для lock()
`unlock()`	Попытка разблокировать не захваченный мьютекс это тоже UB

- std::mutex поддерживает “очень простой“ интерфейс с сюрпризами
- Какой главный сюрприз в этом интерфейсе?

RAII синхронизация

• Разумеется, в таких условиях делать lock() и unlock() руками никто никогда не будет.

```
for(int i = 0; i < 100; ++i) {
  lock_guard<mutex> lkmforx;
  x += 1;
}
```

• Защёлка std::lock_guard<T> это RAII обёртка на любым классом, поддерживающим интерфейс из методов lock() и unlock().

• По логике это scoped lock (сравнить со scoped ptr) без копирования и перемещения.

Интермедия: безопасность исключений

- Когда мы говорим, что некий объект безопасен относительно исключений, что мы имеем в виду?

Напомним основное

• Код, в котором при исключении могут утечь ресурсы, оказаться в несогласованном состоянии объекты и прочее, называется небезопасным относительно исключений.
• Базовая гарантия: исключение при выполнении операции может изменить состояние программы, но не вызывает утечек и оставляет все объекты в согласованном (но не обязательно предсказуемом) состоянии.
• Строгая гарантия: при исключении гарантируется неизменность состояния программы относительно задействованных в операции объектов (commit/rollback).
• Гарантия бессбойности: функция не генерирует исключений (noexcept).

Пример Каргилла

```
template <typename T> class MyVector {
  T *buffer_ = nullptr;
  size_t capacity_, size_ = 0;
public:
  MyVector(const MyVector &rhs) : buffer_(new T[rhs.size_]),
      capacity_(rhs.size_),
      size_(rhs.size_) {
    std::copy(rhs.arr_, rhs.arr_ + rhs.size_, arr_);
}
```

- Например где бы вы вставили catch(...)?

Обсуждение: безопасность потоков

- Когда мы говорим, что некий объект безопасен относительно многопоточности, что мы имеем в виду?

Нулевой уровень безопасности

- У нас есть некий нулевой уровень, это нейтральные объекты, например int.
- Если int защищён синхронизацией, он безопасен, если нет, то нет.
- Можем ли мы спуститься ниже нейтрального уровня?

Хуже нейтральности

- Можем ли мы спуститься ниже нейтрального уровня?
- Да и очень просто. Например: указатель на int.

{ lock_guard<mutex> lk{mforx}; *px += 1; }

- Понимаете ли вы в чём проблема?

Лучше нейтральности

- Безопасным относительно многопоточного окружения является объект, никакие операции с которым в этом окружении не приводят к data race
- Шире говоря: не приводят к неконсистентному состоянию
- В этом смысле безопасность относительно потоков похожа на строгую безопасность относительно исключений
- Не ждёт ли нас на этом пути больше сюрпризов?

Потокобезопасный буффер

• Вспомним буфер, спроектированный для безопасности исключений.

```
template <typename T> struct MyBuffer: public BufImpl {
void pop() {
  size_ -= 1;
  destroy(buffer_ + size_);
}
  T top() const { return buffer_[size_ - 1 ]; }
  bool empty() const { return (size_ == 0); }
// прочие методы
};
```

• Как сделать его безопасным для использовани я с несколькими потоками?

Потокобезопасный буффер

- Похоже все эти методы нужно защитить мьютексом.

```
mutex bufmut_;
void pop() {
  lock_guard<mutex>{bufmut_};
  size_ -= 1;
  destroy(buffer_ + size_);
}
T top() const; // аналогично
bool empty() const; // аналогично
```

- Сначала простой вопрос. Всё ли хорошо на этом слайде?

Потокобезопасный буффер

- Нельзя забывать про имена для объектов синхронизации.

```
mutex bufmut_;
void pop() {
  lock_guard<mutex> lk{bufmut_};
  size_ -= 1;
  destroy(buffer_ + size_);
}
```

- А теперь всё ли в порядке?

API races

• Представим себе следующий кусок кода, исполняемый сразу двумя потоками на одном глобальном объекте s.

```
if (!s.empty()) {
  auto elem = s.top();
  s.pop();
  use(elem);
}
```

• В буфере остался один объект, оба потока прошли проверку на empty().

API races

• Представим себе следующий кусок кода, исполняемый сразу двумя потоками на одном глобальном объекте s.

```
if (!s.empty()) {
  auto elem = s.top();
  s.pop();
  use(elem);
}
```

• Первый поток считал и снял элемент.

• Далее второй поток пытается считать не существующий более элемент.

API races

• Представим себе следующий кусок кода, исполняемый сразу двумя потоками на одном глобальном объекте s.

```
if (!s.empty()) {
  auto elem = s.top();
  s.pop();
  use(elem);
}
```

• Это конфликт между pop() и empty().

• Строго говоря это не data race и не UB. Но явно что-то идёт не так. Такие случаи называются API races.

API races

• Представим себе следующий кусок кода, исполняемый сразу двумя потоками на одном глобальном объекте s.

```
if (!s.empty()) {
  auto elem = s.top();
  s.pop();
  use(elem);
}
```

• В буфере осталось два объекта, оба потока прошли проверку на empty().

API races

• Представим себе следующий кусок кода, исполняемый сразу двумя потоками на одном глобальном объекте s.

```
if (!s.empty()) {
  auto elem = s.top();
  s.pop();
  use(elem);
}
```

• Оба потока считали первый объект из буфера.

API races

• Представим себе следующий кусок кода, исполняемый сразу двумя потоками на одном глобальном объекте s.

```
if (!s.empty()) {
  auto elem = s.top();
  s.pop();
  use(elem);
}
```

• Оба потока удалили один первый объект, второй оставшийся. В итоге буфер пуст но у обоих потоков копия одного объекта, а второй навсегда потерян.

• Это гонка между pop() и top().

Обсуждение

• Кажется первая попытка сделать буфер потокобезопасным провалилась.

```
mutex bufmut_;
void pop() {
  lock_guard<mutex> lk{bufmut_};
  size_ -= 1;
  destroy(buffer_ + size_);
}
```

• Что делать, что пошло не так?

Попытки исправления ситуации

- Заметим, что никаких API races не будет если объединить pop и top

```
mutex bufmut_;
 bool try_pop(T &loc) {
 lock_guard<mutex> lk{bufmut_};
 if (empty()) return false; // что произойдёт здесь?
 loc = buffer_[size_ - 1]; // сколько времени это займёт?
 size_ -= 1;
 destroy(buffer_ + size_);
 return true;
}
```

- Хороша ли эта идея?

Попытки исправления ситуации

- Правда что тогда делать с безопасностью исключений?

```
mutex bufmut_;
 bool try_pop(T &loc) {
 lock_guard<mutex> lk{bufmut_};
 if (empty()) return false;
 loc = buffer_[size_ - 1];
 size_ -= 1;
 destroy(buffer_ + size_);
 return true;
}
```

- Слишком широкие критические секции это плохо. Есть ли ещё идеи?

Попытки исправления ситуации

- Как вам например такой выход?

mutex bufmut_;
shared_ptr<T> try_pop();

- В целом выходов много и все они спорные.
- Важно отметить: исторически все контейнеры стандартной библиотеки были спроектированы так, как будто никаких потоков нет. Это частично избавляет от API races но гарантирует в лучшем случае слабую нейтральность.
- Интересно то, что API races это не самое плохое.

Задача: покритикуйте swap

• Следующий метод, который часто хочется реализовать безопасным это обмен значениями.

```
template <typename T>
void MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
  if (this == &rhs) return;
  std::lock_guard<mutex> lk1{bufmut_};
  std::lock_guard<mutex> lk2{rhs.bufmut_};
  std::swap(buffer_, rhs.buffer_);
  std::swap(size_, rhs.size_);
  std::swap(capacity_, rhs.capacity_);
}
```

• Что тут может пойти не так?

Deadlocks

• Сценарий ошибки: первый поток зовёт x.swap(y) а второй y.swap(x) для глобальных x и y.

```
if (this == &rhs) return;
std::lock_guard<mutex> lk1{bufmut_};
std::lock_guard<mutex> lk2{rhs.bufmut_};
std::swap(buffer_, rhs.buffer_);
std::swap(size_, rhs.size_);
std::swap(capacity_, rhs.capacity_);
```

• Первый поток взял x.bufmut_.

• Второй поток взял y.bufmut_.

Deadlocks

- Сценарий ошибки: первый поток зовёт x.swap(y) а второй y.swap(x) для глобальных x и y

```
if (this == &rhs) return;
std::lock_guard<mutex> lk1{bufmut_};
std::lock_guard<mutex> lk2{rhs.bufmut_};
std::swap(buffer_, rhs.buffer_);
std::swap(size_, rhs.size_);
std::swap(capacity_, rhs.capacity_);
```

- Первый поток ждёт y.bufmut_
- Второй поток ждёт x.bufmut_
- Они будут ждать вечно. Это мёртвая блокировка, deadlock

Обедающие философы

- Задача была сформулирована ещё Дейкстрой и Хоаром.
- За столом едят несколько философов.
- Каждый может есть или думать.
- Есть каждый может только двумя вилками.
- Задача: написать функцию:

void take(mutex &left, mutex &right) { // ??? }

- Которая корректно берёт и левую и правую вилки.

Обедающие философы

- Суть простейшего решения на C++.

```
void take(mutex &left, mutex &right) {
  for (;;) {
    left.lock();
    if (right.try_lock())
      break;
    left.unlock();
    std::this_thread::yield();
  }
}
```

- Есть стандартный алгоритм.

Обедающие философы и C++

• Стандартная функция std::lock позволяет безопасно захватить произвольное количество мьютексов.

```
template <class Lockable1, class Lockable2, class... LockableN>
void lock (Lockable1& lock1, Lockable2& lock2, LockableN&... lockn);
```

• Для решения проблемы swap её можно использовать напрямую.

```
template <typename T>
void MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
  if (this == &rhs) return;
  std::lock(bufmut_, rhs.bufmut_);
```

• Но тогда придётся делать ручной unlock в конце.

Решение проблемы swap

• Используем adopt_lock чтобы захватить взятый mutex в оболочку lock_guard

```
template <typename T>
void MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
  if (this == &rhs) return;
  std::lock(bufmut_, rhs.bufmut_);
  std::lock_guard<mutex> lk1bufmut_, std::adopt_lock;
  std::lock_guard<mutex> lk2rhs.bufmut_, std::adopt_lock;
  swap(arr_, rhs.arr_);
  swap(size_, rhs.size_);
  swap(used_, rhs.used_);
}
```

• Теперь всё хорошо. Но смотрится несколько странно.

Решение проблемы swap в C++17

• Начиная с 2017 года можно использовать RAII оболочку std::scoped_lock

```
template <typename T>
void MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
  if (this == &rhs) return;
  std::scoped_lock slbufmut_, rhs.bufmut_;
  std::swap(arr_, rhs.arr_);
  std::swap(size_, rhs.size_);
  std::swap(used_, rhs.used_);
}
```

• Она внутри использует алгоритм std::lock и хранит несколько взятых защелок.

Обсуждение

- Проблемы проектирования выглядят гораздо более серьёзными, чем в случае с исключениями.
- Это и понятно: исключения рисуют произвольное количество выходных дуг, а потоки ещё и произвольное количество входных.
- На этом месте следует задать один важный вопрос: а хотим ли мы вообще безопасные относительно потоков контейнеры?
- И если да, то в каком виде?
- Ответ на него придётся несколько отложить.

Одноразовые события

• Пусть некая функция содержит в себе и использование ресурса и его одноразовую инициализацию при необходимости.

```
{
 lock_guard<mutex> lk{resmut};
 if (!resptr)
   resptr = new Resource(); // создание требует синхронизации
}
resptr->use(); // use это const функция, синхронизация не требуется
```

• Похоже этот подход слишком консервативен: все вызовы этой функции, которым уже не надо ничего создавать, будут платить за синхронизацию.

Выход из ситуации: DCL

- Паттерн double-checked lock (DCL), увы, нередко используется.

```
if (!resptr) {
  lock_guard<mutex> lkresmut;
  if (!resptr)
    resptr = new Resource(); // создание требует синхронизации
}
resptr->use(); // use это const функция, синхронизация не требуется
```

- Стало ли существенно лучше?

DCL is totally broken

- Паттерн double-checked lock (DCL), увы, нередко используется.

```
if (!resptr) { // эта проверка не синхронизирована
  lock_guard<mutex> lk{resmut};
  if (!resptr)
    resptr = new Resource(); // создание требует синхронизации
}
resptr->use(); // use это const функция, синхронизация не требуется
```

- В реальности стало хуже: появился тривиальный data race между записью и чтением, это UB.
- Ещё идеи?

Правильный выход: std::once_flag

• Специальный примитив, вместе с std::call_once защищающий однократное создание.

```
resource *resptr;
std::once_flag resflag;
void init_resource()  resptr = new resource();
```

• И где-то далее в коде

```
std::call_once(resflag, init_resource);
resptr->use();
```

• Расходы на это решение существенно меньше, чем на постоянную сериализацию вокруг мьютекса.

Упражнение

- Вам приносят следующий код. Что может пойти не так?

```
volatile int resready = 0; resource *resptr;
void foo() { // will be called by thread 1
  resptr = new resource();
  resready = 1;
}
void bar() {
  while (!resready)  std::this_thread::yield();
  resptr->use();
}
```

Решение

- Разумеется это тривиальный data race и таким образом UB.

```
volatile int resready = 0; resource *resptr;
void foo() { // will be called by thread 1
  resptr = new resource();
  resready = 1;
}
void bar() {
  while (!resready)  std::this_thread::yield();
  resptr->use();
}
```

Обсуждение

- По сути, инциализация ресурса в потоке A это событие, о котором он пытается сообщить потоку B.
- Делать это через volatile плохо, но сама идея хороша.
- Как бы вы решили проблему сообщения о событии?
- Хватит ли вам для этого уже рассмотренных механизмов?

Условные переменные: первая попытка

• Допустим, в языке существовал бы механизм условных переменных.

```
resource *resptr = nullptr;
std::mutex resmut;
std::condition_variable data_cond;
// thread 1                                  // thread 2
lock_guard<mutex> lkresmut;                     data_cond.wait();
resptr = new resource();                        resptr->use();
data_cond.notify_one();
```

• Здесь есть существенная проблема (кроме того, что это псевдокод).

Условные переменные: первая попытка

• Допустим, в языке существовал бы механизм условных переменных.

```
resource *resptr = nullptr;
std::mutex resmut;
std::condition_variable data_cond;
// thread 1                                  // thread 2
lock_guard<mutex> lkresmut;                     data_cond.wait();
resptr = new resource();                        resptr->use();
data_cond.notify_one();
```

• Между этими строчками может пройти больше времени, чем кажется.

• Хотелось бы дождавшись ресурса, сразу взять мьютекс.

Условные переменные: вторая попытка

```
resource *resptr = nullptr;
std::mutex resmut;
std::condition_variable data_cond;
// thread 1                                  // thread 2
lock_guard<mutex> lkresmut;                     data_cond.wait();
resptr = new resource();                        resptr->use();
data_cond.notify_one();                         resmut.unlock();
```

- Это лучше чем ничего, но теперь хотелось бы RAII.
- Тут не сделать lock_guard, так как он не умеет снимать блокировку и ждать.

Уникальные блокировки

• Класс std::unique_lock предоставляет уникальное владение блокировкой. {

```
std::unique_lock<mutex> ul{resmut}; // locked by ctor
res->use();
ul.unlock();
// something on unlocked
ul.lock();
res->use();
```

} // unlocked by dtor

• В принципе его можно даже использовать вместо lock_guard

• Но это расточительно. Признак взятия блокировки – лишнее поле в классе.

Обсуждение

- Возможность вручную вызвать lock означает возможность вызвать его вручную повторно.
- Разумеется unique_lock обложен всем чем можно
  - метод owns_lock проверяет взята ли блокировка
  - и даже если его забыть вызвать, его проверит lock перед mutex::lock и бросит исключение (правда почему-то std::system_error но и на этом...)
- Мы расходуем место для одного лишнего признака и получаем крайне удобный интерфейс.

Условные переменные

```
resource *resptr = nullptr;
std::mutex resmut;
std::condition_variable data_cond;
// thread 1                                  // thread 2
lock_guard<mutex> lk{resmut};                std::unique_lock<mutex>
lk{resmut}
resptr = new resource();                     data_cond.wait(lk) //unlock & wait
data_cond.notify_one();                      resptr->use(); // lock obtained
```

- Увы, тут всё ещё есть небольшая проблема.
- Ожидание data_cond.wait(lk) может закончиться само по себе (spuriously)

Обсуждение

• Как вы думаете, причины по которым spurious wakeup вообще способен случится, это:

a. Настоящие инженерные соображения

b. Исторические причины связанные с тоннами легаси

Условные переменные

• И вот теперь да, в языке могут существовать условные переменные

```
resource *resptr = nullptr;
std::mutex resmut;
std::condition_variable data_cond;
// thread 1                                  // thread 2
lock_guard<mutex> lk{resmut};                std::unique_lock<mutex>
lk{resmut}
resptr = new resource();                     data_cond.wait(lk, [] {return
(resptr != nullptr)}) //unlock & wait
data_cond.notify_one();                      resptr->use(); // lock obtained
```

• Теперь всё идеально?

Условные переменные

- Вызывающему оповещение потоку не нужно держать мьютекс.

```
resource *resptr = nullptr;
std::mutex resmut;
std::condition_variable data_cond;
// thread 1
{
lock_guard<mutex> lkresmut;
resptr = new resource();
}
data_cond.notify_one();
// thread 2
std::unique_lock<mutex> lkresmut;
data_cond.wait(lk, [] { return (resptr != nullptr); });
resptr->use(); // lock obtained
```

Множественные оповещения

• Можно оповестить всех (хотя тут они всё равно все сериализуются на мьютексе).

```
resource *resptr = nullptr;
std::mutex resmut;
std::condition_variable data_cond;
// thread 1
{
lock_guard<mutex> lkresmut;
resptr = new resource();
}
data_cond.notify_all();
// threads 2..n
std::unique_lock<mutex> lkresmut;
data_cond.wait(lk, [] {
return (resptr != nullptr);
});
resptr->use(); // lock obtained
```

Разделяемые блокировки

• Начнём с наивного вопроса. Ниже тело класса. Оно вообще скомпилируется?

```
std::mutex m_; T value_;
T get() const {
  std::unique_lock<std::mutex> lockm_;
  return value_;
}
void modify(const T &newval) {
  std::unique_lock<std::mutex> lockm_;
  value_ = newval;
}
```

Обсуждение: mutable

• Пожалуй наличие внутри класса мьютекса это один из немногих существенных доводов за использование mutable.

```
class S {
  mutable std::mutex m_; T value_;
public:
T get() const {
  std::unique_lock<std::mutex> lockm_;
  return value_;
}
```

• Что означает обещание const на метод?

Обсуждение: mutable против SRP

- Такое чувство что получившийся класс нарушает SRP?

```
class S {
  T value_; // (1)
  mutable std::mutex m_; // (2)
public:
  T get() const {
    std::unique_lock<std::mutex> lockm_; // (2)
  return value_; // (1)
}
```

Разделяемые блокировки

Допустим чтение происходит в 1000 раз чаще. Какая проблема тогда очевидна?

```
std::mutex m_; T value_;
T get() const {
  std::unique_lock<std::mutex> lockm_;
  return value_;
}
void modify(const T &newval) {
  std::unique_lock<std::mutex> lockm_;
  value_ = newval;
}
```

Разделяемые блокировки

• Решение: расшарить мьютекс для чтения и уникально захватить на запись.

```
mutable std::shared_mutex m_; T value_;
T get() const {
  std::shared_lock<std::shared_mutex> lockm_;
  return value_;
}
void modify(const T &newval) {
  std::unique_lock<std::shared_mutex> lockm_;
  value_ = newval;
}
```

Разгадка проста: безблагодатность

• Рассмотрим защёлкивание уникальной защёлки на обычный мьютекс.

```
std::mutex m;
std::unique_lock<std::mutex> lock{m};
```

pthread_mutex_lock + pthread_mutex_unlock

• И на шаренный. Разница гигантская.

```
std::shared_mutex sm;
std::unique_lock<std::shared_mutex> slock{sm};
```

pthread_rwlock_wrlock + pthread_rwlock_unlock

https://quick-bench.com/q/UDLS-Js5aCo35Y64tiYoRgU5D2w

Рекурсивные мьютексы

• Если инварианты класса меняются в нескольких методах, похоже, что каждый из них требует защиты мьютексом.

```
template <typename T> struct sometype {
foo() {
  lock_guard<mutex> lk{mut_};
  // всё остальное
}
bar() {
  lock_guard<mutex> lk{mut_};
  foo();
}
```

Рекурсивные мьютексы

• Специальный класс std::recursive_mutex позволяет себя защёлкивать многократно и ведёт счётчик закрытий и открытий.

```
template <typename T> struct sometype {
foo() {
  lock_guard<std::recursive_mutex> lk{mut_};
  // всё остальное
}
bar() {
  lock_guard<std::recursive_mutex> lk{mut_};
  foo();
}
```

Ограничения по времени

- Ещё один антипаттерн это std::timed_mutex
- Он позволяет ждать себя с таймаутом.

```
auto now = std::chrono::steady_clock::now();
res = test_mutex.try_lock_until(now + 10s);
if (!res) { // ждать надоело
```

- Или блокировать себя не более чем на какое-то время.

```
if (test_mutex.try_lock_for(Ms(100))) {
// у нас есть 100 миллисекунд
}
```

- Разумеется std::recursive_timed_mutex тоже к вашим услугам.

Обсуждение

- Что плохого в таких мьютексах:
- С точки зрения проектирования?
- С точки зрения реализации в операционке?

Проверка интуиции

sizeof(std::once_flag)
sizeof(std::lock_guard<std::mutex>)
sizeof(std::scoped_lock<std::mutex>)
sizeof(std::unique_lock<std::mutex>)
sizeof(std::shared_lock<std::mutex>)
sizeof(std::condition_variable)
sizeof(std::mutex)
sizeof(std::recursive_mutex)
sizeof(std::timed_mutex)
sizeof(std::recursive_timed_mutex)
sizeof(std::shared_mutex)

https://godbolt.org/z/jzeWK8ena