

ЛЕКЦИЯ 15

ПРИМИТИВЫ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.



ОБСУЖДЕНИЕ

- Проблемы проектирования выглядят гораздо более серьезными, чем в случае с исключениями
- Это и понятно: исключения рисуют произвольное количество выходных дуг, а потоки – еще и произвольное количество входных дуг
- На этом месте следует задать один вопрос: а хотим ли мы безопасные относительно потоков контейнеры?
- И если да, то в каком виде?
- Ответ на этот вопрос мы пока отложим

ОДНОРАЗОВЫЕ СОБЫТИЯ

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

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

- Покритикуйте этот код

ОДНОРАЗОВЫЕ СОБЫТИЯ

- Пусть некая функция содержит в себе и использование ресурса и его одnorазовую инициализацию при необходимости

```
{  
    lock_guard<mutex> lk{result};  
    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 ничего не надо создавать, будут платить за синхронизацию.

ВЫХОД ИЗ СИТУАЦИИ: DCL

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

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

- В реальности стало только хуже: появился тривиальный data race между записью и чтением – это UB
- Еще 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 решение существенно меньше, чем на постоянную сериализацию вокруг мьютекса



УПРАЖНЕНИЕ

Вам принесли 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, инициализация ресурса в потоке А – это **событие**, о котором он пытается сообщить потоку В.
- Делать это через `volatile` **плохо**, но сама **идея хороша!**
- Как бы вы решили проблему сообщения о событии?
- Хватит ли вам для этого уже изученных механизмов?

УСЛОВНЫЕ ПЕРЕМЕННЫЕ: 1ST TRY

- Допустим, в языке существовал бы механизм условных переменных

```
resource *resptr = nullptr;
```

```
std::mutex resmut;
```

```
std::conditional_variable data_cond;
```

```
// thread 1
```

```
lock_guard<mutex> lk{resmut};
```

```
resptr = new resource();
```

```
data_cond.notify_one();
```

```
// thread 2
```

```
data_cond.wait();
```

```
resptr->use();
```

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

УСЛОВНЫЕ ПЕРЕМЕННЫЕ: 1ST TRY

- Допустим, в языке существовал бы механизм условных переменных

```
resource *resptr = nullptr;
```

```
std::mutex resmut;
```

```
std::conditional_variable data_cond;
```

```
// thread 1
```

```
lock_guard<mutex> lk{resmut};
```


```
resptr = new resource();
```

```
data_cond.notify_one();
```

```
// thread 2
```

```
data_cond.wait();
```

```
resptr->use();
```



- Между этими строчками может пройти времени больше, чем кажется
- Хотелось бы дождавшись ресурса, сразу взять мьютекс

УСЛОВНЫЕ ПЕРЕМЕННЫЕ: 2ND TRY

- Допустим, в языке существовал бы механизм условных переменных

```
resource *resptr = nullptr;
```

```
std::mutex resmut;
```

```
std::conditional_variable data_cond;
```

```
// thread 1
```

```
lock_guard<mutex> lk{resmut};
```

```
resptr = new resource();
```

```
data_cond.notify_one();
```

```
// thread 2
```

```
data_cond.wait(resmut);
```

```
resptr->use();
```

```
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 unlocked  
    ul.lock();  
    res->use();  
} // unlocked by dtor
```

- В принципе, его можно использовать даже вместо `lock_guard`
- Но это расточительно. Признак взятия блокировки – лишнее поле в классе

ОБСУЖДЕНИЕ

- Возможность вручную вызвать lock означает возможность вызвать его вручную повторно

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` и на этом спасибо...)
- Мы расходуем место для одного лишнего признака и получаем крайне удобный интерфейс

УСЛОВНЫЕ ПЕРЕМЕННЫЕ

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

```
resource *resptr = nullptr;
```

```
std::mutex resmut;
```

```
std::conditional_variable data_cond;
```

```
// thread 1
```

```
lock_guard<mutex> lk{resmut};
```

```
resptr = new resource();
```

```
data_cond.notify_one();
```

```
// thread 2
```

```
std::unique_lock<mutex> lk{resmut};
```

```
data_cond.wait(lk); // unlock & wait
```

```
resptr->use(); // lock obtained
```

- Увы, тут все еще 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 случиться, это
 - Настоящие инженерные соображения
 - Исторические причины, связанные с тоннами легаси
- Проголосуем!

УСЛОВНЫЕ ПЕРЕМЕННЫЕ

- Вызывающему оповещение потоку не нужно держать мьютекс

```
resource *resptr = nullptr;
```

```
std::mutex resmut;
```

```
std::conditional_variable data_cond;
```

```
// thread 1
```

```
{  
    lock_guard<mutex> lk{resmut};  
    resptr = new resource();  
} data_cond.notify_one();
```

```
// thread 2
```

```
std::unique_lock<mutex> lk{resmut};  
data_cond.wait(lk, []  
{return (resptr != nullptr);});  
resptr->use(); // lock obtained
```

МНОЖЕСТВЕННЫЕ ОПОВЕЩЕНИЯ

- Можно оповестить всех (хотя тут они все равно сериализуются на мьютексе)

```
resource *resptr = nullptr;
```

```
std::mutex resmut;
```

```
std::conditional_variable data_cond;
```

```
// thread 1
```

```
{  
    lock_guard<mutex> lk{resmut};  
    resptr = new resource();  
} data_cond.notify_all();
```

```
// thread 2
```

```
std::unique_lock<mutex> lk{resmut};  
data_cond.wait(lk, []  
{return (resptr != nullptr);});  
resptr->use(); // lock obtained
```

ОБСУЖДЕНИЕ

- Исследование трассы и основные события

```
$ strace -f ./a.out >& strace.log
```

```
$ grep clone strace.log
```

```
$ grep futex strace.log
```

РАЗДЕЛЯЕМЫЕ БЛОКИРОВКИ

- Начнем с наивного вопроса. ниже тело класса. Оно вообще скомпилируется?

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

- Что означает обещание `const` на метод?

ОБСУЖДЕНИЕ: MUTABLE VS SRP

- Такое чувство, что класс нарушает SRP

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

- Что означает обещание `const` на метод?

РАЗДЕЛЯЕМЫЕ БЛОКИРОВКИ

- Допустим, чтение происходит в 1000 раз чаще. какая проблема тогда очевидна?

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

РАЗДЕЛЯЕМЫЕ БЛОКИРОВКИ

- Ужасно не хочется тратить такты на синхронизацию чтения

```
mutable std::mutex m_; T value;
```

```
T get() const {
```

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

```
    return value_;
```

```
}
```

```
void modify(const Y &newval) {
```

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

```
    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> lock{m_};
```

```
    return value_;
```

```
}
```

```
void modify(const Y &newval) {
```

```
    std::unique_lock<std::shared_mutex> lock{m_};
```

```
    value_ = newval;
```

```
}
```

РАЗГАДКА ПРОСТА: БЕЗБЛАГОДАТНОСТЬ

- Рассмотрим защелкивание уникальной защелки на обычный мьютекс

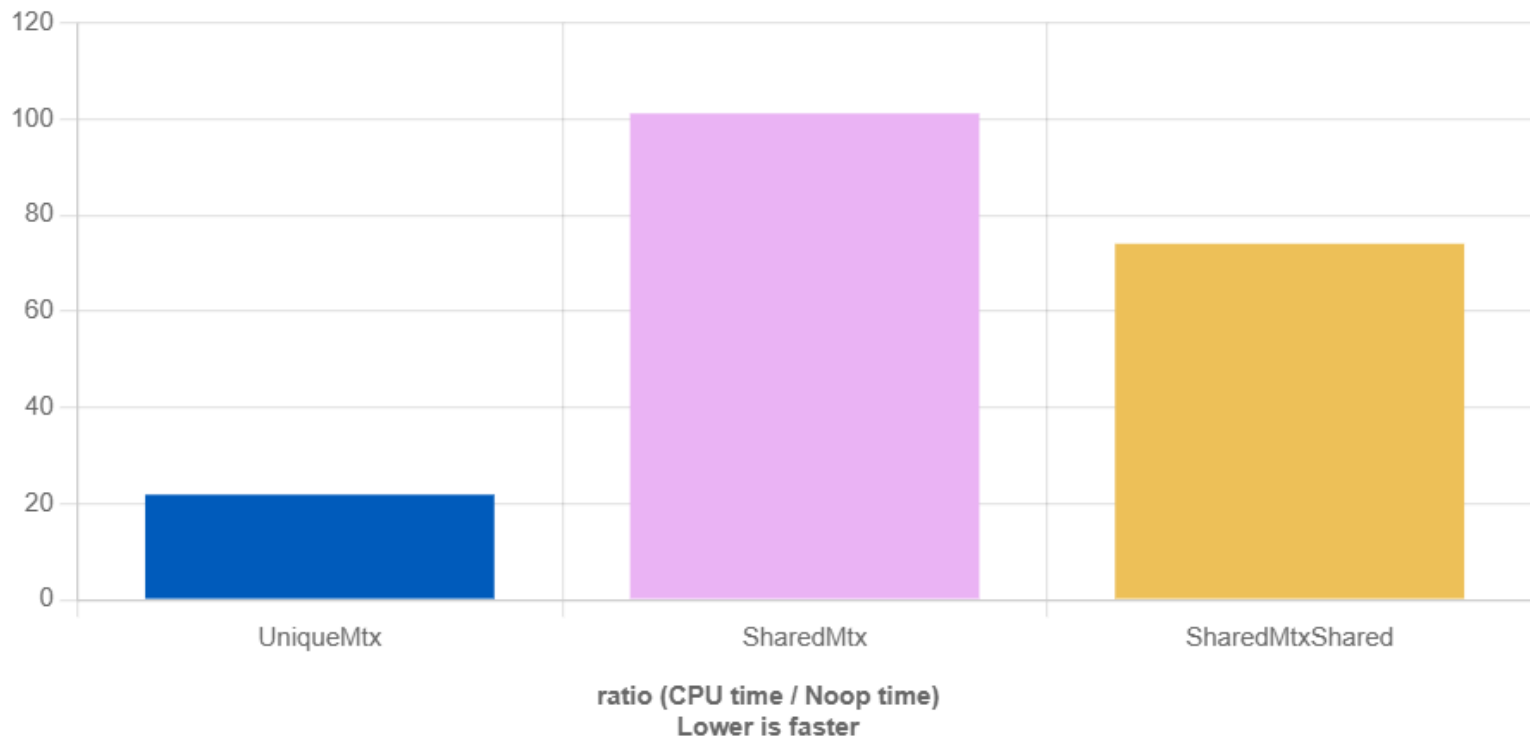
```
std::mutex m;  
std::unique_lock<std::mutex> lock{m};  
pthread_mutex_lock + pthread_mutex_unlock
```

И на шаренный. Разница гигантская

```
std::shared_mutex sm;  
std::unique_lock<std::shared_mutex> slock{m};  
pthread_rwlock_lock + pthread_rwlock_unlock
```

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

РАЗГАДКА ПРОСТА: БЕЗБЛАГОДАТНОСТЬ



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_}; // something else  
    }  
  
    bar() {  
        lock_guard<mutex> lk{mut_};  
        foo();  
    }  
}
```

РЕКУРСИВНЫЕ МЬЮТЕКСЫ

- Специальный класс `std::recursive_mutex` позволяет себя защелкивать многократно и ведет счетчик закрытий и открытий

```
template <typename T> struct sometype {  
    foo() {  
        lock_guard<recursive_mutex> lk{mut_}; // something  
else  
    }  
    bar() {  
        lock_guard<recursive_mutex> lk{mut_};  
        foo();  
    }  
}
```

ОГРАНИЧЕНИЯ ПО ВРЕМЕНИ

- Еще 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

- Каковы, по вашему, размеры основных типов?

```
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::conditional_variable);  
sizeof(std::mutex);  
sizeof(std::shared_mutex);  
sizeof(std::recursive_mutex);  
sizeof(std::timed_mutex);  
sizeof(std::recursive_timed_mutex);
```

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М.: «Вильямс», 2006. – 720 с.
3. Дональд Кнут Искусство программирования, том 2. Получисленные алгоритмы - The Art of Computer Programming, vol.2. Seminumerical Algorithms. — 3-е изд. — М.: «Вильямс», 2007. — 832 с.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦМНО, 1999. – 960 с.
5. Скотт Мейерс, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2017. – 300 с.
6. Скотт Мейерс, Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2016. – 298 с.