Идея по улучшению многопоточного кода

Александр Корнилов

Лаборатория Касперского

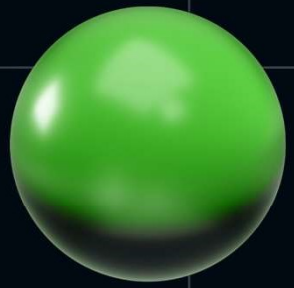
akornilov.82@mail.ru akornilov.82

C++ Russia 2023

План выступления

- Типичные ошибки многопоточного кода
- Принятые практики решения проблемы
- Описание новой идеи в реализации на C++
- Пример прикладного использования нового подхода

Основные причины ошибок



Общие данные, которые совместно используются разными потоками, обычно защищают мьютексами. Однако мьютексы создаются независимо от защищаемых ими данных.

Поэтому достаточно распространены ситуации когда:

- Мьютекс не был заблокирован при обращении к общим данным
- Был заблокирован другой мьютекс, предназначенный для защиты других данных
- Мьютекс заблокирован без обращения к общим данным

Последствия ошибок

- Неожиданные падения в разных местах
- Падения происходят случайно, но могут зависеть от нагрузки на приложение и системного окружения
- Трудно определить и воспроизвести сценарий сбоя
- Нарушение целостности данных
- Ухудшение производительности

```cpp
class Example
{
private:
    int _id;                    // Данные, которые не нуждаются в защите
    mutable std::mutex _mutex1; // Первый мьютекс
    std::string _name;          // Данные, которые, по изначальной задумке автора, должны находится под защитой _mutex1
    std::vector<int> _data1;    // Еще одни данные под защитой первого мьютекса
    mutable std::mutex _mutex2; // Второй мьютекс (предположим, был добавлен позже другим автором, при серьезном расширении функционала класса)
    std::vector<int> _data2;    // Новые данные, которые должны защищаться вторым мьютексом

public:

    // Все правильно: обращени к общим данным происходит при заблокированном _mutex1
    // Мьютекс блокируется с помощью вспомогательного класса, который гарантирует его разблокировку в деструкторе.
    size_t dataSize() const {
        std::lock_guard<std::mutex> lock(_mutex1);
        return _data1.size();
    }

    // Ошибка: забыли заблокировать _mutex1
    std::string name() const {
        return _name;
    }

    // Ошибка: Поле _id не нуждается в защите мьютексом
    size_t id() const {
        std::lock_guard<std::mutex> lock(_mutex1);
        return _id;
    }

    void reset() {
        {
            std::lock_guard<std::mutex> lock(_mutex1); // Все правильно: заблокирован мьютекс для защиты _data1
            _data1.clear();
        }
        {
            std::lock_guard<std::mutex> lock(_mutex1); // Ошибка: для защиты _data2 нужно было заблокировать _mutex2
            _data2.clear();
        }
    }

    void reserve(size_t count) {
        std::lock_guard<std::mutex> lock(_mutex1);
        _data1.reserve(count); // Все правильно: мьютекс для защиты _data1 заблокирован
        _data2.reserve(count); // Ошибка: данные находятся под защитой другого мьютекса, здесь к ним обращаться нельзя
    }

};
```

Принятые методики

- Повышение качества инспекции кода
- Использование готовых библиотек для распараллеливания (например Streams в Java 8)
- Применение статических анализаторов / санитайзеров кода
- Использование только неизменяемых общих данных
- Полный запрет на использование общих данных (например модель акторов в Scala, когда между потоками разрешен только обмен асинхронными сообщениями)
- Модель при которой прямые вызовы методов класса возможны только из породившего его потока, остальные происходят опосредованно, например, через очередь сообщений (модель многопоточности в Qt)
- Компилятор в состоянии полностью проконтролировать любые обращения к общим данным и запретить некорректные попытки (например, язык программирования D или Rust)

Суть идеи

- Мьютекс и защищаемые им данные должны быть неразрывно связаны и обладать общим жизненным циклом
- Не должно осуществляться ручное управление мьютексом, вместо этого необходимо обеспечить *защищенный доступ* к общим данным
- Доступ к общим данным должен быть четко регламентирован
- Все вышеперечисленные условия должны обеспечиваться существующими средствами языка программирования без изменений в его синтаксисе

Что на практике?

- Для реализации идеи лучше всего подходит *шаблонный класс*, назовем его *SharedState*, который будет инкапсулировать в себе общие данные и средства для их защиты
- Все общие данные помещаются в отдельный класс или структуру которым специфицируется шаблонный класс SharedState
- В конструкторе SharedState создается объект общих данных, который напрямую недоступен извне
- SharedState предоставляет четко регламентированный защищенный доступ к общим данным
- У SharedState можно запросить доступ на чтение или модификацию общих данных, а также дожидаться их определенного состояния
- Доступ к общим данным происходит с помощью *лямбда-выражений*

Интерфейс SharedState

- В конструктор передаются все параметры, необходимые для создания объекта общих данных
- Шаблонный метод для просмотра данных называется view и принимает в качестве параметра *std::function*, которая будет вызвана с константной ссылкой на общие данные. Метод view можно специфицировать под любое возвращаемое значение
- Простой метод modify для изменения общих данных, принимает *std::function*, которая будет вызвана с не константной ссылкой на общие данные
- Метод modify, который возвращает класс Action

Интерфейс Action

- Методы для простой модификации общих данных без оповещения об их изменении access и extract

- Методы для модификации общих данных и оповещении об их изменении notifyOne и notifyAll

- Метод when который принимает *std::function*-предикейт для определения подходящего состояния общих данных. Метод возвращает тот же экземпляр объекта Action поэтому можно строить цепочки вызовов:

```
111         while(!exit) {
112             auto task = _state.modify().when([&isWorking](auto& state) {
113                 const auto wakeupIf = state.isFinished() || state.hasTasks();
114                 if (!wakeupIf && isWorking) { isWorking = false; state.waitingBegin(); }
115                 return wakeupIf;
116 
117             }).template extract<std::packaged_task<R()>>([&isWorking, &exit](auto& state) {
118                 exit = state.isFinished();
119                 if (!exit && !isWorking) {
120                     isWorking = true;
121                     state.waitingFinish();
122                 }
123                 return state.takeTask();
124             });
125             if (!exit) { task(); }
126         }
```

Что под капотом?

```cpp
template<typename T>
class SharedState final
{
public:
    class Action;

    template<typename... A, typename = std::enable_if_t<std::is_constructible_v<T, A...>>>
    inline SharedState(A&&... args): _state(std::forward<A>(args)...) {}

    inline void view(std::function<void(const T&)> block) const {
        LockRead lock(_mutex);
        block(_state);
    }

    template<typename R>
    inline R view(std::function<R(const T&)> block) const {
        LockRead lock(_mutex);
        return block(_state);
    }

    inline void modify(std::function<void(T&)> block) {
        LockWriteGuard lock(_mutex);
        block(_state);
    }

    inline Action modify() {
        return Action(*this);
    }

private:
#if __cplusplus >= 201703L
    using Mutex = std::shared_mutex;
    using LockRead = std::shared_lock<Mutex>;
    using LockWriteGuard = std::unique_lock<Mutex>;
    using ConditionVariable = std::condition_variable_any;

#else
    using Mutex = std::mutex;
    using LockRead = std::lock_guard<Mutex>;
    using LockWriteGuard = std::lock_guard<Mutex>;
    using ConditionVariable = std::condition_variable;
#endif // __cplusplus >= 201703L

    using LockWrite = std::unique_lock<Mutex>;

    T _state;
    mutable Mutex _mutex;
    ConditionVariable _notify;
};
```

Что под капотом?

```cpp
template <typename T>
class SharedState<T>::Action final
{
public:
    void access(std::function<void(T&)> block) {
        return block(_parent._state);
    }

    template<typename R>
    inline R extract(std::function<R(T&)> block) {
        return block(_parent._state);
    }

    inline void notifyOne(std::function<void(T&)> block) {
        block(_parent._state);
        _parent._notify.notify_one();
    }

    template<typename R>
    inline R notifyOne(std::function<R(T&)> block) {
        R result = block(_parent._state);
        _parent._notify.notify_one();
        return result;
    }

    inline void notifyAll(std::function<void(T&)> block) {
        block(_parent._state);
        _parent._notify.notify_all();
    }

    template<typename R>
    inline R notifyAll(std::function<R(T&)> block) {
        R result = block(_parent._state);
        _parent._notify.notify_all();
        return result;
    }

    inline Action& when(std::function<bool(T&)> pred) {
        _parent._notify.wait(_lock, [&pred, this] { return pred(_parent._state); });
        return *this;
    }

private:
    friend class SharedState;

    inline Action(SharedState& parent):
        _parent(parent),
        _lock(_parent._mutex)
    {
    }

    SharedState& _parent;
    LockWrite _lock;
};
```

А если с SharedState?

```cpp
class Example2
{
private:
    struct Data1 {
        std::string name;
        std::vector<int> data;
    };

    struct Data2 {
        std::vector<int> data;
    };

    int _id;                    // Данные, которые не нуждаются в защите
    SharedState<Data1> _data1;  // Первые общие данные, защищенные SharedState
    SharedState<Data2> _data2;  // Вторые общие данные, защищенные SharedState

public:

    size_t dataSize() const {
        // Запрашиваем доступ на просмотр данных и специфицируем метод для нужного нам типа возвращаемого значения
        return _data1.view<size_t>([](const auto& s) {
            return s.data.size(); // Внутри этого блока несколько потоков могут читать данные (>= C++17), но не модифицировать
        });
    }

    std::string name() const {
        return _data1.view<std::string>([](const auto& s) {
            return s.name;
        });
    }

    size_t id() const {
        return _id;
    }

    void reset() {
        // Запрашиваем доступ на модификацию общих данных
        _data1.modify([](auto& s) {
           s.data.clear(); // Внтури этого блока осуществляется эксулюзивный доступ к общим данным только для одного потока
        });
        _data2.modify([](auto& s) {
           s.data.clear();
        });
    }

    void reserve(size_t count) {
        _data1.modify([&count](auto& s) {
           s.data.reserve(count);
        });
        _data2.modify([&count](auto& s) {
           s.data.reserve(count);
        });
    }
};
```

Преимущества

- Не нужно явно создавать такие примитивы синхронизации как мьютексы, локеры и condition variables и управлять ими вручную
- Общие данные представляют собой отдельную сущность, а не разбросанные по классу разрозненные поля. Такой подход материализует логическую связь общих данных
- Концептуально мы уходим от категорий заблокирован/разблокирован, а начинаем мыслить в терминах доступа к данным: просмотр/модификация/ожидание
- Автор кода явно выражает свои намерению по отношению к общим данным. Например, объявляет прямо с помощью кода что в следующих строчках хочет посмотреть общие данные. Такое самодокументирование кода может помочь другим людям в его поддержке
- Благодаря использованию std::shared_mutex несколько потоков могут одновременно читать одни и те же данные, что в отдельных случаях может повысить общую производительность
- Все методы SharedState реализованы как inline и объявлены прямо в заголовочном файле, поэтому производительность не пострадает после перехода на SharedState

Правила использования

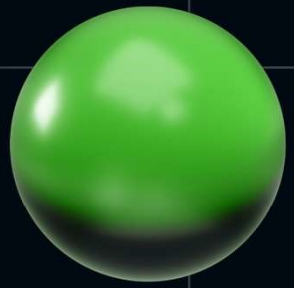
- Объект общих данных должен быть простым и не содержать сложной/неочевидной логики, а также желательно обойтись без ссылок на внешние объекты. Основное его назначение быть тривиальным контейнером для общих данных

- Необходимо минимизировать количество кода в лямбдах доступа к общим данным и ограничится извлечением необходимой информации или ее модификацией. Не стоит вызывать внутри защищенных блоков какие-то тяжеловесные операции или обращаться к тому же экземпляру SharedState

- Сложные манипуляции с извлеченными данными лучше производить в локальных переменных за пределами блока лямбда-выражения. Однако не стоит забывать, что связь извлеченных данных с их источником в объекте общих данных сразу же теряется после выходя из лямбды, т.к. другой поток может их сразу же изменить

- Нельзя сохранять ссылки на объект общих данных и работать с ним в обход регламента

Ссылки на материалы:



Оригинальная статья на Habr: https://habr.com/ru/articles/597271/

Полная реализация SharedState на C++: https://sourceforge.net/p/cpp-mate/code/ci/default/tree/src/main/public/CppMate/SharedState.hpp

Документация (doxygen): https://cpp-mate.sourceforge.io/doc/classCppMate_1_1SharedState.html

Примеры использования SharedState:

Простая реализация Thread Pool: https://sourceforge.net/p/cpp-mate/code/ci/default/tree/src/main/public/CppMate/ThreadPool.hpp

https://cpp-mate.sourceforge.io/doc/classCppMate_1_1ThreadPool.html

Реализация абстрактного кэша: https://sourceforge.net/p/cpp-mate/code/ci/default/tree/src/main/public/CppMate/Cache.hpp

Вопросы?

Спасибо за внимание! ☺