Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ПІДТРИМКА КОНКУРЕНТНОСТІ У C++

Текстова частина до курсової роботи

за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
доцент, Бублик В.В.

_____

(підпис)
“__”_____2022 р.

Виконав студент
Андрійченко Д. С.
“__”_____2022 р.

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри мультимедійних систем,
доцент, к.ф-м.н.
_____О. П. Жежерун (підпис)
„____”_____2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Андрійченку Даниїлу Сергійовичу факультету інформатики 3-го курсу

ТЕМА Підтримка конкурентності у C++

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Вступ
1 Основні визначення
2 Управління потоками
3 Взаємне виключення потоків
4 Умовні змінні
5 Реалізація потокобезпечного патерну Singleton
Висновки
Список використаної літератури
Додатки (за необхідністю)

Дата видачі „___” _____ 2021 р. Керівник _____ (підпис)
Завдання отримав _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ РОБОТИ

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи	14.10.2021	
2.	Пошук тематичної літератури	23.10.2021	
3.	Ознайомлення з науковою літературою	18.01.2022	
4.	Ознайомлення з бібліотечними засобами підтримки паралелізму у C++	01.03.2022	
5.	Визначення структури роботи	01.04.2022	
6.	Написання першого - четвертого розділів роботи. Частковий огляд можливостей стандартної бібліотеки у підтримці паралелілзму у C++	21.04.2022	
7.	Написання п’ятого розділу роботи. Вимірювання часу виконання доступу до різних реалізацій потокобезпечного Сінглтону.	15.05.2022	
8.	Написання висновків курсової роботи	30.05.2022	
9.	Перегляд змісту роботи керівником	01.06.2022	
10.	Корегування роботи згідно із зауваженнями керівника	03.06.2022	
11.	Подача курсової роботи на перевірку	07.06.2022	

Зміст

Вступ

Розробники часто стикаються з необхідністю розробки багатопоточних додатків, тому питання багатопоточності вимагають детального вивчення. Давайте ознайомимося з основними термінами, що використовуються в джерелах інформації про багатопоточність, розглянемо завдання та проблеми багатопоточності та вивчимо засоби стандартної бібліотеки C++, які допоможуть створювати багатопоточні програми.

Розділ 1. Основні визначення

1.1 Види конкурентності

Машина, яка має один процесор з одним обчислювальним блоком або ядром може виконувати тільки одну задачу в конкретний момент часу. Проте вона може переключатися між задачами багато разів за секунду. При виконанні невеличкої частини однієї задачі, потім другої задачі і т.д. складається враження що задачі вирішуються одночасно. Це називається **переключенням задач** або псевдопаралельним виконанням. У таких системах процеси виконуються послідовно, займаючи малі кванти процесорного часу.

По-справжньому паралельне виконання завдань можливе лише в багатопроцесорній системі, оскільки тільки в них є кілька системних конвеєрів для виконання команд.

Це називається **апаратною конкурентністю**.

Усі технічні засоби, прийоми і класи, розглянуті в цій роботі можна буде застосовувати незалежно від того на якому комп’ютері виконується програма – з одним одноядерним процесором чи з декількома багатоядерними. І неважливо, як саме буде реалізовуватись конкурентність – шляхом переключення задач чи за рахунок справжнього апаратного паралелізму.

1.2 Навіщо використовується конкурентність

Існує дві основні причини застосунку конкурентності в програмах:

- **Розділити задачі**

  У деяких випадках програму можно спростити за рахунок винесення механізмів чергування виконання різних слабо взаємопов'язаних

підзадач, що вимагають одночасного виконання, в окрему підсистему багатопоточності.

- **Підвищити продуктивність**

  Існує два способи застосування конкурентності для підвищення продуктивності. Перший і найбільш очевидний – розбити одне завдання на частини і виконувати їх паралельно, скорочуючи цим загальний час виконання.

  Це паралелізм завдань. На словах все досить просто, але процес може виявитися зовсім не простим, тому що, можливо, між різними частинами програми існує безліч залежностей. Розбиття на частини також можливе відносно обробки: один потік виконує одну частину алгоритму, інший потік — іншу або відносно даних: всі потоки виконують одну і ту саму операцію з різними частинами даних. Останній підхід називається **паралелізмом даних**.

1.3 Коли конкурентність не потрібна

По суті, єдиною причиною відмови щодо використання конкурентності є негативний баланс переваг і витрат. Код, що реалізує конкурентність, часто складніше зрозуміти,

тому створення та підтримка багатопоточного коду пов'язані з прямими інтелектуальними витратами, а додаткова складність може спричинити збільшення

кількості помилок. Якщо потенційний приріст продуктивності недостатньо великий або розбиття завдань виражено нечітко і це не може виправдати додаткові витрати часу на розробку, необхідні для досягнення успіху, а також додаткові витрати, пов'язані з підтримкою багатопотокового коду, використовувати конкурентність немає сенсу.

Розділ 2. Управління потоками

Кожна програма на C++ має як мінімум один потік, що запускається середовищем виконання C++, — потік, що виконує функцію main(). Потім програма може запустити додаткові потоки, точкою входу в які є інша функція. Після чого ці потоки та початковий потік виконуються одночасно. Аналогічно завершенню програми при виході з main() потік завершується при поверненні з функції, зазначеної як точка входу.

2.1 std::thread

Основний клас створення нових потоків в C++ - це std::thread.

Визначення класу:

```
class thread
{
public:
    // Типи
    class id;
    typedef implementation-defined native_handle_type; //
необов'язково

    // Конструктори і десктруктор
    thread() noexcept;
    ~thread();

    template<typename Callable, typename ...Args>
    explicit thread(Callable&& func,Args&&... args);

    // Копіювання і переміщення
    thread(thread const& other) = delete;
    thread& operator=(thread const& other) = delete;

    thread(thread&& other) noexcept;
    thread& operator=(thread&& other) noexcept;

    void swap(thread& other) noexcept;

    void join();
    void detach();
    bool joinable() const noexcept;

    id get_id() const noexcept;
```

```
    native_handle_type native_handle();

    static unsigned hardware_concurrency() noexcept;
};

void swap(thread& lhs,thread& rhs);
```

Об'єкт класу представляє один потік виконання. Новий потік починає виконання відразу після побудови об'єкта std::thread. Виконання починається з функції верхнього рівня, яка передається як аргумент у конструктор std::thread.

Значення цієї функції, що повертається, ігнорується, а якщо в ній буде кинуто виняток, який не буде оброблено в цьому ж потоці, то викличеться std::terminate.

Передати значення, що повертається, або виключення з нового потоку назовні можна через std::promise або через глобальні змінні (робота з якими вимагатиме синхронізації, див. std::mutex і std::atomic).

Об'єкти std::thread також можуть бути не пов'язані з жодним потоком (після default construction, move from, detach або join), і потік виконання може бути не пов'язаний з жодним об'єктом std::thread (після detach).

Жодні два об'єкти std::thread не можуть представляти той самий потік виконання; std::thread не можна копіювати (не CopyConstructible або CopyAssignable), але можна переміщати (є MoveConstructible і MoveAssignable).

2.2 Запуск потоку, join та detach

Потоки запускаються створенням об'єкта std::thread, у якому визначається завдання, що виконується в потоці. У найпростішому випадку це завдання є звичайною функцією. Ця функція виконується у власному потоці, доки не поверне управління, після чого потік зупиняється. Що б не збирався робити потік і звідки він не запускався, його запуск з використанням стандартної бібліотеки C++ завжди зводиться до створення об'єкта std::thread:

```
void do_some_work();
std::thread my_thread(do_some_work);
```

std::thread працює з будь-яким Callable об’єктом, тому конструктору std::thread можна також передати екземпляр класу з оператором виклику функції:

```
class background_task{
public:
   void operator()() const {
      do_something();
      do_something_else();
   }
};
background_task f;
std::thread my_thread(f);
```

В даному випадку наданий функціональний об'єкт копіюється в сховище, що належить новоствореному потоку виконання, і викликається звідти. Тому важливо, щоб копія діяла аналогічно до оригіналу, інакше результат може не відповідати очікуваному.

За допомогою лямбда-виразу попередній приклад можна записати так:

```
std::thread my_thread([]
{
   do_something();
   do_something_else();
});
```

Після запуску потоку, потрібно прийняти однозначне рішення, чекати його завершення (join) або пустити його на самоплив (detach). Якщо не ухвалити рішення до знищення об'єкта std::thread, то програма завершиться (деструктор std::thread викличе std::terminate()). Рішення потрібно ухвалювати до того, як об'єкт std::thread буде знищений. Сам потік цілком міг би завершитися задовго до його приєднання або від'єднання. Якщо його від'єднати, то за умови, що він все ще виконується, він буде виконуватися, і цей процес може продовжуватися ще довго і після знищення об'єкта std::thread. Виконання буде припинено лише тоді, коли зрештою відбудеться повернення з функції потоку. Якщо не чекати завершення

потоку, необхідно переконатися, що дані, до яких він звертається, будуть дійсними, доки він не закінчить працювати з ними.

Дочекатися завершення потоку можна, викликавши join() для пов'язаного екземпляра std::thread. Виклик join() призводить до очищення об'єкта std::thread, тому об'єкт std::thread більше не пов'язаний із завершеним потоком. Мало того, він не пов'язаний з жодним потоком. Це означає, що join() можна викликати для конкретного потоку лише один раз: як тільки був викликаний метод join(), об'єкт std::thread втрачає можливість приєднання, а метод joinable() поверне значення false.

Виклик методу detach() для об'єкта std::thread дозволяє потоку виконуватись у фоновому режимі, безпосередня взаємодія з ним не потрібна. Можливість дочекатися завершення цього потоку зникає: якщо потік від'єднується, отримати об'єкт std::thread, що посилається, неможливо, тому такий потік більше не можна приєднати. Від'єднані потоки фактично виконуються у фоновому режимі, володіння та керування ними передаються в бібліотеку середовища виконання C++, яка гарантує правильне вивільнення ресурсів, пов'язаних з потоком при виході з нього. Як правило, такі потоки є дуже тривалими, працюючи протягом практично всього часу життя програми і виконуючи фонове завдання, наприклад відстежуючи стан файлової системи, видаляючи записи з кеш-пам'яті об'єктів, що не використовуються, або оптимізуючи структури даних. Метод detach() не можна викликати на об'єктові std::thread, який не має пов'язаного з ним потоку виконання. Ця вимога аналогічна тому, що пред'являється до виклику методу join(), і перевірку можна провести так само - викликати для об'єкта t типу std::thread метод t.detach() можливо, тільки якщо метод t.joinable() поверне значення true.

2.3 Передача аргументів

Завдяки варіативному шаблону конструктора std::thread передача аргументів об'єкту, що викликається, або функції зводиться до простої передачі додаткових аргументів конструктору std::thread. Але важливо врахувати, що за замовчуванням аргументи копіюються у внутрішнє сховище, де до них може отримати доступ новостворений потік виконання, а потім передаються об'єкту, що викликається, або функції як r-значення (rvalues), ніби вони тимчасові. Так робиться, навіть якщо відповідний параметр функції очікує посилання. Розглянемо приклад:

```
void f(int i, std::string const& s);
std::thread t(f, 3, "hello");
```

В результаті створюється новий потік виконання, пов'язаний з t який викликає функцію f(3,"hello"). Зверніть увагу: навіть якщо f як другий параметр приймає std::string, рядковий літерал передається як char const* і перетворюється на std::string тільки в контексті нового потоку. Це стає особливо важливим, коли, як показано далі, наданий аргумент є вказівником на локальну змінну:

```
void f(int i, std::string const& s);
void oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer, "%i", some_param);
    std::thread t(f, 3, buffer);
    t.detach();
}
```

Тут це указник на буфер локальної змінної, який передається у новий потік. І висока ймовірність того, що вихід із функції oops відбудеться, перш ніж буфер буде в новому потоці перетворений на std::string, що викличе невизначену поведінку. Рішенням є приведення до типу std::string перед передачею буфера в конструктор std::thread:

```
void f(int i, std::string const& s);
void oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer, "%i", some_param);
```

```
    std::thread t(f, 3, std::string(buffer));
    t.detach();
}
```

В даному випадку причиною виникнення проблеми є надія на передбачуване перетворення вказівника на буфер в об'єкт std::string, очікуваний як параметр функції, але це перетворення відбувається занадто пізно, оскільки конструктор std::thread копіює надані значення як є, не перетворюючи на очікуваний тип аргументу.

Якщо працювати зі std::bind вже доводилося, то в семантиці передачі параметрів не буде нічого нового, оскільки і операція конструктора std::thread, і операція std::bind визначені в рамках одного і того ж механізму. Тобто, можна, наприклад, передати покажчик на компонентну функцію за умови, що другим аргументом надано відповідний покажчик на об'єкт:

```
class X
{
public:
    void do_lengthy_work();
};
X my_x;
std::thread t(&X::do_lengthy_work, &my_x);
```

Цей код викличе my_x.do_lengthy_work() у новому потоці, оскільки як вказівник на об'єкт надається адреса my_x. Такому виклику компонентної функції також можна надавати аргументи: третій аргумент конструктора std::thread буде першим аргументом компонентної функції тощо.

В сценарії, коли аргумент не можна скопіювати, а можна тільки перемістити, слід явно викликати std::move при передачі аргументу в параметри конкструктора потоку:

```
void process_big_object(std::unique_ptr<big_object>);
std::unique_ptr<big_object> p(new big_object);
p->prepare_data(42);
std::thread t(process_big_object,std::move(p));
```

Право володіння big_object спочатку передається внутрішньому сховищу

новоствореного потоку, а потім переходить до process_big_object.

2.4 Id, native_handle

- std::thread::get_id() повертає ID потоку. Можна використовувати для логування або як ключ асоціативного контейнера потоків.
- std::thread::native_handle() повертає специфічний для операційної системи handle потоку, який можна передавати в методи WinAPI або pthreads для більш гнучкого управління потоками.

2.5 std::jthread

Розглянемо приклад:

```
struct func;
void f()
{
    int some_local_state = 0;
    func my_func(some_local_state);
    std::thread t(my_func);
    try
    {
        do_something_in_current_thread();
    }
    catch (...)
    {
        t.join();
        throw;
    }
    t.join();
}
```

Оскільки для того щоб уникнути завершення програми потрібно викликати метод join чи detach, потрібно врахувати усі можливі шляхи виходу об'єкту std::thread за scope, як нормальні, так і виняткові, і бажано забезпечити для цього простий і лаконічний механізм.

Така проблема викликає дежавю і на думку спадає RAII:

```
class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_): t(t_) {}
```

```
    ~thread_guard()
    {
        if (t.joinable())
        {
            t.join();
        }
    }

    thread_guard(thread_guard const&) = delete;
    thread_guard& operator=(thread_guard const&) = delete;
};
struct func;
void f()
{
    int some_local_state = 0;
    func my_func(some_local_state);
    std::thread t(my_func);
    thread_guard g(t);
    do_something_in_current_thread();
}
```

Коли виконання поточного потоку досягає кінця функції f, локальні об'єкти знищуються у порядку, зворотному порядку їх побудови. Отже, спочатку знищується об'єкт g типу thread_guard, а його деструкторі відбувається приєднання до потоку. Це спостерігається навіть при винятковому завершенні виконання функції.

Також, враховуючи підтримку переміщення std::thread існує можливість зобов’язати thread_guard володіти потоком і це допомогло би уникнути небажаних наслідків в тому випадку, якщо thread_guard переживе поток на який посилається. Реалізація такого класу – scoped_guard і простий приклад наведені нижче:

```
class scoped_thread
{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_): t(std::move(t_))
    {
        if (!t.joinable())
            throw std::logic_error("No thread");
    }

    ~scoped_thread()
```

```
    {
        t.join();
    }

    scoped_thread(scoped_thread const&) = delete;
    scoped_thread& operator=(scoped_thread const&) = delete;
};
struct func;
void f()
{
    int some_local_state = 0;
    scoped_thread g(std::thread t(func (some_local_state)));
    do_something_in_current_thread();
}
```

Новий потік замість створення для нього окремої іменованої змінної передається безпосередньо у scoped_thread.

В C++20 з'явився новий клас std::jthread, який автоматично приєднує поток, як це робить scoped_thread. Більше того, він пропонує інтерфейс для зупинки. На відміну від std::thread, std::jthread містить внутрішній закритий член типу std::stop_source, який зберігає stop-state. Конструктор jthread приймає функцію, яка приймає std::stop_token як свій перший аргумент. Цей аргумент передається у функцію з stop_source, і дозволяє функції перевірити, чи була запитана зупинка під час її виконання, та завершитись у разі потреби. Також існує можливість зв'язати callback функцію з подією зупинки потоку завдяки std::stop_callback.

2.6 Управління поточним потоком

Заголовок <thread> також містить декілька методів для управління поточним потоком. Всі вони знаходяться в просторі імен std::this_thread:

```
namespace std
{
    class thread;
    namespace this_thread
    {
        thread::id get_id() noexcept;

        void yield() noexcept;
```

```
        template<typename Rep, typename Period>
        void sleep_for(
               std::chrono::duration<Rep,Period> duration);

        template<typename Clock, typename Duration>
        void sleep_until(
               std::chrono::time_point<Clock,Duration> time);
    }
}
```

- std::this_thread::yield() підказує планувальнику потоків перепланувати виконання, призупинивши поточний потік і надавши перевагу іншим потокам. Точна поведінка цієї функції залежить від реалізації, зокрема від механіки використовуваного планувальника ОС та стану системи. Наприклад, планувальник реального часу first-in-first-out (SCHED_FIFO в Linux) припиняє поточний потік і поміщає його в кінець черги потоків з однаковим пріоритетом, готових до запуску (якщо немає інших потоків з таким же пріоритетом, yield не робить нічого).
- std::this_thread::get_id() працює аналогічно std::thread::get_id().
- std::this_thread::sleep_for(sleep_duration) блокує виконання поточного потоку на час sleep_duration.
- std::this_thread::sleep_until(wake_time) блокує виконання поточного потоку до моменту часу wake_time.

2.7 Вибір кількості потоків в ході виконання програми

Для того щоб визначити оптимальну кількість потоків для распаралелювання задачі можна скористатися наступним підходом:

```
template <typename Iterator, typename T>
T some_parallel_algorithm(Iterator first, Iterator last, T
initValue)
{
    unsigned long const length = std::distance(first, last);
    if (!length)
    {
        return initValue;
    }
    unsigned long const min_per_thread = 25;
    unsigned long const max_threads =
```

```
        (length + min_per_thread - 1) / min_per_thread;
    unsigned long const hardware_threads =
        std::thread::hardware_concurrency();
    unsigned long const num_threads = std::min(
        hardware_threads != 0 ? hardware_threads : 2,
        max_threads);
    unsigned long const block_size = length / num_threads;

    // process data using num_threads and block_size
}
```

Якщо вхідний діапазон порожній, повертається вихідне значення, зазначене як значення параметра init. В іншому випадку в діапазоні є хоча б один елемент, тому,

щоб отримати максимальну кількість потоків, кількість оброблюваних елементів можна розділити на мінімальний розмір блоку. Це зроблено для того, щоб не створювати 32 потоки на 32-ядерному комп'ютері, якщо в діапазоні тільки

п'ять значень.

Кількість потоків, що запускаються - це мінімальне значення з отриманого в результаті розрахунків максимуму та кількості апаратних потоків. Не слід запускати більше потоків, ніж може підтримувати наявне обладнання (призводячи до появи так званого перевищення лімітів), оскільки перемикання контексту при перевищенні кількості потоків спричинить зниження продуктивності. Якщо під час виклику std::thread::hardware_concurrency() повертається 0, кількість вибирається на ваш розсуд, в даному випадку вибрано 2. Запустити занадто багато потоків небажано, оскільки на одноядерному комп'ютері це сповільнить роботу.

Розділ 3. Взаємне виключення потоків (Mutual exclusion)

Уявіть, що ви живете в одній квартирі з приятелем. У вас одна кухня та одна ванна на двох. Зазвичай ванною не користуються одночасно кілька людей, і те, що сусід занадто довго хлюпається у воді, змушуючи вас чекати своєї черги, не може не дратувати. Можливо, одному з вас захочеться запекти в духовці ковбаски, у той час як у іншого там готуються кекси, і з цього теж нічого доброго не вийде. Ну і всім знайоме почуття досади, коли при спільно використовуваному устаткуванні ви на півдорозі до вирішення якогось завдання раптом виявляєте, що хтось взяв щось потрібне вам в даний момент або щось змінив, а ви розраховували, що все залишиться у колишньому стані або на своїх місцях.

Те саме відбувається і з потоками. Якщо вони спільно використовують дані, для них потрібні правила, що визначають, який потік і до яких даних може отримати доступ, коли і як будь-які оновлення даних будуть передаватися іншим потокам, що цікавляться цими даними. Некоректна робота із загальними даними – одна з основних причин помилок, пов'язаних із конкурентністю.

Коли справа доходить до спільної роботи з даними кількох потоків, всі проблеми виникають через наслідки зміни цих даних. Якщо всі дані, що спільно використовуються, доступні тільки для читання, проблем не буде, оскільки дані, що зчитуються одним потоком, не залежать від того, читає інший потік ті ж дані чи ні. Але якщо один або кілька потоків, що спільно використовують дані, починають вносити в них зміни, створюються серйозні передумови для виникнення проблем. У разі слід забезпечити прийнятність кінцевих результатів.

Припустимо, ви купуєте квиток у кіно. Якщо кінотеатр великий, квитки продаватимуть одразу кілька касирів, обслуговуючи одночасно кілька

людей. Якщо хтось у цей час купує квиток на той самий сеанс в іншій касі, то вибір місця залежить від того, хто першим його замовить, ви або інший. Якщо залишилося лише кілька місць, черговість може стати вирішальною: можлива справжня гонка за останніми квитками. Це приклад стану гонки: яке місце ви отримаєте і чи взагалі отримаєте, залежить від порядку двох покупок.

При конкурентності станом гонки є все, що залежить від порядку виконання операцій у двох і більше потоках щодо один одного: потоки беруть участь у гонці з виконання відповідних операцій. У стандарті C++ також визначається поняття гонки за даними, що позначає конкретний тип стану гонки, що виникає через одночасної зміни того самого об'єкта. Перегони за даними викликають небезпечну невизначену поведінку.

Помилки у стані гонки виникають, коли для завершення операції потрібно виконання кількох інструкцій процесора. Стан гонки найчастіше важко визначити та складно відтворити.

Є кілька способів, що дозволяють впоратися із проблемними станами перегонів. Найпростіший варіант — укласти структуру даних у механізм захисту, щоб гарантувати, що проміжні стани, в яких порушені інваріанти, буде видно лише потоку, який виконує зміни. З позиції інших потоків, що звертаються до цієї структури даних, такі зміни або ще не почнуться, або вже завершаться. Стандартна бібліотека C++ надає такі механізми.

3.1 std::mutex

Основним механізмом захисту спільно використовуваних даних, забезпеченим стандартом C++, є м'ютекс.

Отже, є спільно використовувана структура даних, наприклад список, і його потрібно захистити від стану гонки і можливих порушень інваріантів. Напевно, непогано було б отримати можливість помічати всі фрагменти коду, що звертаються до структури даних, як взаємовиключні, щоб при

виконанні одного з них будь-яким іншим потоком, який намагається отримати доступ до цієї структури даних, був би змушений чекати, поки перший потік не завершить виконання такого фрагмента. Тоді потік не міг би побачити порушений інваріант, крім тих випадків, коли він сам виконував модифікацію. Саме це буде отримано при використанні примітива синхронізації під назвою "м'ютекс", що означає взаємне виключення (mutual exclusion). Перед отриманням доступу до спільно використовуваної структури даних м’ютекс, пов'язаний з нею, блокується, а коли доступ до неї закінчується, блокування з нього знімається. Бібліотека потоків гарантує, що як тільки один потік заблокує певний м'ютекс, всі інші потоки, які намагаються його заблокувати, повинні будуть чекати, поки потік, який успішно заблокував м'ютекс, його не розблокує. Тим самим гарантується, що всі потоки бачать несуперечливе уявлення даних, що спільно використовуються, без порушених інваріантів. М'ютекси - головний механізм захисту даних, доступний в C++, але панацеєю від усіх бід їх не назвеш: важливо структурувати код таким чином, щоб захистити потрібні дані і уникнути станів гонки, властивих інтерфейсам. У мьютексів є й власні проблеми як взаємне блокування і захист або занадто великого, або занадто малого обсягу даних.

Клас std::mutex – це примітив синхронізації, який може використовуватись для захисту загальних даних від одночасного доступу кількох потоків.

```
class mutex
{
   mutex(mutex const&) = delete;
   mutex& operator=(mutex const&) = delete;

   constexpr mutex() noexcept;
   ~mutex();

   void lock();
   void unlock();
   bool try_lock();
};
```

std::mutex пропонує ексклюзивну, нерекурсивну семантику володіння:

- Потік володіє мьютексом з моменту успішного виклику методів lock або try_lock до unlock.
- Коли потік володіє мьютексом, решта потоків блокуються (при виклику lock) або отримують false (при виклику try_lock), якщо вони намагаються претендувати на володіння мьютексом.
- Потік виклику не повинен володіти мьютексом до виклику lock або try_lock. Інакше – невизначена поведінка і програма, наприклад, може потрапити в deadlock.
- Поведінка програми не визначена, якщо м'ютекс знищується, все ще заблокований, або якщо потік завершується, не розблокувавши м'ютекс.
- std::mutex не є копійованим, ні переміщуваним.
- М'ютекс повинен бути розблокований тим потоком виконання, який його заблокував, інакше поведінка не визначена.

std::mutex зазвичай не захоплюється безпосередньо, оскільки при цьому потрібно пам'ятати про необхідність виклику unlock() на всіх шляхах виходу з функції, у тому числі, що виникають через видачу винятків. Стандартною бібліотекою C++ надаються класи std::lock_guard, std::unique_lock або std::scoped_lock (починаючи з C++17) для безпечнішого управління захопленням мьютексів.

Приклад використання м'ютексу:

```
std::map<std::string, std::string> pages;
std::mutex mutex;
void save_page(const std::string& url)
{
    // simulate a long page fetch
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::string result = "some content";

    std::lock_guard<std::mutex> guard(mutex);
    pages[url] = result;
}
int main()
{
```

```
    std::thread t1(save_page, "https://foo");
    std::thread t2(save_page, "https://bar");
    t1.join();
    t2.join();

    // safe to access g_pages without lock now, as the threads
are joined
    for (const auto& pair : pages)
    {
        std::cout << pair.first << " => " << pair.second
                << '\n';
    }
}
```

У прикладі вище використовуються глобальні змінні структури даних і мьютексу. Іноді в такому використанні глобальних змінних є певний сенс, однак у більшості випадків м'ютекс і захищені дані поміщаються в один клас. Це відповідає стандартним правилам об'єктно-орієнтованого проектування: поміщення їх в один клас є ознакою пов'язаності один з одним, дозволяючи інкапсулювати функціональність та забезпечити захист. В даному випадку save_page стане методом класу, а м'ютекс і дані, що захищаються - закритими членами класу, що значно спростить визначення того, який код має доступ до даних і, отже, який код повинен заблокувати м'ютекс. Якщо всі методи класу блокують м'ютекс перед доступом до даних, що захищаються, і розблокують його після завершення доступу, дані будуть надійно захищені від будь-якого коду, що звертається до них. Однак, це не завжди так: якщо один з методів класу повертає вказівник або посилання на дані, що захищаються, то в захисті буде прорублена велика діра. Тепер звернутися до захищених даних і, можливо, їх змінити, не блокуючи м'ютекс, зможе будь-який код, який має доступ до цього покажчика або посилання. Тому захист даних за допомогою м'ютексу вимагає ретельного опрацювання інтерфейсу. Крім перевірки того, що методи не повертають покажчики або посилання коду, що їх викликає, важливо також переконатися, що вони не передають ці покажчики або посилання тим функціям, які викликаються ними і не контролюються вами.

Така передача не менш небезпечна: ці функції можуть зберігати покажчик або посилання там, де їх пізніше можна використовувати без захисту, що надається мьютексом. У цьому сенсі особливо небезпечні функції, які надаються під час виконання програми як аргументів чи іншим способом. На жаль, допомогти впоратися з проблемою такого роду бібліотека потоків C++ не в змозі, завдання блокування потрібного м'ютексу для захисту даних покладається на програміста. У той же час можна скористатися рекомендацією, яка допоможе в подібних випадках: не передавайте вказівники та посилання на захищені дані за межі блокування жодним способом: ні повертаючи їх з функції, ні зберігаючи в видимій пам'яті, ні передаючи в якості аргументів функцій, наданим користувачем.

Застосування м'ютексу або іншого механізму для захисту даних, що спільно використовуються, не дає повної гарантії захищеності від стану гонки. Розглянемо структуру даних стека. Нехай над нашим стеком можна проводити наступні операції: можна помістити в стек новий елемент методом push(), витягти елемент зі стека методом pop(), прочитати верхній елемент за допомогою top(), перевірити, чи стек не є порожнім, за допомогою empty(), та прочитати кількість елементів стеку методом size().

```
template<typename T, typename Container = std::deque<T>>
class stack
{
public:
   // constructors and other stuff

   bool empty() const;
   size_t size() const;
   T& top();
   T const& top() const;
   void push(T const&);
   void push(T&&);
   void pop();
};
```

Проблема в тому, що покладатись на результати роботи функцій empty() та size() не можна. Хоча на момент виклику вони, ймовірно, і були

достовірними, але після повернення з функції будь-який інший потік може звернутися до стеку та заштовхнути у нього нові елементи (push()), або виштовхнути існуючі (pop()), причому до того, як потік, що викликає empty() або size(), зможе скористатися цією інформацією.

Зокрема, якщо екземпляр стека не є спільно використовуваним, то за допомогою наступного коду цілком безпечно перевірити його на порожнечу методом empty(), а потім викликати top() для доступу до верхнього елемента:

```
stack<int> s;
if (!s.empty())
{
   int const value = s.top();
   s.pop();
   do_something(value);
}
```

Для однопоточного коду такий варіант не тільки безпечний, а й цілком очікуваний, оскільки виклик top() при порожньому стеку викликає невизначену поведінку.

При спільно використовуваному об'єкті стека ця послідовність викликів перестає бути безпечною через можливість виклику pop() з іншого потоку, в результаті чого між викликами empty() і top() видаляється останній елемент.

Отже, це класичний стан гонки, що не усувається внутрішнім використанням мьютексу для захисту вмісту стека і є наслідком застосування цього інтерфейсу.

Для того щоб уникнути таких проблем потрібно ретельно продумувати конктрукцію інтерфейсу потокобезпечних структур даних.

3.2 std::timed_mutex

Клас timed_mutex - це примітив синхронізації, який можна використовувати для захисту загальних даних від одночасного доступу кількох потоків.

Подібно до м'ютексу, timed_mutex пропонує ексклюзивну, нерекурсивну семантику володіння. Крім того, timed_mutex дозволяє спробувати захопити timed_mutex з таймаутом за допомогою методів try_lock_for() і try_lock_until().

```
class timed_mutex
{
public:
    timed_mutex();
    timed_mutex(timed_mutex const&) = delete;
    timed_mutex& operator=(timed_mutex const&) = delete;
    ~timed_mutex();

    void lock();
    void unlock();
    bool try_lock();

    template <typename Rep, typename Period>
    bool try_lock_for(
        std::chrono::duration<Rep, Period> const&
              timeout_duration);
    template <typename Clock, typename Duration>
    bool try_lock_until(
        std::chrono::time_point<Clock, Duration> const&
              timeout_time);
};
```

Метод try_lock_for():

- Намагається заблокувати м'ютекс. Потік очікує доти, доки не закінчиться зазначений час очікування або не буде отримано блокування, залежно від того, що настане раніше. При успішному отриманні блокування повертає true, інакше повертає false.
- Якщо timeout_duration менше або дорівнює timeout_duration.zero(), то функція поводиться як try_lock().
- Ця функція може блокувати потік довше, ніж timeout_duration через затримки в роботі планувальника або конкуренції за ресурси між

потоками.

- Стандарт рекомендує використовувати steady_clock для вимірювання тривалості. Якщо натомість реалізація використовує system_clock, час очікування також може бути чутливим до коригування годинника.
- Якщо try_lock_for викликається потоком, який уже володіє м'ютексом, поведінка не визначена.

Метод try_lock_until() працює так само, як try_lock_for(), але приймає std::chrono::time_point як аргумент. Якщо timeout_time вже минув, ця функція поводиться як try_lock().

3.3 RAII механізми для блокування м’ютексів

Не рекомендується використовувати клас std::mutex безпосередньо, тому що потрібно пам'ятати про виклик unlock на всіх шляхах виконання функції, у тому числі тих, які завершуються кидком виключення. Тобто, якщо між викликами lock і unlock буде згенеровано виняток, а ви цього не передбачте, то м'ютекс не звільниться, а заблоковані потоки так і залишаться чекати. Проблема безпеки блокувань мьютексів у C++ threading library вирішена досить звичайним для C++ способом - застосуванням техніки RAII.

3.3.1 std::lock_guard

```
template <class Mutex>
class lock_guard
{
public:
    typedef Mutex mutex_type;
    explicit lock_guard(mutex_type& m);
    lock_guard(mutex_type& m, adopt_lock_t);
    ~lock_guard();
    lock_guard(lock_guard const&) = delete;
    lock_guard& operator=(lock_guard const&) = delete;
};
```

Це простий клас, конструктор якого викликає метод lock для заданого об'єкта, а деструктор викликає unlock. Також у конструктор класу std::lock_guard можна передати аргумент std::adopt_lock - індикатор, що

означає, що mutex вже заблоковано і блокувати його не треба. std::lock_guard не містить жодних інших методів, і його не можна копіювати або переносити.

```
int i = 0;
std::mutex mutex;  // protects i

void safe_increment()
{
    const std::lock_guard<std::mutex> lock(mutex);
    ++i;
    std::cout << "i: " << i << "; in thread #"
              << std::this_thread::get_id() << '\n';

    // mutex is automatically released
    // when lock goes out of scope
}
```

lock_guard потрібен для зворотньої сумісності, а також є хороший вибором, коли потрібно заблокувати лише один м’ютекс на всій області видимості. У новому коді з підтримкою C++17 компілятором радять надавати перевагу scoped_lock.

3.3.2 std::unique_lock

Клас unique_lock - це універсальна оболонка володіння м'ютексом, що надає можливість відкладеного блокування, обмежені за часом спроби блокування, рекурсивне блокування, передачу володіння блокуванням та використання з condition variables.

Обмежені за часом спроби блокування працюють так само, як у класі std::timed_mutex. Для цього пов'язаний м'ютекс має бути TimedLockable.

Якщо передача володіння блокуванням або інші дії, що вимагають std::unique_lock, не передбачаються, краще скористатися класом std::scoped_lock з C++17.

Визначення класу std::unique_lock

```
template <class Mutex>
class unique_lock
{
```

```
public:
   typedef Mutex mutex_type;

   unique_lock() noexcept;
   explicit unique_lock(mutex_type& m);
   unique_lock(mutex_type& m, adopt_lock_t);
   unique_lock(mutex_type& m, defer_lock_t) noexcept;
   unique_lock(mutex_type& m, try_to_lock_t);

   template <typename Clock, typename Duration>
   unique_lock(
      mutex_type& m,
      std::chrono::time_point<Clock, Duration> const&
absolute_time);

   template <typename Rep, typename Period>
   unique_lock(
      mutex_type& m,
      std::chrono::duration<Rep, Period> const&
relative_time);

   ~unique_lock();

   unique_lock(unique_lock const&) = delete;
   unique_lock& operator=(unique_lock const&) = delete;

   unique_lock(unique_lock&&);
   unique_lock& operator=(unique_lock&&);

   void swap(unique_lock& other) noexcept;

   void lock();
   bool try_lock();

   template <typename Rep, typename Period>
   bool try_lock_for(
      std::chrono::duration<Rep, Period> const&
relative_time);

   template <typename Clock, typename Duration>
   bool try_lock_until(
      std::chrono::time_point<Clock, Duration> const&
absolute_time);

   void unlock();

   explicit operator bool() const noexcept;
   bool owns_lock() const noexcept;
   Mutex* mutex() const noexcept;
   Mutex* release() noexcept;
};
```

Відкладене блокування

Клас std::unique_lock забезпечує трохи більш гнучкий підхід, порівняно з std::lock_guard: екземпляр std::unique_lock не завжди володіє пов'язаним із ним мьютексом. Конструктору як другий аргумент можна передавати не тільки об'єкт std::adopt_lock, що змушує об'єкт блокування керувати блокуванням мьютексу, але і об'єкт відстрочки блокування std::defer_lock, що показує, що м'ютекс при конструюванні повинен залишатися розблокованим. Блокування можна встановити пізніше, викликавши функцію lock() для об'єкта std::unique_lock (але не мьютекса) або передавши об'єкт std::unique_lock функції std::lock().

std::unique_lock займає трохи більше пам'яті і працює дещо повільніше, ніж std::lock_guard. За гнучкість, що полягає в дозволі екземпляру std::unique_lock не володіти мьютексом, доводиться розплачуватися тим, що інформація про стан повинна зберігатися, оновлюватися і перевірятися: якщо екземпляр дійсно володіє мьютексом, деструктор повинен викликати функцію unlock(), інакше — не повинен. Цей прапор можна запросити, викликавши метод owns_lock().

Приклад використання std::unique_lock:

```
struct Box
{
   explicit Box(int num) : num_things{num} {}

   int num_things;
   std::mutex m;
};

void transfer(Box& from, Box& to, int num)
{
   // don't actually take the locks yet
   std::unique_lock<std::mutex> lock1(from.m,
      std::defer_lock);
   std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);

   // lock both unique_locks without deadlock
   std::lock(lock1, lock2);

   from.num_things -= num;
```

```
    to.num_things += num;

    // 'from.m' and 'to.m' mutexes unlocked in 'unique_lock'
dtors
}

int main()
{
    Box acc1(100);
    Box acc2(50);

    std::thread t1(transfer, std::ref(acc1), std::ref(acc2),
10);
    std::thread t2(transfer, std::ref(acc2), std::ref(acc1),
5);

    t1.join();
    t2.join();
}
```

Рекурсивне блокування

std::unique_lock можна використовувати з м'ютексами, що підтримують рекурсивне блокування. Це не означає, що для того самого unique_lock можна кілька разів викликати метод lock(). Це означає, що в одному потоці кілька різних екземплярів std::unique_lock можуть викликати метод lock() для одного і того ж м'ютексу. Повторний виклик методу lock() для одного і того ж екземпляра std::unique_lock призводить до виключення. Детальніше про роботу рекурсивних м'ютексів буде написано далі.

Передача володіння блокуванням

Об'єкти std::unique_lock переміщуються. Володіння м'ютексом може передаватися між екземплярами std::unique_lock шляхом переміщення. У деяких випадках, наприклад, при поверненні екземпляра з функції воно відбувається автоматично, а в інших випадках його необхідно виконувати явним чином викликом функції std::move(). По суті, все залежить від того, чи є джерело l-значенням реальною змінною або посиланням на таку або r-значенням якимось тимчасовим об'єктом. Володіння передається автоматично, якщо джерело є r-значенням, або має передаватися явним

чином, якщо він є l-значенням, щоб уникнути випадкової передачі володіння за межі змінної.

Один з варіантів можливого використання полягає в дозволі функції заблокувати м'ютекс, а потім передати володіння цим блокуванням коду, що викликає, який згодом зможе виконати додаткові дії під захистом цієї ж самої блокування. Відповідний приклад показаний у наступному фрагменті коду, де функція get_lock() блокує м'ютекс, а потім готує дані перед тим, як повернути блокування коду, що викликає:

```
std::unique_lock<std::mutex> get_lock() {
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk;
}
void process_data() {
    std::unique_lock<std::mutex> lk(get_lock());
    do_something();
}
```

Використання з condition variables

Детальніше про умовні змінні буде згодом, а поки що коротко ознайомимось як з ними працювати:

- Має бути хоча б один потік, який чекає, поки якась умова змінна стане істинною. Потік, що очікує, повинен спочатку виконати блокування за допомогою unique_lock. Це блокування передається методу wait(), який звільняє м'ютекс і зупиняє потік, доки не буде отримано сигнал від умовної змінної. Коли це станеться, потік прокинеться і м'ютекс знову заблокується.
- Має бути хоча б один потік, що сигналізує про те, що умова стала істинною. Сигнал може бути надісланий за допомогою notify_one(), при цьому буде розблокований один (будь-який) потік з очікуваючих, або notify_all(), що розблокує всі потоки, що очікують.
- У зв'язку з деякими складнощами при створенні пробуджуючої умови,

можуть відбуватися помилкові пробудження (**spurious wakeup**). Це означає, що потік може бути пробуджений навіть якщо ніхто не сигналізував умовної змінної. Тому необхідно перевіряти, чи вірна умова пробудження вже після того, як потік був пробуджений.

Приклад використання unique_lock з condition variable:

```
std::vector<int> data;
std::condition_variable data_cond;
std::mutex m;

void thread_func1()
{
    std::unique_lock<std::mutex> lock(m);
    data.push_back(10);
    data_cond.notify_one();
}

void thread_func2()
{
    std::unique_lock<std::mutex> lock(m);
    data_cond.wait(lock, [] {
        return !data.empty();
    });
    std::cout << data.back() << std::endl;
}

int main()
{
    std::thread th1(thread_func1);
    std::thread th2(thread_func2);
    th1.join();
    th2.join();
}
```

3.4 Рекурсиве блокування м’ютексу

Спроба потоку заблокувати м'ютекс, яким він уже володіє, приводить при використанні std::mutex до помилки та невизначеної поведінки. Але часом буває потрібно, щоб потік багаторазово отримував той самий м'ютекс, не розблоковуючи його попередньо. Для цього в стандартній бібліотеці C++ передбачений клас std::recursive_mutex. Він працює так само, як і std::mutex, за тим лише винятком, що на один його екземпляр можна з одного потоку

отримати кілька блокувань. Перш ніж м'ютекс зможе бути заблокований іншим потоком, потрібно буде зняти всі раніше встановлені блокування, тому якщо функція lock() викликається три рази, то три рази повинна бути викликана і функція unlock(). При правильному застосуванні std::lock_guard та std::unique_lock все це буде зроблено за вас автоматично.

3.4.1 std::recursive_mutex

recursive_mutex пропонує ексклюзивну рекурсивну семантику володіння:

- Потік володіє recursive_mutex протягом періоду часу, який починається, коли він успішно викликає або lock, або try_lock. Протягом цього періоду потік може здійснювати додаткові дзвінки lock або try_lock. Період володіння закінчується, коли потік робить відповідну кількість дзвінків unlock.
- Коли потік володіє recursive_mutex, решта потоків чекатиме (для lock) або отримувати false (для try_lock), якщо вони спробують захопити recursive_mutex.
- Максимальна кількість разів, яку recursive_mutex може бути заблоковано, у стандарті не вказано, але після досягнення цього числа виклики lock будуть кидати std::system_error, а виклики try_lock повертатимуть false.

Поведінка програми не визначена, якщо recursive_mutex знищується, все ще заблокований.

Один з варіантів використання recursive_mutex - це захист спільного стану в класі, функції-члени якого можуть викликати одна одну. Приклад:

```
class X
{
   std::recursive_mutex m;
   std::string shared;
public:
   void fun1()
   {
      std::lock_guard<std::recursive_mutex> lk(m);
      shared = "fun1";
```

```
        std::cout << "in fun1, shared variable is now " <<
shared << '\n';
    }

    void fun2()
    {
        std::lock_guard<std::recursive_mutex> lk(m);
        shared = "fun2";
        std::cout << "in fun2, shared variable is now "
                  << shared << '\n';
        fun1(); // recursive lock becomes useful here
        std::cout << "back in fun2, shared variable is "
                  << shared <<'\n';
    }
};

int main()
{
    X x;
    std::thread t1(&X::fun1, &x);
    std::thread t2(&X::fun2, &x);
    t1.join();
    t2.join();
}

/*
Possible output:
in fun1, shared variable is now fun1
in fun2, shared variable is now fun2
in fun1, shared variable is now fun1
back in fun2, shared variable is fun1
*/
```

3.4.2 std::recursive_timed_mutex

std::recursive_timed_mutex працює аналогічно тому, як працює std::timed_mutex, але надає можливість багаторазового блокування одного мьютексу в одному потоці, як std::recursive_mutex.

3.5 М'ютекси читання-запису

Якщо ми проводимо тільки читання даних, то гонки даних не виникає. Однак, якщо ми хочемо змінювати дані, ми змушені захищати їх від одночасного доступу. Але що робити, якщо більшу частину часу структура даних використовується тільки для читання, а захисту ми потребуємо тільки при рідкісних оновленнях цієї структури. Блокувати потоки при кожному

читанні без потреби не хотілося б, тому що від цього постраждає продуктивність. Тому застосування std::mutex для захисту такої структури даних має похмурі перспективи, оскільки при цьому виключається можливість реалізувати конкурентність при читанні структури даних у той період, коли вона не піддається модифікації, тому потрібен інший вид мьютексу. Цей інший тип м'ютексу зазвичай називають м'ютексом читання - записи, оскільки він допускає два різні типи використання: монопольний доступ для одного потоку запису або загальний одночасний доступ для декількох потоків читання. Стандартна бібліотека C++17 надає два повністю готові м'ютекси такого виду, std::shared_mutex і std::shared_timed_mutex.

Для операцій запису можна використовувати std::lock_guard<std::shared_mutex> і std::unique_lock<std::shared_mutex>. Вони забезпечують монопольний доступ, як і під час використання std::mutex. У потоках, яким не потрібно оновлювати структуру даних, для отримання спільного доступу натомість можна скористатися std::shared_lock<std::shared_mutex>. Цей шаблон класу RAII був доданий в C++14 і застосовується так само, як і std::unique_lock, за винятком того, що кілька потоків можуть одночасно отримати загальне блокування на той самий м'ютекс std::shared_mutex. Обмеження полягає в тому, що якщо будь-який має shared блокування потік спробує отримати монопольне блокування, він буде чекати до тих пір, поки всі інші потоки не знімуть свої блокування. Аналогічно, якщо який-небудь потік має монопольне блокування, ніякий інший потік не може отримати shared або монопольне блокування, поки не зніме блокування перший потік.

3.5.1 std::shared_mutex (C++17)

Визначення класу:

```
class shared_mutex
{
```

```
public:
    shared_mutex(shared_mutex const&)=delete;
    shared_mutex& operator=(shared_mutex const&)=delete;

    shared_mutex() noexcept;
    ~shared_mutex();

    void lock();
    void unlock();
    bool try_lock();

    void lock_shared();
    void unlock_shared();
    bool try_lock_shared();
};
```

Клас shared_mutex – це примітив синхронізації, який може використовуватись для захисту спільних даних від одночасного доступу кількох потоків. На відміну від інших типів мьютексів, які забезпечують ексклюзивний доступ, shared_mutex має два рівні доступу:

- спільний доступ - кілька потоків можуть спільно володіти одним і тим самим мьютексом.
- ексклюзивний доступ (виключне блокування) - лише один потік може мати мьютекс.

У межах одного потоку одночасно може бути отримане лише одне блокування (загальне або ексклюзивне).

Правила блокування shared_mutex:

- Якщо один потік отримав ексклюзивний доступ (через lock, try_lock), ніякі інші потоки не можуть отримати блокування (включаючи спільне).
- Якщо один потік отримав спільне блокування (через lock_shared, try_lock_shared), жоден інший потік не може отримати ексклюзивне блокування, але може отримати спільне блокування.
- Тільки якщо виняткове блокування не було отримано жодним потоком, загальне блокування може бути отримано кількома потоками.

shared_mutex особливо корисні, коли дані у спільному використанні можуть бути безпечно зчитані будь-якою кількістю потоків одночасно, але потік може перезаписувати дані лише тоді, коли жоден інший потік не читає і не записує в цей час.

Приклад використання:

```
class threadsafe_counter
{
public:
   threadsafe_counter() = default;

   // Multiple threads can read the counter's value at the
      same time.
   unsigned int get() const
   {
      std::shared_lock lock(mutex);
      return value;
   }

   // Only one thread/writer can increment/write the counter's
      value.
   unsigned int increment()
   {
      std::lock_guard lock(mutex);
      return ++value;
   }

   // Only one thread/writer can reset/write the counter's
      value.
   void reset()
   {
      std::lock_guard lock(mutex);
      value = 0;
   }

private:
   mutable std::shared_mutex mutex;
   unsigned int value = 0;
};
```

3.5.2 std::shared_timed_mutex (C++14)

std::shared_timed_mutex пропонує таку саму семантику володіння мьютексом, як std::shared_mutex.

Крім того, std::shared_timed_mutex подібно до timed_mutex надає

можливість спробувати претендувати на володіння shared_timed_mutex з таймаутом за допомогою методів try_lock_for(), try_lock_until(), try_lock_shared_for(), try_lock_shared_until().

3.5.3 std::shared_lock (C++14)

Клас shared_lock це аналог std::unique_lock для отримання спільного доступу до даних, що захищаються за допомогою shared_mutex. Він дозволяє відкладене блокування, спробу блокування з таймаутом та передачу права володіння блокуванням. Блокування shared_lock блокує shared_mutex у спільному режимі (щоб заблокувати його в ексклюзивному режимі, можна використовувати std:: unique_lock).

Для роботи з умовними змінними можна використати std::condition_variable_any (std::condition_variable вимагає std::unique_lock і тому підтримує лише виняткове володіння).

3.6 Захоплення декількох м'ютексів одночасно

При великій глибині деталізації блокування для будь-якої операції може бути необхідно заблокувати два або більше м'ютекси. При цьому може виникнути ще одна проблема – взаємне блокування. При взаємному блокуванні один потік чекає завершення виконання операції іншим, тому жоден із потоків не виконує роботи.

Не важко уявити ситуацію коли пара потоків потребує блокування декількох м’ютексів, кожен поток має один заблокований м'ютекс і він очікує розблокування іншого. Продовжити виконання не може жоден із потоків, оскільки кожен чекає, коли інший розблокує свій м'ютекс. Такий сценарій називається взаємним блокуванням і є серйозною проблемою при необхідності заблокувати для виконання однієї операції два м'ютекси і більше.

Загальна порада з обходу взаємного блокування полягає в постійному блокуванні двох м'ютексів в тому самому порядку: якщо завжди блокувати

м'ютекс А перед блокуванням м'ютексу Б, то взаємного блокування ніколи не відбудеться. Іноді цю умову виконати нескладно, оскільки м'ютекси служать різним цілям, але іноді все набагато складніше, наприклад, коли кожен з м'ютексів захищає окремий екземпляр одного й того ж класу.

3.6.1 std::lock

Розглянемо приклад, у якому якась функція виконує дію над двома об'єктами одного класу. Щоб забезпечити коректну роботу і при цьому уникнути впливу змін, що вносяться у режимі конкурентності, слід заблокувати м'ютекси на обох примірниках. Але якщо вибрати певний порядок, наприклад спочатку блокувати м'ютекс для екземпляра, переданого як перший параметр, а потім м'ютекс для екземпляра, переданого як другий параметр, то можна отримати зворотний ефект: варто всього іншого потоку викликати функцію з переставленими місцями параметрами, і ви отримайте взаємне блокування. У стандартній бібліотеці C++ є засіб від цього у вигляді std::lock — функції, здатної одночасно заблокувати два і більше м'ютекси, не ризикуючи викликати взаємне блокування.

```
struct Employee
{
   Employee(std::string id) : id(id) {}
   std::string id;
   std::vector<std::string> lunch_partners;
   std::mutex m;

   std::string output() const
   {
      std::string ret = "Employee " + id + " has lunch
            partners: ";
      for (const auto& partner : lunch_partners)
         ret += partner + " ";
      return ret;
   }
};

void send_mail(Employee&, Employee&)
{
   // simulate a time-consuming messaging operation
```

```
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

void assign_lunch_partner(Employee& e1, Employee& e2)
{
    static std::mutex io_mutex;
    {
        std::lock_guard<std::mutex> lk(io_mutex);
        std::cout << e1.id << " and " << e2.id
                  << " are waiting for locks" << std::endl;
    }

    // use std::lock to acquire two locks without worrying
        about
    // other calls to assign_lunch_partner deadlocking us
    {
        std::lock(e1.m, e2.m);
        std::lock_guard<std::mutex> lk1(e1.m, std::adopt_lock);
        std::lock_guard<std::mutex> lk2(e2.m, std::adopt_lock);
        // Equivalent code (if unique_locks are needed, e.g. for
               condition variables)
        // std::unique_lock<std::mutex> lk1(e1.m,
               std::defer_lock);
        // std::unique_lock<std::mutex> lk2(e2.m,
               std::defer_lock);
        //        std::lock(lk1, lk2);
        // Superior solution available in C++17
        // std::scoped_lock lk(e1.m, e2.m);
        {
            std::lock_guard<std::mutex> lk(io_mutex);
            std::cout << e1.id << " and " << e2.id <<
                " got locks" << std::endl;
        }
        e1.lunch_partners.push_back(e2.id);
        e2.lunch_partners.push_back(e1.id);
    }
    send_mail(e1, e2);
    send_mail(e2, e1);
}

int main()
{
    Employee alice("alice"), bob("bob"),
        christina("christina"), dave("dave");

    // assign in parallel threads because mailing users about
        lunch assignments
    // takes a long time
    std::vector<std::thread> threads;
    threads.emplace_back(assign_lunch_partner,
```

```
        std::ref(alice), std::ref(bob));
    threads.emplace_back(assign_lunch_partner,
        std::ref(christina), std::ref(bob));
    threads.emplace_back(assign_lunch_partner,
        std::ref(christina), std::ref(alice));
    threads.emplace_back(assign_lunch_partner,
        std::ref(dave), std::ref(bob));

    for (auto& thread : threads) thread.join();
    std::cout << alice.output() << '\n' << bob.output() << '\n'
       << christina.output() << '\n' << dave.output() << '\n';
}
```

Коректне розблокування мьютексів при виході з функції в цьому прикладі забезпечується за допомогою std::lock_guard. На додаток до м'ютексу надається параметр std::adopt_lock, щоб показати об'єктам std::lock_guard, що м'ютекси вже заблоковані. Об'єкти повинні опанувати існуюче блокування м'ютексу, а не намагатися заблокувати його в конструкторі. Слід також зазначити, що блокування одного з м'ютексів усередині виклику std::lock може призвести до видачі виключення, у такому разі виняток поширюється із std::lock. Якщо функцією std::lock успішно заблоковано один м'ютекс, а виняток видано при спробі заблокувати інший, перший м'ютекс розблокується автоматично: щодо блокування наданих м'ютексів функція std::lock забезпечує семантику «все або нічого».

Застосування std::lock дозволяє позбавитися взаємних блокувань, коли потрібно заволодіти відразу двома і більше блокуваннями, проте воно не допоможе, якщо блокування захоплюються роз'єднано. У разі, щоб гарантувати обхід взаємних блокувань, розробникам доводиться покладатися на самодисципліну. А це не так просто: взаємоблокування відносяться до однієї з найнеприємніших проблем, з якою доводиться стикатися в багатопотоковому коді, їх виникнення часто неможливо передбачити, оскільки в більшості випадків все працює нормально. Проте існує ряд відносно простих правил, що допомагають створювати код, не схильний до взаємних блокувань.

Всі рекомендації щодо обходу взаємних блокувань зводяться до одного: не чекати завершення операції іншим потоком, якщо є ймовірність, що він також чекає завершення операції поточним потоком:

- Уникайте вкладених блокувань. Не встановлюйте блокування, якщо вже є якесь блокування.
- Утримуючи блокування виклику, уникайте коду, наданого користувачем. Якщо при утриманні блокування викликати код користувача, що встановлює блокування, виявиться порушена рекомендація, що наказує уникати вкладених блокувань, і може виникнути взаємне блокування.
- Встановлюйте блокування у фіксованому порядку. Якщо є потреба встановити дві і більше блокування, але в рамках єдиної операції за допомогою std::lock це неможливо, найкраще, що можна зробити, - встановити їх у кожному потоці в тому самому порядку.
- Використовуйте ієрархію блокування. Будучи окремим випадком визначення порядку блокувань, ієрархія блокувань дозволяє забезпечити засіб перевірки дотримання угоди під час виконання програми. Таку перевірку можна провести під час виконання програми, призначивши номери рівнів кожному м'ютексу та зберігши записи про те, які м'ютекси заблоковані кожним потоком. Цей шаблон отримав дуже широке поширення, але його пряма підтримка в стандартній бібліотеці C++ не забезпечується, тому потрібно створити власний тип м'ютексу hierarchical_mutex.

3.6.2 std::try_lock

Аналог std::lock для спроби блокування кількох м'ютексів. try_lock не призведе до взаємного блокування, навіть якщо не буде певного порядку блокування. Тому він намагається заблокувати кожен з переданих об'єктів lock_1, lock_2, ..., lock_n, викликаючи їх метод try_lock в тому порядку, в

якому вони передані.

Якщо виклик try_lock для якогось аргументу завершується невдало, подальші виклики try_lock не виконуються, а викликається unlock для всіх заблокованих об'єктів та повертається індекс об'єкта, який не вдалося заблокувати, починаючи з 0.

Якщо виклик try_lock для будь-якого аргументу призводить до виключення, викликається unlock для всіх заблокованих об'єктів перед прокиданням виключення нагору.

Значення, що повертається: -1 при успішному виконанні або 0-based індекс об'єкта, який не вдалося заблокувати.

3.6.3 std::scoped_lock (C++17)

C++17 надається спосіб блокування декількох мьютексів одночасно у вигляді нового RAII-шаблону std::scoped_lock<>. Він практично еквівалентний std::lock_guard<>, за винятком того, що є варіативним шаблоном, що приймає як параметри - список типів мьютексів, а як аргументи конструктора — список мьютексів. Надані конструктору м'ютекси блокуються з використанням такого ж алгоритму, як і в std::lock, і коли конструктор завершує роботу, вони виявляються заблокованими, а потім розблокуються в деструкторі.

3.7 Одноразовий виклик функції за допомогою std::call_once та std::once_flag

Припустимо, що є спільно використовуваний ресурс, створення якого настільки затратно, що займатися цим хочеться лише в крайній необхідності, коли користувач звернувся до цього ресурсу: можливо, він відкриває підключення до бази даних або виділяє великий обсяг пам'яті. Подібна відкладена (або лінива) ініціалізація (lazy initialization) досить часто зустрічається в однопотоковому коді — кожна операція, яка потребує ресурсу, спочатку перевіряє, чи він ініціалізований, і, якщо не був, перш ніж

скористатися цим ресурсом, ініціалізує його. Якщо спільно використовуваний ресурс безпечний при отриманні до нього конкурентного доступу, єдиною частиною, яка потребує захисту під час перетворення коду на багатопотоковий, є ініціалізація. Можна було б захистити ініціалізацію мьютексом у багатопотоковому додатку:

```
struct some_resource
{
   void do_something() {}
};

std::shared_ptr<some_resource> resource_ptr;
std::mutex resource_mutex;

void foo()
{
   std::unique_lock<std::mutex> lk(resource_mutex);
   if (!resource_ptr)
   {
      resource_ptr.reset(new some_resource);
   }
   lk.unlock();
   resource_ptr->do_something();
}

int main()
{
   foo();
}
```

Але це може призвести до непотрібного блокування потоків, що використовують ресурс. Причина в тому, що кожен потік буде змушений очікувати на розблокування м'ютексу, щоб перевірити, чи ресурс уже не був ініціалізований. Ця проблема настільки поширена, що багато хто намагався придумати більш відповідний спосіб вирішення даного завдання, включаючи відомий шаблон блокування з подвійною перевіркою: спочатку вказівник зчитується без отримання блокування, яке встановлюється, тільки якщо він має значення NULL. Після отримання блокування покажчик перевіряється ще раз на той випадок, якщо між першою перевіркою та отриманням блокування даним потоком ініціалізація була виконана

якимось іншим потоком:

```
void double_checked_locking()
{
   if (!resource_ptr)
   {
      std::lock_guard<std::mutex> lk(resource_mutex);
      if (!resource_ptr)
      {
         resource_ptr.reset(new some_resource);
      }
   }
   resource_ptr->do_something();
}
```

Щоб справитися з цією ситуацією, стандартна бібліотека C++ надає компоненти std::once_flag та std::call_once. Замість блокування м'ютексу та явної перевірки вказівника кожен потік може безпечно скористатися функцією std::call_once, знаючи, що до моменту повернення управління з цієї функції покажчик буде ініціалізований будь-яким потоком. Необхідні для цього дані синхронізації зберігаються в екземплярі std::once_flag, і кожен екземпляр std::once_flag відповідає іншій ініціалізації. Задіяння функції std::call_once зазвичай пов'язані з меншими витратами проти явним використанням мьютексу, особливо коли ініціалізація вже було виконано. Тому перевагу слід надавати саме їй. Приклад вище можна було б змінити так:

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;

void init_resource()
{
   resource_ptr.reset(new some_resource);
}

void foo()
{
   std::call_once(resource_flag, init_resource);
   resource_ptr->do_something();
}
```

Один із сценаріїв, що передбачає ймовірність стану гонки під час ініціалізації, до C++11 був пов'язаний із застосуванням локальної змінної,

оголошеної з ключовим словом static. Ініціалізація такої змінної визначена так, щоб вона виконувалася при першому проходженні потоку керування через оголошення. Це означало, що кілька потоків, що викликають функцію, у прагненні першими виконати визначення могли спричинити стан гонки. На багатьох компіляторах, що передували C++11, це створювало реальні проблеми, оскільки почати ініціалізацію могли відразу кілька потоків або вони могли намагатися використовувати під час ініціалізації, запущеної в іншому потоці. У C++11 ця проблема була вирішена: ініціалізація визначена так, що виконується тільки в одному потоці, і жодні інші потоки не будуть продовжувати виконання доти, доки ця ініціалізація не буде завершена. Коли потрібна лише одна глобальна змінна, цією властивістю можна скористатися як альтернатива std::call_once:

```
class MyClass;

MyClass& get_instance()
{
    static MyClass instance;
    return instance;
}
```

Отже, std::call_once:

- Виконує об'єкт, що викликається f рівно один раз, навіть якщо він викликається одночасно з декількох потоків.
- Якщо на момент виклику call_once прапор вказує, що f вже викликано, call_once відразу ж завершується (пасивний виклик call_once).
- Інакше call_once викликає f: std::invoke(sdt::forward<_Fn>(_Fx), std::forward<_Args>(_Ax)...);. На відміну від конструктора std::thread або std::async, аргументи не переміщуються і копіюються, оскільки їх потрібно передавати в інший потік виконання. (Активний виклик call_once).
- Якщо виклик функції кидає виняток, він передається в call_once, і прапор

не встановлюється, щоб зробити інший виклик (exceptional виклик call_once).

- Якщо цей виклик функції завершився успішно (returning виклик call_once), прапорець встановлюється, і решта викликів call_once з тим же прапором будуть гарантовано пасивними.
- Всі активні виклики з тим самим прапором утворюють послідовність, що складається з нуля або більше exceptional викликів, за якими слідує один returning виклик.
- Якщо паралельні виклики call_once виконують різні функції f, то не визначено, яка функція f буде викликана. Функція, що виконується, виконується в тому ж потоці, що і call_once.
- Ініціалізація локальної статичної змінної гарантовано відбувається лише один раз, навіть при виклику з кількох потоків, і може бути ефективнішою, ніж еквівалентний код, що використовує std::call_once.

Розділ 4. Умовні змінні

Якщо якийсь потік очікує, доки інший потік завершить виконання свого завдання, є кілька варіантів розвитку подій. По-перше, перший потік може постійно перевіряти стан прапора спільно використовуваних даних, захищених мьютексом, а другий потік буде зобов'язаний встановити прапор після завершення свого завдання. Це дуже накладно з двох міркувань: постійно перевіряючи стан прапора, потік витрачає цінний процесорний час, а коли м'ютекс заблокований очікуваним потоком, його не можна заблокувати жодним іншим потоком. Другий варіант передбачає введення очікуваного потоку в режим сну на короткий проміжок часу між перевірками за допомогою функції std::this_thread::sleep_for(). Це вже набагато краще, оскільки потік, перебуваючи в режимі сну, не витрачає процесорний час марно, але хороший період перебування в ньому підібрати досить важко. Занадто короткий період сплячки між перевірками — і потік, як і раніше, витрачає час процесора на занадто часті перевірки, занадто довгий період сплячки — і потік вийде з неї пізніше, що призведе до непотрібної затримки. Третім і найкращим варіантом є використання засобів зі стандартної бібліотеки C++, призначених для очікування настання будь-якої події. Основним механізмом реалізації такого очікування є умовна змінна. Концептуально вона пов'язана з якоюсь умовою, і один або кілька потоків можуть очікувати на виконання цієї умови. Коли інший потік виявить, що умова виконана, він може сповістити про це один або кілька потоків, що чекають умовну змінну, щоб розбудити їх та дозволити продовжити роботу.

Стандартна бібліотека C++ надає не одну, а дві реалізації умовної змінної: std:: condition_variable та std:: condition_variable_any. Обидві вони оголошені у заголовку <condition_variable>. В обох випадках для відповідної синхронізації їм потрібно працювати з м'ютексом: перша реалізація обмежується роботою тільки зі std::mutex, а друга може

працювати з будь-якими типами, які працюють як м'ютекс, про що свідчить суфікс _any. Якщо не потрібна додаткова гнучкість, перевагу слід віддавати реалізації std::condition_variable.

4.1 std::condition_variable

Клас condition_variable — це примітив синхронізації, який може використовуватися для блокування потоку або кількох потоків, доки інший потік не змінить загальну змінну (не виконає умову) і не повідомить про це condition_variable.

Визначення класу:

```
class condition_variable
{
public:
   condition_variable();
   ~condition_variable();

   condition_variable(condition_variable const&) = delete;
   condition_variable& operator=(condition_variable const&) =
delete;

   void notify_one() noexcept;
   void notify_all() noexcept;

   void wait(std::unique_lock<std::mutex>& lock);

   template <typename Predicate>
   void wait(std::unique_lock<std::mutex>& lock,
      Predicate pred);

   template <typename Clock, typename Duration>
   std::cv_status wait_until(
      std::unique_lock<std::mutex>& lock,
      const std::chrono::time_point<Clock, Duration>&
            absolute_time);

   template <typename Clock, typename Duration, typename
Predicate>
   bool wait_until(
      std::unique_lock<std::mutex>& lock,
      const std::chrono::time_point<Clock, Duration>&
            absolute_time, Predicate pred);

   template <typename Rep, typename Period>
   std::cv_status wait_for(
```

```
        std::unique_lock<std::mutex>& lock,
        const std::chrono::duration<Rep, Period>&
               relative_time);

    template <typename Rep, typename Period,
        typename Predicate>
    bool wait_for(
        std::unique_lock<std::mutex>& lock,
        const std::chrono::duration<Rep, Period>& relative_time,
        Predicate pred);
};

void notify_all_at_thread_exit(condition_variable&,
                               std::unique_lock<std::mutex>);
```

Потік, який має намір змінити загальну змінну, повинен:

- захопити std::mutex (зазвичай через std::lock_guard)
- виконати модифікацію, поки утримується блокування мьютексу
- виконати notify_one або notify_all на std::condition_variable (блокування потрібно зняти перед повідомленням, щоб не виявилось, що потоки прокинулись, а м’ютекс все ще утримується потоком який викликав notify)

Навіть якщо загальна змінна є атомарною, все одно потрібно використовувати м'ютекс для коректного сповіщення потоків, що очікують. Будь-який потік, який очікує настання події від std::condition_variable, повинен:

- за допомогою std::unique_lock<std::mutex> отримати блокування того ж мьютексу, який використовується для захисту загальної змінної.
- перевірити, що потрібна умова ще не виконана.
- викликати метод wait, wait_for або wait_until. Операції очікування звільняють м'ютекс і зупиняють виконання потоку.
- коли отримано повідомлення, сплив тайм-аут або сталося помилкове пробудження, потік прокидається, і м'ютекс повторно блокується. Потім потік повинен перевірити, що умова дійсно виконана, і відновити очікування, якщо пробудження було помилковим.

Замість трьох останніх кроків можна скористатися перевантаженням методів wait, wait_for і wait_until, що приймає предикат для перевірки умови та виконує три останні кроки.

std::condition_variable працює тільки з std::unique_lock<std::mutex>. Це обмеження забезпечує максимальну ефективність на деяких платформах. std::condition_variable_any працює з будь-яким BasicLockable об'єктом, наприклад, з std::shared_lock.

Condition variables допускають одночасний виклик методів wait, wait_for, wait_until, notify_one та notify_all з різних потоків.

Приклад використання:

```
std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [] { return ready; });

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing
        completed\n";

    // Manual unlocking is done before notifying, to avoid
        waking up
    // the waiting thread only to block again (see notify_one
        for details)
    lk.unlock();
    cv.notify_one();
}

int main()
{
    std::thread worker(worker_thread);
```

```
    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for
processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, [] { return processed; });
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
/*
Possible output:
main() signals data ready for processing
Worker thread is processing data
Worker thread signals data processing completed
Back in main(), data = Example data after processing
*/
```

4.2 std::condition_variable_any

Цей тип умовної змінної має такий самий інтерфейс, як std::condition_variable, але може використовуватися не тільки з std::unique_lock<std::mutex>, а з будь-яким типом, що блокується. Працює повільніше, ніж std:: condition_variable. Може використовуватися, наприклад, для роботи зі std::shared_lock.

4.3 std::notify_all_at_thread_exit

Стандартна бібліотека надає ще одну функцію для використання в ситуаціях, коли за допомогою condition_variable хочемо дочекатися завершення потоку.

Навіщо це потрібно? Допустимо, ми хочемо дочекатися завершення detached потоку, у цьому випадку ми не можемо використовувати метод join для очікування завершення потоку. Тоді ми вирішуємо, що потрібно

використовувати condition_variable, щоб повідомити, що потік завершується. Але якщо ми просто в кінці функції, виконуваної в окремому потоці, додамо cv.notify_all(), то отримаємо поведінку відмінну від тієї, яка нам потрібна. Незважаючи на те, що ця команда буде останньою у функції потоку, потік на ній ще не закінчує виконання. Після виклику notify_all у цьому ж потоці відбуватиметься знищення thread_local змінних, будуть викликатися їхні деструктори та виконуватись будь-які дії. Тобто насправді повідомлення було надіслано ще до того, як потік завершився.

Тоді як насправді дочекатися повного завершення detached потоку? Для цього стандартна бібліотека надає функцію std::notify_all_at_thread_exit.

```
void notify_all_at_thread_exit( std::condition_variable& cond,
                                std::unique_lock<std::mutex> lk );
```

Вона чекає завершення потоку, у тому числі знищення thread_local змінних, і останніми діями в потоці виконує:

```
lk.unlock();
cv.notify_all();
```

Приклад використання:

```
std::mutex m;
std::condition_variable cv;
bool ready = false;
std::string result;

void thread_func()
{
    thread_local std::string thread_local_data = "42";

    std::unique_lock<std::mutex> lk(m);

    // assign a value to result using thread_local data
    result = thread_local_data;
    ready = true;

    std::notify_all_at_thread_exit(cv, std::move(lk));
}
// 1. destroy thread_locals;
// 2. unlock mutex;
// 3. notify cv.

int main()
```

```
{
    std::thread t(thread_func);
    t.detach();

    // do other work
    // ...

    // wait for the detached thread
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{ return ready; });

    // result is ready and thread_local destructors have
       finished, no UB
    assert(result == "42");
}
```

Розділ 5. Реалізація потокобезпечного патерну Singleton

5.1 Підхід з застосуванням блокувань

Припустимо, у вас є клас, який реалізує добре відомий шаблон Singleton, і ви хочете зробити його потокобезпечним. Очевидним підходом є забезпечення взаємної ексклюзивності шляхом додавання блокування. Таким чином, якщо два потоки одночасно викликають Singleton::getInstance, лише один з них створить екземпляр.

```
LockBasedSingleton* LockBasedSingleton::instance = nullptr;
std::mutex LockBasedSingleton::myMutex;

LockBasedSingleton& LockBasedSingleton::getInstance()
{
    std::lock_guard<std::mutex> myLock(myMutex);
    if (!instance)
    {
        instance = new LockBasedSingleton();
    }
    return *instance;
}
```

Недоліком цього рішення є те, що воно може бути дорогим. Кожен доступ до

Singleton вимагає блокування, але насправді це блокування потрібне лише один раз - під час ініціалізації instance. Якщо getInstance викликається n разів під час виконання програми, навіщо платити за n блокувань, коли ви знаєте, що n − 1 з них непотрібні?

5.2 Double-Checked Locking Pattern

DCLP створено, щоб запобігти цьому.

Однак це не так просто, як показує стаття Мейерса-Александреску “C++ and the Perils of Double-Checked Locking”. У цій статті автори описують кілька помилкових спроб реалізувати DCLP на C++, розбираючи кожну спробу, щоб пояснити, чому це небезпечно. Нарешті, на сторінці 12 вони показують безпечну реалізацію, але яка залежить від невизначених бар’єрів пам’яті,

що залежать від платформи.

```
Singleton* Singleton::getInstance() {
   Singleton* tmp = m_instance;
   ...                      // insert memory barrier
   if (tmp == NULL) {
      Lock lock;
      tmp = m_instance;
      if (tmp == NULL) {
         tmp = new Singleton;
         ...                // insert memory barrier
         m_instance = tmp;
      }
   }
   return tmp;
}
```

Тут ми бачимо, як double-checked locking pattern отримав свою назву: ми блокуємо м'ютекс лише тоді, коли вказівник m_instance має значення NULL, що серіалізує першу групу потоків, які випадково бачать це значення. Потрапивши в блокування, m_instance перевіряється вдруге, так що тільки перший потік створить синглтон.

Це дуже близько до робочої реалізації. Просто відсутній якийсь бар'єр пам'яті у виділених рядках. На той час, коли автори писали статтю, не було переносимої функції у C++, яка могла б заповнити пропуски. Тепер із C++11 є.

5.2.1 Acquire and release fences

Ви можете безпечно завершити наведену вище реалізацію, використовуючи acquire and release fences. Однак, щоб зробити цей код дійсно переносимим, ви також повинні загорнути m_instance в атомарний тип C++11 і маніпулювати ним за допомогою relaxed атомарних операцій. Ось отриманий код:

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance()
{
   Singleton* tmp =
m_instance.load(std::memory_order_relaxed);
```

```
    std::atomic_thread_fence(std::memory_order_acquire);
    if (tmp == nullptr)
    {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr)
        {
            tmp = new Singleton;
            std::atomic_thread_fence(std::memory_order_release);
            m_instance.store(tmp, std::memory_order_relaxed);
        }
    }
    return tmp;
}
```

Це працює надійно навіть у багатоядерних системах, оскільки memory fences встановлюють *synchronizes-with* зв'язок між потоком, який створює синглтон, і будь-яким наступним потоком, який пропускає блокування.

Acquire and release fences можуть правильно реалізувати DCLP і повинні мати можливість генерувати оптимальний машинний код на більшості сучасних багатоядерних пристроїв, але вони не вважаються дуже модними.

5.2.2 Атомарні операції з обмеженнями впорядкування пам'яті

Переважний спосіб досягнення такого ж ефекту — використовувати атомарні операції з низькорівневими обмеженнями впорядкування. Це можливо завдяки тому що функція load класу std::atomic приймає різні значення memory orders.

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance()
{
    Singleton* tmp =
m_instance.load(std::memory_order_acquire);
    if (tmp == nullptr)
    {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr)
        {
```

```
            tmp = new Singleton;
            m_instance.store(tmp, std::memory_order_release);
        }
    }
    return tmp;
}
```

5.2.3 Послідовно узгоджені атомарні операції

Якщо ви пропустите необов'язковий аргумент std::memory_order для всіх функцій атомарної бібліотеки, значення за замовчуванням — std::memory_order_seq_cst, що перетворюється всі атомарні змінні на послідовно узгоджені (sequentially consistent).

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance()
{
    Singleton* tmp = m_instance.load();
    if (tmp == nullptr)
    {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load();
        if (tmp == nullptr)
        {
            tmp = new Singleton;
            m_instance.store(tmp);
        }
    }
    return tmp;
}
```

Зазвичай програмісти використовують саме такий підхід з послідовно узгодженими атомарними змінними, але в такому випадку, згенерований машинний код, як правило, менш ефективний, ніж у попередніх прикладах.

5.3 std::call_once як альтернатива DCLP

Замість DCLP також можна використати std::call_once і це є цілком життєздатним варіантом, оскільки перший (успішний) виклик синхронізується з усіма наступними викликами, як було описано вище, коли ми розглядали std::call_once і std::once_flag. Також цілком ймовірно що компілятори реалізують цей спосіб у вигляді, що дуже нагадує DCLP. З

використанням call_once код буде виглядати наступним чином:

```
CallOnceSingleton* CallOnceSingleton::instance = nullptr;
std::once_flag CallOnceSingleton::initInstanceFlag;

CallOnceSingleton& CallOnceSingleton::getInstance()
{
   std::call_once(initInstanceFlag,
&CallOnceSingleton::initSingleton);
   return *instance;
}

void CallOnceSingleton::initSingleton()
{
   instance = new CallOnceSingleton;
}
```

5.4 Сінглтон Мейерса

Також в C++11, щоб отримати потокобезпечний синглтон ви можете просто використовувати статичну ініціалізацію.

```
Singleton& Singleton::getInstance()
{
   static Singleton instance;
   return instance;
}
```

Стандарт C++11 пише наступне:

If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.

Компілятор має заповнити деталі реалізації, і DCLP є очевидним вибором. Немає гарантії, що компілятор використовуватиме DCLP, але так сталось, що деякі (можливо, більшість) компіляторів C++11 це роблять.

5.5 Вимірювання продуктивності вище наведених методів

Мірою для порівняння виступатиме час витрачений на виклик методу getInstance 10 мільйонів разів з кожного потоку. Для цього реалізуємо простий тестувальник, який дасть нам змогу виміряти час:

```
template <typename Singleton>
std::chrono::duration<double> getTime()
{
   constexpr auto tenMill = 10000000;
   const auto begin = std::chrono::steady_clock::now();
   for (size_t i = 0; i <= tenMill; ++i)
   {
      Singleton::getInstance();
   }
   return std::chrono::steady_clock::now() - begin;
}

template <typename Singleton>
std::chrono::duration<double> getSumConcurrentTime()
{
   using durationsVectorType =
       std::vector<std::future<std::chrono::duration<double>>>;
   durationsVectorType
       durations(std::thread::hardware_concurrency());
   for (auto& duration : durations)
   {
      duration = std::async(std::launch::async,
            getTime<Singleton>);
   }
   std::chrono::duration<double> sum{0};
   for (auto& duration : durations)
   {
      sum += duration.get();
   }
   return sum;
}

void printTimeInMs(const std::chrono::duration<double> time)
{
   using std::chrono::duration_cast;
   using std::chrono::milliseconds;
   std::cout << duration_cast<milliseconds>(time).count()
      << '\n';
}

int main()
{
  printTimeInMs(getSumConcurrentTime<LockBasedSingleton>());
  printTimeInMs(getSumConcurrentTime<AtomicAcquireRelease>());
  printTimeInMs(
      getSumConcurrentTime<AtomicSequentialSingleton>());
  printTimeInMs(getSumConcurrentTime<CallOnceSingleton>());
  printTimeInMs(getSumConcurrentTime<MeyersSingleton>());
  return 0;
}
```

На моєму комп’ютері, на якому std::thread::hardware_concurrency() показує 12 доступних апаратних потоків, а також виконуючи проект з увімкненою оптимізацією, доступ до сінглтону методом getInstance з кожного з потоків 10 мільйонів разів та провівши даний експеримент тричі для більшої об’єктивності отриманих результатів часи виконання склали:

Час виконання методу, мілісекунди/ № спроби	Застосування блокування	Атомарні операції з acquire та release обмеженнями впорядкування пам’яті	Послідовно узгоджені атомарні операції	std::call_once	Сінглтон Мейерса
1	228 753	132	129	1 165	71
2	232 788	124	125	1 115	71
3	230 077	124	124	1 125	68
Середній час	~230 539	~126	~126	~1135	~70

Як можна побачити, застосування блокування без DCLP показує просто жахливі результати, реалізація DCLP з атомарними операціями, як послідовно узгодженими так і acquire and release виявилась досить швидкою, std::once повільніше, проте все ще значно краще простого застування блокування. Найшвидшим же виявилась реалізація сінглтону Мейерса, яка використовує статичну ініціалізацію C++11. Тому go-to рішенням можна назвати саме цей підхід.

Висновки

З виходом стандарту C++11 багато чого змінилося. У ньому є не тільки модель пам'яті,

орієнтована на використання багатопоточності, а й стандартна бібліотека, куди входять класи управління потоками (див. розділ 2), захист спільно використовуваних даних (див. розділ 3), синхронізація операцій між потоками (див. розділ 4) та низькорівневі атомарні операції (див. розділ 5). У цій роботі ми розглянули тільки частину із наявного інструментарію стандартної бібліотеки, а також застосували набуті знання для реалізації багатопоточної безпечної версії патерну Одинак різними шляхами.

Список використаної літератури

1. **“Concurency in Action”, 2nd Edition, Anthony Williams [Друковане видання]**
2. **“Concurrency with Modern C++”, Rainer Grimm [Друковане видання]**
3. **Документація Concurrency support library [Електронний ресурс]**
   Concurrency support library - cppreference.com
4. Основні визначення [Електронний ресурс]
   Computer multitasking - Wikipedia
   Multithreading (computer architecture) - Wikipedia
   Race condition - Wikipedia
   Deadlock - Wikipedia
5. Модель памяті в C++ і std::atomic [Електронний ресурс]
   std::atomic. Модель пам'яті C++ на прикладах
6. C++ Deadlocks Lei Mao [Електронний ресурс]
   C++ Deadlocks - Lei Mao's Log Book
7. Вікіпедія про Double-Checked Locking
   Double-checked locking - Wikipedia
8. **“C++ and the Perils of Double-Checked Locking” Scott Meyers, Andrei Alexandresku [Електронний ресурс]**
   dcl.dvi (aristeia.com)
9. **“Double-Checked Locking is Fixed In C++11” Jeff Preshing [Електронний ресурс]**
   Double-Checked Locking is Fixed In C++11 (preshing.com)
10. “Thread-Safe Initialization of a Singleton” Rainer Grimm [Електронний ресурс]
    Thread-Safe Initialization of a Singleton - ModernesCpp.com