top КОЛЛЕДЖ

Часть 3. Многопоточность

Асинхронные операции в коде

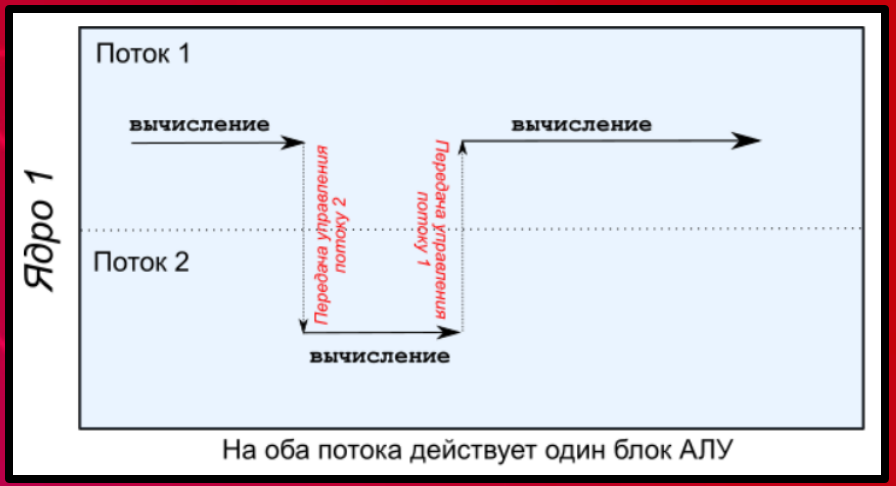
Снова операционные системы

- На операционных системах мы столкнулись с понятием многопоточности.
  Многопоточность — свойство программы или среды иметь несколько потоков выполнения инструкций одновременно (буквально одновременно на разных физических ядрах, либо создавать видимость одновременного выполнения).

Параллельная многопоточность

Каждый поток независим, работает в отдельном блоке АЛУ

Конкурентная многопоточность

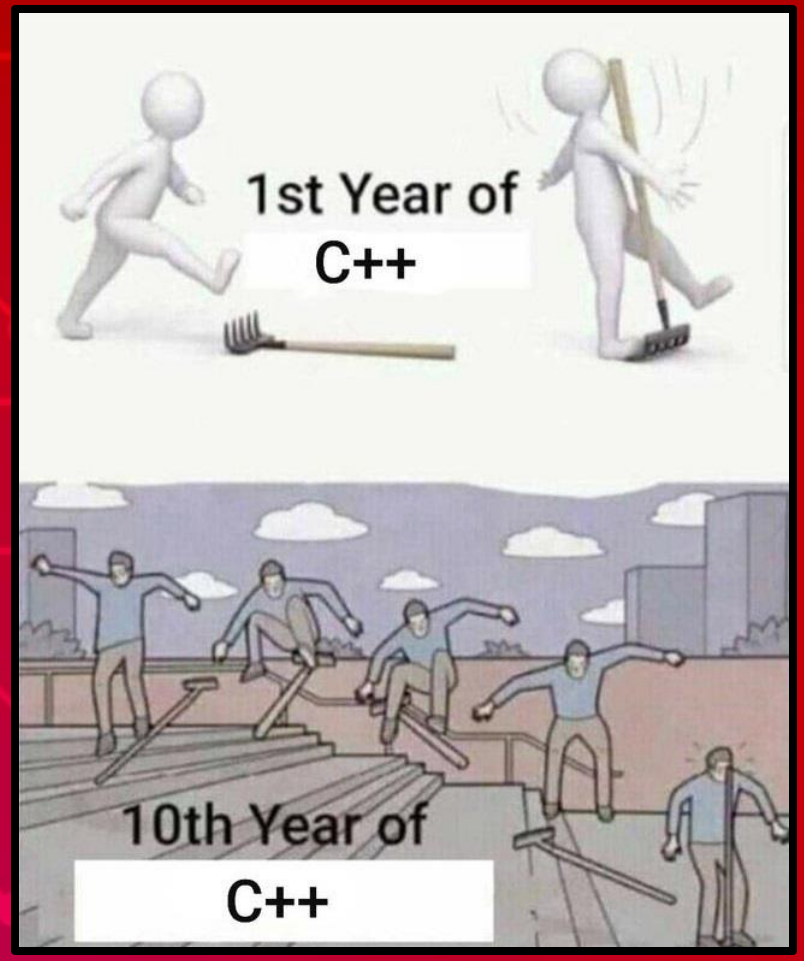


На оба потока действует один блок АЛУ

Многопоточность в C++

top КОЛЛЕДЖ

- В стандарте C++11 появилась возможность создавать многопоточные приложения (важное уточнение: *именно на уровне стандарта, а не конкретной реализации*).
- До этого стандарта в C++ можно было создавать такие приложения, но с использованием отдельных реализаций потоков вроде POSIX threads.



std::thread

top КОЛЛЕДЖ

- Для создания нового потока используется специальный объект стандартной библиотеки, имеющий тип `std::thread` и определенный в заголовочном файле `<thread>`.
- Данный объект при создании запускает процедуру (в виде указателя на функцию) с переданными аргументами в отдельном потоке. Для определения переменной этого типа используется следующий синтаксис:

```
std::thread thr({имя процедуры}, {аргументы процедуры}…);
```

- Заметьте, что данный объект запускает в другом потоке именно ***процедуру*** — то есть функцию с возвращаемым типом `void`.

std::thread

top КОЛЛЕДЖ

- Любой объект данного типа, объявленный в коде, должен рано или поздно принять одно из двух состояний.

Присоединение (join)
Ожидание завершения процедуры в потоке

Отсоединение (detach)
Поток отвязывается от объекта и управляется ОС

- Если этого не сделать, при удалении объекта из памяти, операционная система, отвечающая за выделение памяти другим потокам и процессам, будет вынуждена его «убить», что приводит к неопределенному поведению программы.

Пример использования

top КОЛЛЕДЖ

```cpp
// Некоторая процедура, которая вызывается в другом потоке
void procedure(std::string value)
{
    for (int i = 0; i < 5; ++i)
    {
        std::cout << value;
        std::this_thread::sleep_for( _Rel_time: std::chrono::milliseconds(500));
    }
}

// Изначально существует главный единственный поток, в котором работает функция main
int main()
{
    // Создаем другой поток, который выполняется с главным одновременно
    std::thread thr(procedure, "Hello world from the \033[93mother thread\033[0m\n");

    procedure( value: "Hello world from the \033[92mmain thread\033[0m\n");

    thr.join(); // Ожидаем завершение процедуры в потоке

    return 0;
}
```

```
Hello world from the main thread
Hello world from the other thread
Hello world from the other thread
Hello world from the main thread
Hello world from the main thread
Hello world from the other thread
Hello world from the other thread
Hello world from the main thread
Hello world from the main thread
Hello world from the other thread
```

std::this_thread

top КОЛЛЕДЖ

- В прошлом примере вы могли заметить использование функции с длинным названием `std::this_thread::sleep_for`.
- В реализации потоков в C++ существует возможность управлять текущим потоком по отношению к выполняющейся инструкции. Существуют следующие функции, которые можно использовать:
  - `std::this_thread::sleep_for` — функция для остановки работы потока (*засыпание*) на заданное время (например, тип `std::chrono::milliseconds` отвечает за миллисекунды);
  - `std::this_thread::get_id` — функция для получения идентификатора потока;
  - `std::this_thread::yield` — функция для принудительной передачи управления другому потоку из текущего.

Зачем нужны другие потоки

top КОЛЛЕДЖ

- Как правило, для работы программ достаточно одного единственного потока, однако в некоторых случаях необходимо выполнять несколько операций одновременно, например:
  - Отображение графического интерфейса в главном потоке и одновременное с ним тяжелое вычисление во второстепенном (в данном случае поток может быть нужен для отображения прогресса выполнения вычислений, иначе интерфейс программы не будет реагировать на действия пользователя во время этого вычисления);
  - Одновременное считывание данных и их обработка (например, ввод с клавиатуры и реакция на этот ввод на экране в играх);
  - Для распределения вычислений на несколько ядер для их ускорения.

Вопрос

top КОЛЛЕДЖ

Можно ли в `std::thread` запустить функцию, возвращающую значение?

Данные из другого потока

top КОЛЛЕДЖ

- Можно проверить, что при использовании объекта std::thread в другом потоке нельзя вызывать функцию, возвращающую некоторое значение. Точнее вызвать можно, а вот получить результат не получится.
- Существует два варианта получить данные из другого потока: переписать сигнатуру функции, либо использовать замыкание.

```
// Некоторая функция, которая вызывается в другом потоке
std::string procedure()
{
    srand(_Seed: time(_Time: nullptr));
    std::string result;
    for (int i = 0; i < 10; ++i)
    {
        char symb = 'A' + (rand() % 26);
        result.push_back(_Ch: symb);
    }
    return result;
}

int main()
{
    std::thread thr(procedure); // А как результат вернуть?

    std::string result = procedure(); // Тут результат легко получить

    thr.join();

    std::cout << result << std::endl;

    return 0;
}
```

Данные из другого потока

top КОЛЛЕДЖ

- В первом случае можно создать *дополнительный аргумент* в функции, по которому передается переменная типа «указатель на возвращаемый тип», а фактический возвращаемый тип функции задать равным `void`.
- Использование ссылок в данном случае не допускается из-за ограничений конструктора типа `std::thread`.

```
// Некоторая функция, которая вызывается в другом потоке
void procedure(std::string * result)
{
    result->clear();
    srand(_Seed: time(_Time: nullptr));
    for (int i = 0; i < 10; ++i)
    {
        char symb = 'A' + (rand() % 26);
        result->push_back(_Ch: symb);
    }
}

int main()
{
    std::string result;
    // Возвращаем результат по указателю
    std::thread thr(procedure, &result);

    thr.join();

    std::cout << result << std::endl;

    return 0;
}
```

Данные из другого потока

top КОЛЛЕДЖ

- Для второго случая воспользуемся *замыканием*. Это позволит избежать изменения сигнатуры используемой функции, благодаря чему использование ее в других местах по коду отличаться не будет, но при этом ее можно будет использовать и в другом потоке.
- В замыкании необходимо осуществить захват возвращаемого значения по ссылке. Если у функции имеются аргументы, то их нужно продублировать и в замыкании.

```
// Некоторая функция, которая вызывается в другом потоке
std::string procedure()
{
    srand(_Seed: time(_Time: nullptr));
    std::string result;
    for (int i = 0; i < 10; ++i)
    {
        char symb = 'A' + (rand() % 26);
        result.push_back(_Ch: symb);
    }
    return result;
}

int main()
{
    std::string result;
    // Используем замыкание
    auto lambda []()->void closure = [&result]() { result = procedure(); };
    std::thread thr(closure);

    thr.join();

    std::cout << result << std::endl;

    return 0;
}
```

std::future

top КОЛЛЕДЖ

- Обе представленные решения являются не очень интуитивными, хотелось бы иметь объект потока, который также может запускать функции и получать при завершении их результат.
- Такой объект в C++ есть — `std::future` из заголовочного файла `<future>`. Для создания такого объекта используется следующий синтаксис:

```
std::future<{возвращаемый тип}> future = std::async({функция}, …);
```

- Для получения данных используется метод get:

```
{тип} result = future.get();
```

std::future

top КОЛЛЕДЖ

- Функция `std::async` призвана для создания потоков для выполнения функций асинхронно. Однако у нее есть перегрузка: она позволяет определить политику запуска (*launch policy*) для операции.
- Может быть задано две политики запуска:

Асинхронный запуск
`std::launch::async`
Функция вызывается сразу при создании объекта std::future

Отложенный запуск
`std::launch::deferred`
Функция запускается при первом вызове get на объекте std::future

- Используется функция следующим образом:

```
std::async({политика}, {функция}, …)
```

Пример использования

top КОЛЛЕДЖ

```cpp
// Некоторая функция, которая вызывается в другом потоке
std::string function()
{
    srand(_Seed: time(_Time: nullptr));
    std::string result;
    for (int i = 0; i < 10; ++i)
    {
        char symb = 'A' + (rand() % 26);
        result.push_back(_Ch: symb);
    }
    return result;
}

int main()
{
    // Используем std::future с асинхронным запуском функции
    std::future<std::string> fut = std::async(_Policy: std::launch::async, & _Fnarg: function);

    std::cout << fut.get() << std::endl;

    return 0;
}
```

HRPUZHPHUD

Вопрос

top КОЛЛЕДЖ

Помните, какие проблемы возникают при использовании нескольких потоков с общими данными?

Синхронизация потоков

top КОЛЛЕДЖ

- Если несколько потоков работают одновременно с одними данными, то может возникнуть ситуация под названием ***состояние гонки данных***.
- Это состояние легко продемонстрировать, достаточно вызвать в главном и второстепенном потоках одну и ту же функцию с множеством инструкций, обращающихся к одним данным. В качестве примера возьмем вывод в консоль, осуществляющийся с помощью общей глобальной переменной `std::cout`.

```
void print(int value)
{
    for (int i = 0; i < 10; ++i)
    {
        std::cout << "Hello "; // много операций
        std::cout << "world ";
        std::cout << "from ";
        std::cout << "thread #" << value;
        std::cout << "\n";
    }
}
```

```
Hello world from thread #1
Hello world from thread #1Hello
Hello world from thread #world from thread #1
Hello world from thread #1
Hello world 2
Hello world from from thread #1
Hello world from thread #1
Hello world from thread #thread #2
Hello 1
Hello world from thread #1
```

Синхронизация потоков

top КОЛЛЕДЖ

- Для предотвращения подобного поведения используется ***синхронизация потоков***. Синхронизация потоков основана на введении критической секции, внутри которой доступ к одному общему ресурсу предоставляется не более, чем одному потоку, в любой момент времени.
- Для обеспечения синхронизации могут использоваться следующие примитивы:
  - Атомарные операции и типы;
  - Семафоры;
  - Мьютексы;
  - R/W-регистры;
  - Барьеры.

Атомарные типы

top КОЛЛЕДЖ

- Также в C++ существуют предопределенные типы, для которых определены атомарные операции. Напомню, что ***атомарные операции*** — это операции, для которых определен порядок выполнения инструкций чтения/записи. Другими словами операция либо выполняется целиком, либо не выполняется вовсе, аналогично транзакциям.
- Такие типы представлены в заголовочном файле `<atomic>`. Все арифметические и логические операции над ними можно выполнять, не заботясь о введении других примитивов синхронизации:

```
atomic_char,   atomic_int,   atomic_bool (C++11)
atomic_float,   atomic_double (C++20)
```

Мьютексы

top КОЛЛЕДЖ

- Самым популярным механизмом синхронизации потоков (еще со времен языка C) являются мьютексы за счет простоты их реализации и использования.
- В C++ мьютексы представлены типом `std::mutex` из заголовочного файла `<mutex>`.
- Для определения начала критической секции необходимо захватить блокировку с помощью метода `lock`. Для определения конца необходимо отпустить блокировку с использованием метода `unlock`.
- *Замечание*: мьютекс должен быть общим среди всех потоков, где он применяется.

```cpp
// Общий для всех потоков мьютекс легче всего
// определить как глобальную переменную
std::mutex mtx;

void print(int value)
{
    for (int i = 0; i < 10; ++i)
    {
        mtx.lock(); // Начало критической секции
        std::cout << "Hello "; // много операций
        std::cout << "world ";
        std::cout << "from ";
        std::cout << "thread #" << value;
        std::cout << "\n";
        mtx.unlock(); // Конец критической секции
    }
}
```

```
Hello world from thread #1
Hello world from thread #1
Hello world from thread #1
Hello world from thread #2
Hello world from thread #2
Hello world from thread #2
```

Вопрос

top КОЛЛЕДЖ

Что произойдет, если после получения потоком блокировки будет выброшено исключение?

Проблемы мьютексов

top КОЛЛЕДЖ

- Как вы знаете, при выбросе исключения происходит развертка стека вверх, поэтому все операции после выброса исключения не выполняются. Таким образом, метод `unlock` вызван не будет, и блокировка не снимется.
- Один из вариантов избежания такой ситуации — это снимать блокировку при отлавливании исключения, однако это может потенциально вести к неопределенному поведению программы.

```
mtx.lock(); // Начало критической секции
std::cout << "Hello "; // много операций
std::cout << "world ";
std::cout << "from ";

// Представим, что здесь произошло исключение.
// В таком случае, блокировка не будет отпущена.

std::cout << "thread #" << value;
std::cout << "\n";
mtx.unlock(); // Конец критической секции
```

Концепция RAII

- Решение такой проблемы в C++ было найдено снова в лице автоматического срабатывания деструктора при покидании области видимости, аналогично тому как это делают интеллектуальные указатели.
- Смысл в том, чтобы создать некоторый объект, который в конструкторе блокировку получит, а в деструкторе снимет.
- Существует концепция в C++ для управления ресурсами с помощью создания и удаления объектов. Она имеет название ***Resource Acquisition Is Initialization*** (захват ресурса является инициализацией, ***RAII***), и, в том числе, используется в синхронизации потоков.

Концепция RAII

top КОЛЛЕДЖ

- Итак, для использования такой концепции используется тип std::lock_guard, который при инициализации обращается к некоторому мьютексу и блокирует его.

```
std::lock_guard<std::mutex> lg(mtx);
```

- При удалении объекта из памяти, то есть при срабатывании деструктора, например, во время выхода из области видимости или развертки стека при исключениях, снова происходит обращение к мьютексу и блокировка снимается.

```
void print(int value)
{
    for (int i = 0; i < 10; ++i)
    {
        // Используем RAII для захвата блокировки
        std::lock_guard<std::mutex> lg(mtx);
        std::cout << "Hello "; // много операций
        std::cout << "world ";
        std::cout << "from ";
        std::cout << "thread #" << value;
        std::cout << "\n";
    } // Здесь локальная переменная lg
      // удаляется и блокировка снимается
}
```