Конкуренция, состояние гонки. Race condition

Вадим Калашников
Lead Software Engineer
в компании Wildberries

Проверка связи

Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара или закройте страницу и заново присоединитесь к вебинару
- откройте вебинар в другом браузере
- перезагрузите компьютер/ноутбук и заново попытайтесь зайти

Поставьте в чат:

\+ если меня видно и слышно

— если нет

Вадим Калашников

О спикере:

- Разработчик на C++ более 15 лет
- Опыт разработки в областях: backend, embedded, kernel development, системное программирование, сети.
- С 2023 года Lead Software Engineer в компании Wildberries

Вспомним прошлое занятие

Вопрос: как запустить функцию в отдельном потоке?

Вспомним прошлое занятие

Вопрос: как запустить функцию в отдельном потоке?

Ответ: создать объект класса thread и передать в конструктор этого класса ссылку на эту функцию

Вспомним прошлое занятие

Вопрос: как связать основной поток и дочерние потоки?

Вспомним прошлое занятие

Вопрос: как связать основной поток и дочерние потоки?

Ответ: при помощи функций join() или detach()

Вспомним прошлое занятие

Вопрос: как передать в поток параметр по ссылке?

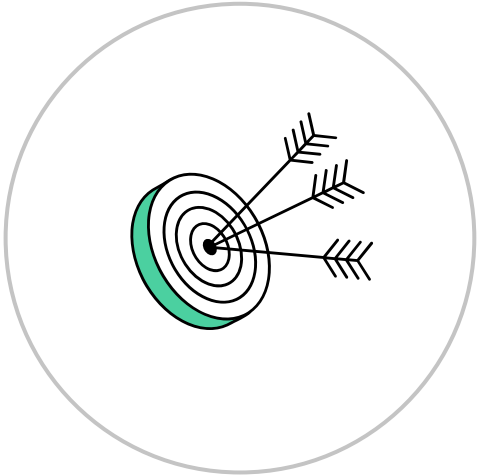
Вспомним прошлое занятие

Вопрос: как передать в поток параметр по ссылке?

Ответ: при помощи функции std::ref()

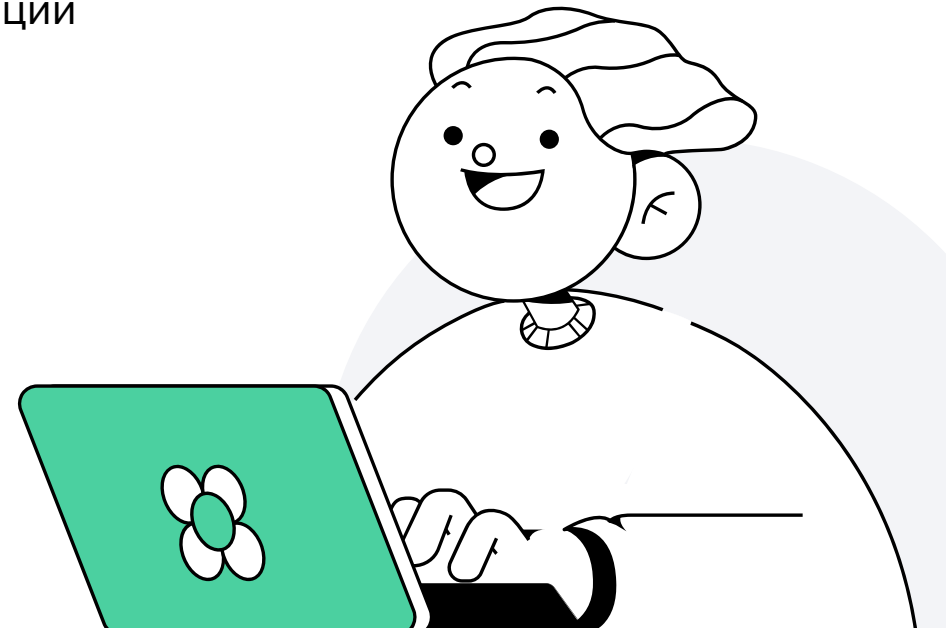
Цели занятия

- Узнаем, что такое мьютексы, условные переменные и атомарные операции
- Создадим программы, защищённые от гонки данных
- Разберём подходы к созданию синхронизирующихся параллельных программ



План занятия

1. Использование мьютексов в C++
2. Взаимные блокировки
3. Синхронизация параллельных операций
4. Атомарные операции
5. Итоги
6. Домашнее задание



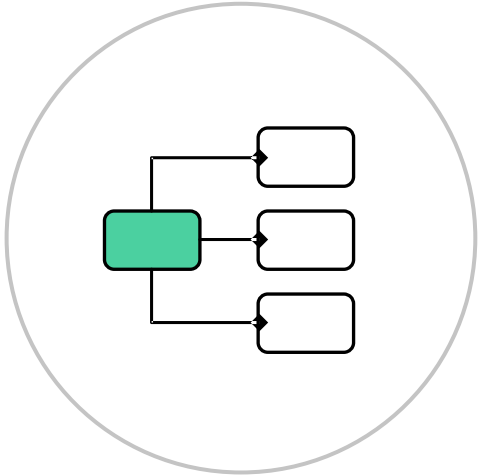
*Нажмите на нужный раздел, чтобы перейти

Использование мьютексов в C++

Главная проблема параллелизма

Если потоки разделяют какие-то данные, необходимы правила, регулирующие, какой поток в какой момент к каким данным может обращаться и как сообщить об изменениях другим потокам, использующим те же данные.

Некорректное использование разделяемых данных — одна из основных причин ошибок, связанных с параллелизмом



Состояние гонки (race condition) — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода

Гонка за данными (data race) — ситуацию, когда гонка возникает из-за одновременной модификации одного объекта, что приводит к неопределённому поведению

Способы защиты разделяемых данных

Средства языка C++

Программирование без блокировок

Программная транзакционная память

Данные снабжаются защитным механизмом, который гарантирует, что только поток, выполняющий модификацию, может видеть промежуточные состояния, в которых инварианты нарушены.

С точки зрения всех остальных потоков, обращающихся к той же структуре данных, модификация либо ещё не началась, либо уже завершилась

Способы защиты разделяемых данных

Средства языка C++

Программирование без блокировок

Программная транзакционная память

Структура данных формируется так, чтобы модификация представляла собой последовательность неделимых изменений, каждое из которых сохраняет инварианты.

В этом случае приходится учитывать нюансы модели памяти и разбираться, какие потоки потенциально могут увидеть те или иные наборы значений

Способы защиты разделяемых данных

Средства языка C++

Программирование без блокировок

Программная транзакционная память

Изменения структуры данных рассматривают как **транзакцию**, поэтому они обрабатываются аналогично обновлению базы данных внутри транзакции. Требуемая последовательность изменений и чтений данных сохраняется в журнале транзакций, а затем атомарно фиксируется.

Если фиксация невозможна, потому что структуру данных в это время модифицирует другой поток, транзакция перезапускается

**Мьютекс, mutex (от mutual exclusion) —
взаимное исключение**

**Библиотека Thread Library гарантирует,
что если один поток захватил некоторый мьютекс,
все остальные потоки, пытающиеся захватить
тот же мьютекс, будут вынуждены ждать,
пока захвативший поток не освободит его**

Защита данных с помощью мьютексов

Не передавайте указатели и ссылки на защищённые данные за пределы области видимости блокировки никаким способом, будь то возврат из функции, сохранение в видимой извне памяти или передача в виде аргумента пользовательской функции:

```
1  mutex m;
2  void func()
3  {
4       m.lock();
5       //do something
6       m.unlock();
7  }
```

```
1  mutex m;
2  void func()
3  {
4       lock_guard<mutex> grd(m);
5       //do something
6  }
7
```

Выявление состояний гонки

Использование мьютексов или других механизмов для защиты разделяемых данных **не гарантирует**, что гонок можно не опасаться.

Необходимо следить, чтобы данные были защищены:

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

Между вызовами empty() и top() другой поток мог вызвать pop() и удалить элемент

Выявление состояний гонки

Поток 1	Поток 2
`if (!s.empty()) {`	
`int const value = s.top();`	`if (!s.empty()){`
	`int const value = s.top();`
`s.pop();`	
`do_something(value);`	`s.pop();`
`}`	`do_something(value);` `}`

Между двумя обращениями к `top()` никто не может модифицировать стек, так что оба потока увидят одно и то же значение.

Между обращениями к `pop()` нет обращений к `top()`. Следовательно, одно из двух хранившихся в стеке значений никто даже не прочитает, а другое будет обработано дважды

Взаимные блокировки

2

Может сложиться ситуация, когда два потока для выполнения некоторой операции должны захватить два мьютекса, но при этом каждый поток захватил только один мьютекс и ждёт другого.

Ни один поток не может продолжить, так как каждый ждёт, пока другой освободит нужный ему мьютекс.

Такая ситуация называется взаимоблокировкой

Проблема взаимоблокировок

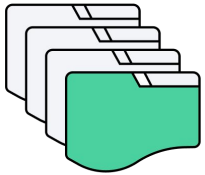
Избежать взаимоблокировок поможет общая рекомендация: всегда захватывайте мьютексы в одном и том же порядке. Если мьютекс A всегда захватывается раньше мьютекса B, взаимоблокировка не возникнет

Проблема взаимоблокировок

В стандартной библиотеке существует функция `std::lock()`, которая умеет захватывать сразу два и более мьютексов без риска получить взаимоблокировку.

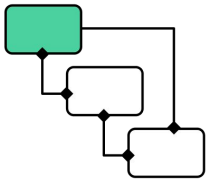
В стандарте C++ 17 появился дополнительный шаблонный класс `std::scoped_lock<>`. Он совмещает `std::lock_guard<>` и `std::lock()`, т. е. принимает список типов мьютексов в качестве параметров шаблона и список мьютексов в качестве аргументов конструктора

Рекомендации, как избежать взаимоблокировок

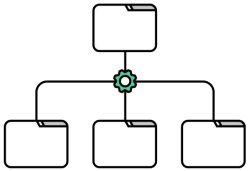


Избегайте вложенных блокировок

Старайтесь не вызывать пользовательский код, когда удерживаете мьютекс



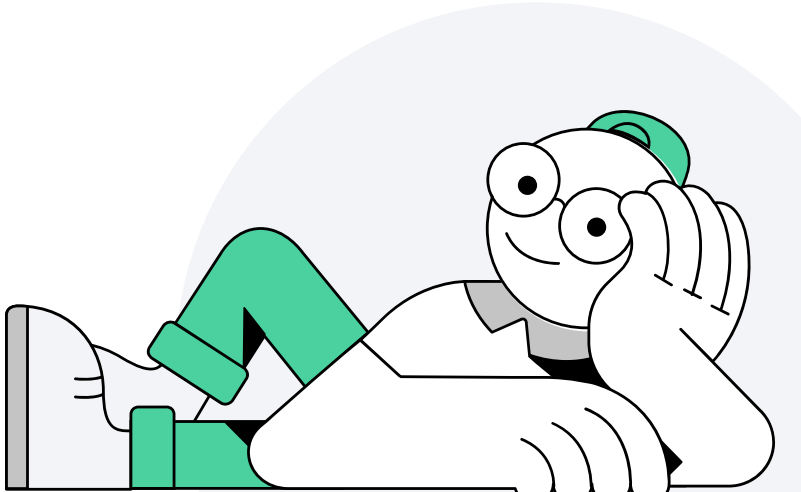
Захватывайте мьютексы в фиксированном порядке



Пользуйтесь иерархией блокировок

Блокировка с помощью std::unique_lock

Шаблон `std::unique_lock` обладает бóльшей гибкостью, чем `std::lock_guard`, потому что несколько ослабляет инварианты — экземпляр `std::unique_lock` не обязан владеть ассоциированным с ним мьютексом



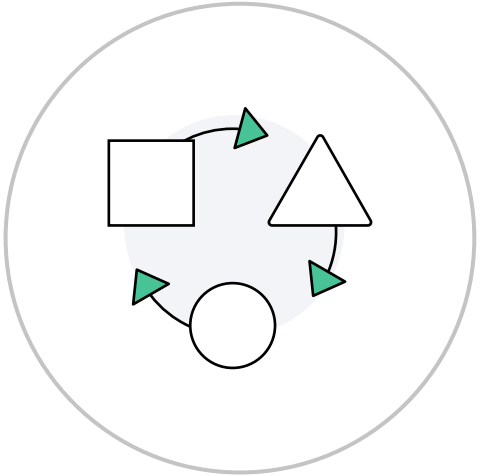
Синхронизация параллельных операций

3

Необходимость синхронизации

В общем случае часто возникает ситуация, когда поток должен ожидать какого-то события или истинности некоторого условия.

Необходимость в синхронизации операций — настолько распространённый сценарий, что в стандартную библиотеку C++ включены специальные механизмы для этой цели — условные переменные и будущие результаты (`std::future`)



Синхронизация при помощи мьютексов

```
bool flag;
mutex m;
void wait_for_flag()
{
     std::unique_lock<std::mutex> lk(m3);
    while (!flag)
    {
        lk.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        lk.lock();
    }
}
```

Условные переменные

Предпочтительный способ синхронизации событий это использовать средства стандартной библиотеки C++, которые позволяют потоку ждать события.

Самый простой механизм ожидания события, возникающего в другом потоке, дают **условные переменные (std::condition_variable)**

Условные переменные

Концептуально условная переменная ассоциирована с каким-то событием или иным условием, причём один или несколько потоков могут ждать, когда это условие окажется выполненным.

Если некоторый поток решит, что условие выполнено, он может известить об этом один или несколько потоков, ожидающих условную переменную. В результате этого они возобновят работу в общем случае. Часто возникает ситуация, когда поток должен ожидать какого-то события или истинности некоторого условия

Атомарные операции

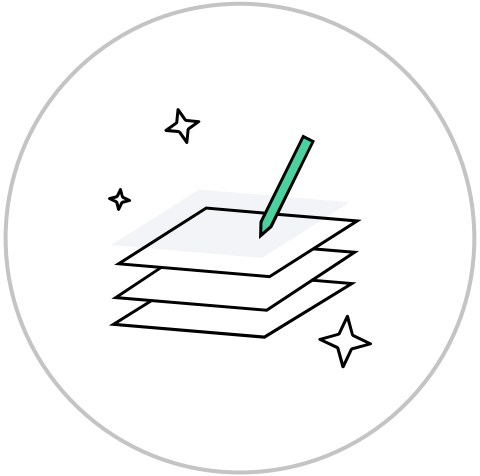
4

Модель памяти

Одна из самых важных особенностей стандарта C++11 — не новые синтаксические конструкции и не новые библиотечные средства, а новая **модель памяти**, учитывающая многопоточность.

У модели памяти есть две стороны:

- базовые **структурные** аспекты, относящиеся к размещению программы в памяти
- аспекты, связанные с **параллелизмом**



Модель памяти

- Каждая переменная — объект, в том числе и переменные, являющиеся членами других объектов
- Каждый объект занимает по меньшей мере одну ячейку памяти
- Переменные фундаментальных типов, например int или char, занимают в точности одну ячейку памяти вне зависимости от размера, даже если являются соседними или элементами массива
- Соседние битовые поля размещаются в одной ячейке памяти

Атомарные типы данных

Если два обращения к одной и той же ячейке памяти из разных потоков не упорядочены и одно или оба обращения не являются **атомарными,** при этом одно или оба обращения являются операциями записи, имеет место гонка за данными, что приводит к неопределённому поведению.

Под **атомарными** понимаются неделимые операции. Ни из одного потока в системе невозможно увидеть, что такая операция выполнена наполовину, — она либо выполнена целиком, либо не выполнена вовсе.

Все стандартные атомарные типы определены в заголовке <atomic>

Атомарные типы данных

Доступ ко всем атомарным типам производится с помощью специализаций шаблона класса `std::atomic<>`

Шаблон класса `std::atomic<>` содержит основной шаблон, который можно использовать для создания атомарного варианта пользовательского типа

Атомарные типы данных

Атомарный тип	Соответствующая специализация
atomic bool	std:atomic<bool>
atomic char	std:atomic<char>
atomic schar	std:atomic<signed char>
atomic uhar	std:atomic<unsigned char>
atomic_int	std:atomic<int>
atomic uint	std:atomic<unsigned>
atomic short	std:atomic<short>
atomic ushort	std:atomic<unsigned short>
atomic_long	std:atomic<long>
atomic ulong	std:atomic<unsigned long>
atomic_llong	std:atomic<long long>
atomic_llong	std:atomic<unsigned long long>
atomic charl6_t	std:atomic<charl6_t>
atomic char32 t	std:atomic<char32_t>
atomic wchar t	std:atomic<wchar_t>

Атомарные типы данных

Стандартные атомарные типы не допускают копирования
и присваивания в обычном смысле, то есть не имеют копирующих
конструкторов и операторов присваивания

Атомарные операции

Атомарные операции разбиты на три категории:

- операции сохранения, для которых можно задавать упорядочение
- операции загрузки, для которых можно задавать упорядочение
- операции чтения — модификации — записи, для которых можно задавать упорядочение

Атомарные операции

Любые операции над атомарным типом должны быть атомарными,
а для присваивания и конструирования копированием нужны два объекта.

Никакая операция над двумя разными объектами не может быть
атомарной. При копировании и присваивании нужно сначала прочитать
значение первого объекта, а потом записать его во второй.

Простейший стандартный атомарный тип `std::atomic_flag`
представляет булевский флаг. Объекты этого типа могут находиться
в одном из двух состояний: установлен или сброшен

Упорядочение доступа к памяти

Последовательное согласование

- `memory_order_seq_cst`

Захват-освобождение

- `memory_order_consume`
- `memory_order_acquire`
- `memory_order_release`
- `memory_order_acq_rel`

Ослабленное

- `memory_order_relaxed`

Если не указано противное, для любой операции над атомарными типами подразумевается упорядочение **memory_order_seq_cst** — самое ограничительное из всех

Найдите ошибку

```
1  mutex m;
2
3  void func1()
4  {
5       m.lock();
6       cout << " thread 1" << endl;
7  }
8  void func2()
9  {
10      m.lock();
11      cout << " thread 2" << endl;
12 }
```

Найдите ошибку

```
1  mutex m1;
2  mutex m2;
3  void func_1()
4  {
5      scoped_lock l{ m1, m2 };
6       lock(m1, m2)
7      cout << “thread 1" << endl;
8  }
9
10 void func_2()
11 {
12     scoped_lock l{ m1, m2 };
13      lock(m1, m2)
14     cout << “thread 2" << endl;
15 }
```

Итоги

Итоги занятия

1. Узнали, что такое мьютексы, условные переменные и атомарные операции

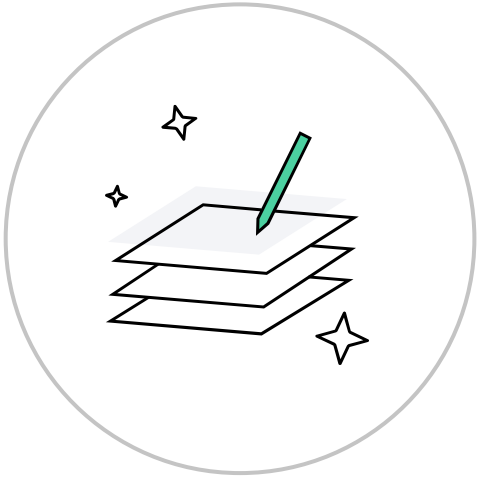
2. Сделали программы, защищённые от гонки данных

3. Разобрали подходы к созданию синхронизирующихся параллельных программ

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы о домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Задавайте вопросы и пишите отзыв о лекции