Асинхронное программирование

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

Вспоминаем прошлое занятие

Вопрос: что такое гонка данных?

Вспоминаем прошлое занятие

Вопрос: что такое гонка данных?

Ответ: попытка нескольких потоков модифицировать одну и ту же область памяти

Вспоминаем прошлое занятие

Вопрос: что такое взаимная блокировка?

Вспоминаем прошлое занятие

Вопрос: что такое взаимная блокировка?

Ответ: захват двух и более зависящих друг от друга мьютексов разными потоками

Вспоминаем прошлое занятие

Вопрос: для чего используются атомарные типы данных?

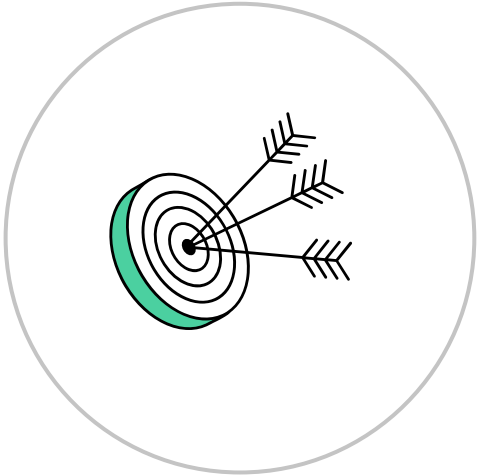
Вспоминаем прошлое занятие

Вопрос: для чего используются атомарные типы данных?

Ответ: для защиты от гонки данных

Цели занятия

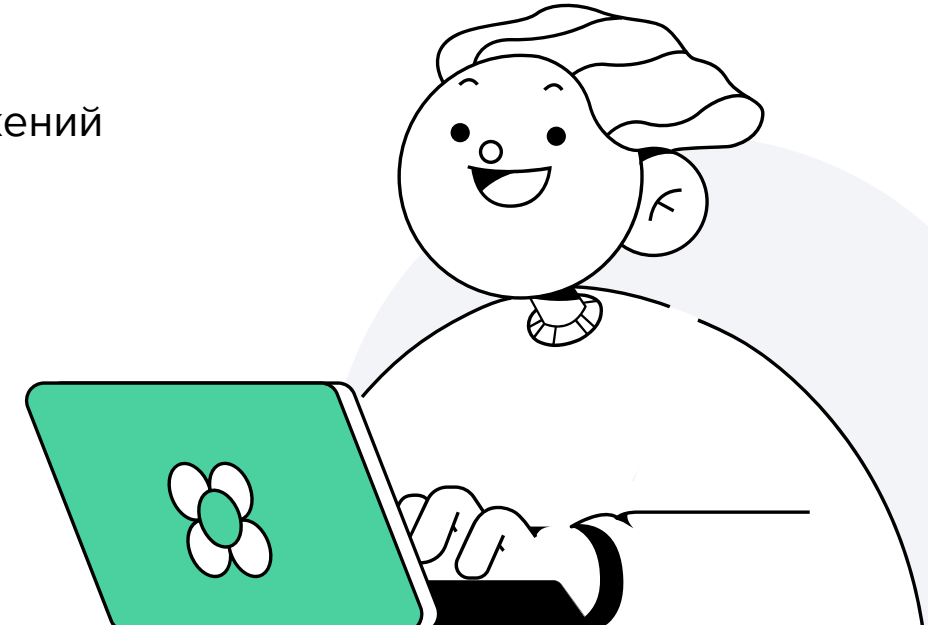
- Узнаем, что такое асинхронное программирование и чем оно отличается от программирования на основе потоков
- Создадим параллельные версии стандартных функций
- Разберём принципы тестирования параллельного кода



План занятия

1. Принципы асинхронного программирования
2. Ожидание событий на основе std::future
3. Ожидание в нескольких потоках
4. Тестирование многопоточных приложений
5. Итоги
6. Домашнее задание

*Нажми на нужный раздел для перехода



Принципы асинхронного программирования

1

Недостатки программирования на основе потоков

Функция **std::async** позволяет запустить **асинхронную задачу**, результат которой прямо сейчас не нужен. В таких вызовах функциональный объект, переданный в **std::async**, рассматривается как задача (task).

Фундаментальное различие между подходами на основе потоков и на основе задач — воплощение более высокого уровня абстракции при использовании **std::async**

Программирование на основе задач

Функция **std::async** позволяет запустить **асинхронную задачу,** результат которой прямо сейчас не нужен. В таких вызовах функциональный объект, переданный в **std::async**, рассматривается как задача (task).

Фундаментальное различие между подходами на основе потоков и на основе задач — воплощение более высокого уровня абстракции при использовании **std::async**

```
1  void doAsyncWork()
2  {
3       //do something
4  }
5
6  int main()
7  {
8       thread t(doAsyncWork); // поток
9       auto fut = async(doAsyncWork); //задача
10      return 0;
11 }
```

Стратегии запуска

Особенность **std::async** — будучи вызванной со стратегией запуска по умолчанию, она не гарантирует создания нового программного потока.

Она, скорее, разрешает планировщику организовать для указанной функции возможность запуска потоком, который запрашивает результат. Интеллектуальные планировщики воспользуются предоставленной свободой, если в системе превышена подписка или не хватает потоков

Стратегии запуска

Имеются две стандартные стратегии, задаваемые параметрами в функции **std::async**:

1. Стратегия std::launch::async означает, что функция должна выполняться асинхронно, т. е. в другом потоке

2. Стратегия std::launch::deferred означает, что функция может выполняться только тогда, когда для фьючерса, возвращённого **std::async**, вызывается функция-член **get()** или **wait()**, т. е. выполнение функции откладывается до тех пор, пока не будет выполнен такой вызов.

→ Если не вызывается ни **get()**, ни **wait()**, функция не выполняется

Ожидание событий на основе std::future

2

Ожидание одноразовых событий

В стандартной библиотеке C++ существует возможность обнаружить одноразовые события с помощью будущего результата.

Для этого в поток необходимо передать специфический объект-будущее.

Затем поток может периодически в течение короткого времени ожидать этот объект-будущее, проверяя, произошло ли событие, а между проверками заниматься другим делом

Ожидание одноразовых событий

В стандартной библиотеке C++ есть два вида будущих результатов, реализованных в форме двух шаблонов классов:

- **уникальные будущие результаты** (std::future<>) — только один объект может ссылаться на одно событие
- **разделяемые будущие результаты** (std::shared_future<>) — несколько объектов могут ссылаться на одно событие

Шаблоны объявлены в заголовке <future>

Ассоциирование с помощью std::package_task

Шаблон класса **std::packaged_task<>** связывает будущий результат с функцией или объектом, допускающим вызов.

При вызове объекта **std::packaged_task<>** ассоциированная функция или допускающий вызов объект вызывается и делает будущий результат готовым, сохраняя возвращённое значение в виде ассоциированных данных

Ассоциирование с помощью std::package_task

Параметром шаблона класса **std::packaged_task<>** является сигнатура функции.

При конструировании экземпляра **std::packaged_task** необходимо передать функцию или допускающий вызов объект, который принимает параметры указанных типов и возвращает значение типа, преобразуемого в указанный тип возвращаемого значения

Ассоциирование с помощью std::promise

Шаблон **std::promise<T>** даёт возможность задать значение (типа T), которое впоследствии можно будет прочитать с помощью ассоциированного объекта **std::future<T>**.

Пара **std::promise/std::future** реализует один из возможных механизмов такого рода. Ожидающий поток приостанавливается в ожидании будущего результата, тогда как поток, поставляющий данные, может с помощью promise установить ассоциированное значение и сделать будущий результат готовым

Ассоциирование с помощью std::promise

Чтобы получить объект **std::future**, ассоциированный с этим обещанием **std::promise**, необходимо вызвать функцию-член **get_future()**.

После установки значения обещания с помощью функции-члена **set_value()** будущий результат становится готовым, и его можно использовать для получения установленного значения.

Если уничтожить объект **std::promise**, не установив значение, то в будущем результате будет сохранено исключение

Рекурсивный запуск std::async

Использование описанных выше средств синхронизации в качестве строительных блоков позволяет сосредоточиться на самих нуждающихся в синхронизации операциях, а не на механизмах реализации.

В частности, код можно упростить, применяя более **функциональный** (в смысле **функционального программирования**) подход к программированию параллелизма.

Вместо того чтобы напрямую разделять данные между потоками, мы можем снабдить каждый поток необходимыми ему данными, а результаты вычисления предоставить другим потокам, которые в них заинтересованы, с помощью будущих результатов

Ожидание в нескольких потоках

3

Использование std::shared_future

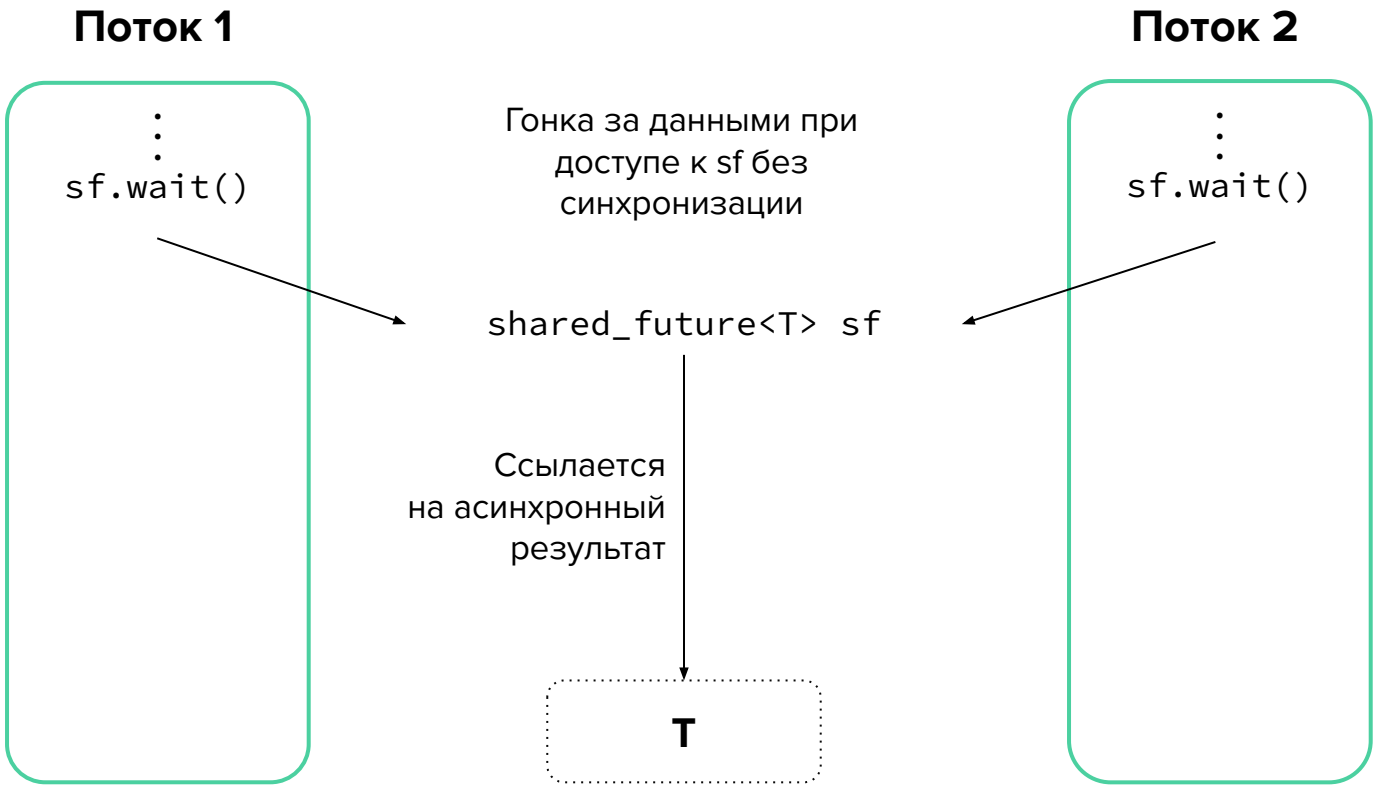
Работа с одним объектом **std::future** из нескольких потоков без дополнительной синхронизации может закончиться гонкой за данными и неопределённым поведением

Использование std::shared_future

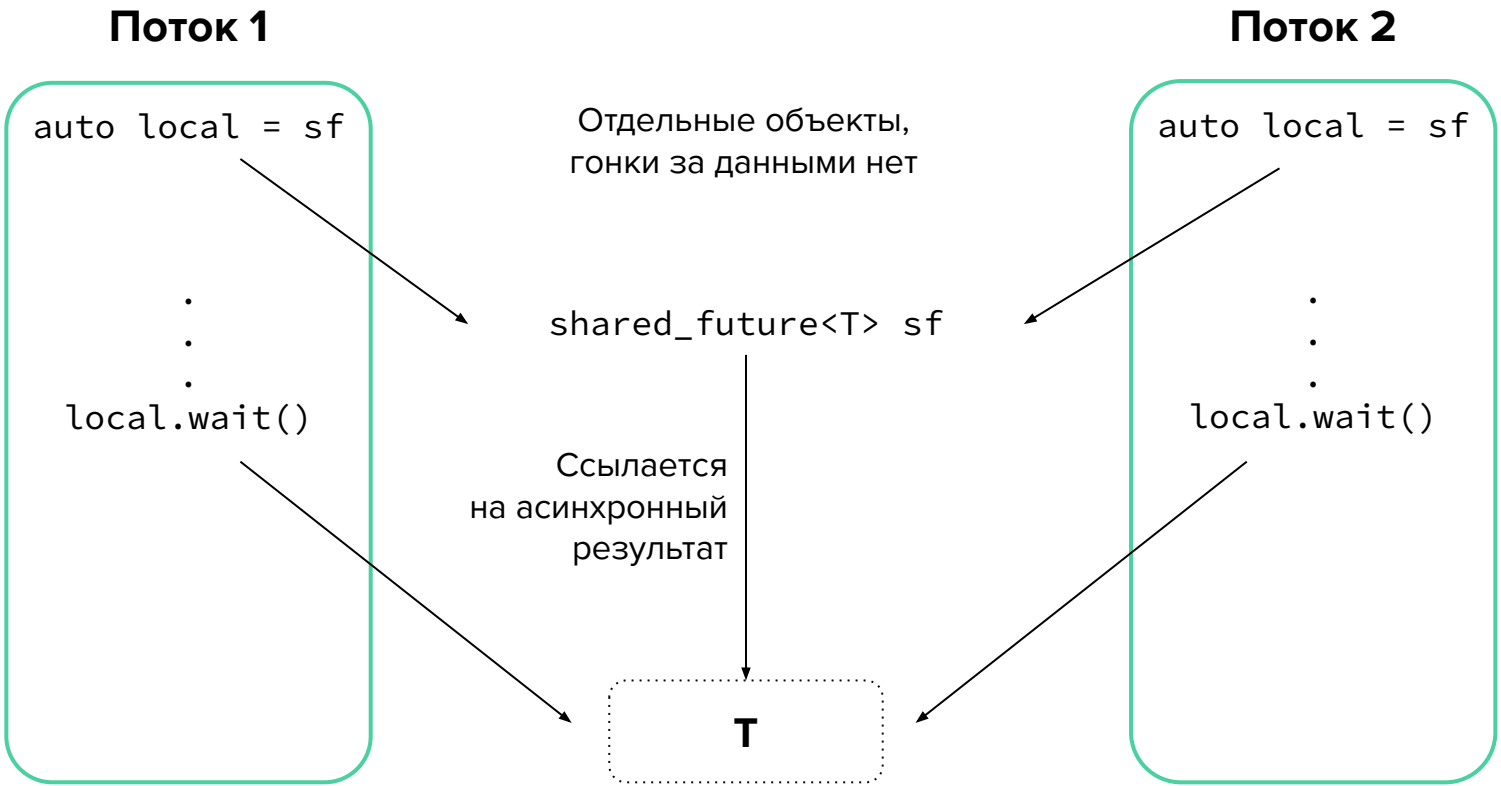
Если алгоритм параллельной программы требует, чтобы одного события могли ждать несколько потоков, то на этот случай предусмотрен шаблон класса **std::shared_future**.

std::future допускает только перемещение: владение можно передавать от одного экземпляра другому, но в каждый момент на асинхронный результат ссылался лишь один экземпляр. А экземпляры **std::shared_future** допускают и копирование, то есть на одно и то же ассоциированное состояние могут ссылаться несколько объектов

Использование std::shared_future



Использование std::shared_future



Тестирование многопоточных приложений

4

Типы ошибок, связанных с параллелизмом

Взаимоблокировка

Активная блокировка

Блокировка в ожидании завершения ввода/вывода или поступления данных из внешнего источника

Гонка за данными

Нарушение инвариантов

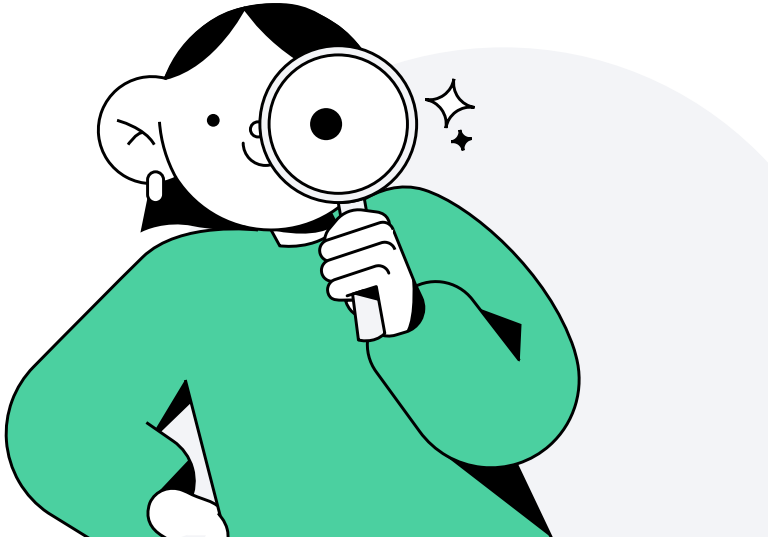
Проблемы со временем жизни

Методы поиска ошибок

1. Какие данные нужно защищать от одновременного доступа?
2. Как вы обеспечиваете защиту этих данных?
3. В каком участке программы могут в этот момент находиться другие потоки?
4. Какие мьютексы удерживает этот поток?
5. Какие мьютексы могут удерживать другие потоки?
6. Существуют ли ограничения на порядок выполнения операций в этом и каком-либо другом потоке? Как гарантируется соблюдение этих ограничений?
7. Верно ли, что данные, загруженные этим потоком, всё ещё действительны? Не могло ли случиться так, что их изменили другие потоки?
8. Если предположить, что другой поток может изменить данные, то к чему это приведёт и как гарантировать, что этого никогда не случится?

Методы поиска ошибок

Тестирование многопоточного кода гораздо сложнее однопоточного, потому что точный порядок выполнения потоков не детерминирован и может изменяться от запуска к запуску



Методы поиска ошибок

Перед многопоточным тестированием имеет смысл устранить параллельность из тестов. Если проблема проявляется даже при однопоточной работе, то это обычная ошибка, не имеющая отношения к параллелизму.

При этом если проблема исчезает при работе в **одноядерной** системе (даже при наличии нескольких одновременно работающих потоков), но появляется в **многоядерной** или **многопроцессорной**, то, возможно, есть состояние гонки и ошибка, связанная с синхронизацией или упорядочением доступа к памяти

Методы поиска ошибок

При реализации параллельного тестирования необходимо учитывать дополнительные параметры тестовой среды:

- Что понимается под «несколькими потоками» в каждом случае?
- Достаточно ли в системе процессорных ядер, чтобы каждый поток работал на отдельном ядре?
- Какова архитектура процессора, на котором будут прогонять тест?
- Как обеспечить подходящее планирование для циклов while в тестах?

Методы поиска ошибок

Упрощение тестирования происходит, если соблюдаются
следующие принципы:

- обязанности всех функций и классов чётко очерчены
- каждая функция короткая и решает ровно одну задачу
- тесты способны полностью контролировать окружение тестируемого кода
- код, выполняющий конкретную тестируемую операцию, находится приблизительно в одном месте, а не разбросан по всей системе

Методы тестирования многопоточного кода

1. Тестирование грубой силой
2. Комбинаторное имитационное тестирование
3. Использование специальных библиотек

Что появится на экране?

```
1  int returnInt()
2  {
3       cout << 42 << endl;
4       return 42;
5  }
6
7
8  int main()
9  {
10      future<int> res = async(launch::deferred, returnInt);
11 }
```

Что появится на экране?

```
double division(double x) {
     if (x == 0)   {
          throw "x == 0";
     }
     return 1 / x;
}

int main() {
     try {
          std::future<double> f = std::async(division, 0);
          double y = f.get();
     }
     catch (logic_error err) {
          cerr << err.what() << endl;
     }
}
```

Итоги

5

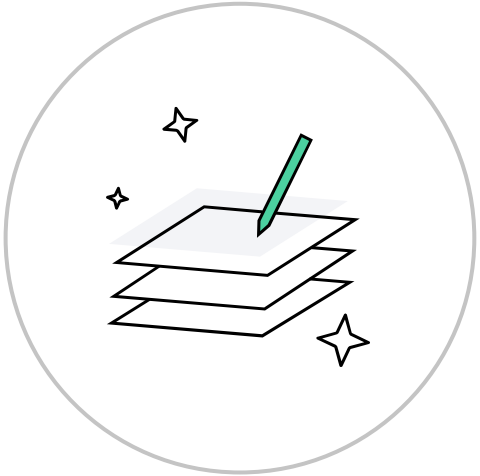
Итоги занятия

1. Узнали, как запускать асинхронные задачи
2. Сделали параллельную функцию суммирования
3. Разобрали принципы синхронизации с помощью std::future

Домашнее задание

Давайте посмотрим ваше **домашнее задание**.

1. Вопросы по 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