Параллельное программирование и суперкомпьютерный кодизайн

Смирнов А.В. asmirnov@srcc.msu.ru

Раздел 3. Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

Раздел 3. План.

Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

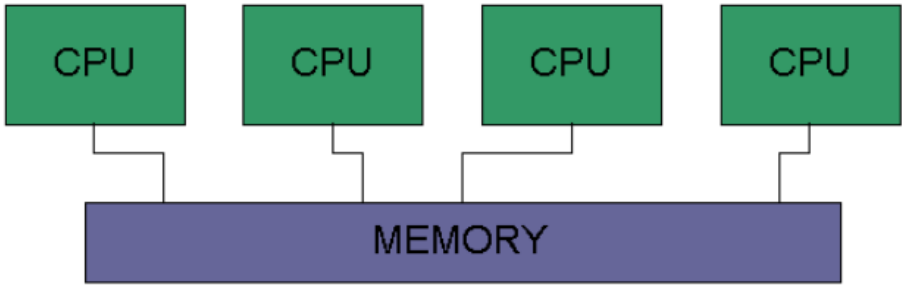
- Введение, общая память и библиотеки
- Использование потоков, передача в них параметров и возврат значений (c++11)
- Параллельная работа в общей памяти, механизмы обеспечения синхронизации (c++11)
- `std::async` и суммирование вектора (c++11)
- Make и Makefile. Основы использования git.
- Модули (c++20)
- Параллельные алгоритмы STL (c++17)
- Введение в корутины (c++20 и немного c++23)

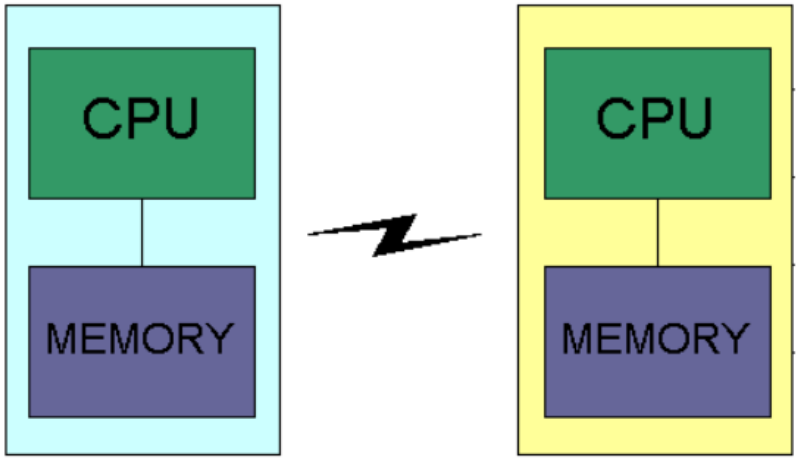
Раздел 3. План.

Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

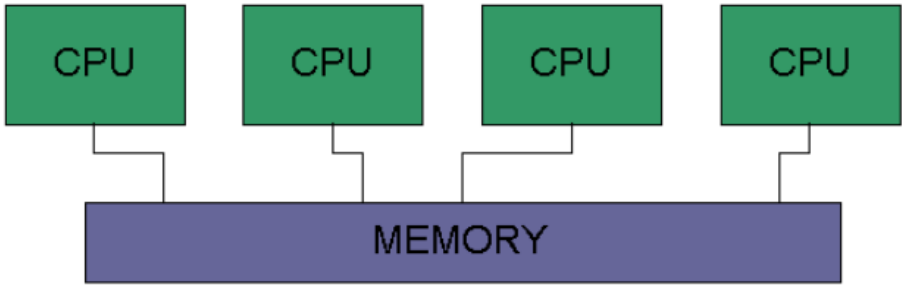
- **Введение, общая память и библиотеки**
- Использование потоков, передача в них параметров и возврат значений (c++11)
- Параллельная работа в общей памяти, механизмы обеспечения синхронизации (c++11)
- `std::async` и суммирование вектора (c++11)
- Make и Makefile. Основы использования git.
- Модули (c++20)
- Параллельные алгоритмы STL (c++17)
- Введение в корутины (c++20 и немного c++23)

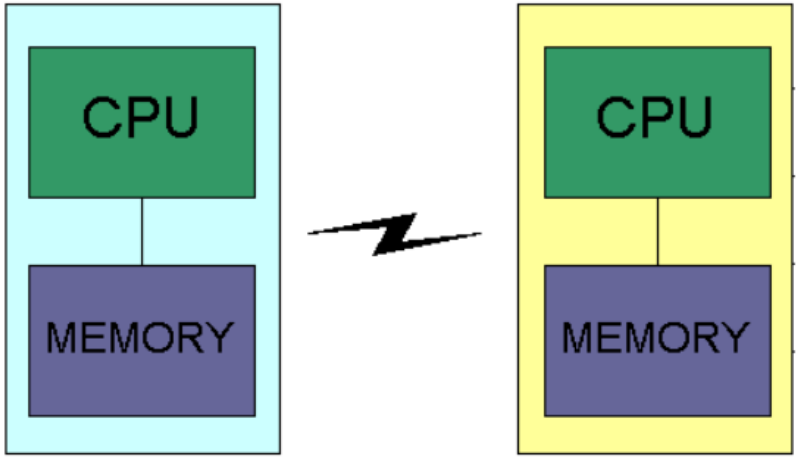
Виды доступа к памяти





Виды доступа к памяти





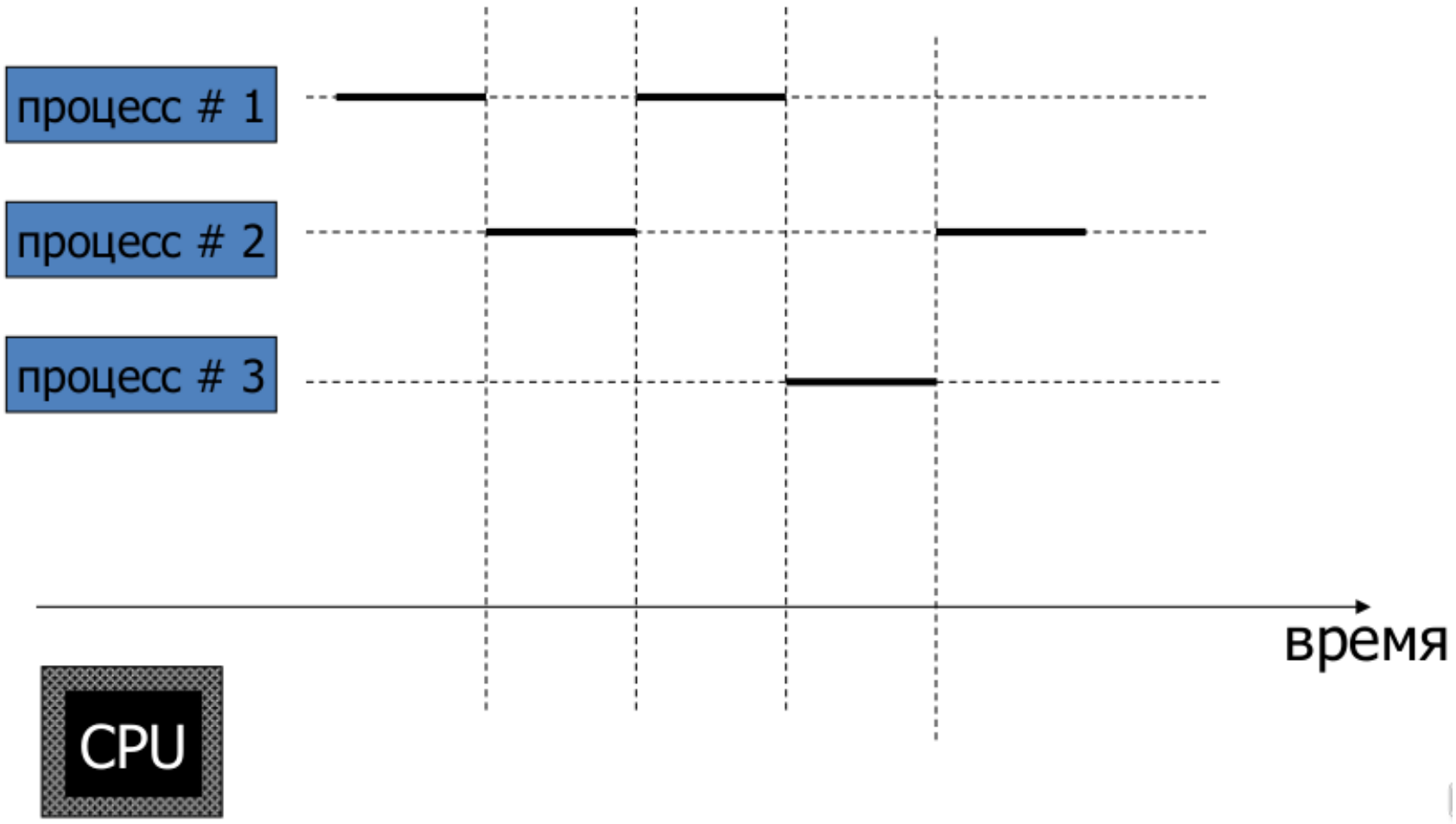
- Программирование в общей памяти позволяет практически не заботиться о задержках доступа к памяти

Процессы

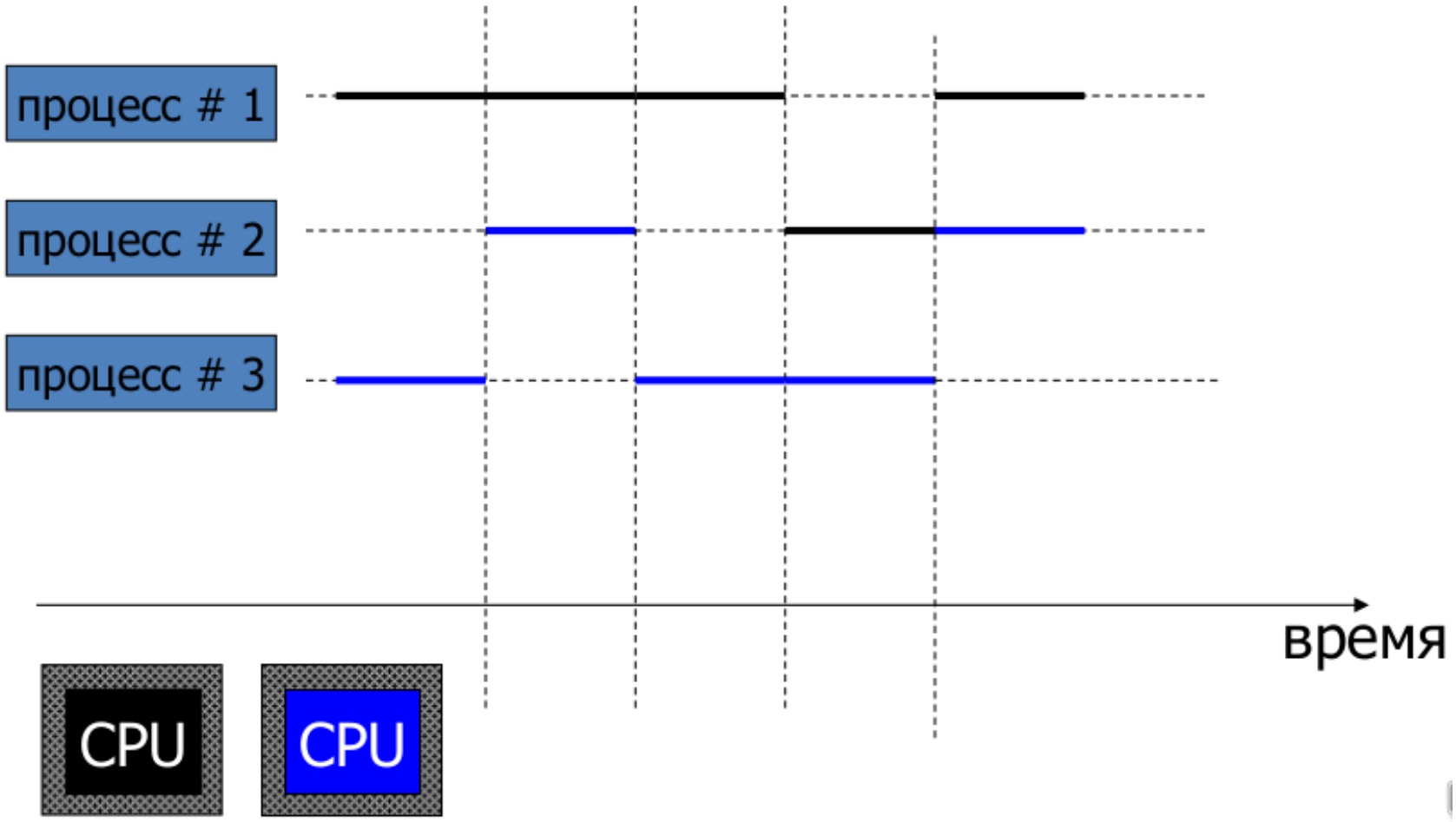
Процесс – это экземпляр выполняемой программы.

- Контекст:
  – регистры;
  – таблица трансляции адресов памяти;
  – ...
- Адресное пространство:
  – код программы;
  – статические данные;
  – стек;
  – разделяемая память;
  – динамическая память (куча).

Планирование процессов



Планирование процессов

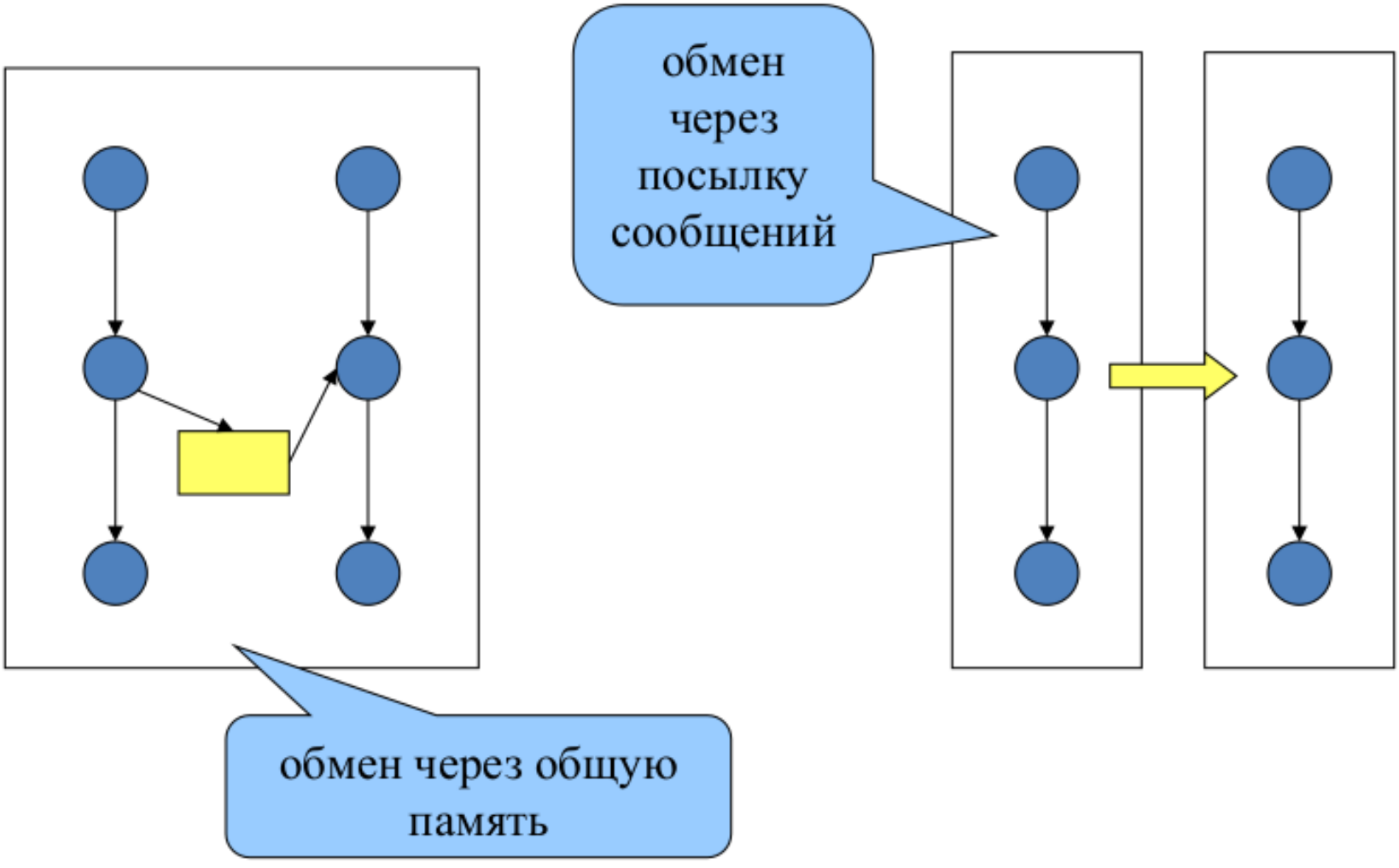


Потоки

Тредами (потоки, нити, threads) называются параллельно выполняющиеся потоки управления в рамках одного процесса

- Потоки одного процесса разделяют его адресное пространство.

Потоки и процессы



Различие потоков и процессов

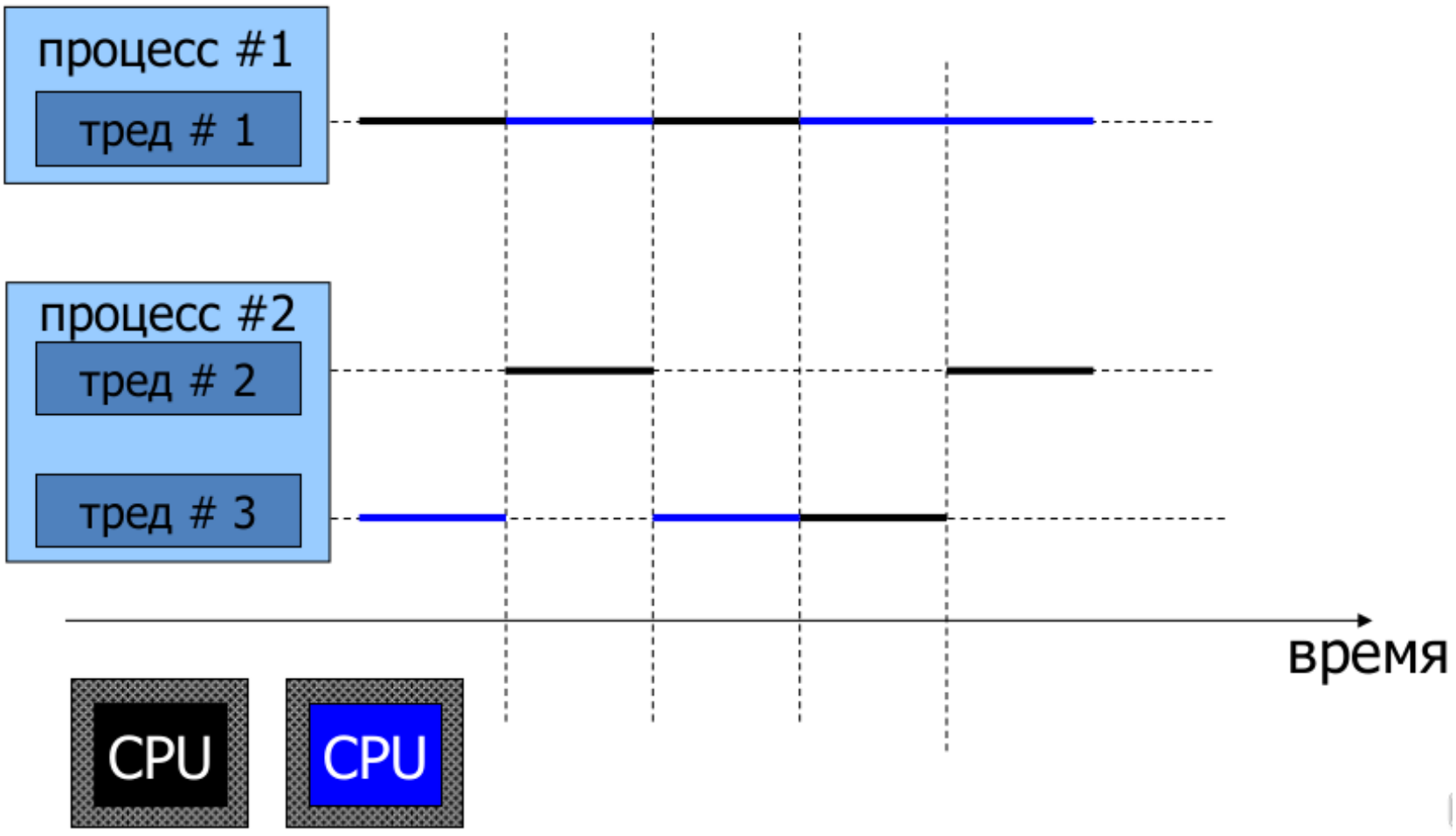
- Различные потоки выполняются в одном адресном пространстве.
- Различные процессы выполняются в разных адресных пространствах.
- Потоки имеют «собственный» стек и набор регистров. Глобальные данные являются общими.
- Как локальные, так и глобальные переменные процессов являются «собственными».

Различные виды потоков

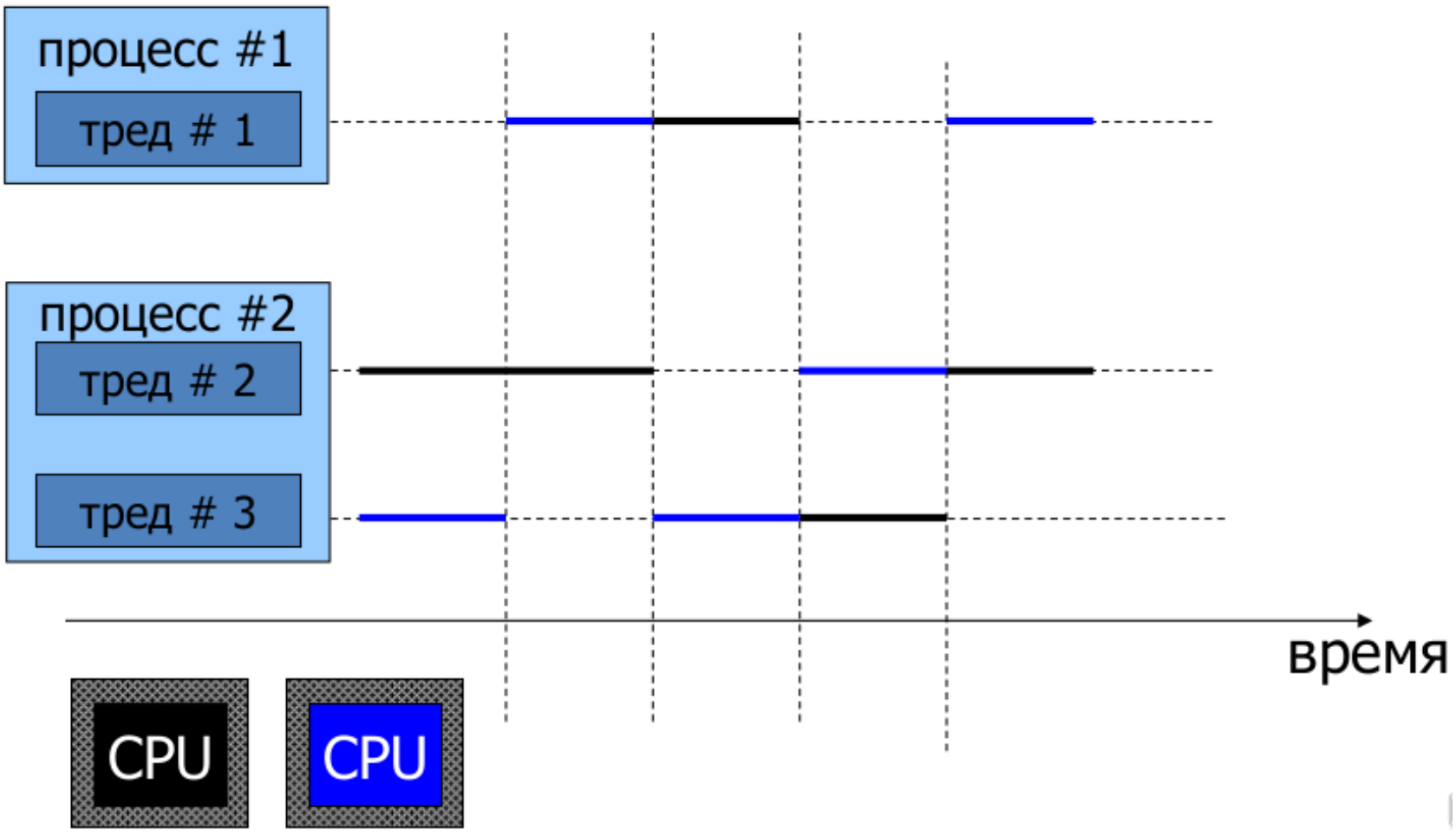
Два вида потоков

- Пользовательские потоки (не используются механизмы ядра) – Такие потоки не «видны» ОС и поэтому не могут выполняться разными ядрами, если принадлежат одному процессу
- Потоки ядра (потоки ОС) – Такие потоки могут использовать преимущества многоядерности, но их переключение требует бОльших ресурсов, чем пользовательских

Планирование пользовательских потоков



Планирование потоков ядра



Различные виды потоков

Два вида потоков

- Пользовательские потоки (не используются механизмы ядра) – Такие потоки не «видны» ОС и поэтому не могут выполняться разными ядрами, если принадлежат одному процессу
- Потоки ядра (потоки ОС) – Такие потоки могут использовать преимущества многоядерности, но их переключение требует бОльших ресурсов, чем пользовательских

Библиотеки, которые мы будем применять, использую потоки ядра

Библиотеки для использования потоков

- PTHREADS – переносимая библиотека для разработки многопоточных программ
- Стандартизована POSIX Section 1003.1c
- Дает возможность получить эффективность близкую к максимально-возможной
- Опирается на системную библиотеку libstdc++ или аналогичную

Библиотеки для использования потоков

- c++11 threads – реализация потоком в modern C++
- В posix-совместимых системах "под капотом"использует PTHREADS, на других системах может иметь альтернативную реализацию
- Мы будем использовать c++11 threads, PTHREADS будут приводиться только в качестве примера

Стандарты языка C и C++

- C99 – классический стандарт языка C. Является развитием стандарта C90 или ANSI C (последний совсем устарел и имеет множество неудобных ограничений)

Стандарты языка C и C++

- C99 – классический стандарт языка C. Является развитием стандарта C90 или ANSI C (последний совсем устарел и имеет множество неудобных ограничений)
- C++98 – классический стандарт C++. В каком-то смысле, "C с классами"

Стандарты языка C и C++

- C99 – классический стандарт языка C. Является развитием стандарта C90 или ANSI C (последний совсем устарел и имеет множество неудобных ограничений)
- C++98 – классический стандарт C++. В каком-то смысле, "C с классами"
- C++11, в свое время названный Modern C++, революционное обновление языка C++. Ключевые обновления - rvalue references, move семантика, lambda-функции, большое обновление стандартной библиотеки и многое другое

Стандарты языка C и C++

- C99 – классический стандарт языка C. Является развитием стандарта C90 или ANSI C (последний совсем устарел и имеет множество неудобных ограничений)
- C++98 – классический стандарт C++. В каком-то смысле, "C с классами"
- C++11, в свое время названный Modern C++, революционное обновление языка C++. Ключевые обновления - rvalue references, move семантика, lambda-функции, большое обновление стандартной библиотеки и многое другое
- C++14 – в большей степени необходимые добавления для C++11

Стандарты языка C и C++

- C99 – классический стандарт языка C. Является развитием стандарта C90 или ANSI C (последний совсем устарел и имеет множество неудобных ограничений)
- C++98 – классический стандарт C++. В каком-то смысле, "C с классами"
- C++11, в свое время названный Modern C++, революционное обновление языка C++. Ключевые обновления - rvalue references, move семантика, lambda-функции, большое обновление стандартной библиотеки и многое другое
- C++14 – в большей степени необходимые добавления для C++11
- C++17 – structural bindings, string_view, optional, variant, std::filesystem, constexpr if, parallel stl

Стандарты языка C и C++

- C99 – классический стандарт языка C. Является развитием стандарта C90 или ANSI C (последний совсем устарел и имеет множество неудобных ограничений)
- C++98 – классический стандарт C++. В каком-то смысле, "C с классами"
- C++11, в свое время названный Modern C++, революционное обновление языка C++. Ключевые обновления - rvalue references, move семантика, lambda-функции, большое обновление стандартной библиотеки и многое другое
- C++14 – в большей степени необходимые добавления для C++11
- C++17 – structural bindings, string_view, optional, variant, std::filesystem, constexpr if, parallel stl
- C++20, C++23, C++2c

Вопросы?

Раздел 3. План.

Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

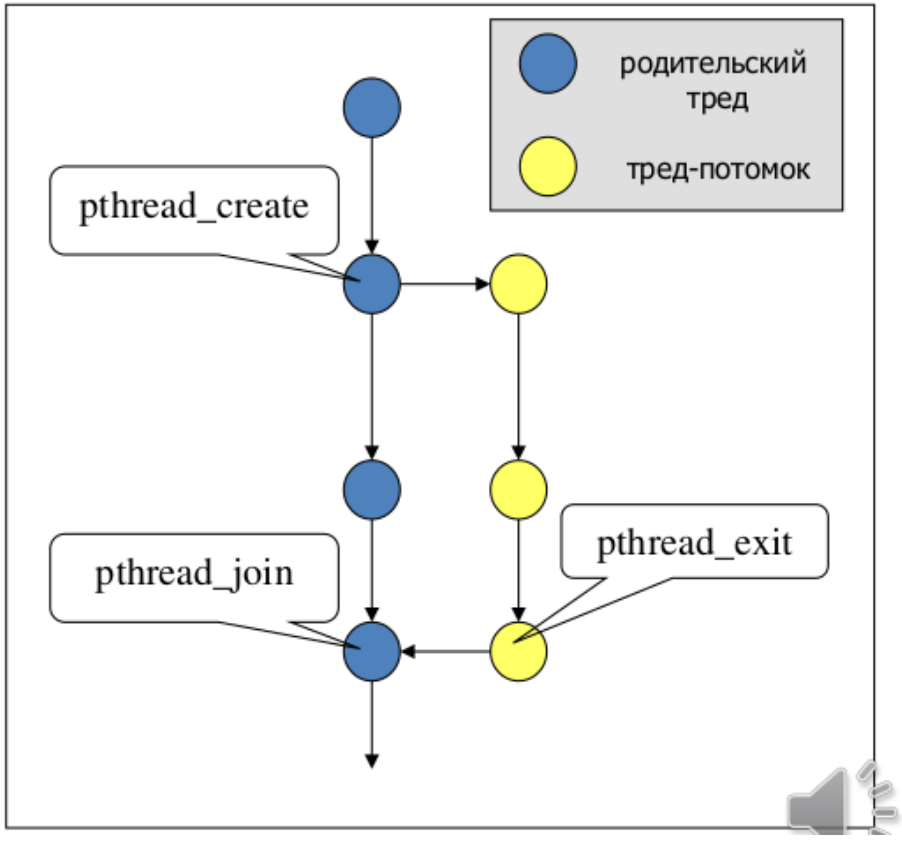
- Введение, общая память и библиотеки
- **Использование потоков, передача в них параметров и возврат значений (c++11)**
- Параллельная работа в общей памяти, механизмы обеспечения синхронизации (c++11)
- `std::async` и суммирование вектора (c++11)
- Make и Makefile. Основы использования git.
- Модули (c++20)
- Параллельные алгоритмы STL (c++17)
- Введение в корутины (c++20 и немного c++23)

Как это было в PTHREADS

```
int pthread_create (
  pthread_t * outHandle,
  pthread_attr_t *inAttribute,
  void *(*inFunction)(void *),
  void *inArg
);

void pthread_exit(void *inReturnValue)

int pthread_join(
    pthread_t inHandle,
    void **outReturnValue,
);
```



Простейшая программа с c++11 threads

```
#include <iostream>
#include <thread>
void hello()
{
    std::cout<<"Hello Other World"<<std::endl;
}
int main()
{
    std::thread t(hello);
    t.join();
}
```

Простейшая программа с c++11 threads

```
#include <iostream>
#include <thread>
void hello()
{
    std::cout<<"Hello Other World"<<std::endl;
}
int main()
{
    std::thread t(hello);
    t.join();
}
```

- Компиляция: g++ -o name name.cpp -pthread

Ответ на главный вопрос жизни, вселенной...

Передача параметров

```
#include <iostream>
#include <thread>
void hello(int n)
{
    std::cout<<"The answer is "<<n<<std::endl;
}
int main()
{
    std::thread t(hello, 42);
    t.join();
}
```

Как это вообще работает?

- `std::thread t(hello);`
- `std::thread t(hello, 42);`

Как это вообще работает?

- `std::thread t(hello);`
- `std::thread t(hello, 42);`
- Здесь же разное количество параметров, и оно может быть вообще любым

Как это вообще работает?

- `std::thread t(hello);`
- `std::thread t(hello, 42);`
- Здесь же разное количество параметров, и оно может быть вообще любым

Вариативные шаблоны

- Вариативный шаблон или шаблон с переменным числом аргументов в программировании — шаблон с заранее неизвестным числом аргументов, которые формируют один или несколько так называемых пакетов параметров.
- Вариативный шаблон позволяет использовать параметризацию типов там, где требуется оперировать произвольным количеством аргументов, каждый из которых имеет произвольный тип. Он может быть очень удобен в тех ситуациях, когда сценарий поведения шаблона может быть обобщён на неизвестное количество принимаемых данных.
- Вариативные шаблоны поддерживаются в C++ начиная со стандарта C++11.

Вариативные шаблоны

```
template<typename T> T sum(T first) {
   return first;
}

template<typename T, typename... Args>
     T sum(T first, Args... args) {
          return first + sum(args...);
     }

int main()
{
     std::cout<<sum(20, 10, 10, 2))<<std::endl;
}
```

Конструкторы класса thread

- «Пустой» поток – `thread();`

Конструкторы класса thread

- «Пустой» поток – `thread();`
- Перемещение потока – `thread( thread&& other );`

Конструкторы класса thread

- «Пустой» поток – `thread();`
- Перемещение потока – `thread( thread&& other );`
- Вызов функции (функтора) в потоке – `template< class Function, class... Args > explicit thread( Function&& f, Args&&... args );`

Конструкторы класса thread

- «Пустой» поток – `thread();`
- Перемещение потока – `thread( thread&& other );`
- Вызов функции (функтора) в потоке – `template< class Function, class... Args > explicit thread( Function&& f, Args&&... args );`
- Запрет копирования – `thread(const thread&) = delete;`

Конструкторы класса thread

Так можно?

- ```
  std::thread a(help, 42);
  std::thread b = a;
  ```
  ```

Конструкторы класса thread

Так можно?

- ```
  std::thread a(help, 42);
  std::thread b = a;
  ```

- ```
  std::thread a;
  a = std::thread(help, 42);
  ```

Конструкторы класса thread

Так можно?

- ```
  std::thread a(help, 42);
  std::thread b = a;
  ```

- ```
  std::thread a;
  a = std::thread(help, 42);
  ```

- ```
  std::thread a(help, 42);
  std::thread b = std::move(a);
  ```
  ```

Что еще за std::move?

- l-value – то, что может быть расположено слева от =

Что еще за std::move?

- l-value – то, что может быть расположено слева от =
- r-value – то, что может быть расположено справа от =

Что еще за std::move?

- l-value – то, что может быть расположено слева от =
- r-value – то, что может быть расположено справа от =
- Всякий l-value это r-value, но не наоборот. Вы можете написать `int a = 2;`, но не можете написать `int a; 2 = a;`. Но при этом некоторый код может не работать для lvalue по определению

Что еще за std::move?

- l-value – то, что может быть расположено слева от =
- r-value – то, что может быть расположено справа от =
- Всякий l-value это r-value, но не наоборот. Вы можете написать `int a = 2;`, но не можете написать `int a; 2 = a;`. Но при этом некоторый код может не работать для lvalue по определению
- l-value reference – ссылка на объект, которая может быть слева от знака равно, то есть ссылается на что-то реальное `int a; int& b = a; a = 2; std::cout«b;`

Что еще за std::move?

- l-value – то, что может быть расположено слева от =
- r-value – то, что может быть расположено справа от =
- Всякий l-value это r-value, но не наоборот. Вы можете написать `int a = 2;`, но не можете написать `int a; 2 = a;`. Но при этом некоторый код может не работать для lvalue по определению
- l-value reference – ссылка на объект, которая может быть слева от знака равно, то есть ссылается на что-то реальное `int a; int& b = a; a = 2; std::cout<<b;`
- Чем это отличается от указателя?

Что еще за std::move?

- l-value – то, что может быть расположено слева от =
- r-value – то, что может быть расположено справа от =
- Всякий l-value это r-value, но не наоборот. Вы можете написать `int a = 2;`, но не можете написать `int a; 2 = a;`. Но при этом некоторый код может не работать для lvalue по определению
- l-value reference – ссылка на объект, которая может быть слева от знака равно, то есть ссылается на что-то реальное
  `int a; int& b = a; a = 2; std::cout<<b;`
- Чем это отличается от указателя?
  `int a; int* b = &a; a = 2; std::cout<< *b;`
  Обращение к ссылке (reference) не требует взятий адреса и перехода по адресу, семантика сохраняется

Что еще за std::move?

- l-value – то, что может быть расположено слева от =
- r-value – то, что может быть расположено справа от =
- Всякий l-value это r-value, но не наоборот. Вы можете написать `int a = 2;`, но не можете написать `int a; 2 = a;`. Но при этом некоторый код может не работать для lvalue по определению
- l-value reference – ссылка на объект, которая может быть слева от знака равно, то есть ссылается на что-то реальное
  `int a; int& b = a; a = 2; std::cout«b;`
- Чем это отличается от указателя?
  `int a; int* b = &a; a = 2; std::cout« *b;`
  Обращение к ссылке (reference) не требует взятий адреса и перехода по адресу, семантика сохраняется r-value reference, int&& (с базовым классом пример бессмысленный) – ссылка, существующая только справа от знака равенства, временная или чьи ресурсы можно задействовать, инвалидируя ссылку

Что еще за std::move?

Функция std::move() — это стандартная библиотечная функция, которая конвертирует передаваемый аргумент в r-value. Мы можем передать l-value в функцию std::move(), и std::move() вернет нам ссылку r-value.

- ```
  std::thread a(help, 42);
  std::thread b = std::move(a);
  b.join();
  ```
- Попытка завершить поток a приведет к ошибке
  ```

Передача параметров

```
void f(int n, std::string& s);

std::thread oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",some_param);
    std::thread t(f,3,buffer);
    return t;
}
```

Казалось бы, что могло пойти не так?

Казалось бы, что могло пойти не так?

Передача параметров

```
void f(int n, std::string& s);

std::thread oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",some_param);
    std::thread t(f,3,buffer);
    return t;
}
```

Казалось бы, что могло пойти не так?

- Мы хотели передать C++ строку, а передали C строку. Конвертация произойдет в потоке, но к этому моменту локальная переменная buffer может уже не существовать и быть заполнена чем-то иным

Передача параметров

Правильное решение при запуске потоков – максимально явно конвертировать параметры в нужные типы.

```
void f(int n, std::string& s);

std::threed correct(int some_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",some_param);
    std::thread t(f,3,
        std::string(buffer)
    );
    return t;
}
```

Как передаются параметры в функции?

```
int func(int a) {
    a = 42;
    return 0;
}
int main()
{
    int a = 21;
    int b = func(a);
    std::cout<<a<<" | "<<b<<std::endl;
}
```

- Что будет напечатано?

Как передаются параметры в функции?

```
int func(int a) {
    a = 42;
    return 0;
}
int main()
{
    int a = 21;
    int b = func(a);
    std::cout<<a<<" | "<<b<<std::endl;
}
```

- Что будет напечатано?
- 21 0, поскольку параметры в функции передаются по значению

Как сделать, чтобы функция изменила параметр?

Вариант, пришедший из C

```
int func(int* a) {
    *a = 42;
    return 0;
}
int main()
{
    int a = 21;
    int b = func(&a);
    std::cout<<a<<" | "<<b<<std::endl;
}
```

Как сделать, чтобы функция изменила параметр?

Вариант, который можно использовать в C++

```
int func(int& a) {
    a = 42;
    return 0;
}
int main()
{
    int a = 21;
    int b = func(a);
    std::cout<<a<<" | "<<b<<std::endl;
}
```

Как этим воспользоваться для потоков?

Так не сработает!

```
int func(int a) {
    a = 42;
    return 0;
}
int main()
{
    int a = 21;
    std::thread t(func, a);
    t.join();
    std::cout<<a<<std::endl;
}
```

Как этим воспользоваться для потоков?

А так не скомпилируется!

```
int func(int& a) {
    a = 42;
    return 0;
}
int main()
{
    int a = 21;
    std::thread t(func, a);
    t.join();
    std::cout<<a<<std::endl;
}
```

Как этим воспользоваться для потоков?

Используем std::ref

```
int func(int& a) {
    a = 42;
    return 0;
}
int main()
{
    int a = 21;
    std::thread t(func, std::ref(a));
    t.join();
    std::cout<<a<<std::endl;
}
```

Как этим воспользоваться для потоков?

А что будет, если сделать так?

```
int func(int a) {
    a = 42;
    return 0;
}
int main()
{
    int a = 21;
    std::thread t(func, std::ref(a));
    t.join();
    std::cout<<a<<std::endl;
}
```

А как получить возвращаемый результат

Есть недостатки...

- Способ, описанный выше не есть в полной мере возвращение значения

А как получить возвращаемый результат

Есть недостатки...

- Способ, описанный выше не есть в полной мере возвращение значения
- Недостаток состоит в том, что функция, выполняемая в потоке, ссылается на переменную, расположенную в месте вызова функции. Если функция, запустившая поток, завершила работу, подобная переменная теряет смысл, вы приходите к неопределенному поведению

А как получить возвращаемый результат

Есть недостатки...

- Способ, описанный выше не есть в полной мере возвращение значения
- Недостаток состоит в том, что функция, выполняемая в потоке, ссылается на переменную, расположенную в месте вызова функции. Если функция, запустившая поток, завершила работу, подобная переменная теряет смысл, вы приходите к неопределенному поведению
- Можно делать переменную глобальной, но это усложняет задачу

А как получить возвращаемый результат

Есть недостатки...

- Способ, описанный выше не есть в полной мере возвращение значения
- Недостаток состоит в том, что функция, выполняемая в потоке, ссылается на переменную, расположенную в месте вызова функции. Если функция, запустившая поток, завершила работу, подобная переменная теряет смысл, вы приходите к неопределенному поведению
- Можно делать переменную глобальной, но это усложняет задачу
- Решение: `std::future` и `std::promise`

А как получить возвращаемый результат

- Решение: `std::future` и `std::promise`

Каждый объект std::promise связан с объектом std::future . Это пара классов, один из которых (std::promise) отвечает за установку значения, а другой (std::future) – за его получение.

А как получить возвращаемый результат?

```
#include <future>
void func(int& a, std::promise<int> result) {
    a = 42;
    result.set_value(1);
}
int main()
{
    int a = 21;
    std::promise<int> result;
    auto future = result.get_future();
    std::thread t(func, std::ref(a),
        std::move(result));
    t.join();
    std::cout<<a<<"|"<<future.get()<<"\n";
}
```

Вопросы?

Раздел 3. План.

Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

- Введение, общая память и библиотеки
- Использование потоков, передача в них параметров и возврат значений (c++11)
- **Параллельная работа в общей памяти, механизмы обеспечения синхронизации (c++11)**
- `std::async` и суммирование вектора (c++11)
- Make и Makefile. Основы использования git.
- Модули (c++20)
- Параллельные алгоритмы STL (c++17)
- Введение в корутины (c++20 и немного c++23)

Потоки, асинхронность, а где же параллельность?

Давайте реализуем простой пример сложения двух векторов в параллельном режиме

Потоки, асинхронность, а где же параллельность?

Давайте реализуем простой пример сложения двух векторов в параллельном режиме

Этот пример исключительно технический, чтобы показать различные паттерны параллелизации, на деле он ускоряется плохо. Позже поговорим почему

Потоки, асинхронность, а где же параллельность?

```
std::random_device rd; std::mt19937 gen(rd());
std::uniform_real_distribution<double>
    uid(0., 1.);
constexpr size_t n = 32 * 1024 * 1024;
std::vector<double> a (n);
std::vector<double> b (n);
std::vector<double> c (n);
std::generate(a.begin(),a.end(),
    [&uid,&gen]()->double {return uid(gen);}
);
std::generate(b.begin(),b.end(),
    [&uid,&gen]()->double {return uid(gen);}
);
for (size_t i = 0; i != n; ++i) {
    c[i] = a[i] + b[i];
}
```

Параллельная реализация сложения

```
void sum(std::vector<double>& a,
         std::vector<double>& b,
         std::vector<double>& c,
         size_t start, size_t end) {
    for (size_t i = start; i != end; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

Параллельная реализация сложения

```
int main() {
    // vector declaration and generation here
    constexpr size_t tnumber = 4;
    std::vector<std::thread> threads(tnumber);
    for (size_t i = 0; i!=tnumber; ++i) {
          threads[i] = std::thread(sum,
          std::ref(a), std::ref(b), std::ref(c),
          i*n/tnumber, (i+1)*n/tnumber);
    }
    for (size_t i = 0; i!=tnumber; ++i) {
          threads[i].join();
    }
}
```

Параллельная реализация сложения

Примечание: задача слишком проста и упирается в скорость работы памяти. Ускорения скорей всего почти не будет

- Детально с проблемой скорости работы памяти мы познакомимся на другой лекции

Параллельная реализация сложения

Какие могут быть недостатки у такого подхода в общем случае?

Параллельная реализация сложения

Какие могут быть недостатки у такого подхода в общем случае?

- Здесь мы сами разделили задание на предположительно равные части. В общем случае они могут не быть равными

Параллельная реализация сложения

Какие могут быть недостатки у такого подхода в общем случае?

- Здесь мы сами разделили задание на предположительно равные части. В общем случае они могут не быть равными
- Более корректный вариант это позволить дочерним потокам самим брать задания из очереди заданий и выполнять их
- Попробуем реализовать и это (да еще и не одним способом)

(и, да, для сложения векторов подойдет и текущий подход, но это просто пример)

Что если завести общий счетчик?

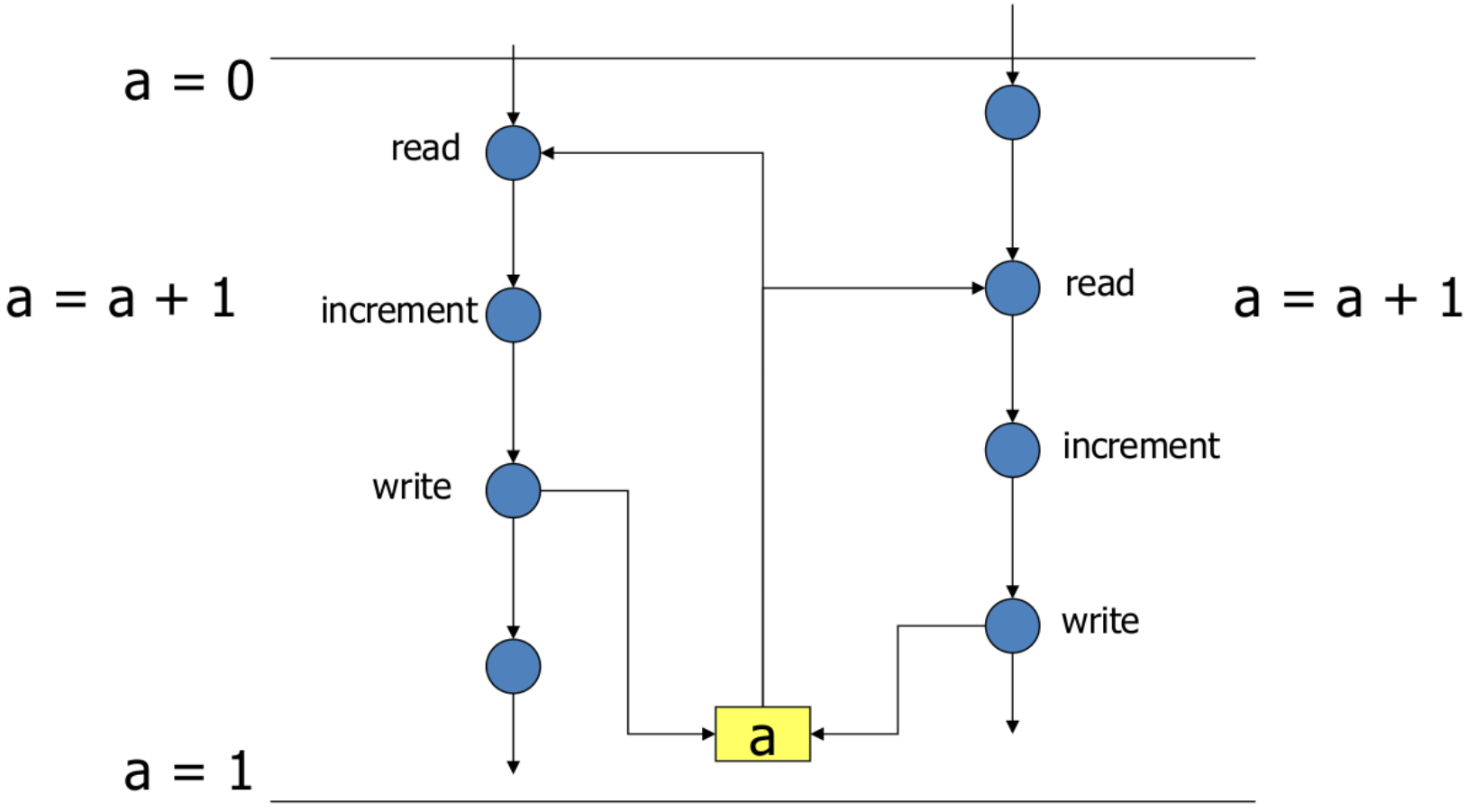
```
void sum(std::vector<double>& a,
         std::vector<double>& b,
         std::vector<double>& c,
         size_t& i, size_t n) {
    while (i < n) {
        int j = i++;
        c[j] = a[j] + b[j];
    }
}
```

Что если завести общий счетчик?

```
int main() {
    // vector declaration and generation here
    std::vector<std::thread> threads(4);
    size_t i = 0;
    for (size_t j = 0; j!=tnumber; ++j) {
          threads[j] = std::thread(sum,
          std::ref(a), std::ref(b), std::ref(c),
          std::ref(i), n);
    }
    for (size_t i = 0; i!=tnumber; ++i) {
          threads[i].join();
    }
}
```

Казалось бы, что могло пойти не так?

Казалось бы, что могло пойти не так?



Неделимые или атомарные операции

- Неделимой называется операция, в момент выполнения которой состояние общих переменных не может «наблюдаться» другими тредами
- В задаче, описанной выше, нам нужно добиться того, чтобы поток не смог прочитать переменную, изменяемую другим потоком

Различные способы обеспечения неделимости

- Атомарные переменные
- Барьерная синхронизация
- Механизм сигналов

Различные способы обеспечения неделимости

- **Атомарные переменные**
- Барьерная синхронизация
- Механизм сигналов

Атомарные переменные

- Атомарная переменная – переменная, операции изменения которой неделимы

Атомарные переменные

- Атомарная переменная – переменная, операции изменения которой неделимы
- Основываются на инструкциях процессора, обеспечивающих неделимость

Атомарные переменные

- Атомарная переменная – переменная, операции изменения которой неделимы
- Основываются на инструкциях процессора, обеспечивающих неделимость
- Ранее могли реализоваваться при помощи специфических для компиляторов инструкций типа `__sync_fetch_and_add`

Атомарные переменные

- Атомарная переменная – переменная, операции изменения которой неделимы
- Основываются на инструкциях процессора, обеспечивающих неделимость
- Ранее могли реализоваваться при помощи специфических для компиляторов инструкций типа `__sync_fetch_and_add`
- Имеют полную поддержку в C++11 при помощи `std::atomic`

Атомарные переменные

- Подключение `#include <atomic>`

Атомарные переменные

- Подключение `#include <atomic>`
- Объявление `std::atomic<int> i` (можно использовать различные типы)

Атомарные переменные

- Подключение `#include <atomic>`
- Объявление `std::atomic<int> i` (можно использовать различные типы)
- Работают все возможные операции инкремента, присваивания от обычных чисел `i++`, `i = 1`. Все эти операции являются атомарными. Аналогично, можно базовому типу присвоить атомарный.

Атомарные переменные

- Подключение `#include <atomic>`
- Объявление `std::atomic<int> i` (можно использовать различные типы)
- Работают все возможные операции инкремента, присваивания от обычных чисел `i++`, `i = 1`. Все эти операции являются атомарными. Аналогично, можно базовому типу присвоить атомарный.
- Почему не работает вот такая инициализация?
  `std::atomic<int> i = 0;`

Атомарные переменные

- Подключение `#include <atomic>`
- Объявление `std::atomic<int> i` (можно использовать различные типы)
- Работают все возможные операции инкремента, присваивания от обычных чисел `i++`, `i = 1`. Все эти операции являются атомарными. Аналогично, можно базовому типу присвоить атомарный.
- Почему не работает вот такая инициализация? `std::atomic<int> i = 0;`
- Можно вот так. Что это? `std::atomic<int> i = {};`

Атомарные переменные

```
#include <atomic>

void sum(std::vector<double>& a,
         std::vector<double>& b,
         std::vector<double>& c,
         std::atomic<size_t>& i, size_t n) {
    while (true) {
        int j = i++;
        if (j >= n) {
            break;
        }
        c[j] = a[j] + b[j];
    }

}
```

Атомарные переменные

```
int main() {
    // vector declaration and generation here
    std::vector<std::thread> threads(4);
    std::atomic<size_t> i = {};
    for (size_t j = 0; j!=tnumber; ++j) {
          threads[j] = std::thread(sum,
          std::ref(a), std::ref(b), std::ref(c),
          std::ref(i), n);
    }
    for (size_t i = 0; i!=tnumber; ++i) {
          threads[i].join();
    }
}
```

Различные способы обеспечения неделимости

- Атомарные переменные
- **Барьерная синхронизация**
- Механизм сигналов

Барьерная синхронизация (мьютексы)

- Мьютекс – примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода

Барьерная синхронизация (мьютексы)

- Мьютекс – примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода
- Классический мьютекс можно представить в виде переменной, которая может находиться в двух состояниях: в заблокированном и в незаблокированном. При входе в свою критическую секцию поток вызывает функцию перевода мьютекса в заблокированное состояние, при этом поток блокируется до освобождения мьютекса, если другой поток уже владеет им. При выходе из критической секции поток вызывает функцию перевода мьютекса в незаблокированное состояние

Барьерная синхронизация (мьютексы)

- Мьютекс – примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода
- Классический мьютекс можно представить в виде переменной, которая может находиться в двух состояниях: в заблокированном и в незаблокированном. При входе в свою критическую секцию поток вызывает функцию перевода мьютекса в заблокированное состояние, при этом поток блокируется до освобождения мьютекса, если другой поток уже владеет им. При выходе из критической секции поток вызывает функцию перевода мьютекса в незаблокированное состояние
- Имеют полную поддержку в C++11 при помощи `std::mutex`

Барьерная синхронизация (мьютексы)

- Подключение `#include <mutex>`

Барьерная синхронизация (мьютексы)

- Подключение `#include <mutex>`
- Объявление `std::mutex m`

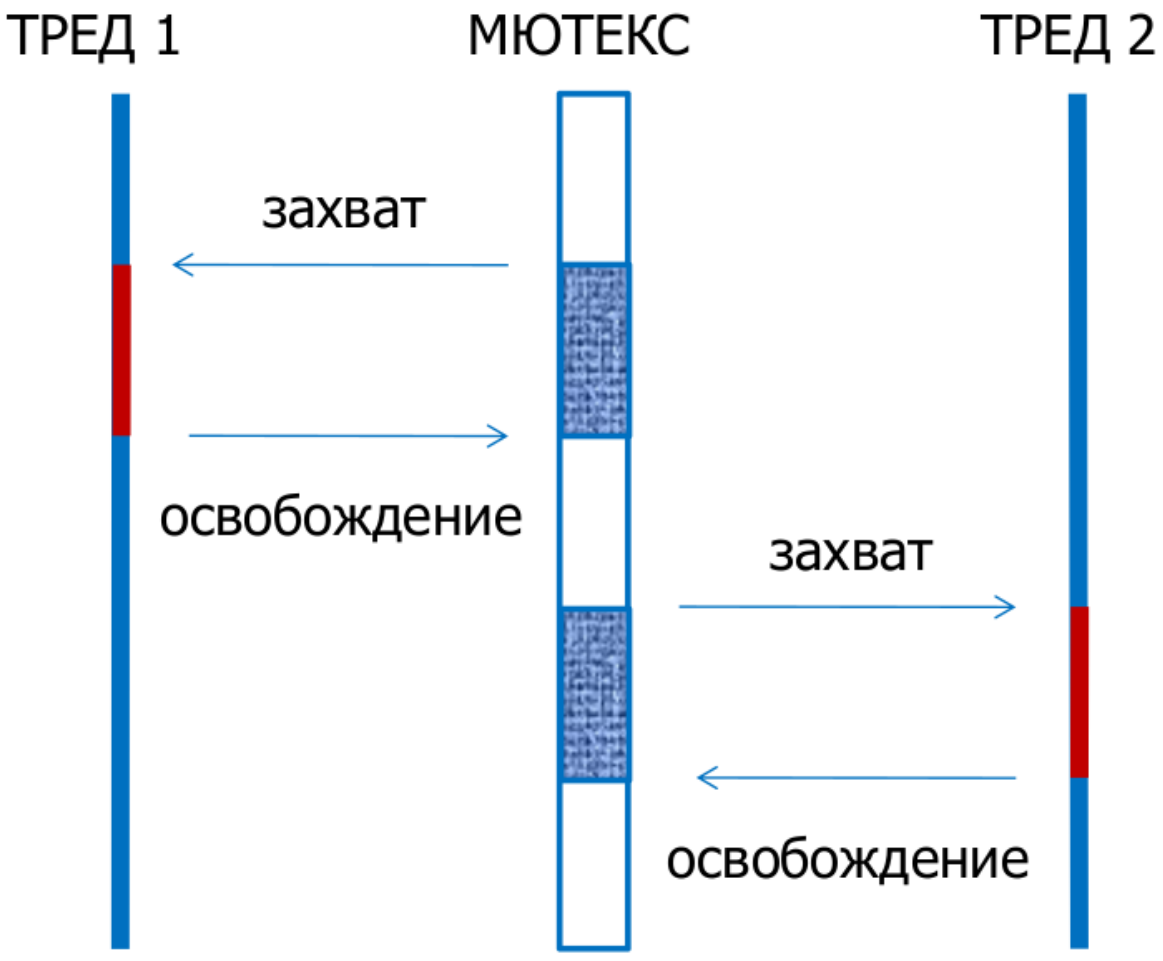
Барьерная синхронизация (мьютексы)

- Подключение `#include <mutex>`
- Объявление `std::mutex m`
- Захват мьютекса `m.lock()`, освобождение `m.unlock()`, попытка захвата `m.try_lock()`

Барьерная синхронизация (мьютексы)

- Подключение `#include <mutex>`
- Объявление `std::mutex m`
- Захват мьютекса `m.lock()`, освобождение `m.unlock()`, попытка захвата `m.try_lock()`
- Важно не забывать освобождать мьютекс, не пытаться захватить повторно той же веткой

Барьерная синхронизация (мьютексы)



Барьерная синхронизация (мьютексы)

```
#include <mutex>

void sum(std::vector<double>& a,
         std::vector<double>& b,
         std::vector<double>& c,
         std::mutex& m, size_t& i, size_t n) {
    while (true) {
        m.lock();
        int j = i++;
        m.unlock();
        if (j >= n) {
            break;
        }
        c[j] = a[j] + b[j];
    }
}
```

Барьерная синхронизация (мьютексы)

```
int main() {
    // vector declaration and generation here
    std::vector<std::thread> threads(4);
    size_t i = 0;
    std::mutex m;
    for (size_t j = 0; j!=tnumber; ++j) {
          threads[j] = std::thread(sum,
          std::ref(a), std::ref(b), std::ref(c),
          std::ref(m), std::ref(i), n);
    }
    for (size_t i = 0; i!=tnumber; ++i) {
          threads[i].join();
    }
}
```

Варианты мьютексов

- Обычные мьютексы `std::mutex`
- Рекурсивные мьютексы `std::recursive_mutex` – позволяют потоку захватывать несколько раз, но столько же раз нужно освобождать
- Разделяемые мьютексы `std::shared_mutex` (C++17) – добавляет методы разделяемой блокировки для чтения; может быть сколько угодно раздлеляемых блокировок одновременно, но уникальная блокировка не совмещается с разделяемой
- Дополнительно мьютексы с возможностью пытаться устанавливать блокировку ограниченное время `std::timed_mutex` и аналоги

Барьерная синхронизация (мьютексы)

- Один из недостатков это необходимость прослеживать в коде, чтобы мьютекс обязательно был разблокирован

Барьерная синхронизация (мьютексы)

- Один из недостатков это необходимость прослеживать в коде, чтобы мьютекс обязательно был разблокирован

Улучшение – `std::lock_guard`

Барьерная синхронизация (мьютексы)

- Один из недостатков это необходимость прослеживать в коде, чтобы мьютекс обязательно был разблокирован

Улучшение – `std::lock_guard`

- Следует концепции RAII – resource allocation is initialization
- Освобождает мьютекс при деаллокации

Барьерная синхронизация (мьютексы)

```
#include <mutex>

void sum(std::vector<double>& a,
         std::vector<double>& b,
         std::vector<double>& c,
         std::mutex& m, size_t& i, size_t n) {
    while (true) {
        int j;
        { // works without these brackets...
          std::lock_guard<std::mutex> lock(m);
          j = i++;
        } //...but worse. why?
        if (j >= n) break;
        c[j] = a[j] + b[j];
    }
}
```

Барьерная синхронизация (мьютексы)

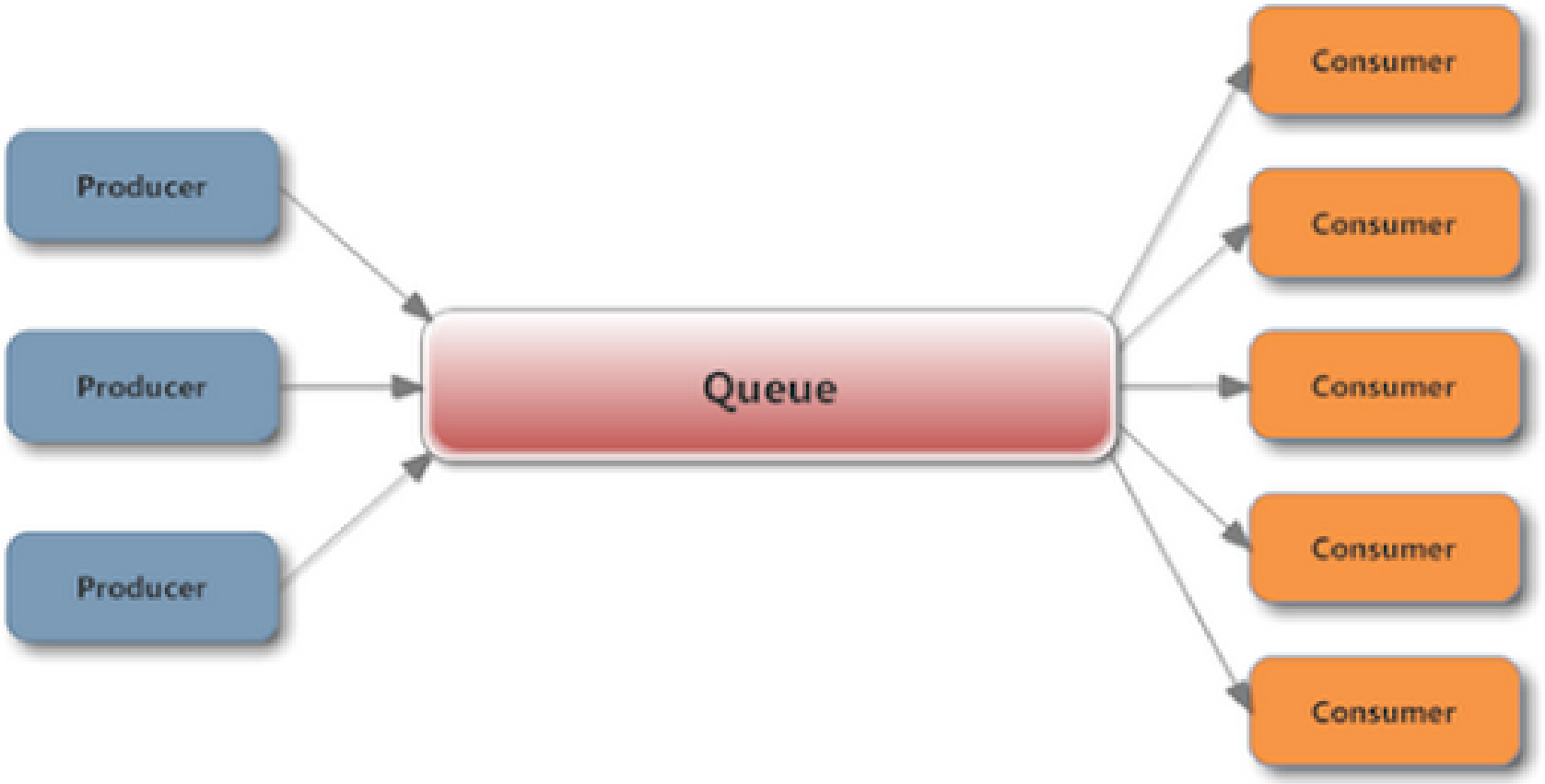
```
int main() {
    // vector declaration and generation here
    std::vector<std::thread> threads(4);
    size_t i = 0;
    std::mutex m;
    for (size_t j = 0; j!=tnumber; ++j) {
          threads[j] = std::thread(sum,
          std::ref(a), std::ref(b), std::ref(c),
          std::ref(m), std::ref(i), n);
    }
    for (size_t i = 0; i!=tnumber; ++i) {
          threads[i].join();
    }
}
```

Барьерная синхронизация (мьютексы)

Варианты замков

- Обычные `std::lock_guard`
- Многоразовые `std::unique_lock` – как и мьютекс, поддерживают операции lock и unlock, освобождают при вызове деструктора
- Разделяемые `std::shared_lock` (C++14) – для разделяемых мьютексов
- Расширенные `std::scoped_lock` (C++17) – как обычный, но сразу можно захватить несколько мьютексов

Паттерн producer-consumer



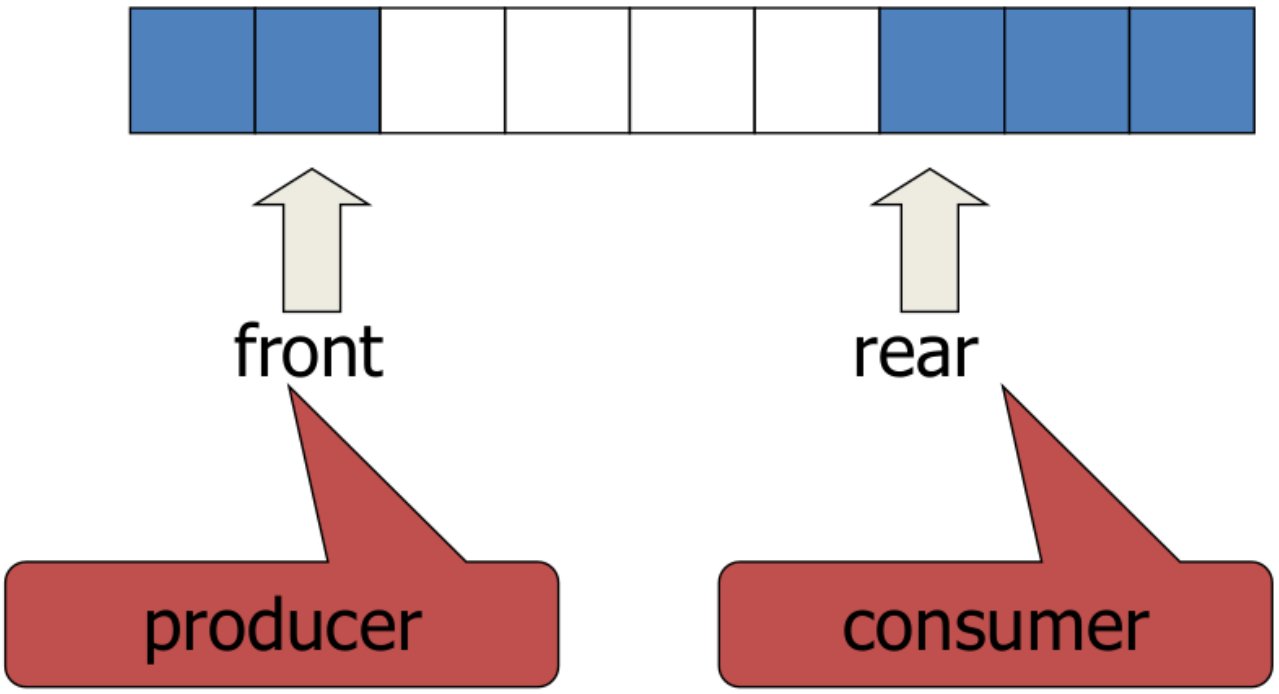
Паттерн producer-consumer

Отступление – нужно реализовать очередь

Паттерн producer-consumer

Отступление – нужно реализовать очередь

- Разработать руками кольцевой буфер (pure C way)



Паттерн producer-consumer

Отступление – нужно реализовать очередь

- Использовать `std::vector`. Чем плохо?

Паттерн producer-consumer

Отступление – нужно реализовать очередь

- Использовать `std::vector`. Чем плохо?
- Данный тип контейнера предлагает последовательную память. Класть в конец в нормально (особенно если сделать лимит на длину), забирать из начала приводит к реаллокации

Паттерн producer-consumer

Отступление – нужно реализовать очередь

- Использовать `std::list`. Чем плохо?

Паттерн producer-consumer

Отступление – нужно реализовать очередь

- Использовать `std::list`. Чем плохо?
- `std::list` эффективно работает со списками, в которые можно класть и забирать с разных сторон, но добавление каждого нового элемента это выделение памяти

Паттерн producer-consumer

Отступление – нужно реализовать очередь

- Использовать `std::deque` (двусторонняя очередь).
- `std::deque` – наверное, лучший базовый контейнер для хранения очередь. Он выделяет память блоками на фиксированное количество элементов, используя `list` для хранения цепочки блоков, поэтому время на добавление или освобождение ограничено константой, а выделения памяти происходят существенно реже

Паттерн producer-consumer

Отступление – нужно реализовать очередь

- Использовать `std::queue` (односторонняя очередь).
- `std::queue` – контейнер более высокого уровня, реализующийся над `std::deque` (по умолчанию) или `std::list`. Предоставляет пользователю операции добавления с одной стороны и взятия с другой. Ничего не добавляет относительно `std::deque`, но концептуально лучше подходит для нашей задачи, оставляя меньше возможностей ошибиться и взять элемент не с той стороны

Паттерн producer-consumer

```
constexpr size_t n = 32 * 1024 * 1024;
std::vector<double> a(n);
std::vector<double> b(n);
std::vector<double> c(n);
std::mutex m;
std::queue<size_t> q;
bool stop = false;
```

Паттерн producer-consumer

```
void consumer() {
  while (true) {
    size_t j;
    {
      std::unique_lock<std::mutex> ul(m);
      if (stop) break;
      if (q.empty()) continue;
      j = q.front();
      q.pop();
    }
    c[j] = a[j] + b[j];
  }
}
```

Паттерн producer-consumer

```
void producer() {
    for (size_t i = 0; i!=n; ++i) {
        std::lock_guard<std::mutex> ul(m);
        q.push(i);
    }
    stop = true;
}
```

Паттерн producer-consumer

```
int main() {
    // filling vectors here
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j] = std::thread(consumer);
    }
    auto threadP = std::thread(producer);
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j].join();
    }
    threadP.join();
}
```

Паттерн producer-consumer

Какая здесь проблема?

Паттерн producer-consumer

Какая здесь проблема?

- Постоянная борьба за захват мьютекса. Нужен механизм сигналов!

Различные способы обеспечения неделимости

- Атомарные переменные
- Барьерная синхронизация
- **Механизм сигналов**

Механизм сигналов

- Недостаток мьютексов – попытка захватить его может занимать ресурсы, поток не знает, когда мьютекс освободится

Механизм сигналов

- Недостаток мьютексов – попытка захватить его может занимать ресурсы, поток не знает, когда мьютекс освободится
- Требуется механизм сигналов – чтобы один поток мог сообщить другому, что что-то случилось. Для этого используются так называемые переменные состояния

Механизм сигналов

- Недостаток мьютексов – попытка захватить его может занимать ресурсы, поток не знает, когда мьютекс освободится
- Требуется механизм сигналов – чтобы один поток мог сообщить другому, что что-то случилось. Для этого используются так называемые переменные состояния
- Полная поддержка в C++11 через `std::condition_variable`

Механизм сигналов

- Подключение: `#include condition_variable`, объявление `std::condition_variable cv`

Механизм сигналов

- Подключение: `#include condition_variable`, объявление `std::condition_variable cv`
- Использование через `cv.wait(unique lock, condition);` То есть перед ожиданием поток должен запереть мьютекс при помощи unique lock, после чего он ждет сигнала. condition – это условие, которое проверяется в момент получения сигнала на то, стоит ли потоку просыпаться или же поспать еще

Механизм сигналов

- Подключение: `#include condition_variable`, объявление `std::condition_variable cv`
- Использование через `cv.wait(unique lock, condition);` То есть перед ожиданием поток должен запереть мьютекс при помощи unique lock, после чего он ждет сигнала. condition – это условие, которое проверяется в момент получения сигнала на то, стоит ли потоку просыпаться или же поспать еще
- Другая ветка использует `cv.notify_one` или `cv.notify_all`, чтобы пробудить один или несколько ожидающих потоков

Механизм сигналов

- Подключение: `#include condition_variable`, объявление `std::condition_variable cv`
- Использование через `cv.wait(unique lock, condition);` То есть перед ожиданием поток должен запереть мьютекс при помощи unique lock, после чего он ждет сигнала. condition – это условие, которое проверяется в момент получения сигнала на то, стоит ли потоку просыпаться или же поспать еще
- Другая ветка использует `cv.notify_one` или `cv.notify_all`, чтобы пробудить один или несколько ожидающих потоков
- Попробуем написать программу (сначала плохо)

Механизм сигналов

```
#include <mutex>
#include <condition_variable>
condition_variable cv;
```

Паттерн producer-consumer

```
std::condition_variable cv;
constexpr size_t n = 32 * 1024 * 1024;
std::vector<double> a(n);
std::vector<double> b(n);
std::vector<double> c(n);
std::mutex m;
std::queue<size_t> q;
bool stop = false;
```

Паттерн producer-consumer

```
void consumer() {
  while (true) {
    size_t j;
    {
      std::unique_lock<std::mutex> ul(m);
      cv.wait(ul);
      if (stop) break;
      if (q.empty()) continue;
      j = q.front();
      q.pop();
    }
    c[j] = a[j] + b[j];
  }
}
```

Паттерн producer-consumer

```
void producer() {
    for (size_t i = 0; i!=n; ++i) {
        std::lock_guard<std::mutex> ul(m);
        q.push(i);
        cv.notify_one();
    }
    stop = true;
    cv.notify_all();
}
```

Паттерн producer-consumer

```
int main() {
    // filling vectors here
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j] = std::thread(consumer);
    }
    auto threadP = std::thread(producer);
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j].join();
    }
    threadP.join();
}
```

Механизм сигналов

Какие же могут быть у этой программы проблемы?

Механизм сигналов

Механизм сигналов

Какие же могут быть у этой программы проблемы?

- Поток может рано приступить к работе! Продюсер будит консьюмеров, когда они еще не дошли до точки, на которой будут спать. Потом будить их некому, и программа зависает

Механизм сигналов

Какие же могут быть у этой программы проблемы?

- Поток может рано приступить к работе! Продюсер будит консьюмеров, когда они еще не дошли до точки, на которой будут спать. Потом будить их некому, и программа зависает
- Spurious wakeup – ситуация, когда поток просыпается, хотя ему никто не присылал сигнала! (чуть позже)

Механизм сигналов

Какие же могут быть у этой программы проблемы?

- Поток может рано приступить к работе! Продюсер будит консьюмеров, когда они еще не дошли до точки, на которой будут спать. Потом будить их некому, и программа зависает
- Spurious wakeup – ситуация, когда поток просыпается, хотя ему никто не присылал сигнала! (чуть позже)

Общее правило – не использовать переменные состояния без условия!

Механизм сигналов

Как работает переменная состояния с условием?

```
std::unique_lock<std::mutex> ul(m);
cv.wait(ul, [](){return !start;});
```

Механизм сигналов

Как работает переменная состояния с условием?

```
std::unique_lock<std::mutex> ul(m);
cv.wait(ul, [](){return !start;});
```

- В тот момент, когда код выходит на эту строчку, мьютекс m должен быть захвачен (иначе неопределенное поведение)

Механизм сигналов

Как работает переменная состояния с условием?

```
std::unique_lock<std::mutex> ul(m);
cv.wait(ul, [](){return !start;});
```

- В тот момент, когда код выходит на эту строчку, мьютекс m должен быть захвачен (иначе неопределенное поведение)
- Программа атомарно
  - проверяет, выполняется ли условие,
  - если нет, то отпускает мьютекс и встает на ожидание сигнала
  - если да, то проскакивает дальше

Механизм сигналов

Как работает переменная состояния с условием?

```
std::unique_lock<std::mutex> ul(m);
cv.wait(ul, [](){return !start;});
```

- Программа ~~атомарно~~
  - проверяет, выполняется ли условие,
  - если нет, то отпускает мьютекс и встает на ожидание сигнала
  - если да, то проскакивает дальше
- Само условие записано в виде лямбда-функции, то есть функции, которую не пришлось определять отдельно и давать ей имя. Синтаксис лямбда-функции начинается с [] (внутри скобок могут быть символы захвата, но это потом), затем () то есть список параметров функции (здесь не нужен), затем тело функции

Механизм сигналов

Как работает переменная состояния с условием?

```
std::unique_lock<std::mutex> ul(m);
cv.wait(ul, [](){return !start;});
```

- В момент пробуждения (приход сообщения) программа
  - захватывает мьютекс (эквивалентно `ul.lock()`) -
  проверяет, выполняется ли условие,
  - если нет, то засыпает снова, освобождая mutex

Механизм сигналов

Как работает переменная состояния с условием?

```
std::unique_lock<std::mutex> ul(m);
cv.wait(ul, [](){return !start;});
```

- В момент пробуждения (приход сообщения) программа
  - захватывает мьютекс (эквивалентно `ul.lock()`) -
  проверяет, выполняется ли условие,
  - если нет, то засыпает снова, освобождая mutex
- Проблема Spurious wakeup – поток может проснуться, хотя ему никто не присылал сигнал

Паттерн producer-consumer

```
std::condition_variable cv;
constexpr size_t n = 32 * 1024 * 1024;
std::vector<double> a(n);
std::vector<double> b(n);
std::vector<double> c(n);
std::mutex m;
std::queue<size_t> q;
bool stop = false;
```

Паттерн producer-consumer

```
void consumer() {
  while (true) {
    size_t j;
    {
      std::unique_lock<std::mutex> ul(m);
      cv.wait(ul,[](){
        return ((!q.empty()) || stop);
      });
      if (q.empty()) break; //?? why not stop
      j = q.front();
      q.pop();
    }
    c[j] = a[j] + b[j];
  }
}
```

Паттерн producer-consumer

```
void producer() {
    for (size_t i = 0; i!=n; ++i) {
        std::lock_guard<std::mutex> ul(m);
        q.push(i);
        cv.notify_one();
    }
    stop = true;      // <--- there is a potential e
    cv.notify_all();
}
```

Паттерн producer-consumer

```
int main() {
    // filling vectors here
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j] = std::thread(consumer);
    }
    auto threadP = std::thread(producer);
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j].join();
    }
    threadP.join();
}
```

Паттерн producer-consumer

Какую проблему мы не учли в этом решении?

Паттерн producer-consumer

Какую проблему мы не учли в этом решении?

- consumer проснулся
- consumer захватил mutex
- consumer проверил stop, не выполняется
- producer изменил stop
- producer послал сигнал
- consumer встал на ожидание, но больше не получит сигнал

Как это исправить?

Паттерн producer-consumer

Какую проблему мы не учли в этом решении?

- consumer проснулся
- consumer захватил mutex
- consumer проверил stop, не выполняется
- producer изменил stop
- producer послал сигнал
- consumer встал на ожидание, но больше не получит сигнал

Как это исправить?

Менять переменную под мьютексом

`https://www.modernescpp.com/index.php/c-core-guidelines-be-aware-of-the-traps\-of-condition-variables/`

Паттерн producer-consumer

```
std::condition_variable cv;
constexpr size_t n = 32 * 1024 * 1024;
std::vector<double> a(n);
std::vector<double> b(n);
std::vector<double> c(n);
std::mutex m;
std::queue<size_t> q;
bool stop = false;
```

Паттерн producer-consumer

```
void consumer() {
  while (true) {
    size_t j;
    {
      std::unique_lock<std::mutex> ul(m);
      cv.wait(ul,[](){
        return ((!q.empty()) || stop);
      });
      if (q.empty()) break; //?? why not stop
      j = q.front();
      q.pop();
    }
    c[j] = a[j] + b[j];
  }
}
```

Паттерн producer-consumer

```
void producer() {
    for (size_t i = 0; i!=n; ++i) {
        std::lock_guard<std::mutex> ul(m);
        q.push(i);
        cv.notify_one();
    }
    {
        std::lock_guard<std::mutex> ul(m);
        stop = true;
    }
    cv.notify_all();
}
```

Паттерн producer-consumer

```
int main() {
    // filling vectors here
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j] = std::thread(consumer);
    }
    auto threadP = std::thread(producer);
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j].join();
    }
    threadP.join();
}
```

Паттерн producer-consumer

Какую проблему мы не учли в этом решении?

Паттерн producer-consumer

Какую проблему мы не учли в этом решении?

Паттерн producer-consumer

Какую проблему мы не учли в этом решении?

- Горшочек, не вари!
- Если производитель готовит объекты быстрее, чем их прорабатывают потребители, то очередь может расти безгранично, и может произойти неожиданное использование памяти. Это надо решать!

Паттерн producer-consumer

```
std::condition_variable cv;
std::condition_variable cvf;
constexpr size_t n = 32 * 1024 * 1024 ;
constexpr size_t ql = 1024;
std::vector<double> a (n);
std::vector<double> b (n);
std::vector<double> c (n);
std::mutex m;
std::queue<size_t> q;
bool stop = false;
```

Паттерн producer-consumer

```
void consumer() {
  while (true) {
    size_t j;
    {
      std::unique_lock<std::mutex> ul(m);
      cv.wait(ul,[](){
        return ((!q.empty()) || stop);
      });
      if (q.empty()) break;
      j = q.front();
      q.pop();
      cvf.notify_one();
    }
    c[j] = a[j] + b[j];
  }
}
```

Паттерн producer-consumer

```
void producer() {
    for (size_t i = 0; i!=n; ++i) {
        std::unique_lock<std::mutex> ul(m);
        cvf.wait(ul,[](){
            return (q.size() < ql);
        });
        q.push(i);
        cv.notify_one();
    }
    {
        std::lock_guard<std::mutex> ul(m);
        stop = true;
    }
    cv.notify_all();
}
```

Паттерн producer-consumer

```
int main() {
    // filling vectors here
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j] = std::thread(consumer);
    }
    auto threadP = std::thread(producer);
    for (size_t j = 0; j!=tnumber; ++j) {
        threads[j].join();
    }
    threadP.join();
}
```

Паттерн producer-consumer

- Вот только теперь этот код хорошо прорабатывает паттерн producer-consumer без лишних ожиданий, загрузки процессора или переполнения памяти. При этом мы использовали одного поставщика, но он прекрасно справляется и с многими.
- И да, это было совершенно не нужно для нашего игрушечного примера про сложение векторов.

Основные паттерны разделения работы

Априорное разделение

- Мы изначально решаем, какую часть будет делать который поток
- Плюсы – простота реализации

Основные паттерны разделения работы

Априорное разделение

- Мы изначально решаем, какую часть будет делать который поток
- Плюсы – простота реализации
- Почему же не ограничиться этим способом и использовать его всегда?

Основные паттерны разделения работы

Априорное разделение

- Мы изначально решаем, какую часть будет делать который поток
- Плюсы – простота реализации
- Минусы – крайне ограниченная применимость ввиду невозможности балансировать нагрузку

Основные паттерны разделения работы

Lock-free

- Использование атомарных переменных в качестве счетчиков
- Плюсы – простота реализации
- Минусы – ограниченная применимость, поскольку подходит только когда можно явно выделить какой-либо счетчик

Основные паттерны разделения работы

Мьютексы

- Использование мьютексов для критических секций
- Плюсы – большая универсальность, чем с атомарными счетчиками, относительная простота
- Минусы – борьба многих потоков за мьютекс это большие накладные расходы. Не предоставляет полной универсальности, как следующий вариант.

Основные паттерны разделения работы

Producer-consumer с сигналами

- Плюсы – универсальность
- Минусы – сложность реализации; накладные расходы для очень простых задач. Имеет смысл выбирать именно его, если требуется оптимизация, а выполнение работы поставщиком или работником существенно превышает инкремент счетчиков и операции по работе с очередью.

Потоки, асинхронность, а где же параллельность?

Мы реализовали простой пример сложения двух векторов в параллельном режиме

Этот пример исключительно технический, чтобы показать различные паттерны параллелизации, на деле он ускоряется плохо. Почему?

Потоки, асинхронность, а где же параллельность?

Мы реализовали простой пример сложения двух векторов в параллельном режиме

Этот пример исключительно технический, чтобы показать различные паттерны параллелизации, на деле он ускоряется плохо. Почему?

- Ограничены скоростью работы памяти. В цикле одна операция, два чтения и одна запись.

Потоки, асинхронность, а где же параллельность?

Мы реализовали простой пример сложения двух векторов в параллельном режиме

Этот пример исключительно технический, чтобы показать различные паттерны параллелизации, на деле он ускоряется плохо. Почему?

- Ограничены скоростью работы памяти. В цикле одна операция, два чтения и одна запись.
- В следующем разделе мы разберем пример, не упирающийся (настолько) в память. И заодно разберем принцип работы std::async

Вопросы?

Раздел 3. План.

Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

- Введение, общая память и библиотеки
- Использование потоков, передача в них параметров и возврат значений (c++11)
- Параллельная работа в общей памяти, механизмы обеспечения синхронизации (c++11)
- **`std::async` и суммирование вектора (c++11)**
- Make и Makefile. Основы использования git.
- Модули (c++20)
- Параллельные алгоритмы STL (c++17)
- Введение в корутины (c++20 и немного c++23)

std::async

Если нужно более простое возвращение значения, возможно, имеет смысл воспользоваться `std::async`

std::async

```
#include <future>
int func(int& a) {
    a = 42;
    return 1;
}
int main()
{
    int a = 21;
    std::future<int> future =
        std::async(func, std::ref(a));
    do_other_stuff();
    std::cout<<a<<"|"<<future.get()<<"\n";
}
```

std::async

Если нужно более простое возвращение значения, возможно, имеет смысл воспользоваться `std::async`

- Тот же синтаксис запуска, что и у `std::thread`
- Возвращаемое значение имеет тип `std::future`
- Метод `get` у `future` блокирует до завершения асинхронного задания

std::async

Если нужно более простое возвращение значения, возможно, имеет смысл воспользоваться `std::async`

- Тот же синтаксис запуска, что и у `std::thread`
- Возвращаемое значение имеет тип `std::future`
- Метод `get` у `future` блокирует до завершения асинхронного задания

Все не совсем так, есть нюансы

Варианты запуска `std::async`

- `auto result = std::async(std::launch::deferred, function, parameters);` ничего не делает до тех пор, пока не произойдет вызов `result.get()`, в тот момент выполняя функцию в текущем потоке

Варианты запуска `std::async`

- `auto result = std::async(std::launch::deferred, function, parameters);` ничего не делает до тех пор, пока не произойдет вызов `result.get()`, в тот момент выполняя функцию в текущем потоке
- `auto result = std::async(std::launch::async, function, parameters);` заставляет выполниться в другом потоке

Варианты запуска `std::async`

- `auto result = std::async(std::launch::deferred, function, parameters);` ничего не делает до тех пор, пока не произойдет вызов `result.get()`, в тот момент выполняя функцию в текущем потоке
- `auto result = std::async(std::launch::async, function, parameters);` заставляет выполниться в другом потоке
- `auto result = std::async(std::launch::async | std::launch::deferred, function, parameters);` оставляет решение за библиотекой (поведение по умолчанию)

std::async

Если вызвать std::async без присвоения результата куда-либо, код будет заблокирован в этой точке до момента завершения вычисления функции. Почему?

std::async

Если вызвать std::async без присвоения результата куда-либо, код будет заблокирован в этой точке до момента завершения вычисления функции. Почему?

- Время существования задания на выполнение функции в `std::async` привязано к времени возвращенного им `std:future`. Если результат никуда не присвоен, то создается временный объект, у него вызывается деструктор, и система должна дождаться выполнения функции.

std::thread vs std::async

std::thread

- \+ Больше контроля за выполнением потоков («ручное управление»)
- \- Сложный доступ к результату выполнения
- \- Проблемы балансировки и перегрузки системы (oversubscription)

std::thread vs std::async

std::thread

- \+ Больше контроля за выполнением потоков («ручное управление»)
- \- Сложный доступ к результату выполнения
- \- Проблемы балансировки и перегрузки системы (oversubscription)

std::async

- \+ больше свободы системе в плане балансировки нагрузки (система решает запускать новый поток или нет)
- \+ легкий доступ к результату через std::future
- \- отсутствие низкоуровнего контроля

std::thread vs std::async

std::thread

- \+ Больше контроля за выполнением потоков («ручное управление»)
- \- Сложный доступ к результату выполнения
- \- Проблемы балансировки и перегрузки системы (oversubscription)

std::async

- \+ больше свободы системе в плане балансировки нагрузки (система решает запускать новый поток или нет)
- \+ легкий доступ к результату через std::future
- \- отсутствие низкоуровнего контроля

еще есть std::packaged_task...

Thread oversubscription

Oversubscription - ситуация, когда число потоков превышает число физически доступных ядер.

Thread oversubscription

Oversubscription - ситуация, когда число потоков превышает число физически доступных ядер.

- \+ Небольшое превышение может позитивно сказаться на производительности: когда один поток ожидает на операциях с памятью, другой выполняет вычисления. Это позволяет задействовать внутрипроцессорный параллелизм и, тем самым, ускорить выполнение программы.

Thread oversubscription

Oversubscription - ситуация, когда число потоков превышает число физически доступных ядер.

- \+ Небольшое превышение может позитивно сказаться на производительности: когда один поток ожидает на операциях с памятью, другой выполняет вычисления. Это позволяет задействовать внутрипроцессорный параллелизм и, тем самым, ускорить выполнение программы.
- \- Переключение контекста потока приводит к непродуктивным накладным расходам. Поэтому если число потоков намного превышает число ядер, то переключение контекста может существенно снизить производительность программы.

Thread oversubscription

Oversubscription - ситуация, когда число потоков превышает число физически доступных ядер.

- \+ Небольшое превышение может позитивно сказаться на производительности: когда один поток ожидает на операциях с памятью, другой выполняет вычисления. Это позволяет задействовать внутрипроцессорный параллелизм и, тем самым, ускорить выполнение программы.
- \- Разные потоки, как правило, работают с разными данными. Поэтому при переключении ядра между потоками из кэш-памяти вытесняются данные, с которыми работал поток. Когда поток опять ставится на выполнение, увеличивается время на доступ к данным, т.к. требуется их загрузка в кэш из основной памяти.

Пример – суммирование вектора

```
double sum_vector(std::vector<double>& a,
      size_t start, size_t end) {
      double res = 0.;
      for (size_t i = start; i!=end; i++) {
            res = res + a[i];
      }
      return res;
}
```

Пример – суммирование вектора

```
int main() {
  constexpr size_t n = 256 * 1024 * 1024;
  std::vector<double> a(n);
  // fill with random here
  auto start =
       std::chrono::steady_clock::now();
  double res = sum_vector(a, 0, n);
  auto end =
       std::chrono::steady_clock::now();
  auto diff = end-start;
  int time = std::chrono::duration_cast<
    std::chrono::milliseconds>(diff).count();
  std::cout << time << " ms" << std::endl;
}
```

Пример – суммирование вектора

Замеряем время (компилируем с -O3, AMD Ryzen 7 3750H, 4 cores

Пример – суммирование вектора

Замеряем время (компилируем с -O3, AMD Ryzen 7 3750H, 4 cores

- Последовательное выполнение – 222 ms

Пример – суммирование вектора

```
double sum_vector(std::vector<double>& a,
      size_t start, size_t end) {
      double res = 0.;
      for (size_t i = start; i!=end; i++) {
            res = res + a[i];
      }
      return res;
}
```

Пример – суммирование вектора

Теперь реализация с async (только содержимое main внутри замера времени)

```
double res = 0.;
size_t block = n/NTHREADS;
std::future<double> f[NTHREADS];
for (int i = 0; i!=NTHREADS; ++i) {
    f[i] = std::async(
        std::launch::async,
        sum_vector, std::ref(a),
        i*block, (i+1)*block);
}
for (int i = 0; i!=NTHREADS; ++i) {
    res += f[i].get();
}
```

Пример – суммирование вектора

Замеряем время (компилируем с -O3, AMD Ryzen 7 3750H, 4 cores

- Последовательное выполнение – 222 ms
- 2 потока – 140 ms

Пример – суммирование вектора

Замеряем время (компилируем с -O3, AMD Ryzen 7 3750H, 4 cores

- Последовательное выполнение – 222 ms
- 2 потока – 140 ms
- 4 потока – 103 ms

Пример – суммирование вектора

Замеряем время (компилируем с -O3, AMD Ryzen 7 3750H, 4 cores

- Последовательное выполнение – 222 ms
- 2 потока – 140 ms
- 4 потока – 103 ms
- 8 потоков – 87 ms (oversubscription)

Пример – суммирование вектора

Замеряем время (компилируем с -O3, AMD Ryzen 7 3750H, 4 cores

- Последовательное выполнение – 222 ms
- 2 потока – 140 ms
- 4 потока – 103 ms
- 8 потоков – 87 ms (oversubscription)
- В режиме `std::launch::deferred` ускорения не происходит. В режиме выбора за библиотекой на моем компьютеры результаты совпали с явно указанным `std::launch::async`

Пример – суммирование вектора

Делаем программу, собирающуюся в разных режимах

```
#ifndef NTHREADS
#define NTHREADS 4
#endif

int main() {
 ..........................................
#ifdef ASYNC
 double res = 0.;
  size_t block = n/NTHREADS;
  std::future<double> f[NTHREADS];
  for (int i = 0; i!=NTHREADS; ++i) {
```

Пример – суммирование вектора

Делаем программу, собирающуюся в разных режимах

```
            f[i] = std::async(
                std::launch::async,
                sum_vector, std::ref(a),
                i*block, (i+1)*block);
    }
    for (int i = 0; i!=NTHREADS; ++i) {
        res += f[i].get();
    }
#else
    double res = sum_vector(a, 0, n);
#endif
 ..........................................
}
```

Пример – суммирование вектора

Делаем программу, собирающуюся в разных режимах

- Программу теперь можно собирать через

```
g++ -DASYNC -DNTHREADS=8 program.cpp
  -o program -pthread
```

Вопросы?

Раздел 3. План.

Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

- Введение, общая память и библиотеки
- Использование потоков, передача в них параметров и возврат значений (c++11)
- Параллельная работа в общей памяти, механизмы обеспечения синхронизации (c++11)
- `std::async` и суммирование вектора (c++11)
- **Make и Makefile. Основы использования git.**
- Модули (c++20)
- Параллельные алгоритмы STL (c++17)
- Введение в корутины (c++20 и немного c++23)

Использование make и Makefile

- Утилита make обычно устанавливается через менеджеры пакетов, например, apt-get install make. Чтобы запустить make в командной строке, нужно в той же папке иметь файл, так и называющийся, Makefile

Использование make и Makefile

- Утилита make обычно устанавливается через менеджеры пакетов, например, apt-get install make. Чтобы запустить make в командной строке, нужно в той же папке иметь файл, так и называющийся, Makefile

Использование make и Makefile

- Утилита make обычно устанавливается через менеджеры пакетов, например, apt-get install make. Чтобы запустить make в командной строке, нужно в той же папке иметь файл, так и называющийся, Makefile

hellomake.cpp

```
#include <hellofunc.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.cpp

```
#include <iostream>
#include <hellofunc.h>

void myPrintHelloMake(void) {

  std::cout<<"Hello makefiles!\n";

  return;
}
```

hellofunc.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

Использование make и Makefile

- Утилита make обычно устанавливается через менеджеры пакетов, например, apt-get install make. Чтобы запустить make в командной строке, нужно в той же папке иметь файл, так и называющийся, Makefile

hellomake.cpp

```
#include <hellofunc.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.cpp

```
#include <iostream>
#include <hellofunc.h>

void myPrintHelloMake(void) {

  std::cout<<"Hello makefiles!\n";

  return;
}
```

hellofunc.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

Обычно вы бы собрали этот код через

```
g++
-o hellomake hellomake.cpp hellofunc.cpp -I.
```

Использование make и Makefile

hellomake.cpp

```
#include <hellofunc.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.cpp

```
#include <iostream>
#include <hellofunc.h>

void myPrintHelloMake(void) {

  std::cout<<"Hello makefiles!\n";

  return;
}
```

hellofunc.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

Makefile. Вариант 1:

```
hellomake: hellomake.cpp hellofunc.cpp
    g++ -o hellomake hellomake.cpp hellofunc.cpp -I.
```

(внимание: отступ перед g++ это табуляция!)

Что важно - повторные вызов make приведет к пересборке только если изменился один из указанных файлов

Использование make и Makefile

hellomake.cpp

```
#include <hellofunc.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.cpp

```
#include <iostream>
#include <hellofunc.h>

void myPrintHelloMake(void) {

  std::cout<<"Hello makefiles!\n";

  return;
}
```

hellofunc.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

Makefile. Вариант 2, отделяем компиляцию и линковку:

```
CC=g++
CFLAGS=-I .
hellomake: hellomake.o hellofunc.o
	${CC} -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.cpp
	${CC} -c -o hellomake.o hellomake.cpp ${CFLAGS}

hellofunc.o: hellofunc.cpp
	${CC} -c -o hellofunc.o hellofunc.cpp ${CFLAGS}
```

Использование make и Makefile

hellomake.cpp

```
#include <hellofunc.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.cpp

```
#include <iostream>
#include <hellofunc.h>

void myPrintHelloMake(void) {

  std::cout<<"Hello makefiles!\n";

  return;
}
```

hellofunc.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

Makefile. Вариант 3, упрощаем код ссылаясь на цель и зависимость:

```
CC=g++
CFLAGS=-I .
OBJ = hellomake.o hellofunc.o
hellomake: ${OBJ}
  ${CC} -o $@ $^

hellomake.o: hellomake.cpp
  ${CC} -c -o $@ $^ ${CFLAGS}
hellofunc.o: hellofunc.cpp
  ${CC} -c -o $@ $^ ${CFLAGS}
```

Использование make и Makefile

hellomake.cpp

```
#include <hellofunc.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.cpp

```
#include <iostream>
#include <hellofunc.h>

void myPrintHelloMake(void) {

  std::cout<<"Hello makefiles!\n";

  return;
}
```

hellofunc.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

Makefile. Вариант 4, убираем повторяющиеся строки

```
CC=g++
CFLAGS=-I .
OBJ = hellomake.o hellofunc.o

hellomake: ${OBJ}
	${CC} -o $@ $^

%.o: %.cpp
	${CC} -c -o $@ $^ ${CFLAGS}
```

Использование make и Makefile

hellomake.cpp

```
#include <hellofunc.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.cpp

```
#include <iostream>
#include <hellofunc.h>

void myPrintHelloMake(void) {

  std::cout<<"Hello makefiles!\n";

  return;
}
```

hellofunc.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

Makefile. Вариант 5, добавляем зависимость от заголовочного файла

```
CC=g++
CFLAGS=-I .
DEPS = hellofunc.h
OBJ = hellomake.o hellofunc.o

hellomake: ${OBJ}
  ${CC} -o $@ $^

%.o: %.cpp ${DEPS}
  ${CC} -c -o $@ $< ${CFLAGS}
```

Использование make и Makefile

Makefile. Вариант 6, добавляем команду для очистки

```
CC=g++
CFLAGS=-I .
DEPS = hellofunc.h
OBJ = hellomake.o hellofunc.o

hellomake: ${OBJ}
	${CC} -o $@ $^

%.o: %.cpp ${DEPS}
	${CC} -c -o $@ $< ${CFLAGS}

.PHONY: clean

clean:
	rm -f *.o hellomake
```

Основы использования git

- Git - это консольная утилита, для отслеживания и ведения истории изменения файлов, в вашем проекте

Основы использования git

- Git - это консольная утилита, для отслеживания и ведения истории изменения файлов, в вашем проекте
- Не стоит путать git и github. Первое это утилита, с которой можно работать вообще локально на вашем компьютере. Второе - один из возможных серверов для ведения совместной разработки и для того, чтобы делиться кодом (хоть и самый известный).

Основы использования git

- Git - это консольная утилита, для отслеживания и ведения истории изменения файлов, в вашем проекте
- Не стоит путать git и github. Первое это утилита, с которой можно работать вообще локально на вашем компьютере. Второе - один из возможных серверов для ведения совместной разработки и для того, чтобы делиться кодом (хоть и самый известный).
- Известные варианты git серверов это https://github.com/, https://bitbucket.org/, https://gitlab.com/. Продвигаются различные российские, например, https://gitverse.ru/ от сбербанка.

Основы использования git

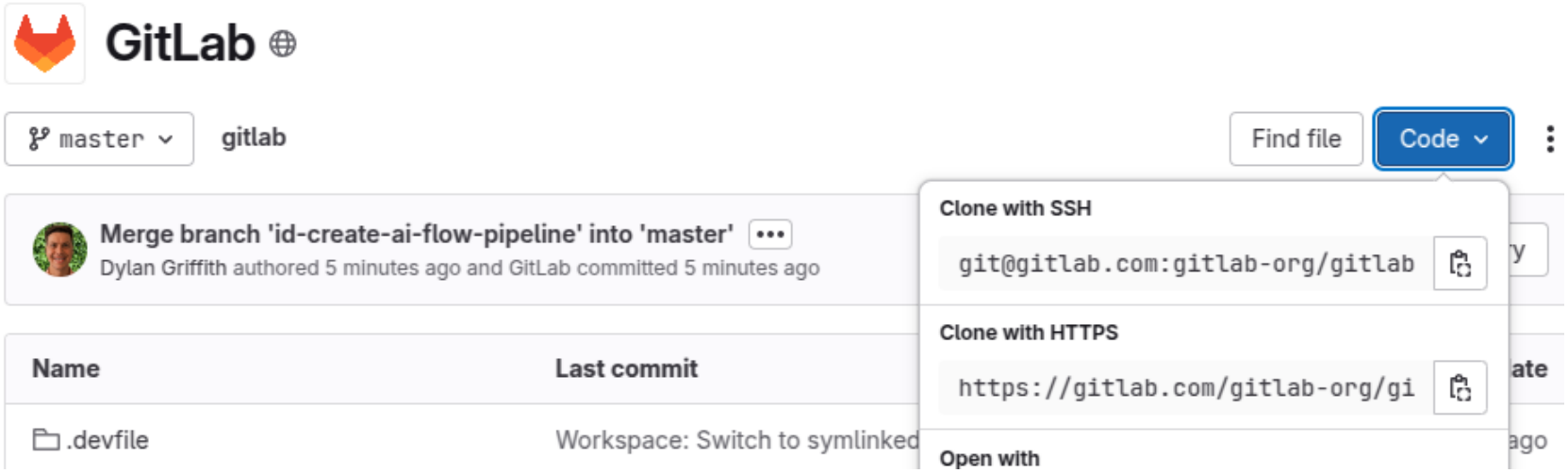
- Git - это консольная утилита, для отслеживания и ведения истории изменения файлов, в вашем проекте
- Не стоит путать git и github. Первое это утилита, с которой можно работать вообще локально на вашем компьютере. Второе - один из возможных серверов для ведения совместной разработки и для того, чтобы делиться кодом (хоть и самый известный).
- Известные варианты git серверов это https://github.com/, https://bitbucket.org/, https://gitlab.com/. Продвигаются различные российские, например, https://gitverse.ru/ от сбербанка.
- Также gitlab интересен тем, что они предлагают код git сервера, который можно развернуть локально, например, такой развернут у нас в НИВЦ: https://gitlab.srcc.msu.ru/

Основы использования git

- Установка под debian-based: sudo apt install git, про остальное смотрите под свою систему. Итогом является то, что вы получаете возможность в командной строке давать команды, начинающиеся с git

Основы использования git

- Установка под debian-based: sudo apt install git, про остальное смотрите под свою систему. Итогом является то, что вы получаете возможность в командной строке давать команды, начинающиеся с git
- Хотя с git можно работать полностью локально, начать лучше с клонирования какого-то репозитория. Например, с gitlab вы можете склонировать код самого gitlab по адресу https://gitlab.com/gitlab-org/gitlab



Основы использования git

- На git-хостингах предлагается выбор, склонировать по ssh или https. Для скачавания чего-то публичного проще использовать https, для регулярной разработки рекомендуется ssh, чтобы единожды залить на сервер свой публичный ключ и не вводить пароль при каждой команде. Итак, к адресу из команды клонирования подпишите git clone и выполните:

Основы использования git

- На git-хостингах предлагается выбор, склонировать по ssh или https. Для скачавания чего-то публичного проще использовать https, для регулярной разработки рекомендуется ssh, чтобы единожды залить на сервер свой публичный ключ и не вводить пароль при каждой команде. Итак, к адресу из команды клонирования подпишите git clone и выполните:
- git clone https://gitlab.com/gitlab-org/gitlab.git
- В результате у вас создастся папка gitlab (можно задать другое название третьим аргументом git clone), которая будет соответствовать текущему состоянию проекта gitlab (может занять время, это большой проект, возможно, лучше склонировать что-то еще)

Основы использования git

- На git-хостингах предлагается выбор, склонировать по ssh или https. Для скачавания чего-то публичного проще использовать https, для регулярной разработки рекомендуется ssh, чтобы единожды залить на сервер свой публичный ключ и не вводить пароль при каждой команде. Итак, к адресу из команды клонирования подпишите git clone и выполните:
- git clone https://gitlab.com/gitlab-org/gitlab.git
- В результате у вас создастся папка gitlab (можно задать другое название третьим аргументом git clone), которая будет соответствовать текущему состоянию проекта gitlab (может занять время, это большой проект, возможно, лучше склонировать что-то еще)
- Перейдите в папку gitlab и выполните git log

Основы использования git

- История разработки в git хранится в виде последовательности “коммитов”. Вот, например, последовательность коммитов по лекциям данного курса.

```
commit bce827c7fc752e1c0c2318a3ef41edf13d8f6d67 (HEAD -> main, origin/main, origin/HEAD)
Author: Alexander Smirnov <asmirnov@srcc.msu.ru>
Date:   Fri Jul 25 13:38:29 2025 +0300

    final notes from last year, and students to keep

commit 2a98870ce7929479c7deec61648e512d86c1e444
Author: Alexander Smirnov <asmirnov@srcc.msu.ru>
Date:   Tue Dec 24 11:39:42 2024 +0300

    notes on how to update lectures

commit d732353c1a8f064dd56f5a0679e1e41e51158df3
Author: Alexander Smirnov <asmirnov@srcc.msu.ru>
Date:   Tue Dec 24 11:39:30 2024 +0300

    2024 lectures update

commit 2e8ecc0f4308a27074787cb9b83cfe265e3276a8
Author: Alexander Smirnov <asmirnov@srcc.msu.ru>
Date:   Tue Dec 24 11:39:13 2024 +0300

    ignoring swp
```

Основы использования git

- История разработки в git хранится в виде последовательности “коммитов”. Коммит это некое изменение в коде, в ряде его файлов, которое всегда можно посмотреть, какие изменения в код оно принесло. Последовательность коммитов образует ветку. На новом проекте она одна (master или main), в реальном большом проекте при совместной разработке их может быть много.

Основы использования git

- История разработки в git хранится в виде последовательности “коммитов”. Коммит это некое изменение в коде, в ряде его файлов, которое всегда можно посмотреть, какие изменения в код оно принесло. Последовательность коммитов образует ветку. На новом проекте она одна (master или main), в реальном большом проекте при совместной разработке их может быть много.
- Всегда в рабочей папке вы можете посмотерть git status, то есть список того, что изменилось, дальше можно будет добавить все измененное (git add .) и сделать новый коммит (git commit -m “commit message”).
- Обратите внимание, что все эти действия локальные, они меняют состояние проекта в вашей папке и никак не влияют на то, что хранится на сервере.

Основы использования git

```
sander@sander-FX705DT:~/kurs$ git status
Текущая ветка: main
Эта ветка соответствует «origin/main».

Изменения, которые будут включены в коммит:
  (используйте «git restore --staged <файл>...», чтобы убрать из индекса)
        переименовано: lecture-notes -> lecture-notes-2024

Изменения, которые не в индексе для коммита:
  (используйте «git add <файл>...», чтобы добавить файл в индекс)
  (используйте «git restore <файл>...», чтобы отменить изменения в рабочем каталоге)
        изменено:      latex/1.tex
        изменено:      latex/2.tex
        изменено:      latex/3.tex
        изменено:      lecture-notes-2024

Неотслеживаемые файлы:
  (используйте «git add <файл>...», чтобы добавить в то, что будет включено в коммит)
        latex/images/gitlab.png
        latex/images/kurs-commits.png
        latex/images/risc-v.jpeg
        plan-2025
```

Основы использования git

- После того, как вы сделаете коммиты, вы увидете на git status, что локальная ветка обгоняет состояние на сервере

```
commit 3d0eddb071547b5ce9f1e8947e4ea3e82a896562 (HEAD -> main)
Author: Alexander Smirnov <asmirnov@srcc.msu.ru>
Date:   Fri Aug 8 12:00:49 2025 +0300

    updates on topics 1-3 with new images on risc-v and git

commit cb14712c8fd6d6a520bc3ea351408d39c6c20559
Author: Alexander Smirnov <asmirnov@srcc.msu.ru>
Date:   Fri Aug 8 11:59:58 2025 +0300

    plan for 2025

commit bce827c7fc752e1c0c2318a3ef41edf13d8f6d67 (origin/main, origin/HEAD)
Author: Alexander Smirnov <asmirnov@srcc.msu.ru>
Date:   Fri Jul 25 13:38:29 2025 +0300

    final notes from last year, and students to keep

commit 2a98870ce7929479c7deec61648e512d86c1e444
Author: Alexander Smirnov <asmirnov@srcc.msu.ru>
Date:   Tue Dec 24 11:39:42 2024 +0300

    notes on how to update lectures
```

Основы использования git

- Если после этого сделать команду git push, то состояние на сервере догонит локальное состояние

Основы использования git

- Если после этого сделать команду git push, то состояние на сервере догонит локальное состояние
- git pull позволит получить обновления на другом компьютере.

Основы использования git

- Если после этого сделать команду git push, то состояние на сервере догонит локальное состояние
- git pull позволит получить обновления на другом компьютере.
- Но какая может быть проблема при таких действиях

Основы использования git

- Если после этого сделать команду git push, то состояние на сервере догонит локальное состояние
- git pull позволит получить обновления на другом компьютере.
- Но какая может быть проблема при таких действиях
- Совместная работа. Если кто-то сделал push в ветку, нельзя сделать в нее push, пока вы тоже не сделали pull, а если вы за это время сделали коммиты, их надо переносить

Основы использования git

- Если после этого сделать команду git push, то состояние на сервере догонит локальное состояние
- git pull позволит получить обновления на другом компьютере.
- Но какая может быть проблема при таких действиях
- Совместная работа. Если кто-то сделал push в ветку, нельзя сделать в нее push, пока вы тоже не сделали pull, а если вы за это время сделали коммиты, их надо переносить
- Совместная работа требует правильной организации с ветвлением, когда создаются ветки под разные задачи, потом они по некоторым правилам сливаются в main. Есть разные подходы, можно читать про gitflow. Сайты типа gitlab позволяют при этом организовывать проверки, не позволяющие осуществить слияние пока, например, не прошли тесты.

Основы использования git

Самое простое использование git для одного человека

- Заведите репозиторий на gitlab, склонируйте его
- После каждого изменения в коде делайте коммит и push
- Перед коммитом проверяйте git status и не делайте слепо git add . , если есть что-то ненужное
- Используйте файл .gitignore чтобы исключить ненужные файлы, создаваемые при работе, но не являющиеся частью кода
- Если работаете не на одном компьютере, то на каждом из них сначала pull, потом работа, commit и push. Это вам позволит избежать конфликтов.

Вопросы?

Раздел 3. План.

Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

- Введение, общая память и библиотеки
- Использование потоков, передача в них параметров и возврат значений (c++11)
- Параллельная работа в общей памяти, механизмы обеспечения синхронизации (c++11)
- `std::async` и суммирование вектора (c++11)
- Make и Makefile. Основы использования git.
- **Модули (c++20)**
- Параллельные алгоритмы STL (c++17)
- Введение в корутины (c++20 и немного c++23)

C++20 и модули

- Курс на отказ от препроцессора в c++
- Улучшение времени сборки программ со сложной зависимостью

C++20 и модули

- Курс на отказ от препроцессора в c++
- Улучшение времени сборки программ со сложной зависимостью
- Давайте попробуем отказаться от #include

C++20 и модули

- Стандартный подход - пара файлов .h и .cpp

```
// main.cpp
#include func.h
int main(){
   func();
}

// func.h
void func();

// func.cpp
#include <iostream>
#include func.h
void func() {
   std::cout << "hello, world!\n";
}
```

C++20 и модули

- Если мы не включаем заголовочный файл, то нам как-то иначе нужно брать информацию о функциях, определенных в другом файле

C++20 и модули

- Если мы не включаем заголовочный файл, то нам как-то иначе нужно брать информацию о функциях, определенных в другом файле
- Новая инструкция import. Она не будет внедрять никакой c++ код!

C++20 и модули

- Если мы не включаем заголовочный файл, то нам как-то иначе нужно брать информацию о функциях, определенных в другом файле
- Новая инструкция import. Она не будет внедрять никакой c++ код!
- Но и из объектных файлов (если мы скомпилировали func) неудобно брать информацию о функциях

C++20 и модули

- Если мы не включаем заголовочный файл, то нам как-то иначе нужно брать информацию о функциях, определенных в другом файле
- Новая инструкция import. Она не будет внедрять никакой c++ код!
- Но и из объектных файлов (если мы скомпилировали func) неудобно брать информацию о функциях
- Компилятором создаются принципиально новые файлы бинарного формата о скомпилированном модуле в директории gcm.cache

C++20 и модули

- Новый подход - файл cxx для модуля (так удобнее отличать от старого)

```
// main.cpp
import func;
int main(){
   func();
}

// func.cxx
module;
#include <iostream>
export module func;
export void func() {
   std::cout << "hello, world!\n";
}
```

C++20 и модули

- Сборка программы

```
g++ -c func.cxx -std=c++20 -fmodules-ts
g++ -c -o main.o main.cpp -std=c++20
    -fmodules-ts
g++ -o main main.o func.o
```

C++20 и модули

- Сборка программы

```
g++ -c func.cxx -std=c++20 -fmodules-ts
g++ -c -o main.o main.cpp -std=c++20
    -fmodules-ts
g++ -o main main.o func.o
```

- Нам нужен стандарт 20 и специальная опция для использования модулей
- После первой команды появится файл gcm.cache/func.gcm (здесь func - имя модуля, а не файла, но можно и использовать одинаковые)

C++20 и модули

- Сборка программы

```
g++ -c func.cxx -std=c++20 -fmodules-ts
g++ -c -o main.o main.cpp -std=c++20
    -fmodules-ts
g++ -o main main.o func.o
```

- Нам нужен стандарт 20 и специальная опция для использования модулей
- После первой команды появится файл gcm.cache/func.gcm (здесь func - имя модуля, а не файла, но можно и использовать одинаковые)
- Поэтому важен порядок сборки - нужно сначала собрать func

C++20 и модули

- Напишем makefile

```
default: main
CXX=g++
OBJ=main.o func.o
CFLAGS=-std=c++20 -fmodules-ts
main: $(OBJ)
	$(CXX) -o $@ $^
%.o: %.cpp
	${CXX} -c -o $@ $< ${CFLAGS}
%.o: %.cxx
	${CXX} -c -o $@ $< ${CFLAGS}
gcm.cache/%.mod: %.cxx
	${CXX} -c $^ ${CFLAGS}
main.o: main.cpp
main.o: gcm.cache/func.mod
func.o: func.cxx
```

C++20 и модули

- Если хочется, можно разделить файл модуля на два, с декларацией и имплементацией. Файлы будут называться по-разному, но описывать один и тот же модуль. Нет стандарта, как называть пары таких файлов, можно, например *func.cxx* и *func_impl.cxx*.
- Впрочем, необходимости в этом нет - ведь заголовочные файлы делались для минимизации включаемого в другой файл кода при компиляции, а здесь такого не происходит
- Компиляция того, что содержится в модуле, происходит в любом случае один раз!
- Так что, пора переводить все на модули?

C++20 и модули

- Поддержка gcc
  `https://gcc.gnu.org/projects/cxx-status.html`

P1103R3	11 (requires `-fmodules-ts`) (No Private Module Fragment, Parser-level Global Module Entity Merging, Global Module Implications of `extern "C/C++"`, or Partition-specific Definition Visibility)
P1766R1	No
P1811R0	11
P1703R1 (superceded by p1857)	11
P1874R1	11
P1979R0	11
P1779R3	11
P1857R3	11
P2115R0	11
P1815R2	No

C++20 и модули

- Поддержка модулей разными компиляторами: gcc-11 (partial), clang-8 (partial), msvc-19.0 (partial), msvc-19.28 (full), apple clang 10.0.1* (partial) (`https://en.cppreference.com/w/cpp/compiler_support/20`)

C++20 и модули

- Поддержка модулей разными компиляторами: gcc-11 (partial), clang-8 (partial), msvc-19.0 (partial), msvc-19.28 (full), apple clang 10.0.1* (partial) (`https://en.cppreference.com/w/cpp/compiler_support/20`)
- Но мало ведь толку от модулей, если они не используются в том коде, который вы используете! Что, например со стандартной библиотекой? Хочется import std

C++20 и модули

- Поддержка модулей разными компиляторами: gcc-11 (partial), clang-8 (partial), msvc-19.0 (partial), msvc-19.28 (full), apple clang 10.0.1* (partial) (`https://en.cppreference.com/w/cpp/compiler_support/20`)
- Но мало ведь толку от модулей, если они не используются в том коде, который вы используете! Что, например со стандартной библиотекой? Хочется import std
- Для этого нужен стандарт c++23, и поддержка на сегодня: clang-17 (partial), msvc-19.36 (full). (`https://en.cppreference.com/w/cpp/compiler_support/23`). Обещано в gcc-15 (добавление 2025)

Вопросы?

Раздел 3. План.

Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

- Введение, общая память и библиотеки
- Использование потоков, передача в них параметров и возврат значений (c++11)
- Параллельная работа в общей памяти, механизмы обеспечения синхронизации (c++11)
- `std::async` и суммирование вектора (c++11)
- Make и Makefile. Основы использования git.
- Модули (c++20)
- **Параллельные алгоритмы STL (c++17)**
- Введение в корутины (c++20 и немного c++23)

Параллельные алгоритмы STL

Стандарт c++17 принес много синтаксического сахара, удобные вещи для работы с шаблонами (constexpr if), но если мы вернемся к теме нашего курса, то основным нововведением кажутся параллельные алгоритмы STL

Параллельные алгоритмы STL

Стандарт c++17 принес много синтаксического сахара, удобные вещи для работы с шаблонами (constexpr if), но если мы вернемся к теме нашего курса, то основным нововведением кажутся параллельные алгоритмы STL

- Но сначала попробуем просто воспользоваться алгоритмами stl для решения задачи суммирования вектора.

Параллельные алгоритмы STL

Стандарт c++17 принес много синтаксического сахара, удобные вещи для работы с шаблонами (constexpr if), но если мы вернемся к теме нашего курса, то основным нововведением кажутся параллельные алгоритмы STL

- Но сначала попробуем просто воспользоваться алгоритмами stl для решения задачи суммирования вектора.
- У нас имелась функция суммирования вектора (или части вектора)

Параллельные алгоритмы STL

Стандарт c++17 принес много синтаксического сахара, удобные вещи для работы с шаблонами (constexpr if), но если мы вернемся к теме нашего курса, то основным нововведением кажутся параллельные алгоритмы STL

- У нас имелась функция суммирования вектора (или части вектора)

```
double sum_vector(std::vector<double>& a,
      size_t start, size_t end) {
      double res = 0.;
      for (size_t i = start; i!=end; i++) {
            res = res + a[i];
      }
      return res;
}
```

Параллельные алгоритмы STL

Стандарт c++17 принес много синтаксического сахара, удобные вещи для работы с шаблонами (constexpr if), но если мы вернемся к теме нашего курса, то основным нововведением кажутся параллельные алгоритмы STL

- У нас имелась функция суммирования вектора (или части вектора)
- Перепишем же ее простым вызовом алгоритма stl

```
double sum_vector(std::vector<double>& a,
    size_t start, size_t end) {
    return std::accumulate(a.begin() + start,
      a.begin() + end, 0.);
}
```

Параллельные алгоритмы STL

- Код с std::accumulate позволяет не писать цикл, да, хорошо, но это все-таки просто небольшое удобство. А можно как-то параллелизовать этот код?

Параллельные алгоритмы STL

- Код с std::accumulate позволяет не писать цикл, да, хорошо, но это все-таки просто небольшое удобство. А можно как-то параллелизовать этот код?
- С std::accumulate нельзя, там старый алгоритм, не поддающийся параллелизации. Поэтому в c++17 появился новый алгоритм и новая функция std::reduce

Параллельные алгоритмы STL

- Код с std::accumulate позволяет не писать цикл, да, хорошо, но это все-таки просто небольшое удобство. А можно как-то параллелизовать этот код?
- С std::accumulate нельзя, там старый алгоритм, не поддающийся параллелизации. Поэтому в c++17 появился новый алгоритм и новая функция std::reduce

```
double sum_vector(std::vector<double>& a,
    size_t start, size_t end) {
    return std::reduce(a.begin() + start,
      a.begin() + end, 0.);
}
```

Параллельные алгоритмы STL

- Давайте замеряем время

Параллельные алгоритмы STL

- Давайте замеряем время
- Одноядерная реализация с циклом: 222ms

Параллельные алгоритмы STL

- Давайте замеряем время
- Одноядерная реализация с циклом: 222ms
- std::accumulate: 222ms

Параллельные алгоритмы STL

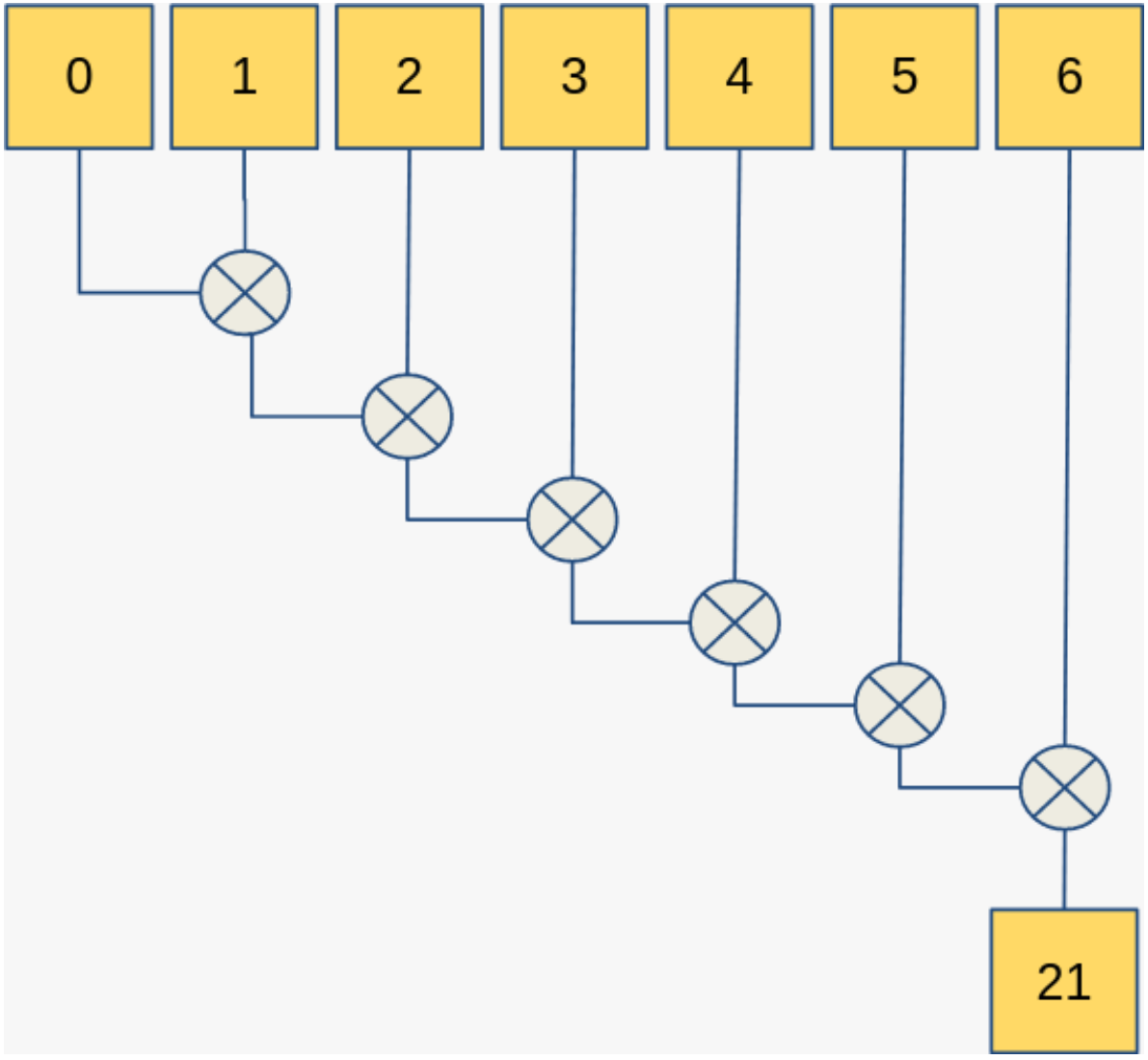
- Давайте замеряем время
- Одноядерная реализация с циклом: 222ms
- std::accumulate: 222ms
- std::reduce: 130ms

Параллельные алгоритмы STL

- Давайте замеряем время
- Одноядерная реализация с циклом: 222ms
- std::accumulate: 222ms
- std::reduce: 130ms

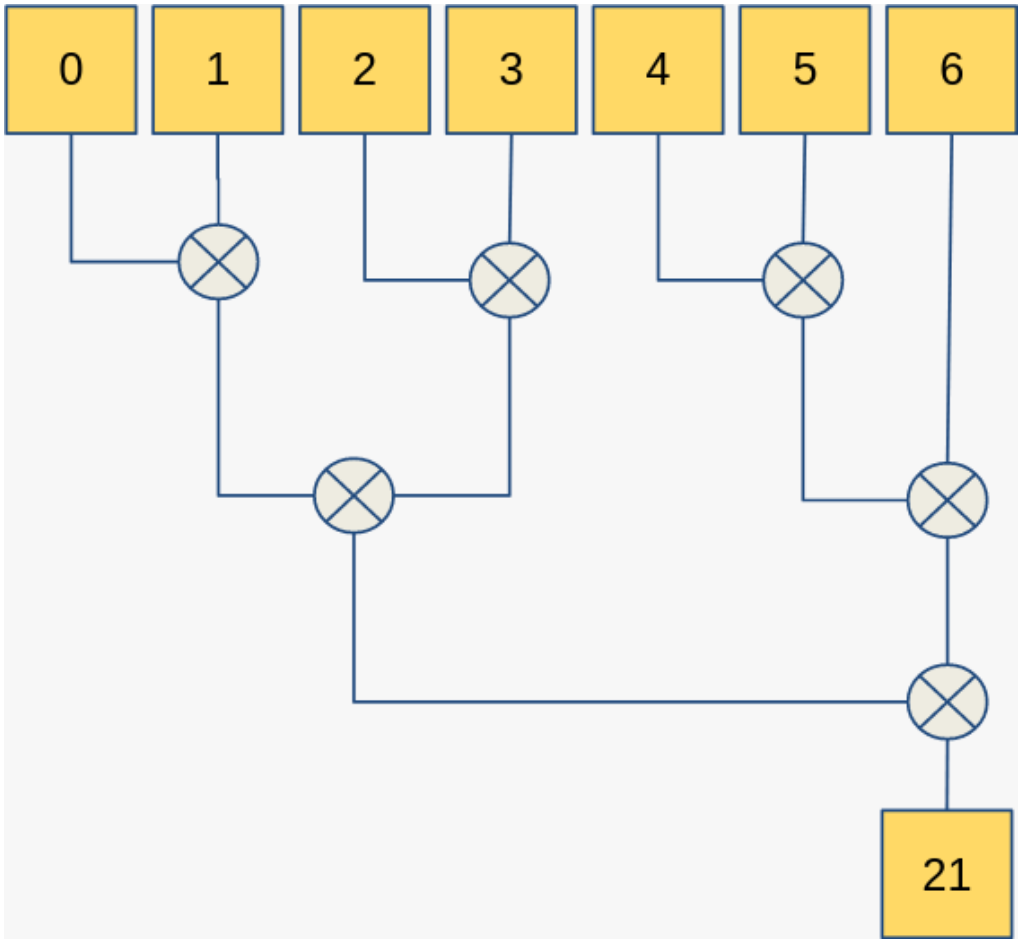
Параллельные алгоритмы STL

- Если обратиться к статьям про алгоритмы stl, они все выдают, что accumulate последовательный



Параллельные алгоритмы STL

- Если обратиться к статьям про алгоритмы stl, они все выдают, что accumulate последовательный, а reduce собирает деревом



Параллельные алгоритмы STL

- Если обратиться к статьям про алгоритмы stl, они все выдают, что accumulate последовательный, а reduce собирает деревом, но в интернете врут, и, на самом деле, код reduce примерно такой:

```
double sum_vector(std::vector<double>& a,
    size_t start, size_t end) {
    double res = 0.;
    for (size_t i = start; i!=end; i+=4) {
        res = res + (a[i] + a[i + 1])
          + (a[i + 2] + a[i + 3]);
    }
    return res;
}
```

Параллельные алгоритмы STL

- Мы можем реализовать код от std::reduce сами и получить то же самое ускорение на одном ядре. Со всей сложностью современных процессоров практически невозможно увидеть “как текут байты”, но в целом логика ускорения в том, что у ядра процессора есть несколько блоков, выполняющих операции и ему удается развести сложении и выполнять его параллельно на самом низком уровне.

Параллельные алгоритмы STL

- Мы можем реализовать код от std::reduce сами и получить то же самое ускорение на одном ядре. Со всей сложностью современных процессоров практически невозможно увидеть “как текут байты”, но в целом логика ускорения в том, что у ядра процессора есть несколько блоков, выполняющих операции и ему удается развести сложении и выполнять его параллельно на самом низком уровне.
- Да, мы здесь исходим из ассоциативности сложения (что для машинной арифметики, вообще говоря, не так), но разбивая сами на блоки через asyc мы тоже этим пользовались.

Параллельные алгоритмы STL

- Мы можем реализовать код от std::reduce сами и получить то же самое ускорение на одном ядре. Со всей сложностью современных процессоров практически невозможно увидеть “как текут байты”, но в целом логика ускорения в том, что у ядра процессора есть несколько блоков, выполняющих операции и ему удается развести сложении и выполнять его параллельно на самом низком уровне.
- Да, мы здесь исходим из ассоциативности сложения (что для машинной арифметики, вообще говоря, не так), но разбивая сами на блоки через asyc мы тоже этим пользовались.
- Но где же параллельность в std::reduce?

Задействуем параллелизацию для алгоритмов stl

```
#include <execution>
...
...
return std::reduce(std::execution::par,
        a.begin() + start,
        a.begin() + end, 0.);
```

- Если вы просто так попробуете это скомпилировать, то программа не скомпонуется. Необходимо добавить к опциям линковки -ltbb, причем поставить это после имени файла.

Задействуем параллелизацию для алгоритмов stl

```
#include <execution>
...
...
return std::reduce(std::execution::par,
        a.begin() + start,
        a.begin() + end, 0.);
```

- Если вы просто так попробуете это скомпилировать, то программа не скомпонуется. Необходимо добавить к опциям линковки -ltbb, причем поставить это после имени файла.
- Что за tbb?

TBB = Thread Building Blocks

- Thread building blocks — библиотека, изначально разработанная Intel (но работающая на различной архитектуре), реализующая различные параллельные алгоритмы
- Библиотекой можно пользоваться и отдельно, но начиная с c++17 она интегрирована для использования при помощи Parallel STL algorithms (этот путь прошли разные библиотеки, например, большие части boost)

TBB = Thread Building Blocks

- Thread building blocks — библиотека, изначально разработанная Intel (но работающая на различной архитектуре), реализующая различные параллельные алгоритмы
- Библиотекой можно пользоваться и отдельно, но начиная с c++17 она интегрирована для использования при помощи Parallel STL algorithms (этот путь прошли разные библиотеки, например, большие части boost)
- И все бы хорошо, но это отдельная библиотека, и даже в 2024 году на суперкомпьютере Ломоносов я так и не смог наладить полноценную компиляцию и работу программ с использованием TBB, а, значит, и Parallel STL Algorithms

Параллельные алгоритмы STL

- Давайте замеряем время
- Одноядерная реализация с циклом: 222ms
- std::accumulate: 222ms
- std::reduce: 130ms
- std::reduce with parallel execution: 82ms
- async with 4 cores: 103ms
- async with 8 cores: 87ms

Параллельные алгоритмы STL

- Давайте замеряем время
- Одноядерная реализация с циклом: 222ms
- std::accumulate: 222ms
- std::reduce: 130ms
- std::reduce with parallel execution: 82ms
- async with 4 cores: 103ms
- async with 8 cores: 87ms
- async with 4 cores and reduce code: 78ms

Параллельные алгоритмы STL

- Давайте замеряем время
- Одноядерная реализация с циклом: 222ms
- std::accumulate: 222ms
- std::reduce: 130ms
- std::reduce with parallel execution: 82ms
- async with 4 cores: 103ms
- async with 8 cores: 87ms
- async with 4 cores and reduce code: 78ms
- async with 8 cores and reduce code: 82ms

Async все-таки выиграл, если применить технику от reduce. Но 8 ядер ухудшают ситуацию. Так как проще писать?)))

Вопросы?

Раздел 3. План.

Многопоточное программирование в общей памяти с использованием стандартов C++11 и выше

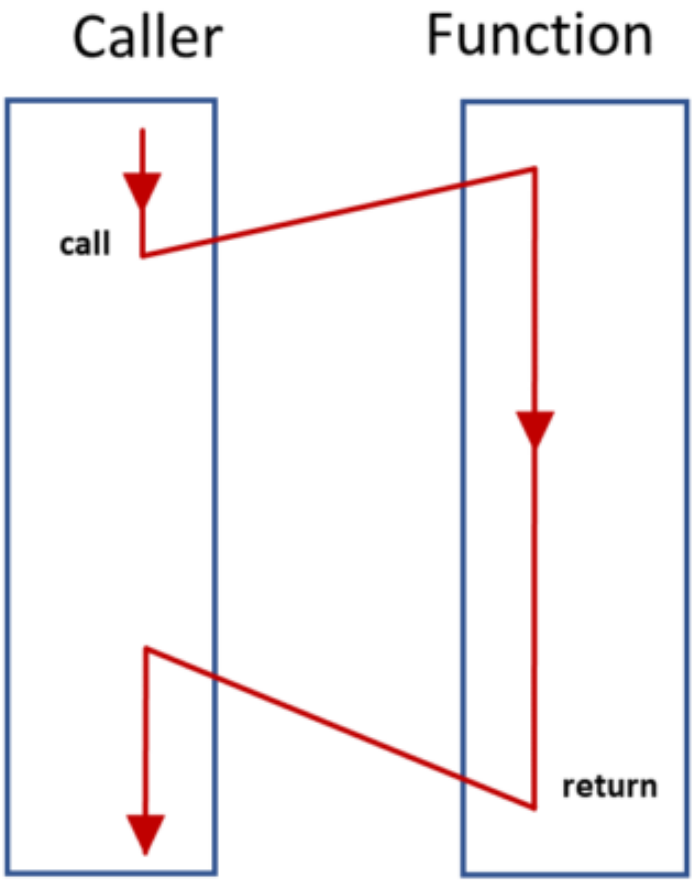
- Введение, общая память и библиотеки
- Использование потоков, передача в них параметров и возврат значений (c++11)
- Параллельная работа в общей памяти, механизмы обеспечения синхронизации (c++11)
- `std::async` и суммирование вектора (c++11)
- Make и Makefile. Основы использования git.
- Модули (c++20)
- Параллельные алгоритмы STL (c++17)
- **Введение в корутины (c++20 и немного c++23)**

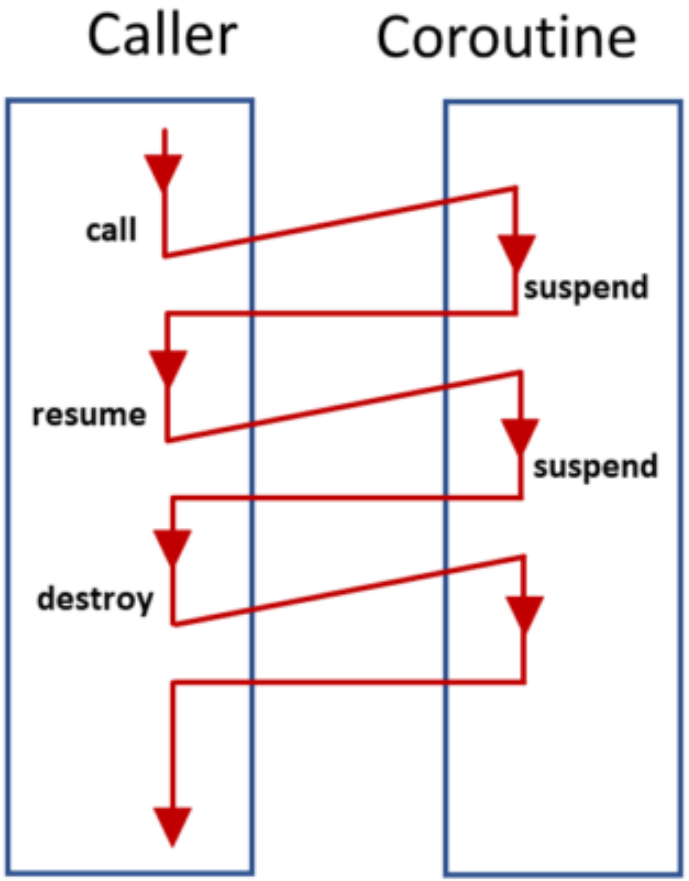
Корутины

Корутины, они же сопрограммы

- Понятие корутин давно существует в других языках. Если “на пальцах”, то корутина — это функция, чье выполнение можно приостановить при помощи промежуточного “return” или “await”, а потом возобновить из другой функции.

Корутины, они же сопрограммы





Корутины, они же сопрограммы

- Понятие корутин давно существует в других языках. Если “на пальцах”, то корутина — это функция, чье выполнение можно приостановить при помощи промежуточного “return” или “await”, а потом возобновить из другой функции.
- Как проще всего реализовать функциональность такой приостановки, используя описанное выше?

Корутины, они же сопрограммы

- Понятие корутин давно существует в других языках. Если “на пальцах”, то корутина — это функция, чье выполнение можно приостановить при помощи промежуточного “return” или “await”, а потом возобновить из другой функции.
- Как проще всего реализовать функциональность такой приостановки, используя описанное выше?
- Конечно, отдельным потоком и mutex/condition_variable. А чем это может нас не устраивать?

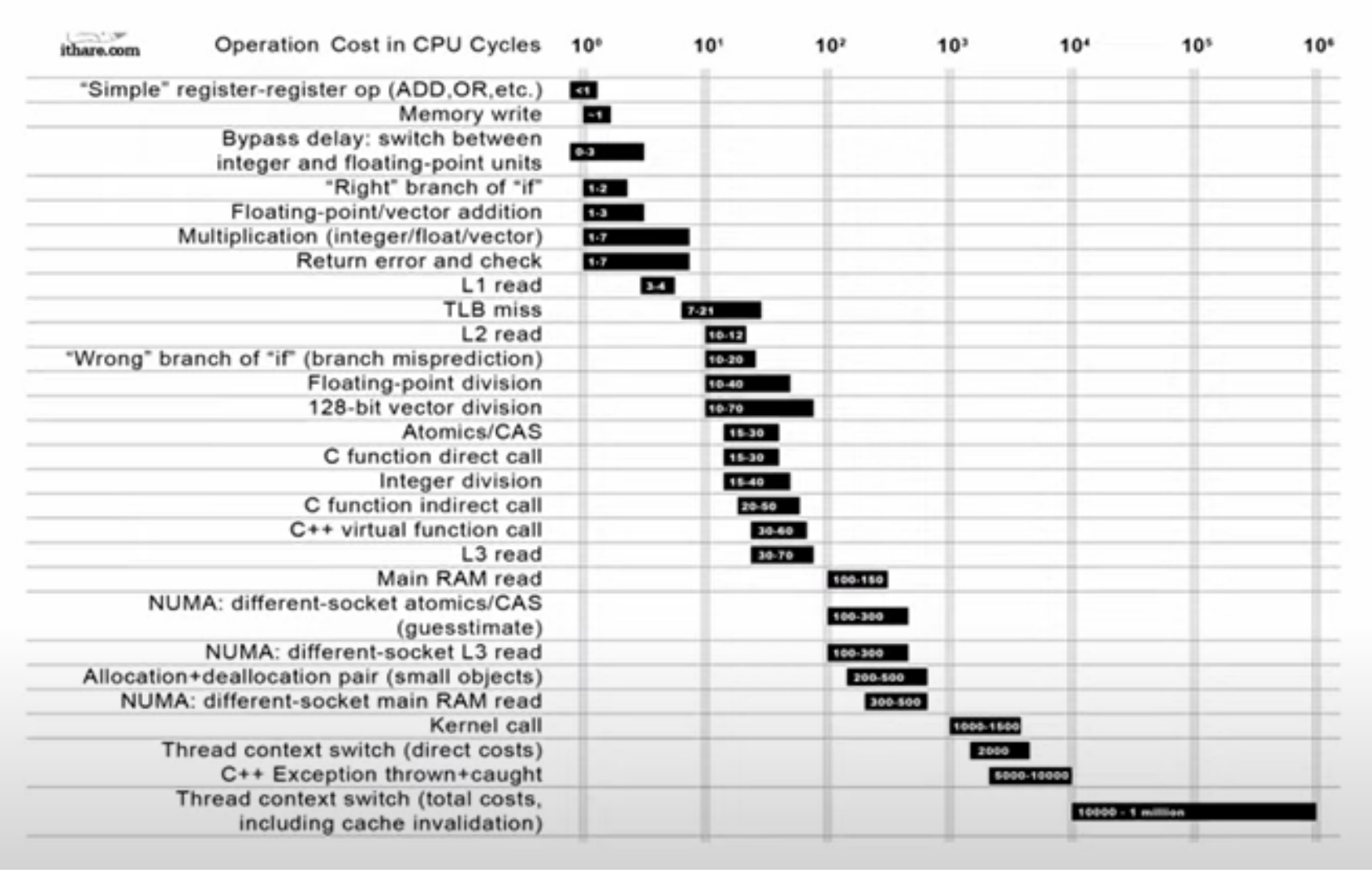
Корутины, они же сопрограммы

- Понятие корутин давно существует в других языках. Если “на пальцах”, то корутина — это функция, чье выполнение можно приостановить при помощи промежуточного “return” или “await”, а потом возобновить из другой функции.
- Как проще всего реализовать функциональность такой приостановки, используя описанное выше?
- Конечно, отдельным потоком и mutex/condition_variable. А чем это может нас не устраивать?
- Основная причина — вовсе не простота синтаксиса, а стоимость переключения контекста

Корутины, они же сопрограммы

- Понятие корутин давно существует в других языках. Если “на пальцах”, то корутина — это функция, чье выполнение можно приостановить при помощи промежуточного “return” или “await”, а потом возобновить из другой функции.
- Как проще всего реализовать функциональность такой приостановки, используя описанное выше?
- Конечно, отдельным потоком и mutex/condition_variable. А чем это может нас не устраивать?
- Основная причина — вовсе не простота синтаксиса, а стоимость переключения контекста
- Корутины это не про параллелизацию, а про асинхронность!

Корутины


ithare.com
Operation Cost in CPU Cycles
10⁰
10¹
10²
10³
10⁴
10⁵
10⁶
"Simple" register-register op (ADD,OR,etc.)
Memory write
Bypass delay: switch between integer and floating-point units
"Right" branch of "if"
Floating-point/vector addition
Multiplication (integer/float/vector)
Return error and check
L1 read
TLB miss
L2 read
"Wrong" branch of "if" (branch misprediction)
Floating-point division
128-bit vector division
Atomics/CAS
C function direct call
Integer division
C function indirect call
C++ virtual function call
L3 read
Main RAM read
NUMA: different-socket atomics/CAS (guesstimate)
NUMA: different-socket L3 read
Allocation+deallocation pair (small objects)
NUMA: different-socket main RAM read
Kernel call
Thread context switch (direct costs)
C++ Exception thrown+caught
Thread context switch (total costs, including cache invalidation)
<1
~1
0-3
1-2
1-3
1-7
1-7
3-4
7-21
10-12
10-20
10-40
10-70
15-30
15-30
15-40
20-50
30-60
30-70
100-150
100-300
100-300
200-500
300-500
1000-1500
2000
5000-10000
10000 - 1 million

Корутины, они же сопрограммы

- Итак, корутина это приостанавливаемая функция, выполняемая в том же потоке, что и вызывающая ее функция, что дает легковесность переключения контекста.

Корутины, они же сопрограммы

- Итак, корутина это приостанавливаемая функция, выполняемая в том же потоке, что и вызывающая ее функция, что дает легковесность переключения контекста.
- Один из типов корутин — стековые корутины, они же зеленые потоки, волокна и горутины. Они отличаются наличием своего стека вызовов (call stack).

Корутины, они же сопрограммы

- Итак, корутина это приостанавливаемая функция, выполняемая в том же потоке, что и вызывающая ее функция, что дает легковесность переключения контекста.
- Один из типов корутин — стековые корутины, они же зеленые потоки, волокна и горутины. Они отличаются наличием своего стека вызовов (call stack).
- Существуют библиотеки, реализующие стековые корутины, как на чистом си, так и под c++ (например, в boost). Однако выделение отдельного стека требует ресурсы, а переключение все равно достаточно дорогое, поэтому в c++ они так и не обрели популярность.

Бесстековые корутины

- Стандарт c++20 приносит бесстековые корутины, под локальные переменные которых выделяется память в куче, но при этом “вызов” или “продолжение” корутыны происходят в обычном стеке, соответственно, дальнейшие вызовы функций из бесстековой корутины производятся на стеке родителя. Что означает, в частности, что приостановка возможна только из основной функции корутины. Впрочем, с этим ограничением можно смириться в обмен на производительность.

Бесстековые корутины

- Стандарт c++20 приносит бесстековые корутины, под локальные переменные которых выделяется память в куче, но при этом “вызов” или “продолжение” корутыны происходят в обычном стеке, соответственно, дальнейшие вызовы функций из бесстековой корутины производятся на стеке родителя. Что означает, в частности, что приостановка возможна только из основной функции корутины. Впрочем, с этим ограничением можно смириться в обмен на производительность.
- В связи с такой структурой реализовать такую корутину в виде библиотеки невозможно, поэтому и ждали, когда оно появится в стандарте.

Бесстековые корутины

- Стандарт c++20 приносит бесстековые корутины, под локальные переменные которых выделяется память в куче, но при этом “вызов” или “продолжение” корутыны происходят в обычном стеке, соответственно, дальнейшие вызовы функций из бесстековой корутины производятся на стеке родителя. Что означает, в частности, что приостановка возможна только из основной функции корутины. Впрочем, с этим ограничением можно смириться в обмен на производительность.
- В связи с такой структурой реализовать такую корутину в виде библиотеки невозможно, поэтому и ждали, когда оно появится в стандарте.
- К сожалению, воспользоваться корутинами достаточно сложно, и код вовсе не становится лаконичным.

Корутины

Код, который хотелось бы видеть

```
generator generate(size_t start, size_t end) {
  for (auto i = start; i < end; ++i) {
    co_yield i;
  }
}

int main() {
  for (auto value: generate(0, 10)) {
    std::cout << value << std::endl;
  }
  return 0;
}
```

Корутины

Код, который хотелось бы видеть

```
generator generate(size_t start, size_t end) {
  for (auto i = start; i < end; ++i) {
    co_yield i;
  }
}

int main() {
  for (auto value: generate(0, 10)) {
    std::cout << value << std::endl;
  }
  return 0;
}
```

Однако, приходится написать очень много кода

Мы должны описать класс generator

- Он должен внутри себя иметь класс iterator, содержащий операторы ++, * и !=
- Также класс generator должен иметь методы begin() и end(), возвращающие iterator

Мы должны описать класс generator

- Он должен внутри себя иметь класс iterator, содержащий операторы ++, * и !=
- Также класс generator должен иметь методы begin() и end(), возвращающие iterator
- Но, главное, чтобы функция могла вызвать co_yield и все это работало, он должен внутри себя содержать структуру promise_type с определенным набором свойств. Давайте попробуем все это реализовать

Корутины

Мы должны описать класс generator

- Он должен внутри себя иметь класс iterator, содержащий операторы ++, * и !=
- Также класс generator должен иметь методы begin() и end(), возвращающие iterator
- Но, главное, чтобы функция могла вызвать co_yield и все это работало, он должен внутри себя содержать структуру promise_type с определенным набором свойств. Давайте попробуем все это реализовать
- Если вложенный тип это promise, то по аналогии обертывающий тип generator это future (не в смысле прямого совпадения типов с описанным выше)

Корутины

get_return_object — Создает экземпляр типа generator (ведь в функции generate нет return!). Она вызовется при “вызове” корутины и возврате изначального значения.

```
class generator {
public:
   struct promise_type {
      using suspend_never=std::suspend_never;
      using suspend_always=std::suspend_always;
      using handle =
      std::coroutine_handle<promise_type>;
      size_t value;
      auto get_return_object() noexcept {
         return generator{
            handle::from_promise(*this)
         };
      }
```

Корутины

```
    suspend_never initial_suspend()
      noexcept { return {}; }
    suspend_always final_suspend()
      noexcept { return {}; }
    void return_void() noexcept {}
    void unhandled_exception()
      { std::terminate(); }
    suspend_always yield_value(size_t v)
      noexcept {
      value = v;
      return {};
    }
};
```

- initial_suspend возвращаемым значением определяет, что корутина сразу начнет работу после создания и дойдет до первого co_yield.

Корутины

```
    suspend_never initial_suspend()
      noexcept { return {}; }
    suspend_always final_suspend()
      noexcept { return {}; }
    void return_void() noexcept {}
    void unhandled_exception()
      { std::terminate(); }
    suspend_always yield_value(size_t v)
      noexcept {
      value = v;
      return {};
    }
};
```

- final_suspend возвращаемым значением говорит не уничтожать корутину после того, как она дошла до конца выполнения.

Корутины

```
    suspend_never initial_suspend()
      noexcept { return {}; }
    suspend_always final_suspend()
      noexcept { return {}; }
    void return_void() noexcept {}
    void unhandled_exception()
      { std::terminate(); }
    suspend_always yield_value(size_t v)
      noexcept {
      value = v;
      return {};
    }
};
```

- return_void ничего не делает, поскольку мы не используем co_return.

Корутины

```
    suspend_never initial_suspend()
      noexcept { return {}; }
    suspend_always final_suspend()
      noexcept { return {}; }
    void return_void() noexcept {}
    void unhandled_exception()
      { std::terminate(); }
    suspend_always yield_value(size_t v)
      noexcept {
      value = v;
      return {};
    }
};
```

- Также мы не обрабатываем исключения

Корутины

```
    suspend_never initial_suspend()
      noexcept { return {}; }
    suspend_always final_suspend()
      noexcept { return {}; }
    void return_void() noexcept {}
    void unhandled_exception()
      { std::terminate(); }
    suspend_always yield_value(size_t v)
      noexcept {
      value = v;
      return {};
    }
};
```

- yield_value это обработка co_yield с сохранением значение в входом в ожидание

Корутины

```
private:
 promise_type::handle m_coro;
public:
 ~generator() noexcept { m_coro.destroy(); }
 class iterator {
 private:
   generator *m_self;
 public:
   bool operator != (iterator second) const {
     return m_self != second.m_self;
   }
   size_t operator*() {
     return m_self->m_coro.promise().value;
   }
```

- m_coro — “указатель на промис”

Корутины

```
private:
 promise_type::handle m_coro;
public:
 ~generator() noexcept { m_coro.destroy(); }
 class iterator {
 private:
    generator *m_self;
 public:
    bool operator != (iterator second) const {
      return m_self != second.m_self;
    }
    size_t operator*() {
      return m_self->m_coro.promise().value;
    }
```

- Деструктор генератора вызывает разрушение фрейма корутины (мы это приостановили в final_suspend)

Корутины

```
private:
 promise_type::handle m_coro;
public:
 ~generator() noexcept { m_coro.destroy(); }
 class iterator {
 private:
    generator *m_self;
 public:
    bool operator != (iterator second) const {
      return m_self != second.m_self;
    }
    size_t operator*() {
      return m_self->m_coro.promise().value;
    }
```

- Вложенный класс итератора содержит указатель на сам генератор

Корутины

```
private:
 promise_type::handle m_coro;
public:
 ~generator() noexcept { m_coro.destroy(); }
 class iterator {
 private:
   generator *m_self;
 public:
   bool operator != (iterator second) const {
     return m_self != second.m_self;
   }
   size_t operator*() {
     return m_self->m_coro.promise().value;
   }
```

- Неравенство итератора для завершения работы цикла

Корутины

```
private:
 promise_type::handle m_coro;
public:
 ~generator() noexcept { m_coro.destroy(); }
 class iterator {
 private:
   generator *m_self;
 public:
   bool operator != (iterator second) const {
     return m_self != second.m_self;
   }
   size_t operator*() {
     return m_self->m_coro.promise().value;
   }
```

- Оператор взятия значения, как и положено, возвращает сохраненное значение из промиса

Корутины

```
    iterator & operator++() {
        m_self->m_coro.resume();
        if (m_self->m_coro.done()) {
            m_self = nullptr;
        }
        return *this;
    }
  private:
    iterator(generator *self): m_self{self} {}
    friend class generator;
};
```

- Оператор инкремента проверяет воззобновляет выполнение корутины - генератора, далее проверяет, завершилась ли корутина, и если да, возвращает nullptr, что потребуется для сравнения с end() — увидим на следующем слайде

Корутины

```
    iterator begin() { return iterator{
            m_coro.done() ? nullptr : this
    }; }
    iterator end() {return iterator{nullptr};}
private:
    explicit generator
      (promise_type::handle coro)
      noexcept: m_coro{coro} {}
};
```

- Метод begin() у итератора проверяет не закончились ли числа (дали пустой интервал), и, если нет, возвращает указатель на класс итератора.

Корутины

```
    iterator begin() { return iterator{
            m_coro.done() ? nullptr : this
    }; }
    iterator end() {return iterator{nullptr};}
private:
    explicit generator
      (promise_type::handle coro)
      noexcept: m_coro{coro} {}
};
```

- Метод end() всегда возвращает нулевой итератор. Они станут равны только когда корутина завершила работу

Корутины

```
    iterator begin() { return iterator{
            m_coro.done() ? nullptr : this
    }; }
    iterator end() {return iterator{nullptr};}
private:
    explicit generator
      (promise_type::handle coro)
      noexcept: m_coro{coro} {}
};
```

- Конструктор генератора. Он должен быть именно такой (с сохранением ссылки на корутину), он вызовется из get_return_object

Корутины

Генератор в c++23

- c++23 представляет первую корутину, включенную в библиотеку (не надо писать свои promise и future).

Генератор в c++23

- c++23 представляет первую корутину, включенную в библиотеку (не надо писать свои promise и future).

```
std::generator<int> fib() {
    co_yield 0;
    auto a = 0;
    auto b = 1;
    for(auto n : std::views::iota(0)) {
        auto next = a + b;
        a = b;
        b = next;
        co_yield next;
    }
}
```

Генератор в c++23

```
int main() {
    for (auto f : fib() |
            std::views::take(10)) {
        std::cout << f << " ";
    }
}
```

- Здесь задействована библиотека ranges из c++20 и ей определенная перегрузка оператора побитового или, работающая похоже на “трубопровод” из командной строки.

Генератор в c++23

```
int main() {
    for (auto f : fib() |
            std::views::take(10)) {
        std::cout << f << " ";
    }
}
```

- Здесь задействована библиотека ranges из c++20 и ей определенная перегрузка оператора побитового или, работающая похоже на “трубопровод” из командной строки.
- Все это выглядит куда более красиво, однако это пока единственная корутина из стандартной библиотеки, а, кроме того, на текущий момент из компиляторов поддержка заявлена только в gcc 14.

Вопросы?

Домашнее задание

Внимание! Задание!

1. Найти максимум в массиве
2. Найти количество положительных элементов в массиве
3. Найти сумму массива (да, это было на лекциях, но реализуйте в коде)
4. Найти первое число, превышающее предварительно заданное

Вы должны написать программу, которая делает одно из написанного выше. Чтобы определить, что, возьмите остаток от деления вашего номера на 4 и прибавьте 1.
В виде формулы это $(number\%4) + 1$

Домашнее задание

Внимание! Задание!

1. реализовать при помощи априорного разделения, использовать packaged_task
2. реализовать при помощи atomic (общий счетчик)
3. реализовать при помощи mutex (без condition_variable), возвращать из потоков значение через future/promise
4. реализовать при помощи mutex + condition_variable
5. реализовать при помощи parallel stl

Вы должны использовать один из методов параллелизации выше. Чтобы определить, возьмите целое частное от деления вашего номера на 4, затем остаток от деления на 5 и прибавьте 1.
В виде формулы это $((number/4)\%5) + 1$

Домашнее задание

Внимание! Задание!

Варианты чисел в массиве:

1. int (32-bit)
2. float
3. long int / long long int (64-bit)
4. double

Чтобы определить ваш вариант, возьмите целое частное от деления вашего номера на 20 и прибавьте 1
В виде формулы это $(number/20) + 1$

Домашнее задание

Внимание! Задание!

- Обратите внимание! Вы можете разрабатывать в какой-то среде, но у проверяющего (меня), может не быть этой среды. У меня вообще может быть другая операционная система. Не стоит присылать файлы проекта и тому подобное

Домашнее задание

Внимание! Задание!

- Обратите внимание! Вы можете разрабатывать в какой-то среде, но у проверяющего (меня), может не быть этой среды. У меня вообще может быть другая операционная система. Не стоит присылать файлы проекта и тому подобное
- В то же время, мне нужны инструкции для сборки, чтобы не заниматься догадками, как правильно собирать программу. Кроме того, это сейчас ваша программа может состоять из одного файла, но представьте, будто вы сдаете полноценный проект, который может быть и многокомпонентным.

Домашнее задание

Внимание! Задание!

- Обратите внимание! Вы можете разрабатывать в какой-то среде, но у проверяющего (меня), может не быть этой среды. У меня вообще может быть другая операционная система. Не стоит присылать файлы проекта и тому подобное
- В то же время, мне нужны инструкции для сборки, чтобы не заниматься догадками, как правильно собирать программу. Кроме того, это сейчас ваша программа может состоять из одного файла, но представьте, будто вы сдаете полноценный проект, который может быть и многокомпонентным.
- Ваша программа - консольная, а значит для ее сборки и запуска не нужна никакая среда. Я буду проверять их, компилируя и запуская в командной строке.

Домашнее задание

Внимание! Задание!

- Обратите внимание! Вы можете разрабатывать в какой-то среде, но у проверяющего (меня), может не быть этой среды. У меня вообще может быть другая операционная система. Не стоит присылать файлы проекта и тому подобное
- В то же время, мне нужны инструкции для сборки, чтобы не заниматься догадками, как правильно собирать программу. Кроме того, это сейчас ваша программа может состоять из одного файла, но представьте, будто вы сдаете полноценный проект, который может быть и многокомпонентным.
- Ваша программа - консольная, а значит для ее сборки и запуска не нужна никакая среда. Я буду проверять их, компилируя и запуская в командной строке.
- Минимум - должна поставляться команда сборки. В идеале - организация сборки, например makefile

Домашнее задание

Внимание! Задание!

Рекомендованная сборка:

- Makefile (или что-то иное типа cmake)

Домашнее задание

Внимание! Задание!

Рекомендованная сборка:

- Makefile (или что-то иное типа cmake)
- Сборка с -D, чтобы в зависимости от параметров была параллельная или линейная версия или количество ядер

Домашнее задание

Внимание! Задание!

Рекомендованная сборка:

- Makefile (или что-то иное типа cmake)
- Сборка с -D, чтобы в зависимости от параметров была параллельная или линейная версия или количество ядер
- Подача единым архивом (или заведите репозиторий)

Домашнее задание

Внимание! Задание!

Рекомендованная сборка:

- Makefile (или что-то иное типа cmake)
- Сборка с -D, чтобы в зависимости от параметров была параллельная или линейная версия или количество ядер
- Подача единым архивом (или заведите репозиторий)
- Сопровождающий текст с комментариями, как оно работает

Домашнее задание

Внимание! Задание!

Рекомендованная сборка:

- Makefile (или что-то иное типа cmake)
- Сборка с -D, чтобы в зависимости от параметров была параллельная или линейная версия или количество ядер
- Подача единым архивом (или заведите репозиторий)
- Сопровождающий текст с комментариями, как оно работает
- Не забудьте вывести ваш номер и написать, какую задачу вы решали!

Вопросы?