Лекция 11

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C++

Лекция 11. Параллельное программирование на языке C++

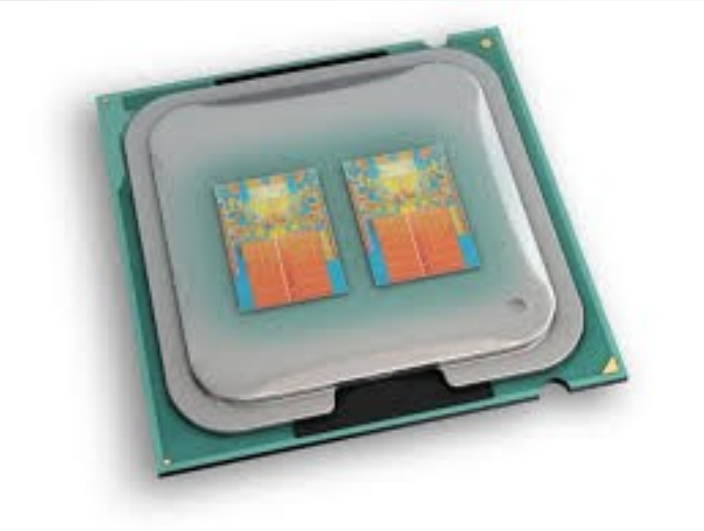
План

1. Общее понятие параллелизма.
2. Многопоточность.
3. Проблемы параллелизма.
4. Реализация многопоточности с использованием класса std::thread
5. Синхронизация конкурентного использования std::cout.
6. Асинхронные вызовы.
7. Автоматическое распараллеливание кода.

1. Общее понятие параллелизма

Под **параллелизмом** понимается одновременное выполнение двух или более операций. В контексте компьютера это означает, что система выполняет несколько независимых операций **параллельно** (одновременно), а не последовательно.

Появление и широкое распространение компьютеров, оборудованных несколькими процессорами или несколькими ядрами на одном кристалле (**многоядерными процессорами**), привело к необходимости разработки программ, поддерживающих параллелизм.



Производители аппаратного обеспечения предпочитают не наращивать тактовую частоту процессора (его быстродействие), а увеличивать количество его ядер. Т. о., **для повышения скорости работы программы нужно создавать параллельные программы.**

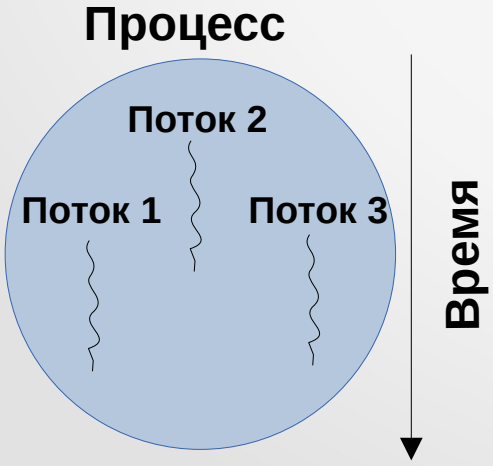
ВАЖНО! Современные процессоры даже при наличии одного ядра могут одновременно выполнять несколько команд. Это называется аппаратным параллелизмом

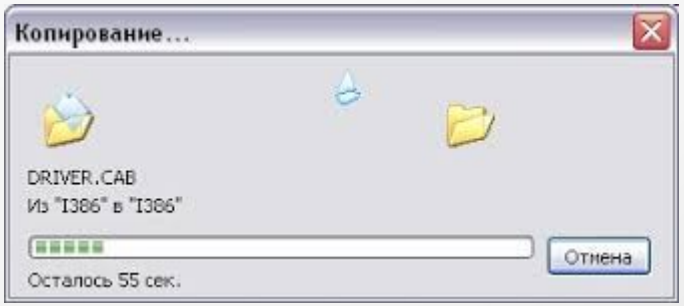
2. Многопоточность

Любой **вычислительный процесс** в современных операционных системах состоит минимум из одного потока.

Поток (**thread**) – это одно из действий внутри процесса, являющееся наименьшей единицей обработки, исполнение которой может быть назначено **планировщиком** операционной системы. Иначе говоря, поток – это отдельный путь выполнения программного кода внутри исполняемого приложения (процесса).

Процесс может быть как **одно-**, так и **многопоточным**.



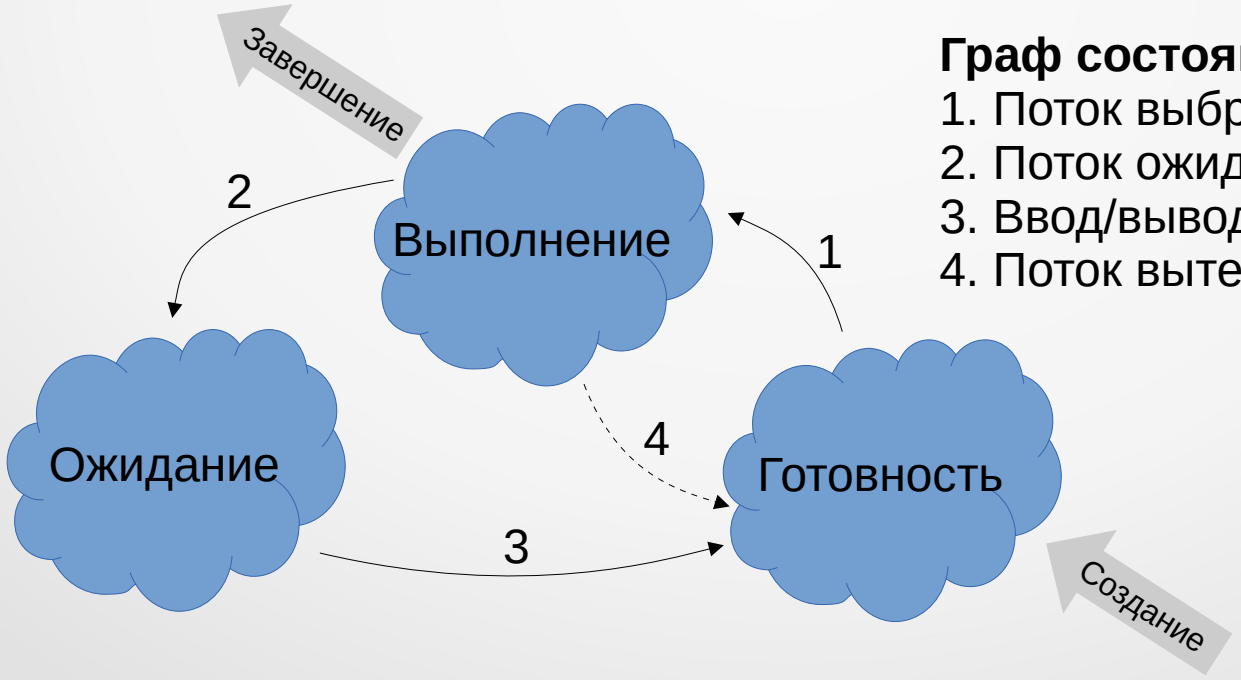


Пример многопоточного приложения в Windows: один поток копирует, другой – выполняет анимацию.

2. Многопоточность

Создание **многопоточных приложений** позволяет добиться реального **параллелизма** в работе компьютера, оборудованного несколькими **процессорами** (или **многоядерным** процессором).

Даже на **однопроцессорном** компьютере использование многопоточных программ позволяет повысить общую **реактивность** системы за счет возможности обхода **блокировок** (приложение, ожидающее ввода/вывода, блокируется системой, но использование нескольких потоков в нем позволяет реагировать на команды пользователя в других потоках).



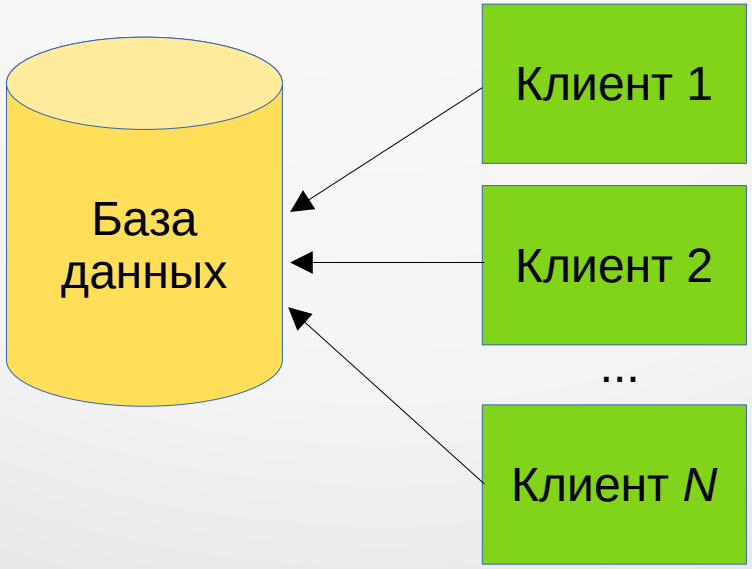
Граф состояний потока:

1. Поток выбран на выполнение.
2. Поток ожидает ввода/вывода.
3. Ввод/вывод завершен.
4. Поток вытеснен планировщиком.

3. Проблемы параллелизма

Конкурентностью называется возможность выполнения нескольких потоков в перекрывающиеся периоды времени, что может приводить к **гонкам** – проблемам **синхронизации** их доступа к некоторым общим ресурсам.

Так называемые, «**гонки за данными**» возникают, например, если поток А еще не завершился, а поток В уже пытается оперировать его данными (которые еще возможно не готовы). В этом случае поведение потоков (и процесса в целом) становится непредсказуемым и может приводить к очень **нетривиальным ошибкам**.



3. Проблемы параллелизма

Рассмотри следующий пример. Пусть в потоке выполняется оператор инкремента языка C:

```
x++; // увеличение значения целочисленной переменной x на 1
```

Байт-код, реализующий данный оператор, можно описать, например, так:

```
load x into register
add 1 to register
store register in x
```

При выполнении этих операций может возникнуть гонка. Пусть, например, x = 5. Тогда:

Шаг	Поток 1	Поток 2	x	register
1	load x into register		5	5
2	add 1 to register		5	6
3	store register in x		6	6
4		load x into register	6	6
5		add 1 to register	6	7
6		store register in x	7	7

3. Проблемы параллелизма

Так тоже допустимо:

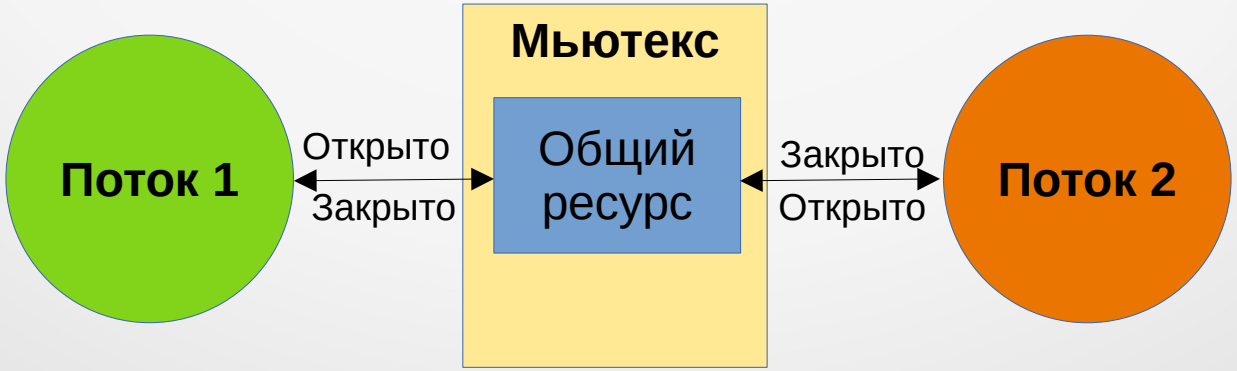
Шаг	Поток 1	Поток 2	x	register
1		load x into register	5	5
2		add 1 to register	5	6
3		store register in x	6	6
4	load x into register		6	6
5	add 1 to register		6	7
6	store register in x		7	7

А вот так уже нет:

Шаг	Поток 1	Поток 2	x	register
1	load x into register		5	5
2	add 1 to register		5	6
3		load x into register	5	5
4	store register in x		5	5
5		add 1 to register	5	6
6	store register in x		6	6

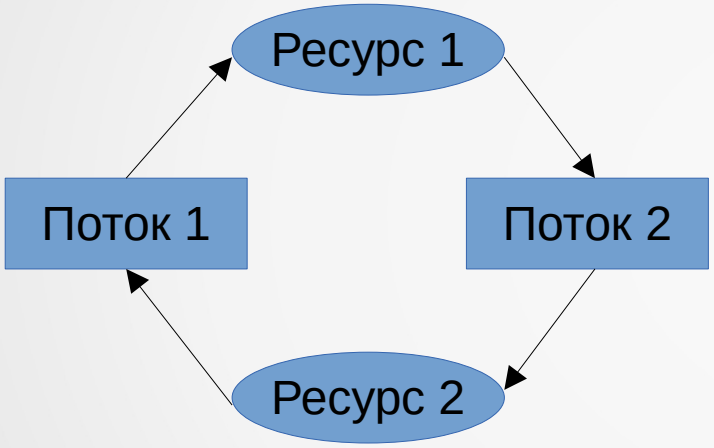
3. Проблемы параллелизма

Для устранения данной проблемы (т. е. синхронизации) потоки используют так называемые **мьютексы**, под которыми понимаются специальные объекты (**семафоры**), имеющие возможность принимать два значения (например, открыто/закрыто). Если поток обращается к мьютексу и он имеет значение «открыто», то он устанавливает его в значение «закрыто» и может монопольно использовать **критическую область кода**. Все другие потоки в этот момент блокируются. Т. о., в один момент времени только один поток может владеть мьютексом. После завершения своей работы с критической областью, поток, заблокировавший мьютекс, устанавливает его в значение «открыто» и другие потоки могут получить к нему (а, соответственно, и к критической области) доступ.



3. Проблемы параллелизма

К сожалению, использование блокировок не всегда решает проблему синхронизации потоков, т. к. на практике при реализации многопоточных приложений могут возникать так называемые **взаимные блокировки** (**клинчи**).



Взаимная блокировка двух потоков, нуждающихся в двух ресурсах

Взаимная блокировка – это ситуация, когда два потока ожидают окончания работы друг друга и, таким образом, ни один из них не может закончиться. При наличии мьютексов взаимная блокировка происходит, когда двум потокам нужны разные мьютексы, каждым из которых владеет другой.

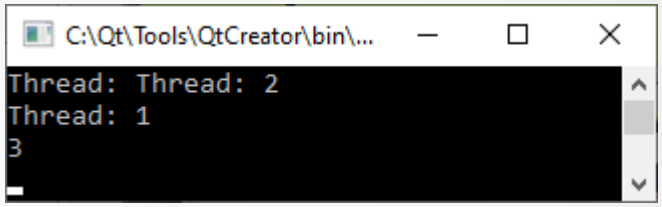
ВАЖНО! Для профилактики клинчей нужно правильным образом проектировать последовательность работы потоков приложения.

4. Реализация многопоточности с использованием класса std::thread

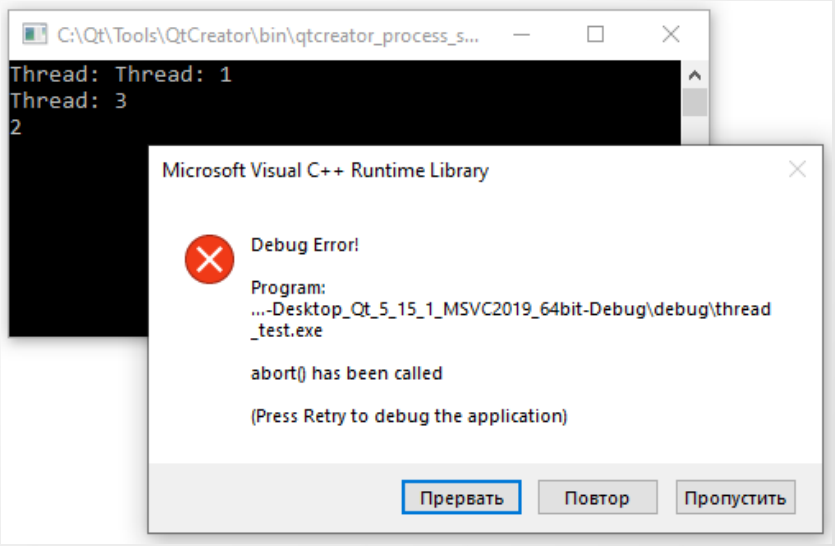
Начиная со стандарта C++11 в стандартной библиотеке STL языка C++ появился класс **std::thread**, инкапсулирующий понятие **потока выполнения**.

```
#include <iostream>
#include <thread>

using namespace std;

// Функция потока
void thread_func(int no)
{
    cout << "Thread: " << no << '\n';
}

int main()
{
    thread t1(thread_func, 1), // Создание потоков
           t2(thread_func, 2),
           t3(thread_func, 3);

    t1.join(); // Синхронизация потоков
    t2.join();
    t3.join();
    return 0;
}
```

Результат работы программы



… без синхронизации

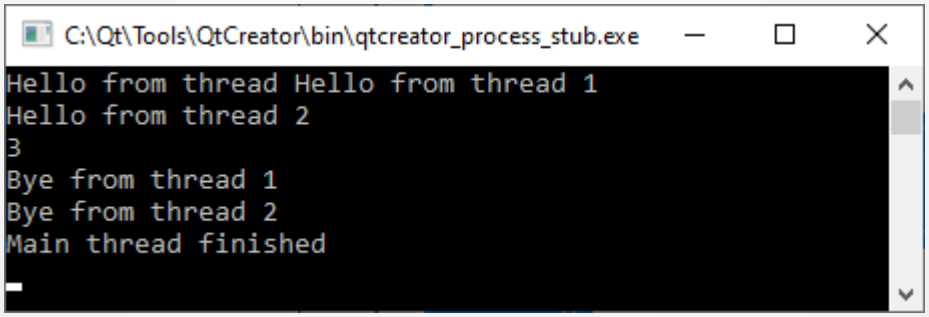


4. Реализация многопоточности с использованием класса std::thread

Еще один пример:

```
#include <iostream>
#include <thread>

using namespace std;

// Функция потока
void thread_func(int i)
{
    this_thread::sleep_for(1ms * i);
    cout << "Hello from thread " << i << '\n';
    this_thread::sleep_for(1s * i);
    cout << "Bye from thread " << i << '\n';
}
```

```
int main()
{
    thread t1(thread_func, 1),
           t2(thread_func, 2),
           t3(thread_func, 3);

    t1.join(); // Синхронизация потоков
    t2.join();
    t3.detach(); // Отсоединение потока
    cout << "Main thread finished" << endl;
    return 0;
}
```

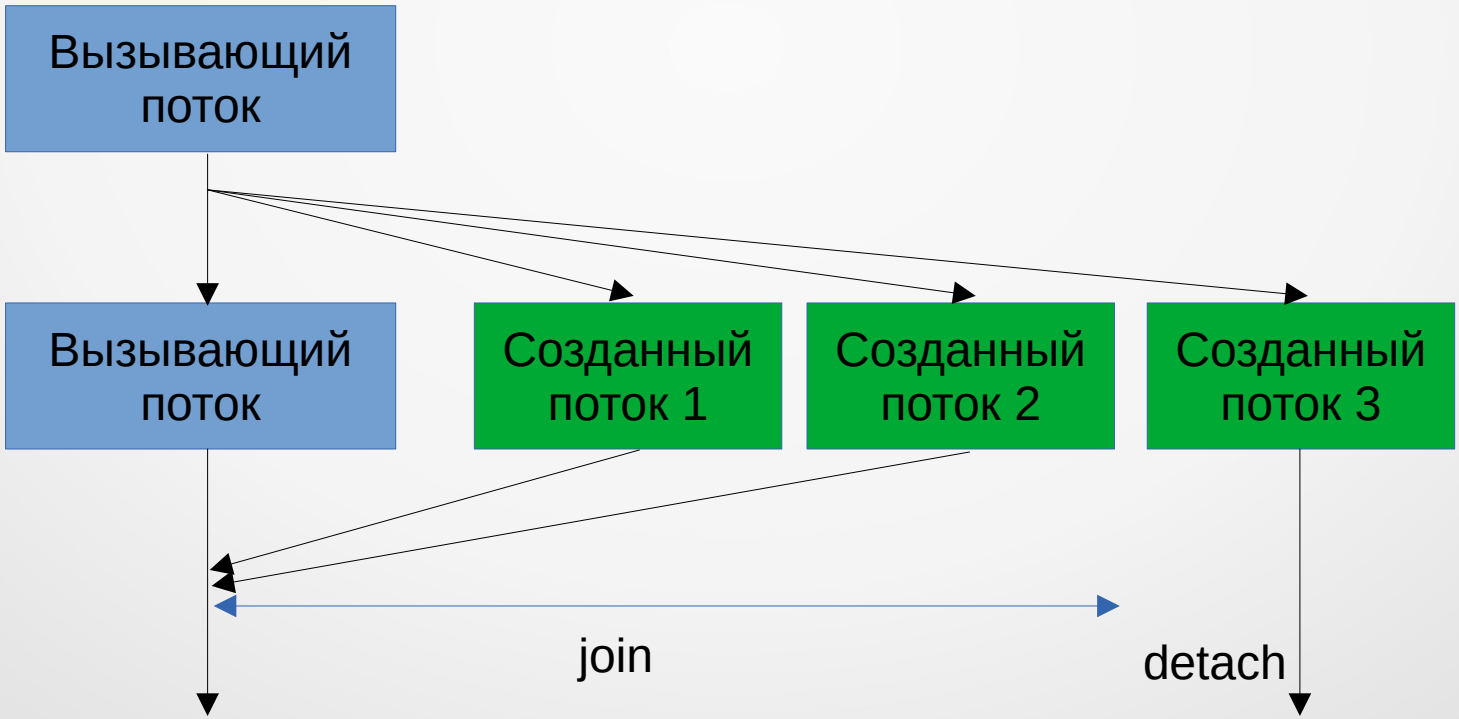


Примечание. Метод join() блокирует вызывающий поток до завершения вызванного потока, а detach() – отсоединяет поток выполнения и делает его независимым от вызывающего.

4. Реализация многопоточности с использованием класса std::thread

Метод **join()** класса **std::thread** блокирует вызывающий поток до завершения вызванного потока.

Метод **detach()** отсоединяет созданный поток выполнения и делает его независимым от вызывающего. **После этого получить доступ к нему становится невозможным**.



4. Реализация многопоточности с использованием класса std::thread

Рассмотрим пример реализации вычисления суммы ряда $\sum_{i=0}^{n} \frac{1}{(i+1)^2}$.

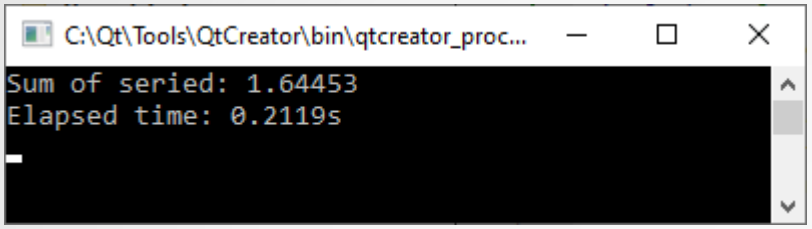
```
#include <algorithm>
#include <vector>
#include <iostream>
#include <thread>
#include <functional>
#include <chrono>

using namespace std;

// Функция, вычисляющая заданный член ряда
double fun(double i)
{
    return 1.0 / (i + 1) / (i + 1);
}
// Потоковая функция вычисления суммы заданного диапазона членов ряда
void thread_func(int begin, int end, double &sum, function<double (double)> f)
{
    for (int i = begin; i < end; i++)
        sum += f(double(i));
}
```

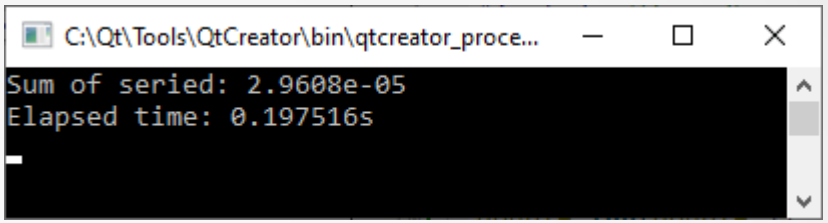
4. Реализация многопоточности с использованием класса std::thread

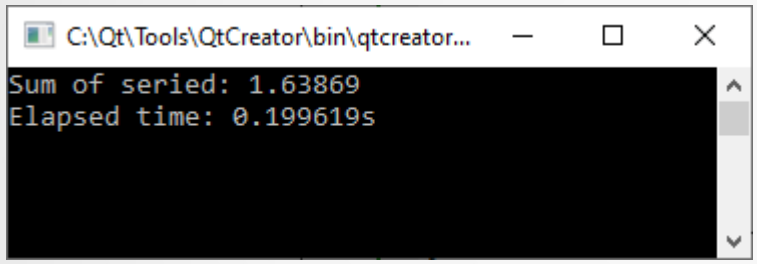
```
int main()
{
    const int max_iter = 10000000, // Количество итераций
              num_thread = 8; // Количество потоков
    int step = max_iter / num_thread; // Количество вычисляемых значений в потоке
    double sum = 0; // Итоговая сумма ряда
    vector<thread> thr(num_thread); // Массив потоков
    auto start = chrono::system_clock::now();

    // Запуск потоков
    for (int i = 0; i < num_thread; i++)
        thr[i] = thread(thread_func, i * step, i == num_thread - 1 ? max_iter : (i + 1) * step,
                        ref(sum), fun);
    // Синхронизация потоков
    for_each(thr.begin(), thr.end(), [](auto& tr) { tr.join(); });
    cout << "Sum of seried: " << sum << '\n';
    cout << "Elapsed time: " << chrono::duration<double>(chrono::system_clock::now() -
                start).count() << "s" << endl;
    return 0;
}
```

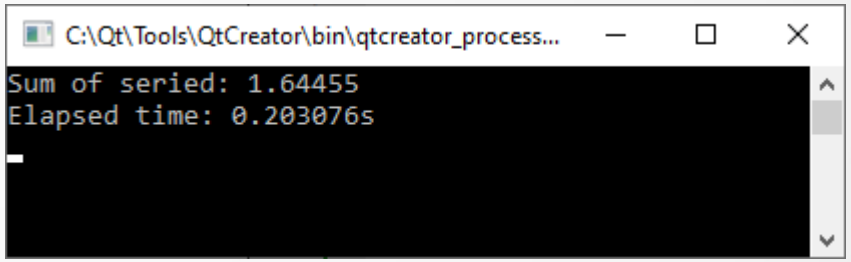
4. Реализация многопоточности с использованием класса std::thread

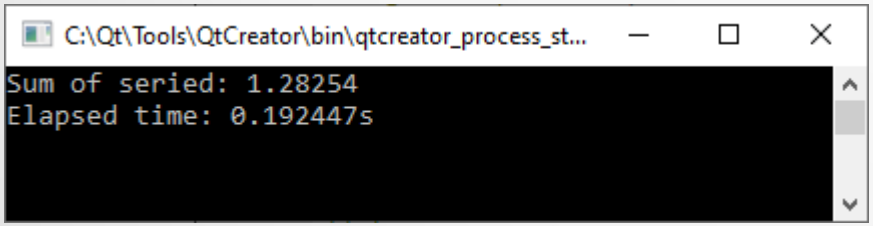
Результаты работы программы при разных запусках

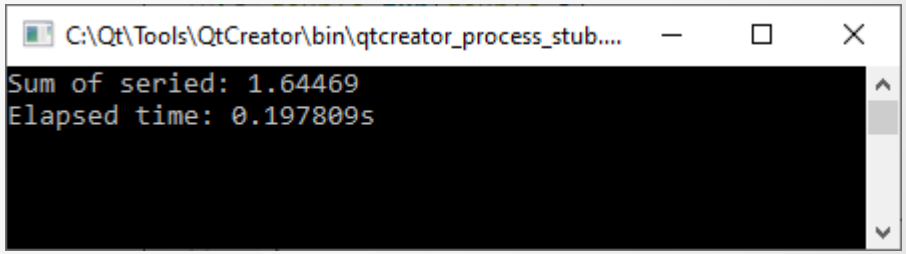










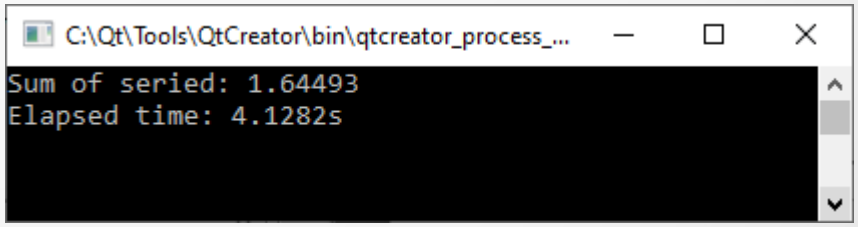


Причина расхождения – гонка за данными в потоковой функции.

4. Реализация многопоточности с использованием класса std::thread

Для исправления ситуации воспользуемся **мьютексом**. Например так:

```
#include <mutex>
// ...

mutex mtx; // Глобальный мьютекс

// ...

// Функция потока
void thread_func(int begin, int end, double &sum, function<double (double)> f)
{

    for (int i = begin; i < end; i++)
    {
        // Использование мьютекса
        lock_guard<mutex> l {mtx};
        sum += f(double(i));
    }
}

// ...
```

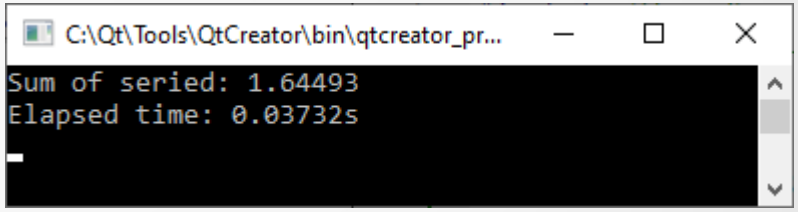


Хотя результаты работы программы перестали различаться, время ее работы существенно выросло.

4. Реализация многопоточности с использованием класса std::thread

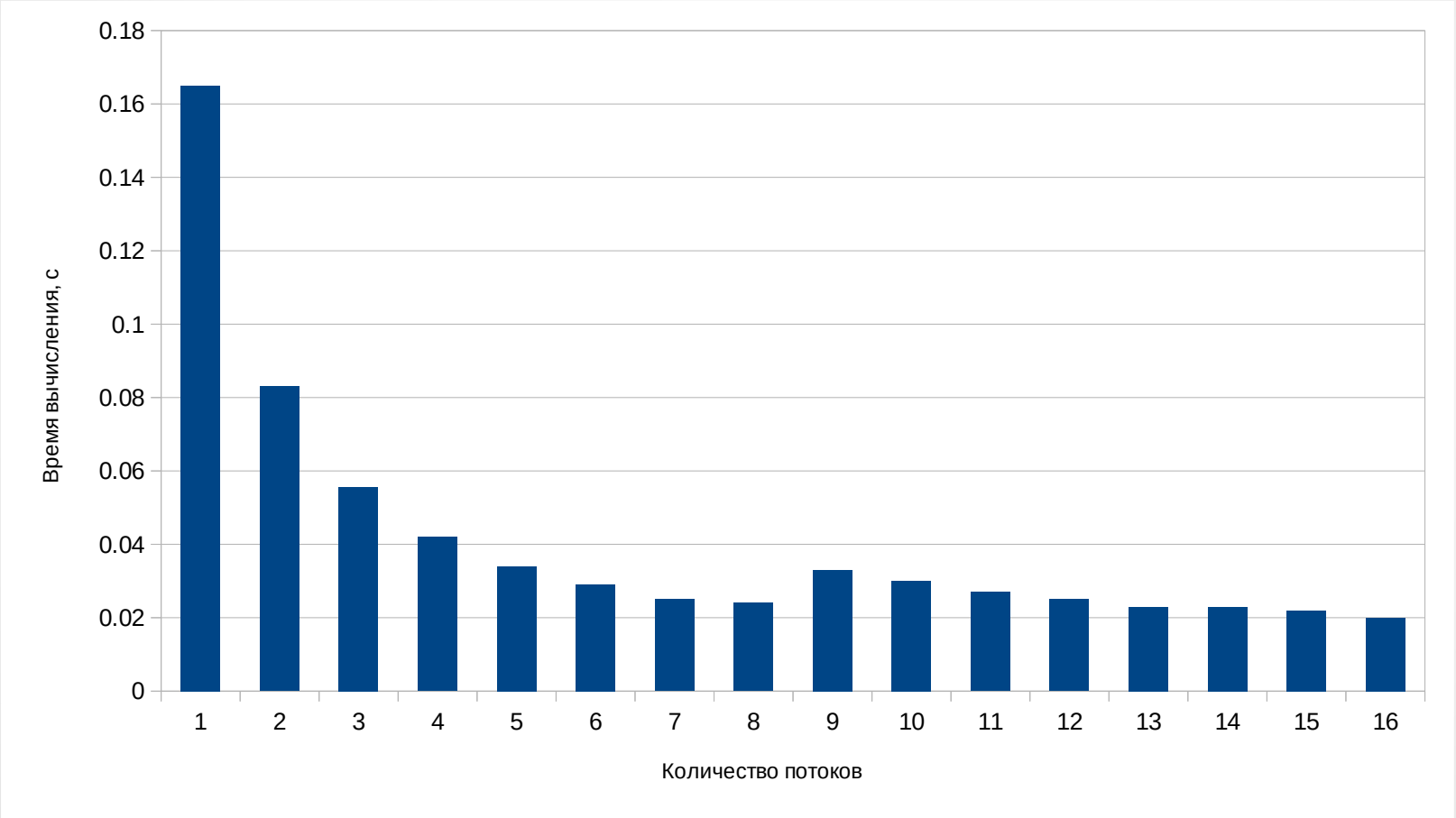
Более правильный способ:

```
// ...
// Функция потока
void thread_func(int begin, int end, double &sum, function<double (double)> f)
{
    double local_sum = 0;

    for (int i = begin; i < end; i++)
        local_sum += f(double(i));

    lock_guard<mutex> l {mtx};
    // Критическая область кода
    sum += local_sum;
}
// ...
```



Время работы существенно уменьшилось

4. Реализация многопоточности с использованием класса std::thread



Время работы программы при различном количестве использованных потоков на процессоре с 16 ядрами (8 физических и 8 виртуальных)

5. Синхронизация конкурентного использования std::cout

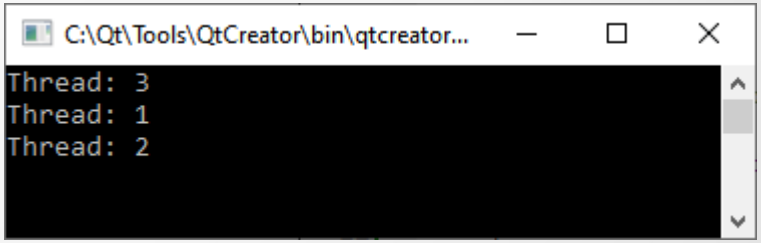
Одним из наиболее часто используемых в вычислительных потоках ресурсов является глобальный объект **std::cout**. Чтобы предотвратить конкуренцию за этот ресурс, можно, например, написать следующий код.

```
#include <iostream>
#include <sstream>
#include <thread>
#include <mutex>

using namespace std;

struct pcout : public stringstream
{
    static inline mutex cout_mutex;
    ~pcout()
    {
        lock_guard<mutex> l(cout_mutex);

        cout << rdbuf();
        cout.flush();
    }
};
```

5. Синхронизация конкурентного использования std::cout

Тогда, например, вышеприведенная программа может быть переписана так:

```
// …

// Функция потока
void thread_func(int no)
{
    pcout() << "Thread: " << no << '\n';
}

int main()
{
    thread t1(thread_func, 1), // Создание потоков
           t2(thread_func, 2),
           t3(thread_func, 3);

    t1.join(); // Синхронизация потоков
    t2.join();
    t3.join();
    return 0;
}
```

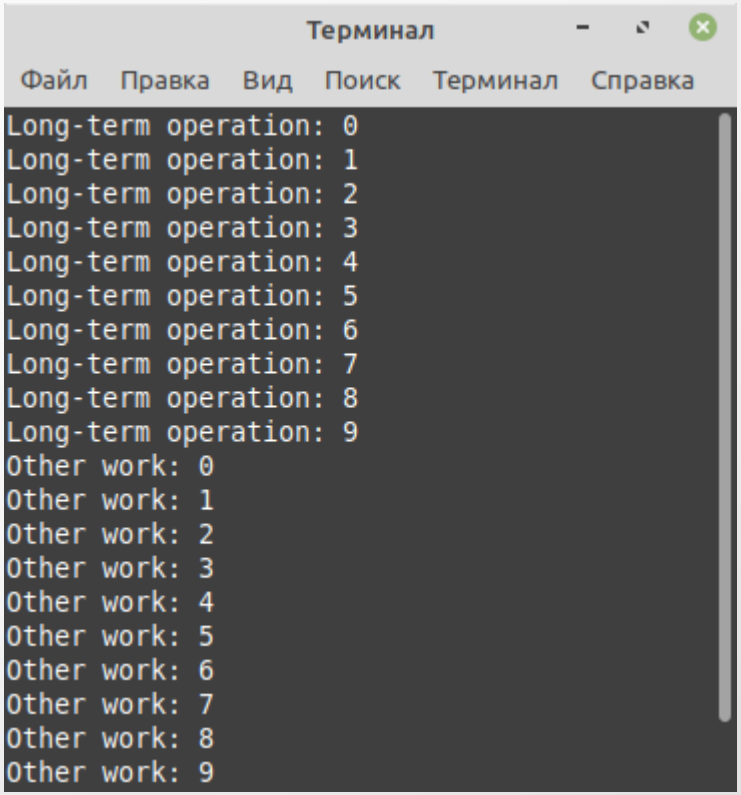


6. Асинхронные вызовы

Если однопоточное приложение выполняет какую-то длительную операцию, то пока она не закончится, приложение не сможет выполнять никаких других задач (обновлять свой интерфейс, реагировать на команды пользователя, …).

```
#include <iostream>
#include <thread>

using namespace std;

// Функция, выполняющая длительную операцию
void do_long_work(void)
{
    for (auto i = 0; i < 10; i++)
    {
        this_thread::sleep_for(1s);
        cout << "Long-term operation: " << i << '\n';
    }
}

int main()
{
    // Запуск длительной процедуры
    do_long_work();
    // Выполнить другую работу в первичном потоке...
    for (auto i = 0; i < 10; i++)
        cout << "Other work: " << i << '\n';
    return 0;
}
```

Пока не закончится длительная операция, программа не может выполнять другую работу



6. Асинхронные вызовы

Для исправления данной ситуации используется механизм **асинхронности** – возможность независимого (асинхронного) выполнения разных частей кода программы.

Для асинхронного запуска в STL используются функция **std::async()**, объявленная в заголовочном файле **future**:

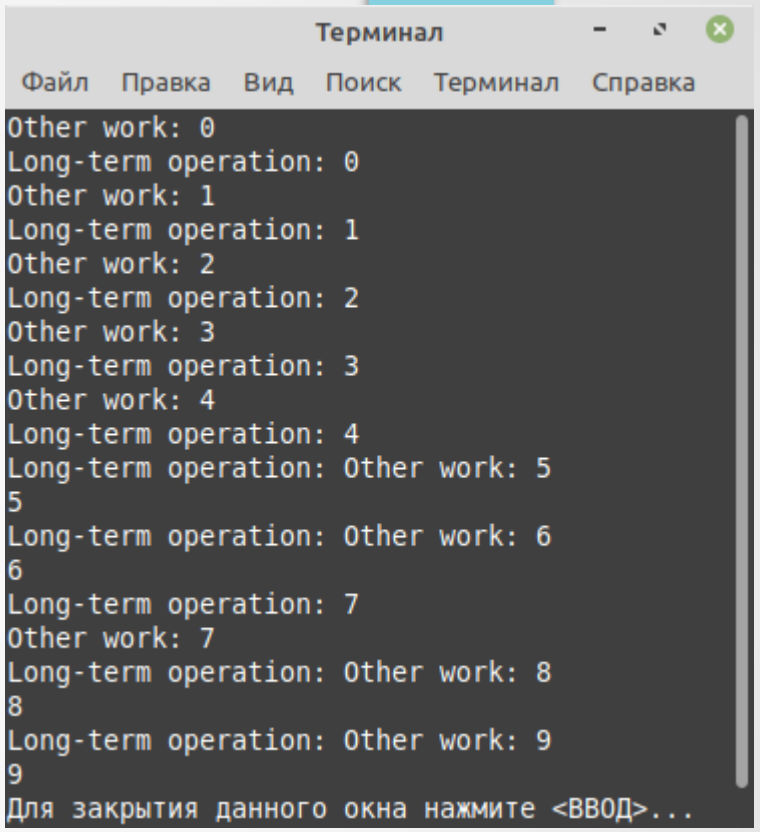
```
std::future<std::invoke_result_t<std::decay_t<Function>,
                    std::decay_t<Args>...>>
    async( std::launch policy, Function&& f, Args&&... args );
```

Т. о., асинхронный вызов инициируется с помощью вызова std::async(), который принимает в качестве аргументов функцию и ее параметры, а также, опционально, флаг, влияющий на политику вызова std::async(). Возвращаемым результатом является объект типа std::future, значение которого будет выставлено по возвращении функции. Для асинхронного запуска параметр **policy** должен иметь значение **launch::async**, тогда функция, переданная в качестве параметра в std::async(), будет запущена в новом потоке.

6. Асинхронные вызовы

Простой пример реализации асинхронности:

```
#include <iostream>
#include <thread>
#include <future>

using namespace std;

// Функция, выполняющая длительную операцию
void do_long_work(void)
{
    for (auto i = 0; i < 10; i++)
    {
        this_thread::sleep_for(0.5s);
        cout << "Long-term operation: " << i << '\n';
    }
}

int main()
{
    auto ret = async(launch::async, do_long_work); // Запуск асинхронной процедуры
    for (auto i = 0; i < 10; i++) // Выполнение другой работы...
    {
        this_thread::sleep_for(0.5s);
        cout << "Other work: " << i << '\n';
    }
    return 0;
}
```



6. Асинхронные вызовы

Рассмотрим пример асинхронного вычисления $\sum_{i=0}^{n} \frac{1}{(i+1)^2} \times \sum_{i=0}^{n} \frac{1}{(i+1)^2}$.

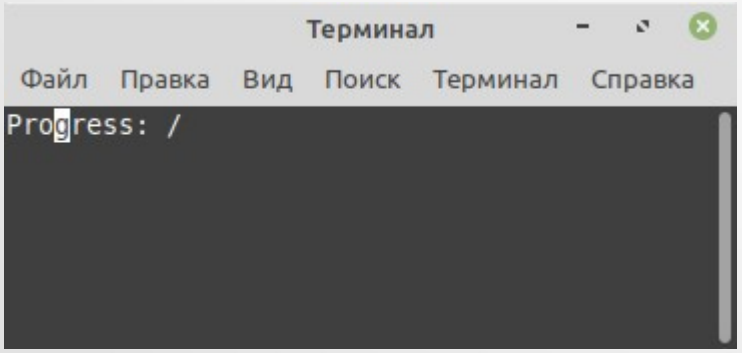
```
#include <iostream>
#include <functional>
#include <future>

using namespace std;

// Потоковая функция индикации прогресса
void progress(bool &is_work)
{
    char chr[] = { '/', '-', '\\', '|' };
    int i = 0;

    while (is_work)
    {
        this_thread::sleep_for(1ms);
        printf("\rProgress: %c", chr[i++ % 4]);
    }
}
```

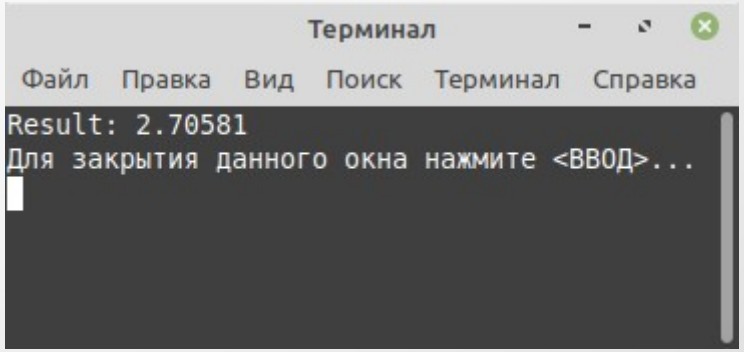
6. Асинхронные вызовы

```
// Функция, вычисляющая заданный член ряда
double fun(double i)
{
    return 1.0 / (i + 1) / (i + 1);
}

// Функция вычисления суммы заданного диапазона членов ряда
double calc_series(int begin, int end, function<double (double)> f)
{
    double sum = 0;

    for (int i = begin; i < end; i++)
        sum += f(double(i));
    return sum;
}
```

6. Асинхронные вызовы

```
int main()
{
    bool is_progress = true;
    double sum = 0;
    auto res1 = std::async(launch::async, calc_series, 0, 100000000, fun),
         res2 = std::async(launch::async, calc_series, 0, 100000000, fun);
    thread t(progress, ref(is_progress));

    t.detach();
    sum = res1.get() * res2.get();
    is_progress = false;
    cout << "\rResult: " << sum << '\n';
    return 0;
}
```



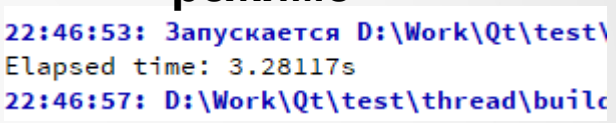


7. Автоматическое распараллеливание кода

Начиная со стандарта C++17, большинство стандартных алгоритмов, описанных в заголовочном файле **algorithm**, могут в качестве параметра принимать **политики выполнения** для работы параллельно.

```
#include <iostream>
#include <algorithm>
#include <chrono>
#include <execution>

using namespace std;

int main()
{
    vector<int> d(50000000);
    auto start = chrono::system_clock::now();

    for_each(execution::par, d.begin(), d.end(), [](auto &it) { it = rand(); });
    sort(execution::par, begin(d), end(d));

    cout << "Elapsed time: " << chrono::duration<double>(chrono::system_clock::now() -
                start).count() << "s" << endl;
    return 0;
}
```

Результат в параллельном режиме



Результат в стандартном режиме