Отчет по лабораторной работе №2

Выполнил Герасимов АД, ИУСбд-01-20, 1132210569

Example 2.1

Упражнения.
1. Доработайте программу из примера 1.2 добавив динамическое распределение области суммирования в заданном массиве.

Создайте соответствующую функцию:

```
void parallel_summ(int arr[], int length_arr, int &results) { ... }
```

Для настройки динамического распределения данных можно использовать следующие параметры:

- минимальное количество данных на поток
```
int min_per_thread = 5;
```

- максимальное необходимое количество потоков
```
int max_threads = (length_arr + min_per_thread - 1) / min_per_thread;
```

- количество потоков в системе
```
int hardware_threads = std::thread::hardware_concurrency();
```

- создаваемое количество потоков
```
int num_threads = std::min(hardware_threads != 0 ? hardware_threads : 2, max_threads);
```

- максимальный размер области данных на поток
```
int block_size = length_arr / num_threads;
```

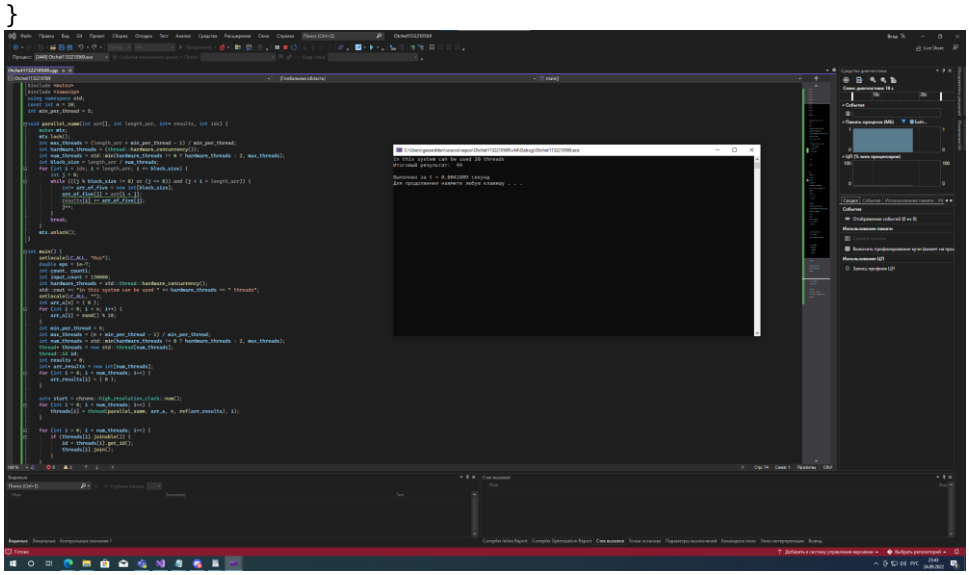
Напоминание, для динамического создания массивов воспользуйтесь:

```
int* results = new int[num_threads];
std::thread* threads = new std::thread[num_threads];
...
delete[] results;
delete[] threads;
```

```
#include <iostream>
#include <ctime>
#include <thread>
#include <chrono>
#include <algorithm>
#include <stdlib.h>
#include <mutex>
#include <iomanip>
using namespace std;
const int n = 20;
int min_per_thread = 5;


void parallel_summ(int arr[], int length_arr, int* results, int idx) {
    mutex mtx;
```

```
    mtx.lock();

    int max_threads = (length_arr + min_per_thread - 1) / min_per_thread;
    int hardware_threads = (thread::hardware_concurrency());
    int num_threads = std::min(hardware_threads != 0 ? hardware_threads : 2,
max_threads);
    int block_size = length_arr / num_threads;

    for (int i = idx; i < length_arr; i += block_size) {
        int j = 0;

        while (((j % block_size != 0) or (j == 0)) and (j + i < length_arr)) {
            int* arr_of_five = new int[block_size];

            arr_of_five[j] = arr[i + j];
            results[i] += arr_of_five[j];

            j++;
        }
        break;
    }
    mtx.unlock();
}

int main() {
    setlocale(LC_ALL, "Rus");
    double eps = 1e-7;
    int count, count1;
    int input_count = 130000;

    int hardware_threads = std::thread::hardware_concurrency();
    std::cout << "in this system can be used " << hardware_threads << " threads";

    setlocale(LC_ALL, "");
    int arr_a[n] = { 0 };
    for (int i = 0; i < n; i++) {
        arr_a[i] = rand() % 10;
    }
    int min_per_thread = 5;
    int max_threads = (n + min_per_thread - 1) / min_per_thread;
    int num_threads = std::min(hardware_threads != 0 ? hardware_threads : 2,
max_threads);

    thread* threads = new std::thread[num_threads];
    thread::id id;
    int results = 0;

    int* arr_results = new int[num_threads];

    for (int i = 0; i < num_threads; i++) {
        arr_results[i] = { 0 };
    }
    cout << '\n';
    int start1 = clock();
    auto start = chrono::high_resolution_clock::now();
    for (int i = 0; i < num_threads; i++) {
        threads[i] = thread(parallel_summ, arr_a, n, ref(arr_results), i);

    }

    for (int i = 0; i < num_threads; i++) {
        if (threads[i].joinable()) {
            id = threads[i].get_id();

            threads[i].join();
```

```
        }
    }
    delete[] threads;
    auto end = chrono::high_resolution_clock::now();
    int end1 = clock();
    double time = (double)(end1 - start1) / CLOCKS_PER_SEC;
    chrono::duration<float> duration = end - start;
    int sum_result = 0;
    for (int i = 0; i < n; i++) {
        cout << ' ' << arr_a[i];

    }
    cout << '\n';
    for (int i = 0; i < num_threads; i++) {
        cout << "Результат " << i << " - " << ' ' << arr_results[i] << '\n';
        sum_result += arr_results[i];
    }
    delete[] arr_results;
    cout << "Итоговый результат:  " << sum_result << '\n';
    cout << '\n' << "Выполнен за t = " << duration.count() << " секунд" << '\n';
    std::system("pause");
}
```



Example 2.2

Домашнее задание(базовое)
1. Вычисление числа ПИ.

Напишите программу для параллельного вычисления числа пи в потоках std::thread.
Число используемых потоков нужно определять автоматически,
исходя из необходимой точности определения числа пи (колличества слогаемых) и
располагаемых ресурсов компьютера, аналогично Exemple 2.1.

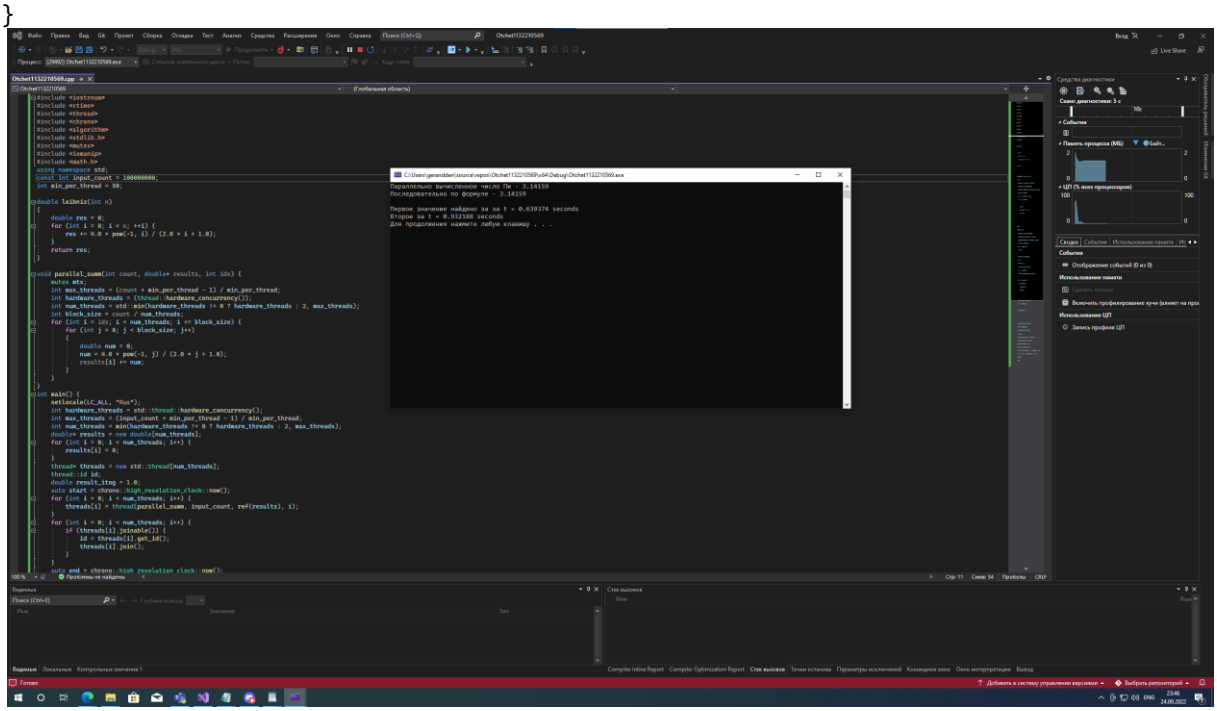
Для вычислений воспользуйтесь формулами:

- Формула Мадхавы-Лейбница (15 век)
  1 - 1/3 + 1/5 - 1/7 + 1/9 - ... = pi/4

```
#include <iostream>
#include <ctime>
```

```cpp
#include <thread>
#include <chrono>
#include <algorithm>
#include <stdlib.h>
#include <mutex>
#include <iomanip>
#include <math.h>
using namespace std;
const int input_count = 1000000;
int min_per_thread = 50;

double leibniz(int n)
{
    double res = 0;
    for (int i = 0; i < n; ++i) {
        res += 4.0 * pow(-1, i) / (2.0 * i + 1.0);
    }
    return res;
}

void parallel_summ(int count, double* results, int idx) {
    mutex mtx;

    int max_threads = (count + min_per_thread - 1) / min_per_thread;
    int hardware_threads = (thread::hardware_concurrency());
    int num_threads = std::min(hardware_threads != 0 ? hardware_threads : 2,
max_threads);
    int block_size = count / num_threads;


    for (int i = idx; i < num_threads; i += block_size) {
        for (int j = 0; j < block_size; j++)
        {
            double num = 0;
            num = 4.0 * pow(-1, j) / (2.0 * j + 1.0);
            results[i] += num;
        }
    }
}


int main() {
    setlocale(LC_ALL, "Rus");

    int hardware_threads = std::thread::hardware_concurrency();
    int max_threads = (input_count + min_per_thread - 1) / min_per_thread;
    int num_threads = min(hardware_threads != 0 ? hardware_threads : 2, max_threads);

    double* results = new double[num_threads];
    for (int i = 0; i < num_threads; i++) {
        results[i] = 0;
    }
    thread* threads = new std::thread[num_threads];
    thread::id id;

    double result_itog = 1.0;
    int start1 = clock();
    auto start = chrono::high_resolution_clock::now();

    for (int i = 0; i < num_threads; i++) {
        threads[i] = thread(parallel_summ, input_count, ref(results), i);
    }

    for (int i = 0; i < num_threads; i++) {
        if (threads[i].joinable()) {
```

```
            id = threads[i].get_id();
            threads[i].join();
        }
    }
    auto end = chrono::high_resolution_clock::now();
    int end1 = clock();
    double time = (double)(end1 - start1) / CLOCKS_PER_SEC;

    for (int i = 0; i < num_threads; i++) {
        result_itog = results[i];
    }

    int start2 = clock();
    auto start3 = chrono::high_resolution_clock::now();
    double aboba = leibniz(input_count);
    auto end3 = chrono::high_resolution_clock::now();
    int end2 = clock();
    double time2 = (double)(end2 - start2) / CLOCKS_PER_SEC;
    delete[] threads;

    cout << "Параллельно вычисленное число Пи - " << result_itog << '\n';
    cout << "Последовательно по формуле - " << aboba << '\n';
    chrono::duration<float> duration = end - start;
    chrono::duration<float> duration1 = end3 - start3;
    cout << '\n' << "Первое значение найдено за за t = " << duration.count() << "
seconds" ;
    cout << '\n' << "Второе за t = " << duration1.count() << " seconds" << '\n';
    system("pause");
    return 0;

}
```



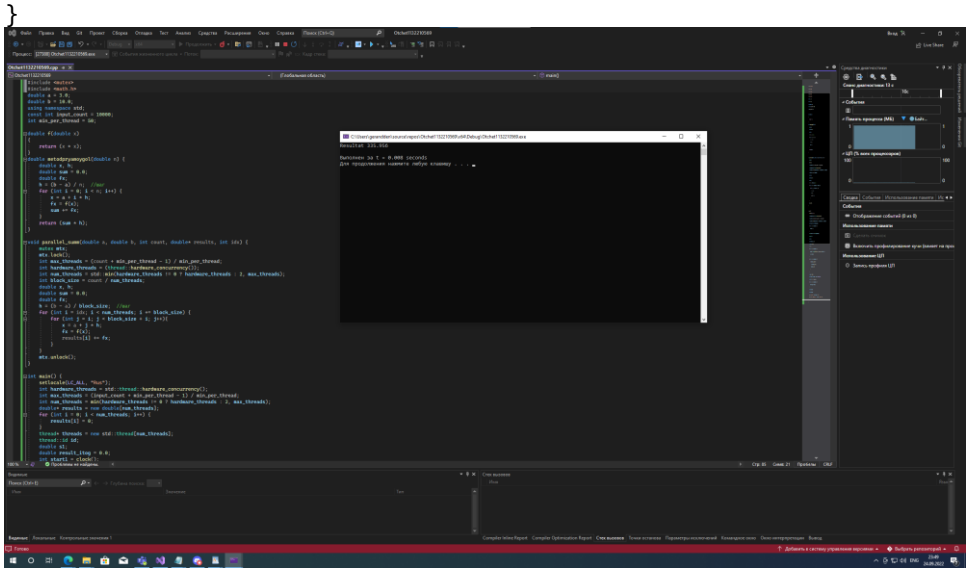
2. Вычисление интеграла.

Аналогичным образом можно разбить на ряд подзадач и вычисление определённого интеграла. Напишите паралельную реализацию
методов интегрирования по следующим алгоритмам:

2.1 Метод прямоугольников (https://ru.wikipedia.org/wiki/Метод_прямоугольников)

```cpp
#include <iostream>
#include <ctime>
#include <thread>
#include <mutex>
#include <math.h>
double a = 3.0;
double b = 10.0;

using namespace std;
const int input_count = 10000;
int min_per_thread = 50;

double f(double x)
{
    return (x * x);
}

double metodpryamoygol(double n) {//максимальное количесвто шагов интегрирования, чем
больше- тем точнее результат
    double x, h;
    double sum = 0.0;
    double fx;

    h = (b - a) / n;  //шаг

    for (int i = 0; i < n; i++) {
        x = a + i * h;
        fx = f(x);
        sum += fx;
    }
    return (sum * h);
}

void parallel_summ(double a, double b, int count, double* results, int idx) {
    mutex mtx;
    mtx.lock();
    int max_threads = (count + min_per_thread - 1) / min_per_thread;
    int hardware_threads = (thread::hardware_concurrency());
    int num_threads = std::min(hardware_threads != 0 ? hardware_threads : 2,
max_threads);

    int block_size = count / num_threads;

    double x, h;
    double sum = 0.0;
    double fx;

    h = (b - a) / block_size;  //шаг
    for (int i = idx; i < num_threads; i += block_size) {
        for (int j = i; j < block_size + i; j++) {
            x = a + j * h;
            fx = f(x);
            results[i] += fx;
        }

    }
    mtx.unlock();
}


int main() {
    setlocale(LC_ALL, "Rus");

    int hardware_threads = std::thread::hardware_concurrency();
    int max_threads = (input_count + min_per_thread - 1) / min_per_thread;
```

```
    int num_threads = min(hardware_threads != 0 ? hardware_threads : 2, max_threads);

    double* results = new double[num_threads];
    for (int i = 0; i < num_threads; i++) {
        results[i] = 0;
    }

    thread* threads = new std::thread[num_threads];
    thread::id id;

    double s1;
    double result_itog = 0.0;
    int start1 = clock();

    s1 = 0.0;

    for (int i = 0; i < num_threads; i++) {

        threads[i] = thread(parallel_summ, a, b, input_count, ref(results), i);
        s1 += results[i];
    }

    for (int i = 0; i < num_threads; i++) {
        if (threads[i].joinable()) {
            id = threads[i].get_id();
            threads[i].join();
        }
    }

    int end1 = clock();
    double time = (double)(end1 - start1) / CLOCKS_PER_SEC;
    double h = (b - a) / input_count;
    for (int i = 0; i < num_threads; i++) {
        cout << "Промежуточный результат : " << i << ' ' << " : " << results[i] << '\n';
        result_itog += results[i];
    }
    delete[] results;

    delete[] threads;

    cout << "Resultat " << result_itog * h << '\n';

    cout << '\n' << "Выполнен за t = " << time << " seconds" << '\n';
    return 0;
}
```



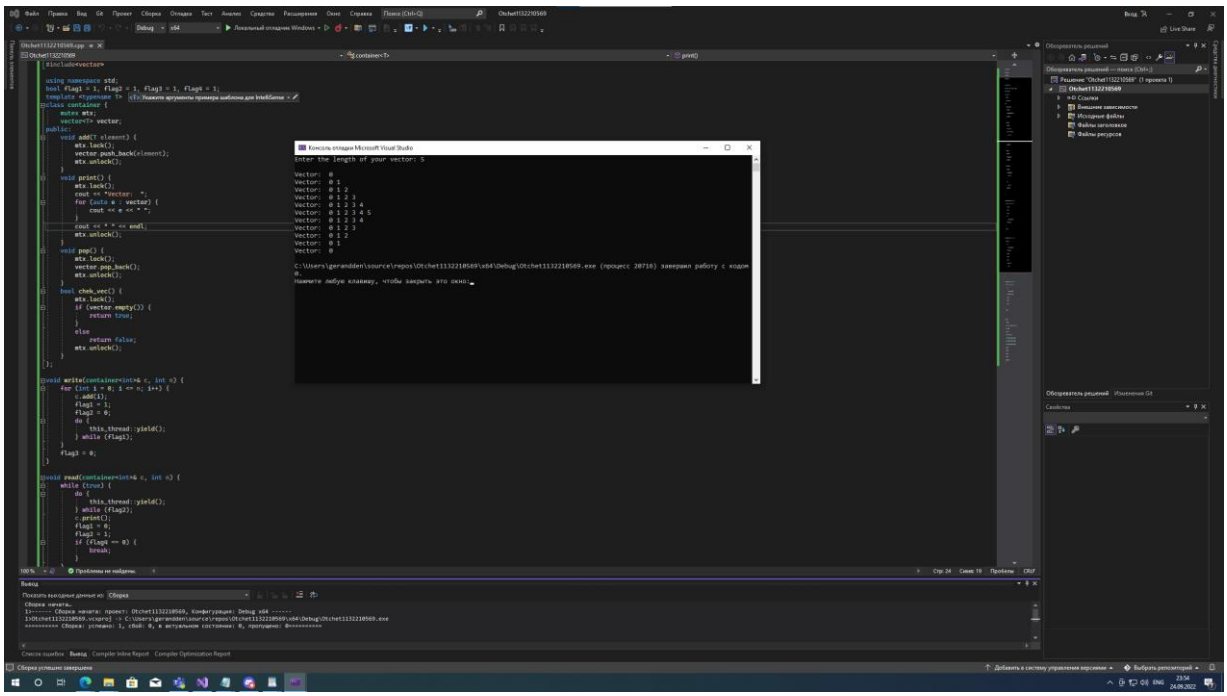
Example 2.3

Упражнения.

1. Напишите программу состоящую из трёх потоков работающих с вектором данных из чисел типа int:
- первый поток последовательно записывает в вектор от 0 до n,
- второй последовательно удаляет значения с конца вектора
- третий выводит состояние вектора после хотябы одного изменения его состояния
Работа программы завершается когда вектор оказывается пустым.

```
#include<iostream>
#include<cstdlib>
#include<thread>
#include<mutex>
#include<vector>

using namespace std;

bool flag1 = 1, flag2 = 1, flag3 = 1, flag4 = 1;

template <typename T>
class container {

    mutex mtx;
    vector<T> vector;

public:

    void add(T element) {
        mtx.lock();
        vector.push_back(element);
        mtx.unlock();
    }

    void print() {
        mtx.lock();
        cout << "Vector: | ";
        for (auto e : vector) {
            cout << e << " ";
        }
        cout << "|" << endl;
        mtx.unlock();
    }

    void pop() {
        mtx.lock();
        vector.pop_back();
        mtx.unlock();
    }

    bool chek_vec() {
        mtx.lock();
        if (vector.empty()) {
            return true;
        }
        else
            return false;
        mtx.unlock();
    }
};
```

```
void write(container<int>& c, int n) {

    for (int i = 0; i <= n; i++) {
        c.add(i);
        flag1 = 1;
        flag2 = 0;
        do {
            this_thread::yield();
        } while (flag1);
    }

    flag3 = 0;
}

void read(container<int>& c, int n) {
    while (true) {
        do {
            this_thread::yield();
        } while (flag2);
        c.print();
        flag1 = 0;
        flag2 = 1;
        if (flag4 == 0) {
            break;
        }
    }

}

void clean(container<int>& c, int n) {
    for (int i = 0; i < n; i++) {
        do {
            this_thread::yield();
        } while (flag1 && flag3);
        c.pop();
        flag2 = 0;
        flag1 = 1;
    }

    flag4 = 0;

}

int main() {

    int length;
    cout << "Enter the length of your vector: ";
    cin >> length;
    cout << "\n";

    container<int> vector;
    thread t1(write, ref(vector), length);
    thread t2(read, ref(vector), length);
    thread t3(clean, ref(vector), length);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```



Домашнее задание(базовое)

1. Ранее мы рассматривали (лекция 3) алгоритм вычисления чисел фибоначи на основе матричного произведения и быстрого
возведения в степень.

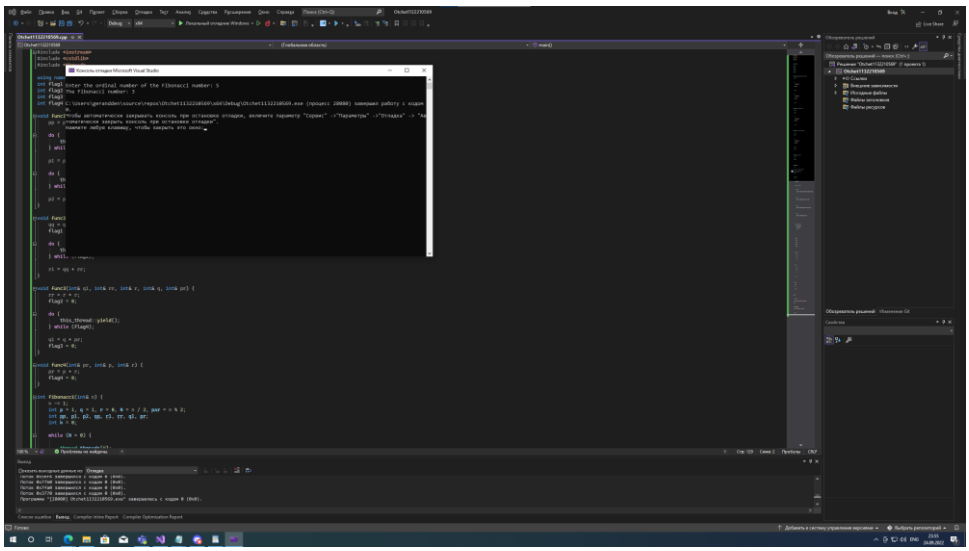
```
def  Fibonacci(n):

    p, q, r = 1, 1, 0
    N = n//2; par = n%2
    while N > 0:

      # Thread_1        Thread_2          Thread_3      Thread_4

        pp = p*p;       qq = q*q;         rr = r*r;     pr = p + r;        #
Такт 1
        p1 = pp + qq;   r1 = qq + rr;     q1 = q*pr;                       #
Такт 2
        p2 = p1 + q1;                                                      #
Такт 3

        if N % 2 == 1:   p, q, r = p2, p1, q1
        else:            p, q, r = p1, q1, r1
        N = N // 2

    if par==0: return r
    else:      return q
```

Напишите программу использующую паралельные потоки для реализации этого алгоритма и оцените время работы.

```
#include <iostream>
```

```cpp
#include <cstdlib>
#include <thread>

using namespace std;
int flag1 = 1;
int flag2 = 1;
int flag3 = 1;
int flag4 = 1;

void func1(int& p, int& pp, int& p1, int& qq, int& p2, int& q1) {
    pp = p * p;

    do {
        this_thread::yield();
    } while (flag1);

    p1 = pp + qq;

    do {
        this_thread::yield();
    } while (flag3);

    p2 = p1 + q1;
}

void func2(int& qq, int& rr, int& r1, int& q) {
    qq = q * q;
    flag1 = 0;

    do {
        this_thread::yield();
    } while (flag2);

    r1 = qq + rr;
}

void func3(int& q1, int& rr, int& r, int& q, int& pr) {
    rr = r * r;
    flag2 = 0;

    do {
        this_thread::yield();
    } while (flag4);

    q1 = q * pr;
    flag3 = 0;
}

void func4(int& pr, int& p, int& r) {
    pr = p + r;
    flag4 = 0;
}

int fibonacci(int& n) {
    n -= 1;
    int p = 1, q = 1, r = 0, N = n / 2, par = n % 2;
    int pp, p1, p2, qq, r1, rr, q1, pr;
    int k = 0;

    while (N > 0) {

        thread threads[4];

        for (int i = 0; i < 4; i++) {
```

```
            if (i == 0) {
                threads[i] = thread(func1, ref(p), ref(pp), ref(p1), ref(qq), ref(p2),
ref(q1));
            }

            if (i == 1) {
                threads[i] = thread(func2, ref(qq), ref(rr), ref(r1), ref(q));
            }

            if (i == 2) {
                threads[i] = thread(func3, ref(q1), ref(rr), ref(r), ref(q), ref(pr));
            }

            if (i == 3) {
                threads[i] = thread(func4, ref(pr), ref(p), ref(r));
            }
        }

        for (int i = 0; i < 4; i++) {
            if (threads[i].joinable()) {
                threads[i].join();
                k++;
            }
        }

        if (k == 4) {
            flag1 = 1;
            flag2 = 1;
            flag3 = 1;
            flag4 = 1;
        }

        if (N % 2 == 1) {//если ввели нечетный номер числа Ф
            p = p2;
            q = p1;
            r = q1;
        }

        else {
            p = p1;
            q = q1;
            r = r1;
        }

        N = N / 2;
    }

    if (par == 0)
        return r;
    else
        return q;
}

int main() {

    int n, result;
    cout << "\nEnter the ordinal number of the Fibonacci number: ";
    cin >> n;

    result = fibonacci(n);
    cout << "The Fibonacci number: " << result << "\n";

    return 0;
}
```



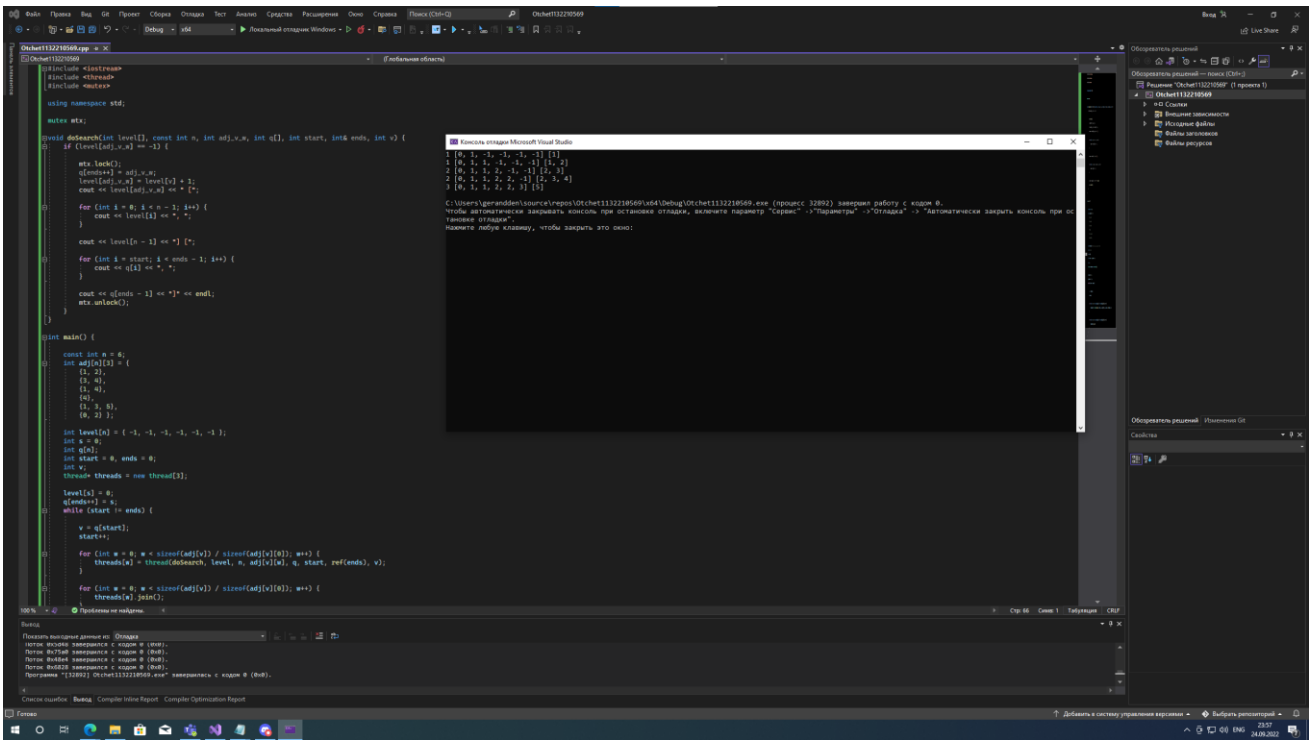
2. Ранее мы рассматривали алгоритм поиска в ширину на ориетированном не взвешенном графе. Состоящего из функций

```
def bfs(graph, s, out=0):
    parents = {v: None for v in graph}
    level = {v: None for v in graph}
    level[s] = 0                            # уровень начальной вершины
    queue = [s]                             # добавляем начальную вершину в очередь
    while queue:                            # пока там что-то есть
        v = queue.pop(0)                    # извлекаем вершину
        for w in graph[v]:                  # запускаем обход из вершины v
            if level[w] is None:            # проверка на посещенность
                queue.append(w)             # добавление вершины в очередь
                parents[w] = v
                level[w] = level[v] + 1     # подсчитываем уровень вершины
        if out: print(level[w], level, queue)
    return level, parents

def PATH (end, parents):
    path = [end]
    parent = parents[end]
    while not parent is None:
        path.append(parent)
        parent = parents[parent]
    return path[::-1]
```

Рассмотрите возможность распараеливания этого алгоритма и напишите соответстствующую программу.

```
*/
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;
```

```
mutex mtx;

void doSearch(int level[], const int n, int adj_v_w, int q[], int start, int& ends, int
v) {
	if (level[adj_v_w] == -1) {

		mtx.lock();
		q[ends++] = adj_v_w;
		level[adj_v_w] = level[v] + 1;
		cout << level[adj_v_w] << " [";

		for (int i = 0; i < n - 1; i++) {
			cout << level[i] << ", ";
		}

		cout << level[n - 1] << "] [";

		for (int i = start; i < ends - 1; i++) {
			cout << q[i] << ", ";
		}

		cout << q[ends - 1] << "]" << endl;
		mtx.unlock();
	}
}

int main() {

	const int n = 6;
	int adj[n][3] = {
		{1, 2},
		{3, 4},
		{1, 4},
		{4},
		{1, 3, 5},
		{0, 2} };

	int level[n] = { -1, -1, -1, -1, -1, -1 };
	int s = 0;
	int q[n];
	int start = 0, ends = 0;
	int v;
	thread* threads = new thread[3];

	level[s] = 0;
	q[ends++] = s;
	while (start != ends) {

		v = q[start];
		start++;

		for (int w = 0; w < sizeof(adj[v]) / sizeof(adj[v][0]); w++) {
			threads[w] = thread(doSearch, level, n, adj[v][w], q, start,
ref(ends), v);
		}

		for (int w = 0; w < sizeof(adj[v]) / sizeof(adj[v][0]); w++) {
			threads[w].join();
		}
	}
}
```


Otchet1132210569.cpp
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
mutex mtx;
void deSearch(int level[], const int n, int adj_v_w, int q[], int start, int& ends, int v) {
if (level[adj_v_w] == -1) {
mtx.lock();
q[ends++] = adj_v_w;
level[adj_v_w] = level[v] + 1;
cout << level[adj_v_w] << " [";
for (int i = 0; i < n - 1; i++) {
cout << level[i] << ", ";
}
cout << level[n - 1] << "] [";
for (int i = start; i < ends - 1; i++) {
cout << q[i] << ", ";
}
cout << q[ends - 1] << "]" << endl;
mtx.unlock();
}
}
int main() {
const int n = 6;
int adj[n][3] = {
{1, 2},
{3, 4},
{1, 4},
{4},
{1, 3, 5},
{0, 2} };
int level[n] = { -1, -1, -1, -1, -1, -1 };
int s = 0;
int q[n];
int start = 0, ends = 0;
int v;
thread* threads = new thread[3];
level[s] = 0;
q[ends++] = s;
while (start != ends) {
v = q[start];
start++;
for (int w = 0; w < sizeof(adj[v]) / sizeof(adj[v][0]); w++) {
threads[w] = thread(deSearch, level, n, adj[v][w], q, start, ref(ends), v);
}
for (int w = 0; w < sizeof(adj[v]) / sizeof(adj[v][0]); w++) {
threads[w].join();
Консоль отладки Microsoft Visual Studio
1 [0, 1, -1, -1, -1, -1] [1]
1 [0, 1, 1, -1, -1, -1] [1, 2]
2 [0, 1, 1, 2, -1, -1] [2, 3]
2 [0, 1, 1, 2, 2, -1] [2, 3, 4]
3 [0, 1, 1, 2, 2, 3] [5]
C:\Users\gerandden\source\repos\Otchet1132210569\x64\Debug\Otchet1132210569.exe (процесс 32892) завершил работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:
Вывод
Поток 0x75a0 завершился с кодом 0 (0x0).
Поток 0x48e4 завершился с кодом 0 (0x0).
Поток 0x6828 завершился с кодом 0 (0x0).
Программа "[32892] Otchet1132210569.exe" завершилась с кодом 0 (0x0).