МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники

ОТЧЕТ

по лабораторной работе № 3.1

по дисциплине «Операционные системы»

ТЕМА: «Процессы и потоки»

Студент гр. 3311 Баймухамедов Р. Р.

Преподаватель Тимофеев А. В.

Санкт-Петербург

2025

Цель работы

Исследовать механизмы создания и управления процессами и потоками в ОС Windows

Задание

Постановка задачи и описание решения

Для выполнения данной лабораторной работы необходимо разработать консольное приложение, которое вычисляет число Пи с точностью N знаков после запятой по следующей формуле:

$$\pi = \left(\frac{4}{1+x_0^2} + \frac{4}{1+x_1^2} + \ldots + \frac{4}{1+x_{N-1}^2} \right) \times \frac{1}{N}, \text{ где } x_i = (i+0.5) \times \frac{1}{N}, \; i = \overline{0, N-1}$$

где N=10000000.

Использовать распределение итераций блоками (размер блока = 10 * 331103) по потокам. Сначала каждый поток по очереди получает свой блок итераций, затем тот поток, который заканчивает выполнение своего блока, получает следующий свободный блок итераций. Освободившиеся потоки получают новые блоки итераций до тех пор, пока все блоки не будут исчерпаны. Создание потоков будет осуществляться с помощью функции Win32 API CreateThread.

Для реализации механизма распределения блоков итераций необходимо сразу в начале программы создать необходимое количество потоков в приостановленном состоянии. Для этого будет использоваться функция Win32 API ResumeThread.

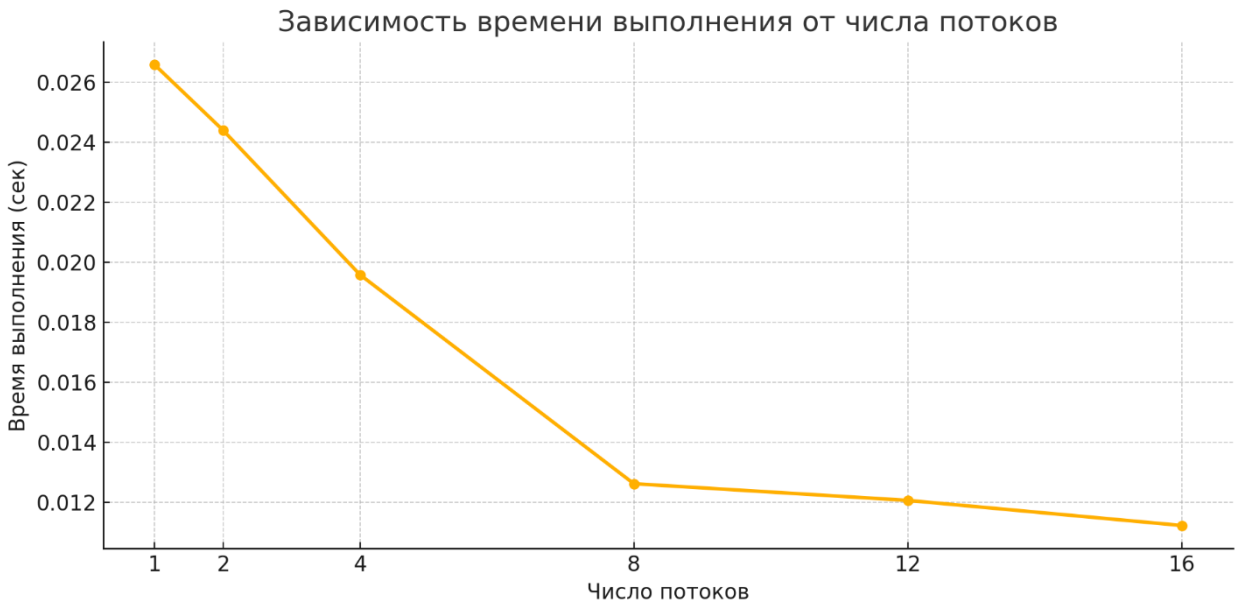
По окончании обработки текущего блока итераций поток не должен завершаться, а должен быть приостановлен с помощью функции Win32 API SuspendThread. Затем потоку должна быть предоставлен следующий свободный блок итераций, и поток должен быть освобожден (ResumeThread).

Провести замеры времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12 и 16)

Пример работоспособности приложения

```
PS C:\GitHub\course-2\OS\lab_03> cd "c:\GitHub\course-2\OS\lab_03\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
Threads: 1 | PI equal 3.14159265358973 | Time: 0.026582700000000 sec
Threads: 2 | PI equal 3.141592653589732 | Time: 0.024399400000000 sec
Threads: 4 | PI equal 3.141592653589732 | Time: 0.019583100000000 sec
Threads: 8 | PI equal 3.141592653589732 | Time: 0.012617300000000 sec
Threads: 12 | PI equal 3.141592653589732 | Time: 0.012061400000000 sec
Threads: 16 | PI equal 3.141592653589732 | Time: 0.011224800000000 sec
```

График



Заключение

В данной лабораторной работе разработано многопоточное приложение для вычисления числа Пи с точностью N=10000000 знаков после запятой с использованием Win 32 API (CreateThread, SuspendThread, ResumeThread). Каждый поток брал блок (по 3311030 итераций), обрабатывал его, приостанавливался при помощи функции Win 32 API SuspendThread и получал новый свободный блок при помощи функции Win 32 API ResumeThread.

На графике видно, что время вычисления числа Пи уменьшается с 0,026 секунд при работе одного потока до 0,012 секунд при шестнадцати потоках. Однако начиная с 8 потоков прирост производительности замедляется и остается примерно на уровне 0,012 секунд. Это происходит из-за ограничений процессора компьютер и накладных расходов на многопоточность.

Код программы

```
#include <windows.h>
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const int N = 10000000;
const int BLOCK_SIZE = 3311030; // 331103 numb of stud bilet
const int TOTAL_BLOCKS = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;

struct thread_data {
    int thread_id;
};

double pi = 0.0;
int current_block = 0;
CRITICAL_SECTION pi_cs;
CRITICAL_SECTION block_cs;

HANDLE* threads;
thread_data* thread_info;
int num_threads;
```

```
DWORD WINAPI compute_pi(LPVOID param) {
    thread_data* data = static_cast<thread_data*>(param);

    while (true) {
        SuspendThread(GetCurrentThread());

        int block_id;

        EnterCriticalSection(&block_cs);
        block_id = current_block++;
        LeaveCriticalSection(&block_cs);

        if (block_id >= TOTAL_BLOCKS) break;

        int start = block_id * BLOCK_SIZE;
        int end = min(start + BLOCK_SIZE, N);
        double local_sum = 0.0;

        for (int i = start; i < end; ++i) {
            double x = (i + 0.5) / N;
            local_sum += 4.0 / (1.0 + x * x);
        }

        EnterCriticalSection(&pi_cs);
        pi += local_sum;
        LeaveCriticalSection(&pi_cs);
    }
    return 0;
}

int main() {
    InitializeCriticalSection(&pi_cs);
    InitializeCriticalSection(&block_cs);

    vector<int> thread_counts = {1, 2, 4, 8, 12, 16};

    for (int count : thread_counts) {
        pi = 0.0;
        current_block = 0;
        num_threads = count;

        threads = new HANDLE[num_threads];
        thread_info = new thread_data[num_threads];

        for (int i = 0; i < num_threads; ++i) {
            thread_info[i].thread_id = i;
            threads[i] = CreateThread(NULL, 0, compute_pi, &thread_info[i], CREATE_SUSPENDED, NULL);
        }

        LARGE_INTEGER frequency, start, end;
        QueryPerformanceFrequency(&frequency);
        QueryPerformanceCounter(&start);

        for (int i = 0; i < num_threads; ++i) {
            ResumeThread(threads[i]);
        }

        bool all_done = false;
        while (!all_done) {
            all_done = true;
            for (int i = 0; i < num_threads; ++i) {
                DWORD code;
                GetExitCodeThread(threads[i], &code);
                if (code == STILL_ACTIVE) {
                    ResumeThread(threads[i]);
                    all_done = false;
                }
            }
        }
```

```
        WaitForMultipleObjects(num_threads, threads, TRUE, INFINITE);
        QueryPerformanceCounter(&end);

        double elapsed_time = static_cast<double>(end.QuadPart - start.QuadPart) / frequency.QuadPart;
        pi /= N;

        cout.precision(15);
        cout << "Threads: " << num_threads << " | PI equal " << pi << " | Time: " << fixed << elapsed_time << " sec" << endl;

        for (int i = 0; i < num_threads; ++i)
            CloseHandle(threads[i]);

        delete[] threads;
        delete[] thread_info;
    }

    DeleteCriticalSection(&pi_cs);
    DeleteCriticalSection(&block_cs);

    return 0;
}
```