**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра Вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе №3**

**по дисциплине «Операционные системы»**

**Тема: «Процессы и потоки»**

Студентки гр. 3312 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Половникова А.С.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тимофеев А.В.

Санкт-Петербург

2025

**Цель работы.**

Исследовать механизмы создания и управленияпроцессами и потоками в ОС Windows.

**Задание 3.1 и 3.2.**

Реализация многопоточного приложения с использованием функций Win32 API.

**Указания к выполнению.**

1. Создайте приложение, которое вычисляет число pi с точностью N знаков после запятой по следующей формуле

где N=10000000.

Используйте распределение итераций блоками (размер блока = 10 \* Nстудбилета) 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 выполняйте 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 освобождения потока из приостановленного состояния используйте функцию 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 предоставлена следующий свободный блок итераций, и поток должен быть освобожден (ResumeThread).

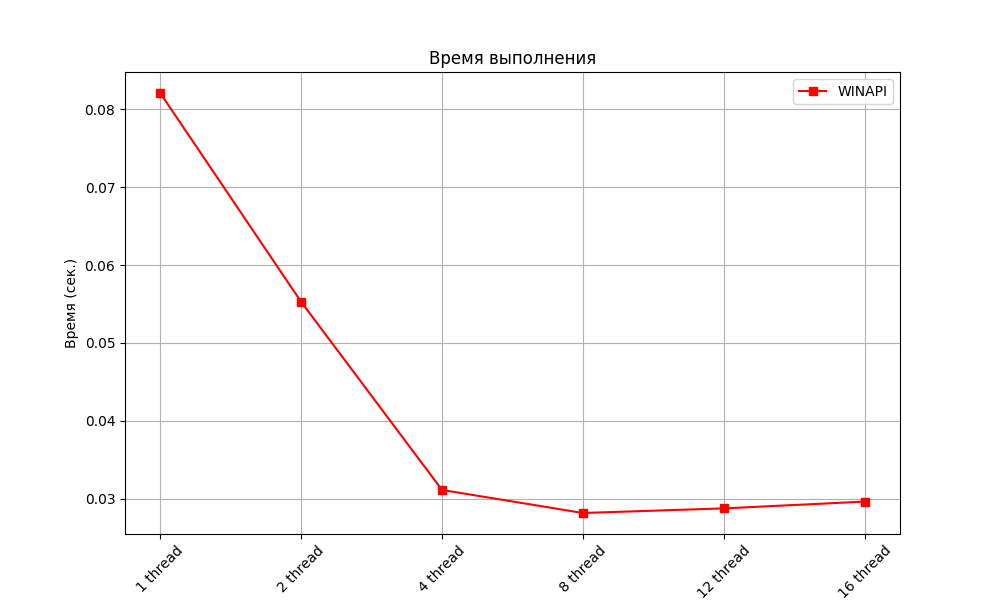
2. Произведите замеры времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12, 16). По результатам измерений постройте график и определите число потоков, при котором достигается наибольшая скорость выполнения. Запротоколируйте результаты в отчет.

3. Подготовьте итоговый отчет с развернутыми выводами по заданию.

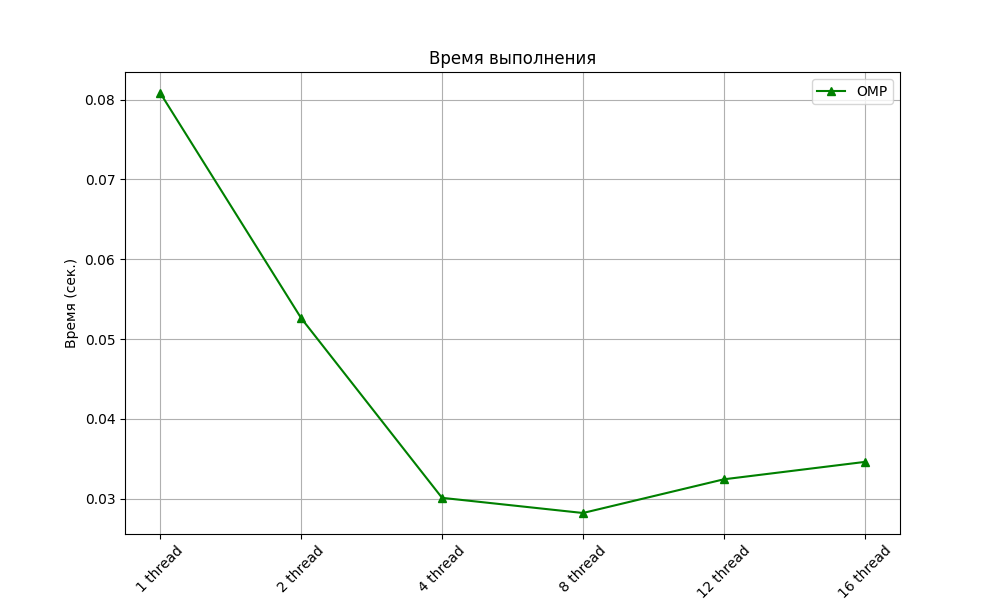
**Анализ данных по количеству потоков (WinAPI и Omp)**

**Данные:**

**WinAPI**

****

**Omp**

****

**Наблюдения:**

**1**. Сравнительный анализ скорости (время исполнения)

1 поток

* WinAPI: 0.0821 с
* OpenMP: 0.0808 с  
  Разница незначительная, OpenMP чуть быстрее.

2 потока

* WinAPI: 0.0553 с
* OpenMP: 0.0526 с  
  Обе технологии дают ~33% ускорения, но OpenMP снова чуть быстрее.

4 потока

* WinAPI: 0.0311 с
* OpenMP: 0.0301 с  
  Здесь почти в 2.6 раза быстрее по сравнению с 1 потоком. Идёт эффективная параллельная реализация.

8 потоков

* WinAPI: 0.0282 с
* OpenMP: 0.0282 с  
  Достигнуто почти предельное ускорение. Прибавка по сравнению с 4 потоками уже незначительная.

12 потоков

* WinAPI: 0.0288 с
* OpenMP: 0.0324 с  
  WinAPI сохраняет производительность, но OpenMP начинает терять эффективность. Возможно, из-за накладных расходов на управление потоками.

16 потоков

* WinAPI: 0.0296 с
* OpenMP: 0.0346 с  
  WinAPI всё ещё держится в рамках разумного, а OpenMP становится медленнее, чем при 8 потоках — явная деградация.

3. Различия технологий

WinAPI:

* Предоставляет низкоуровневое управление потоками, включая создание, приостановку и возобновление потоков вручную.
* Требует больше кода, но более эффективно в высоконагруженных задачах при грамотной реализации.
* Система управления потоками более «ручная», что позволяет оптимизировать нагрузку.

OpenMP:

* Предназначен для упрощения параллельных вычислений, работает на уровне компилятора.
* Отлично масштабируется в пределах количества физических ядер, но хуже справляется с логическими ядрами или сверхпараллелизмом.
* Автоматическая синхронизация и создание пулов потоков могут вносить накладные расходы.

Общие выводы

1. OpenMP показывает отличную производительность до 4–8 потоков, особенно если важна простота реализации.
2. WinAPI обеспечивает более стабильную и предсказуемую работу при большем количестве потоков (12–16), что делает его более подходящим для системных и высокопроизводительных задач.
3. В задачах типа «число π» после определённого порога (обычно 4–8 потоков) выигрыш от многопоточности минимален из-за:
   * Ограничений CPU (количество ядер),
   * Увеличения накладных расходов на синхронизацию,
   * Ограничений пропускной способности памяти.

Текст программы

#include <windows.h>

#include <iostream>

#include <vector>

#include <cmath>

#include <chrono>

#include <omp.h>

const int N = 10000000;

const int STUDENT\_ID = 331222;

const int BLOCK\_SIZE = 10 \* STUDENT\_ID;

CRITICAL\_SECTION cs;

volatile int current\_block = 0;

volatile bool done = false;

double pi = 0.0;

DWORD WINAPI CalculatePiBlock(LPVOID)

{

while (true)

{

int start;

EnterCriticalSection(&cs);

if (current\_block >= N)

{

LeaveCriticalSection(&cs);

break;

}

start = current\_block;

current\_block += BLOCK\_SIZE;

LeaveCriticalSection(&cs);

double local\_sum = 0.0;

int end = min(start + BLOCK\_SIZE, N);

for (int i = start; i < end; ++i) {

local\_sum += pow(-1, i) / (2.0 \* i + 1.0);

}

EnterCriticalSection(&cs);

pi += local\_sum;

LeaveCriticalSection(&cs);

}

return 0;

}

int WinAPI()

{

InitializeCriticalSection(&cs);

for (int threads\_count : {1, 2, 4, 8, 12, 16})

{

pi = 0.0;

current\_block = 0;

std::vector<HANDLE> threads(threads\_count);

auto start\_time = std::chrono::high\_resolution\_clock::now();

for (int i = 0; i < threads\_count; ++i) {

threads[i] = CreateThread(nullptr, 0, CalculatePiBlock, nullptr, 0, nullptr);

}

WaitForMultipleObjects(threads\_count, threads.data(), TRUE, INFINITE);

for (HANDLE t : threads) CloseHandle(t);

auto end\_time = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> duration = end\_time - start\_time;

std::cout.precision(15);

std::cout << "Threads: " << threads\_count << ", Pi ≈ " << (4.0 \* pi) << ", Time: " << duration.count() << "s\n";

}

DeleteCriticalSection(&cs);

return 0;

}

int OPMThreads()

{

for (int threads\_count : {1, 2, 4, 8, 12, 16}) {

double pi = 0.0;

auto start\_time = std::chrono::high\_resolution\_clock::now();

#pragma omp parallel for schedule(dynamic, BLOCK\_SIZE) num\_threads(threads\_count) reduction(+:pi)

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

pi += pow(-1, i) / (2.0 \* i + 1.0);

}

auto end\_time = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> duration = end\_time - start\_time;

std::cout.precision(15);

std::cout << "Threads: " << threads\_count << ", Pi ≈ " << (4.0 \* pi) << ", Time: " << duration.count() << "s\n";

}

return 0;

}

int main()

{

std::cout << "Using WinAPI:\n";

WinAPI();

std::cout << "\nUsing OpenMP:\n";

OPMThreads();

return 0;

}