**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»**

Студента гр. 3312 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Поляков А.И.

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

Санкт-Петербург

2025

**Цель работы.**

Исследовать механизмы создания и управленияпроцессами и потоками в ОС Windows.

**Задание 3.1.**

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).

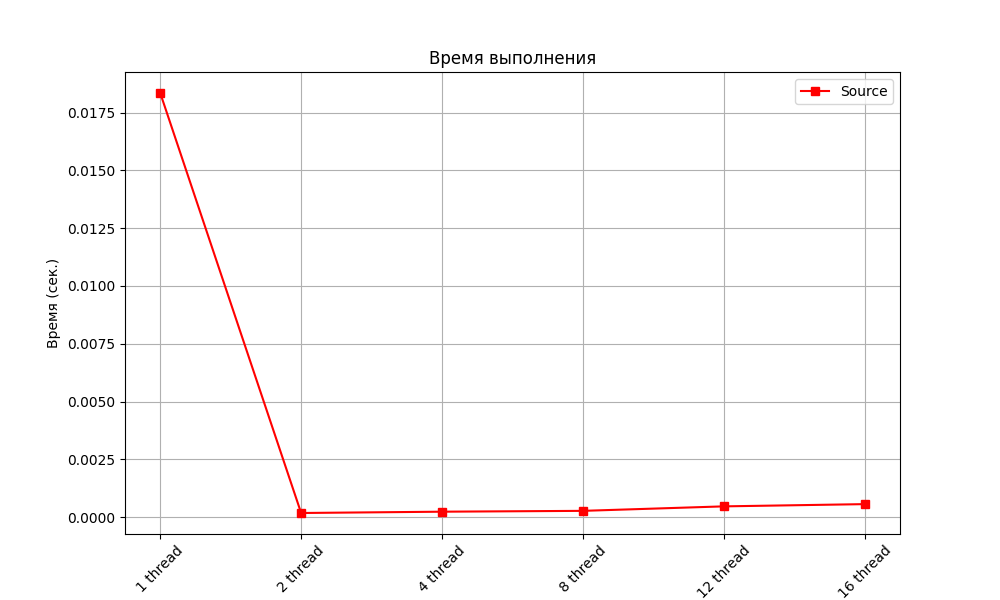
2. Произведите замеры времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12, 16). По результатам измерений постройте график и определите число потоков, при котором достигается наибольшая скорость выполнения. Запротоколируйте результаты в отчет.

3. Подготовьте итоговый отчет с развернутыми выводами по заданию.

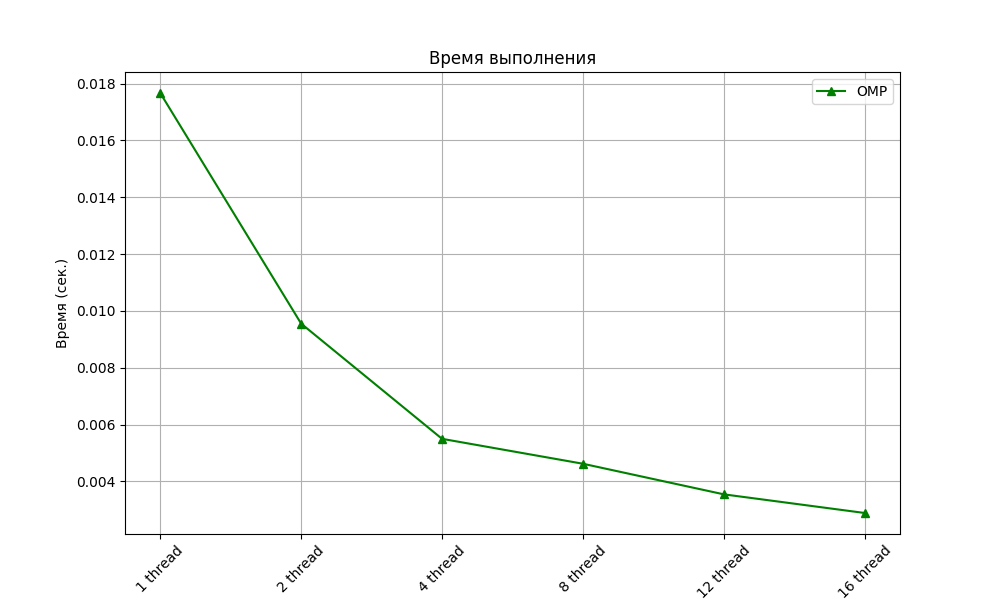
**Анализ данных по количеству потоков (big и small)**

**Данные:**

**WinAPI**

****

**OMP**

****

**Анализ производительности**

**WinAPI:**

1. 1 поток: Базовое время ~0.018 сек.
2. 2 потока: Время резко падает до 0.00017 сек — ускорение более чем в 100 раз.
3. 4–8 потоков: Небольшое увеличение времени (до ~0.00027 сек) — здесь проявляются накладные расходы на создание и синхронизацию потоков.
4. 12–16 потоков: Заметное ухудшение времени — это указывает, что:

* число потоков превышает количество физических/логических ядер, и возникает переключение контекста;
* слишком много потоков — больше затрат на синхронизацию, чем выгоды от распараллеливания.

Вывод: при использовании WinAPI наилучшая производительность достигается при 2–4 потоках. После этого эффективность падает. Это характерно для задач, в которых объём вычислений на поток относительно мал, и накладные расходы начинают доминировать.

**OpenMP:**

| **Потоки** | **Время (с)** | **Ускорение** | **Эффективность** |
| --- | --- | --- | --- |
| **1** | **0.01767** | **1.00×** | **100%** |
| **2** | **0.00955** | **1.85×** | **92.5%** |
| **4** | **0.00550** | **3.21×** | **80.3%** |
| **8** | **0.00462** | **3.82×** | **47.8%** |
| **12** | **0.00354** | **4.99×** | **41.6%** |
| **16** | **0.00289** | **6.11×** | **38.2%** |

**Анализ**

**1**. Ускорение почти линейное до 4 потоков

* До 4 потоков прирост очень хороший (80–92% эффективности).
* Это указывает на то, что Overhead минимален, и ресурсы используются эффективно.

2. Снижение эффективности при увеличении числа потоков

* После 8 потоков прирост замедляется.
* С 8 до 16 потоков ускорение всего 1.6×.
* Эффективность падает до ~38%.

Это нормальное поведение, объясняется:

* Накладными расходами на reduction(+:pi)
* Ограниченной пропускной способностью кэша и памяти
* Законом убывающей отдачи (закон Амдала)
* Влиянием гиперпоточности (если у CPU меньше физических ядер)

**Выводы**

1. WinAPI даёт впечатляющее ускорение при использовании 2 потоков. Это демонстрирует эффективность ручной параллелизации, особенно для коротких, но интенсивных задач.
2. OpenMP:

* **OpenMP успешно распараллеливает** вычисления при больших объёмах данных (N = 100 млн).
* Ускорение до 6× при 16 потоках подтверждает эффективность подхода.
* Максимальная эффективность наблюдается при использовании 2–4 потоков, далее она постепенно падает из-за накладных расходов и аппаратных ограничений.

1. Оптимальное число потоков для WinAPI — 2–4, после чего накладные расходы перевешивают выгоду.

Текст программы

#include "PIWinAPI.h"

#include "Omp.h"

int main()

{

int threadsCount[] = { 1,2,4,8,12,16 };

for (auto i : threadsCount)

{

std::cout << "Threads: " << i << std::endl;

WinAPIThreads(i);

}

std::cout << std::endl << std::endl;

for (auto i : threadsCount)

{

std::cout << "Threads: " << i << std::endl;

OMP(i);

}

}

*PIWinAPI.срр*

#include "PIWinAPI.h"

constexpr int N = 10000000;

constexpr int STUDENT\_ID = 331223;

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

double pi = 0.0;

LONG current\_index = 0;

int total\_blocks = (N + BLOCK\_SIZE - 1) / BLOCK\_SIZE;

HANDLE hMutex;

int thread\_count;

DWORD WINAPI PiWorker(LPVOID lpParam)

{

int thread\_id = \*(int\*)lpParam;

while (true)

{

int block\_index = InterlockedIncrement(&current\_index) - 1;

if (block\_index >= total\_blocks) break;

int start = block\_index \* BLOCK\_SIZE;

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

double local\_sum = 0.0;

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

local\_sum += ((i % 2 == 0 ? 1.0 : -1.0) / (2 \* i + 1));

WaitForSingleObject(hMutex, INFINITE);

pi += local\_sum;

ReleaseMutex(hMutex);

}

return 0;

}

int WinAPIThreads(int thread\_count)

{

std::vector<HANDLE> threads;

std::vector<int> thread\_ids;

hMutex = CreateMutex(NULL, FALSE, NULL);

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

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

{

thread\_ids.push\_back(i);

HANDLE hThread = CreateThread(NULL, 0, PiWorker, &thread\_ids[i], 0, NULL);

threads.push\_back(hThread);

}

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

for (auto h : threads)

CloseHandle(h);

CloseHandle(hMutex);

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

std::cout.precision(15);

std::cout << "Computed PI = " << 4 \* pi << std::endl;

std::cout << "Time: " << std::chrono::duration<double>(end\_time - start\_time).count() << " seconds" << std::endl;

return 0;

}

*ОMP.срр*

#include "Omp.h"

#include <omp.h>

#include <iostream>

constexpr int N = 10000000;

constexpr int STUDENT\_ID = 331223;

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

int OMP(int thread\_count)

{

double pi = 0.0;

double start = omp\_get\_wtime();

#ifdef \_OPENMP

#pragma omp parallel for num\_threads(thread\_count) schedule(static) reduction(+:pi)

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

pi += (i % 2 == 0 ? 1.0 : -1.0) / (2 \* i + 1);

#else

std::cerr << "OpenMP is not enabled. Please enable OpenMP in your compiler settings.\n";

return -1;

#endif

double end = omp\_get\_wtime();

std::cout.precision(15);

std::cout << "Computed PI = " << 4 \* pi << "\n";

std::cout << "Time: " << (end - start) << " seconds\n";

#pragma omp parallel

{

#pragma omp single

std::cout << "Used threads: " << omp\_get\_num\_threads() << std::endl;

}

return 0;

}