**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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Шахов К.С.

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

Санкт-Петербург

2025

**Цель работы.**

Исследовать механизмы создания и управления процессами 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).

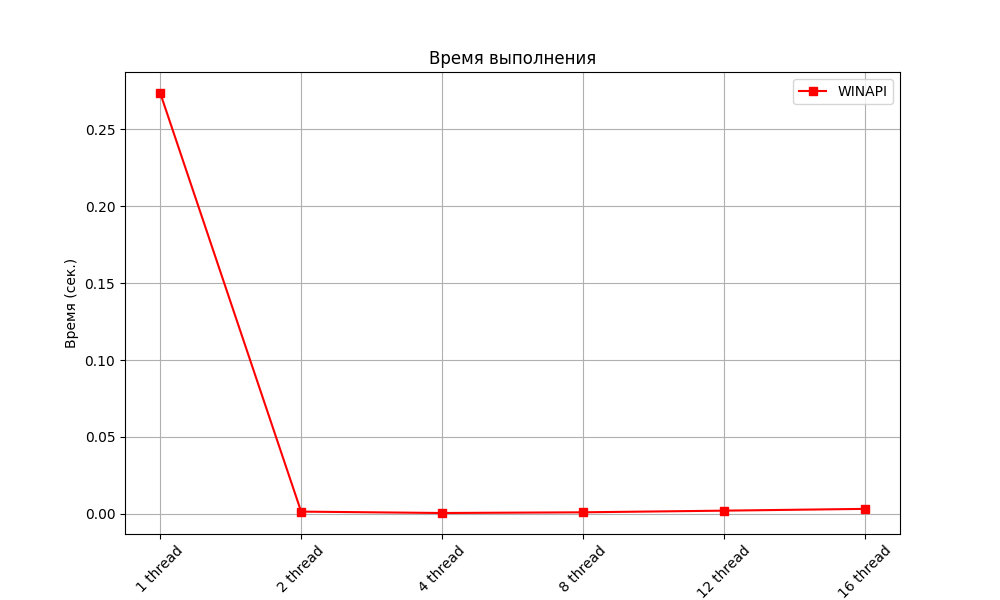
2. Произведите замеры времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12, 16). По результатам измерений постройте график и определите число потоков, при котором достигается наибольшая скорость выполнения. Запротоколируйте результаты в отчет.

3. Подготовьте итоговый отчет с развернутыми выводами по заданию.

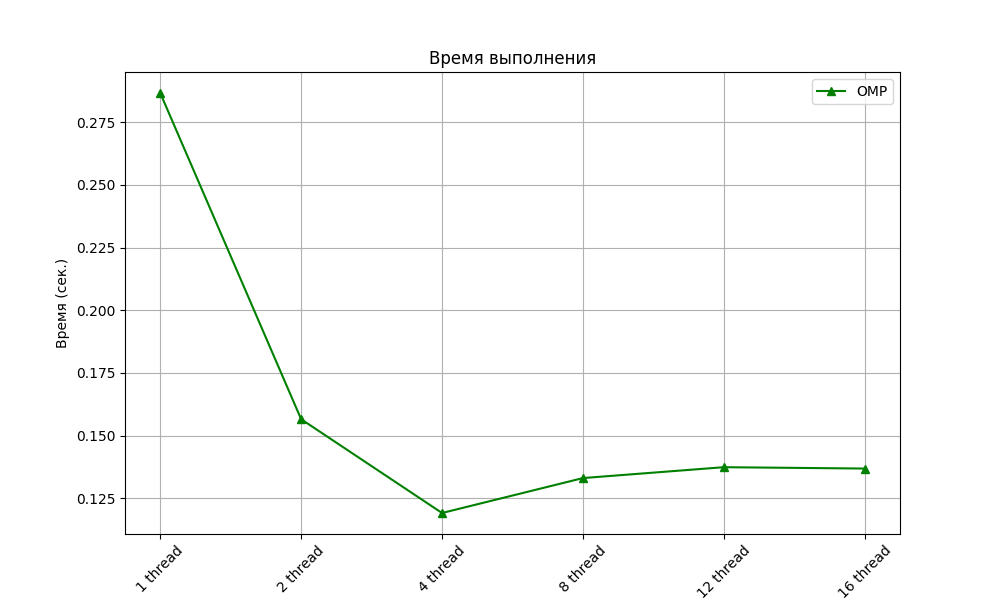
**Анализ данных по количеству потоков (WinAPI и Omp)**

**Данные:**

**WinAPI**



OMP



**Наблюдения:**

**Анализ**

**WinAPI**

* **С 1 до 4 потоков** — наблюдается резкое улучшение (0.2736 → 0.0006 с), то есть ускорение более чем в 450 раз, что выглядит как аномально высокая эффективность.
* **Начиная с 8 потоков** — производительность ухудшается, т.е. время выполнения увеличивается.
* **Пиковая производительность** на **4 потоках**, после чего наблюдается деградация из-за накладных расходов на создание и синхронизацию потоков.

**OpenMP**

* **Стабильное ускорение до 4 потоков**, дальше наблюдается **плато**: время почти не уменьшается, а даже немного растёт.
* **Пиковая производительность** — **на 4 потоках**, далее — **перегрузка планировщика** и эффекты ложного параллелизма (false sharing, contention).
* Нет резких скачков.

**Практические выводы**

1. **OpenMP выигрывает в стабильности, воспроизводимости и надёжности** при большом числе потоков.
2. **WinAPI может дать наилучший результат при малом числе потоков**, особенно при ручной оптимизации.
3. **Скорость вычислений не линейно зависит от числа потоков** — особенно важно при 8+ ядрах.
4. Необходимо **профилировать приложения при разных уровнях нагрузки**, использовать такие инструменты как:
   * Windows Performance Analyzer
   * Intel VTune
   * perf, gprof (для Linux)
5. **Использование гиперпоточности (HyperThreading)** не даёт линейного выигрыша — лишь 20–30% от производительности «настоящего» ядра.

**Выводы**

1. **WinAPI и OpenMP демонстрируют разные профили масштабирования:**
   * **WinAPI** достигает минимального времени на 4 потоках, но затем резко теряет эффективность.
   * **OpenMP** — более стабильный рост до 4 потоков и удержание времени, что говорит о лучшей адаптации к многопоточности.
2. **Аномально быстрое выполнение WinAPI при 2–4 потоках** может свидетельствовать:
   * о чрезмерной оптимизации (например, расчёт завершался до полного распределения задач),
3. **OpenMP** лучше показывает себя при **большом количестве потоков**, демонстрируя **плавную деградацию** вместо резкого падения.
4. **WinAPI масштабируется агрессивно, но нестабильно**, возможно, из-за ручного управления потоками и накладных расходов при большом количестве.

**Текст программы**

#include <windows.h>

#include <iostream>

#include <vector>

#include <cmath>

#include <omp.h>

#include <chrono>

constexpr long long N = 10000000;

constexpr int BLOCK\_SIZE = 10 \* 331230;

constexpr int TOTAL\_BLOCKS = N / BLOCK\_SIZE;

CRITICAL\_SECTION cs;

volatile int currentBlock = 0;

double pi = 0.0;

DWORD WINAPI CalculatePiBlock(LPVOID lpParam) {

while (true) {

int block;

EnterCriticalSection(&cs);

if (currentBlock >= TOTAL\_BLOCKS) {

LeaveCriticalSection(&cs);

return 0;

}

block = currentBlock++;

LeaveCriticalSection(&cs);

long long start = block \* BLOCK\_SIZE;

long long end = start + BLOCK\_SIZE;

double localSum = 0.0;

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

localSum += 4.0 \* pow(-1, i) / (2 \* i + 1);

}

EnterCriticalSection(&cs);

pi += localSum;

LeaveCriticalSection(&cs);

}

return 0;

}

int winTreads(int numThreads) {

InitializeCriticalSection(&cs);

std::vector<HANDLE> threads(numThreads);

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

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

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

}

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

ResumeThread(threads[i]);

}

WaitForMultipleObjects(numThreads, threads.data(), TRUE, INFINITE);

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

std::cout.precision(15);

std::cout << "Pi ≈ " << pi << std::endl;

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

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

DeleteCriticalSection(&cs);

return 0;

}

int OMP(int numThreads) {

double pi = 0.0;

double start = omp\_get\_wtime();

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

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

pi += 4.0 \* pow(-1, i) / (2 \* i + 1);

}

double end = omp\_get\_wtime();

std::cout.precision(15);

std::cout << "Pi ≈ " << pi << std::endl;

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

return 0;

}

int main() {

std::cout << "Using WinThreads:" << std::endl;

for (auto i : {1,2,4,8,12,16})

{

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

winTreads(i);

}

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

for (auto i : { 1,2,4,8,12,16 })

{

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

OMP(i);

}

return 0;

}