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»

по теме «Процессы и потоки»

|  |  |  |
| --- | --- | --- |
| Студентка гр. 2307 |  | Левушкина С.И. |
| Преподаватель |  | Тимофеев А.В. |

Санкт-Петербург, 2024

# **СОДЕРЖАНИЕ**

[СОДЕРЖАНИЕ 2](#_Toc167048740)

[ЦЕЛЬ РАБОТЫ 3](#_Toc167048741)

[ЗАДАНИЯ 3](#_Toc167048742)

[ЗАДАНИЕ 1 (WIN API) 5](#_Toc167048743)

[ЗАДАНИЕ 2 (OPENMP) 6](#_Toc167048744)

[ТАБЛИЦА ЗАМЕРОВ ВРЕМЕНИ 7](#_Toc167048745)

[ВЫВОДЫ 8](#_Toc167048746)

[ТЕКСТ ПРОГРАММЫ 9](#_Toc167048747)

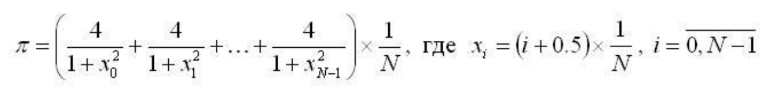
# **ЦЕЛЬ РАБОТЫ**

Исследовать механизмы создания и управления

процессами и потоками в ОС Windows.

# **ЗАДАНИЯ**

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



где N=100000000.

Использовать распределение итераций блоками размером 2307150 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 выполнять 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

использовать функцию Win32 API ResumeThread.

По окончании обработки текущего блока итераций поток не

должен завершаться, а должен быть, например, приостановлен с

помощью функции Win32 API SuspendThread. Затем потоку должен

быть предоставлен следующий свободный блок итераций, и поток

должен быть освобожден, например, с помощью функции Win32 API

ResumeThread.

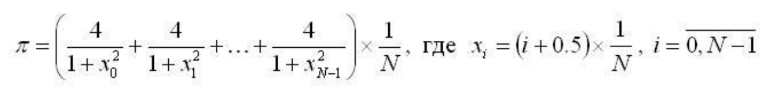
Произвести замеры времени выполнения приложения для

разного числа потоков (1, 2, 4, 8, 12, 16). По результатам измерений

построить график и определить число потоков, при котором достигается

наибольшая скорость выполнения.

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



где N=100000000.

Распределить работу по потокам с помощью OpenMP-директивы for.

Использовать динамическое планирование блоками итераций (размер блока = 2307150).

Произвести замеры времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12, 16). По результатам измерений построить график и определить число потоков, при котором достигается наибольшая скорость выполнения.

# **ЗАДАНИЕ 1 (WIN API)**

За создание и освобождение потоков отвечает функция standart, принимающая на вход количество потоков. При создании потоков в CreateThreads передается указатель на функцию calculation, которая отвечает за вычисление “куска” от числа пи и с помощью мьютекса организует доступ к глобальной переменной pi и current\_block. В цикле в основной функции main прогоняются 5 вариантов количества потоков и замеряется время для каждого.

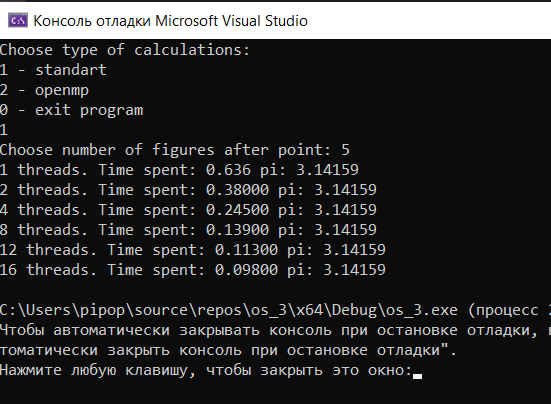


Рисунок 1. Win Api

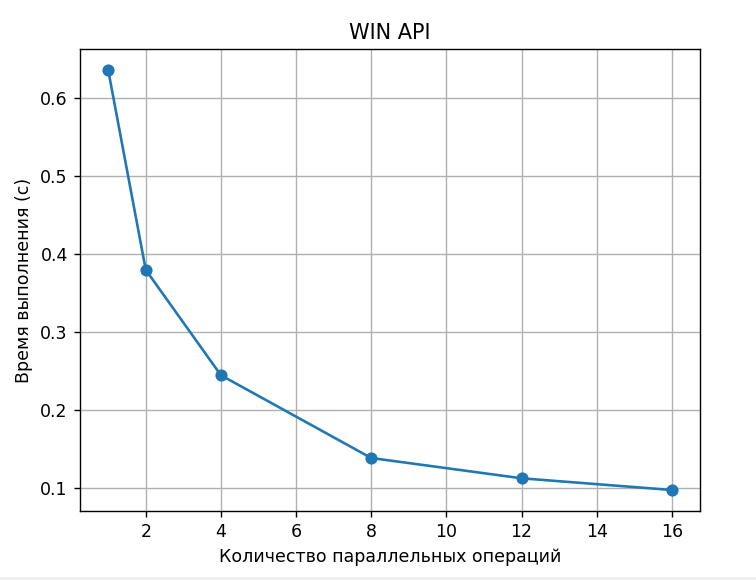


Рисунок 2. График Win Api

# **ЗАДАНИЕ 2 (OPENMP)**

За вычисление числа пи вторым способом отвечает функция openmp. Внутри функции используется директива #pragma omp parallel for для параллельного выполнения цикла. Она создает набор из num\_threads, и каждый из них выполняет часть цикла.

Директива schedule(dynamic, block\_size) указывает, что распределение итераций цикла между потоками будет динамическим, и каждый поток будет получать блок итераций размером block\_size. Это позволяет оптимизировать распределение нагрузки между потоками и снизить время выполнения.

Директива private(temp, local\_pi) указывает, что каждый поток получит свои копии переменных local\_pi, temp, которые будут использоваться для хранения частичного результата вычислений.

В теле цикла используется директива #pragma omp critical для обозначения критической секции, она защищает доступ к общей переменной pi, чтобы избежать состояния гонки.

Также в функции main этот способ прогоняется 5 раз на разном количестве потоков и производится замер времени для каждого варианта.

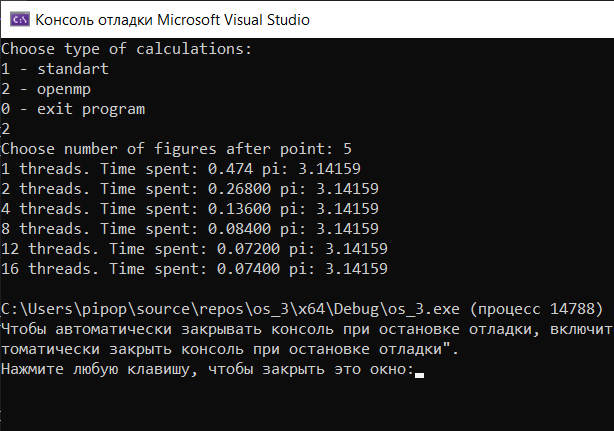


Рисунок 3. OpenMP

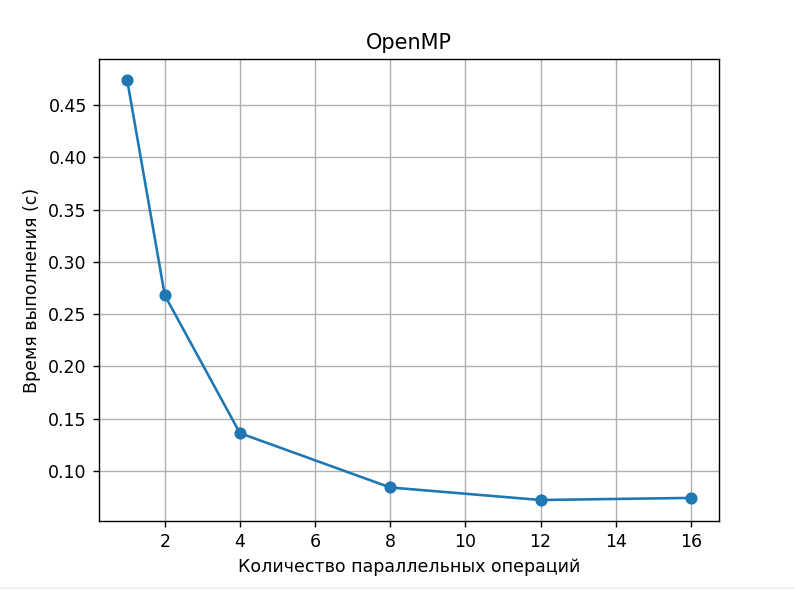


Рисунок 4. График openmp

# **ТАБЛИЦА ЗАМЕРОВ ВРЕМЕНИ**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Кол-во потоков | 1 | 2 | 4 | 8 | 12 | 16 |
| Win32 API | 0.636 | 0.38 | 0.245 | 0.139 | 0.113 | 0.098 |
| OpenMP | 0.474 | 0.286 | 0.136 | 0.084 | 0.072 | 0.074 |

# **ВЫВОДЫ**

В варианте с использованием OpenMP по графику видно, что до 8 потоков время уменьшается примерно в 2 раза с увеличением количества потоков в 2 раза (процессор компьютера, на котором выполнялся код, 8-ядерный, поэтому до 8 потоков включительно могут честно выполняться параллельно). На 16 потоках слегка увеличилось – так как это больше физического предела по потокам, то потоки будут переключаться между ядрами процессора в соответствии с расписанием операционной системы, это минус к скорости.

С использованием технологии Win32 API наилучшее время было при 16, при этом значительное увеличение производительности также приостановилось на 8 потоках.

На графиках также видно, что использование технологии OpenMP немного выигрывает во времени у Win32 API. Скорее всего, это связано с тем, что распределение крупной вычислительной задачи (в данном случае – вычисление числа пи) согласно данным Microsoft хорошо подходит к выполнению на стандарте OpenMP.

# **ТЕКСТ ПРОГРАММЫ**

#include <windows.h>

#include <vector>

#include <cmath>

#include <iostream>

#include <iomanip>

#include <mutex>

#include <ctime>

#include <omp.h>

using namespace std;

const int n = 100000000;

const int block\_size = 2307150;

const int num\_blocks = n / block\_size;

long double pi = 0.0;

clock\_t start\_time, end\_time;

mutex mtx;

int current\_block = 0;

DWORD WINAPI calculation(LPVOID lpParam) {

while (true) {

int start, end;

mtx.lock();

start = current\_block \* block\_size;

end = (current\_block == num\_blocks) ? n : (current\_block + 1) \* block\_size;

current\_block++;

mtx.unlock();

if (start >= n) {

break;

}

long double local\_pi = 0.0;

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

local\_pi += 4.0 / (1.0 + ((k + 0.5) / n) \* ((k + 0.5) / n));

}

mtx.lock();

pi += local\_pi / n;

mtx.unlock();

}

return 0;

}

void standart(int num\_threads) {

vector<HANDLE> threads(num\_threads);

current\_block = 0;

pi = 0.0;

for (int i = 0; i < num\_threads; i++) {

threads[i] = CreateThread(NULL, 0, calculation, NULL, CREATE\_SUSPENDED, NULL);

}

for (int i = 0; i < num\_threads; i++) {

ResumeThread(threads[i]);

}

for (int i = 0; i < num\_threads; i++) {

WaitForSingleObject(threads[i], INFINITE);

CloseHandle(threads[i]);

}

}

void openmp(int num\_threads) {

double temp = 0.0;

double local\_pi = 0.0;

#pragma omp parallel num\_threads(num\_threads) private(temp, local\_pi)

{

#pragma omp for

for (long long int i = 0; i < n; i++) {

temp = (i + 0.5) \* (1.0 / n);

local\_pi += 4.0 / (1.0 + temp \* temp);

}

#pragma omp critical

{

pi += local\_pi;

}

}

}

int main() {

int flag = -1, size = 5;

double elapsed\_time;

int num\_threads[6] = { 1, 2, 4, 8, 12, 16 };

cout << "Choose type of calculations:\n1 - standart\n2 - openmp\n0 - exit program\n";

cin >> flag;

cout << "Choose number of figures after point: ";

cin >> size;

if (flag == 1) {

for (int i = 0; i < 6; i++) {

start\_time = clock();

standart(num\_threads[i]);

end\_time = clock();

elapsed\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

cout << num\_threads[i] << " threads. ";

cout << "Time spent: " << elapsed\_time;

cout << " pi: " << fixed << setprecision(size) << pi << endl;

}

}

else if (flag == 2) {

for (int i = 0; i < 6; i++) {

pi = 0.0;

start\_time = clock();

openmp(num\_threads[i]);

end\_time = clock();

elapsed\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

cout << num\_threads[i] << " threads. ";

cout << "Time spent: " << elapsed\_time;

cout << " pi: " << fixed << setprecision(size) << pi/n << endl;

}

}

else {

exit(0);

}

return 0;

}