**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»**

| Студент |  |  |
| --- | --- | --- |
| Преподаватель |  | Тимофеев А. В. |

Санкт-Петербург

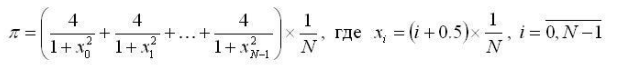
2023

**Цель работы**: исследовать механизмы создания и управления процессами и потоками в ОС Windows.

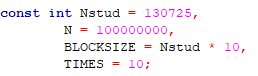
**Задание 3.1.** Реализация многопоточного приложения с использованием функций Win32 API.

**Ход работы**

**Пункт 1** - создание программы, которая вычисляет число pi с точностью N знаков после запятой по следующей формуле



, где N=100000000.

В начале программы установили константные переменные (Рис. 1):

,,

Рис.1

, где Nstud – номер студенческого билета, N – кол-во знаков в числе ПИ после запятой, BLOCKSIZE – размер блока для итераций по потокам и TIMES – кол-во повторений запуска функции для поиска среднего времени выполнения вычислений.

Далее используя распределение итераций по потокам, где 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. Потоки создавались функцией **Win32 API CreateThread**, при этом сначала заданное кол-во потоков создаются приостановленными, а в дальнейшем освобождаются из приостановленного состояния функцией **Win32 API ResumeThread**. По окончании обработки текущего блока итераций поток не завершается, а приостанавливается с помощью функции **Win32 API SuspendThread**. Затем потоку предоставляется следующий свободный блок итераций, и поток освобождается функцией **Win32 API ResumeThread** (Рис. 2).

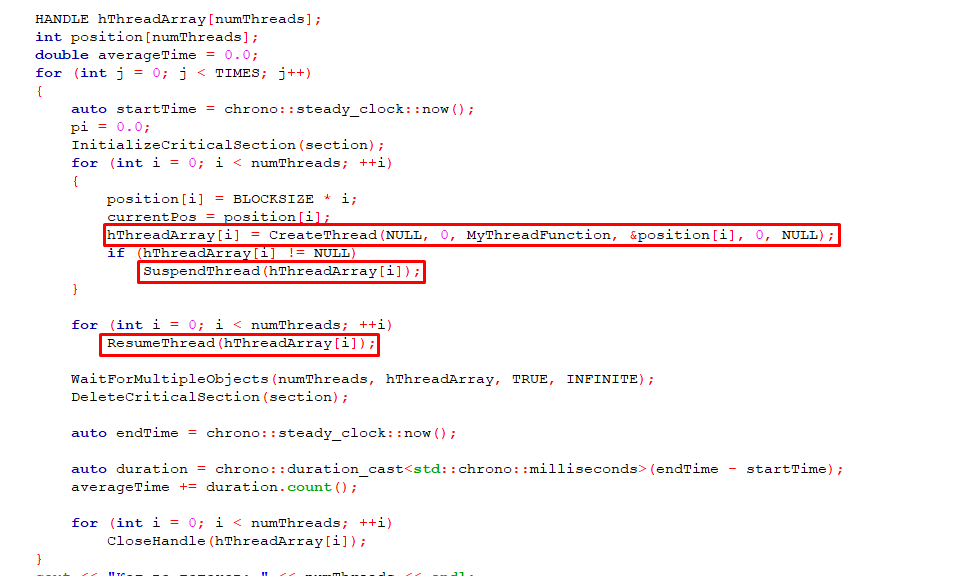


Рис. 2

Основные функции данного кода:

**HANDLE hThreadArray[numThreads];** – объявление массива дескрипторов потоков. Этот массив будет хранить дескрипторы для каждого созданного потока.

**int position[numThreads];** – объявление массива позиций для каждого потока. Этот массив будет хранить начальные позиции для каждого потока в многопоточном расчете числа Пи.

**InitializeCriticalSection(section);** - инициализация критической секции section. Критическая секция используется для синхронизации доступа к общим данным (переменным pi и currentPos) между потоками.

**for (int j = 0; j < TIMES; j++)** – внешний цикл, который повторяется TIMES раз. Этот цикл используется для усреднения времени выполнения расчета числа Пи.

**for (int i = 0; i < numThreads; ++i)** - цикл, в котором создаются потоки. Выполняется numThreads раз, где numThreads - это аргумент функции, указывающий количество потоков, которые нужно создать.

**position[i] = BLOCKSIZE \* i;** – вычисление начальной позиции для текущего потока. Начальная позиция зависит от индекса потока и размера блока BLOCKSIZE.

**currentPos = position[i];** – установка текущей позиции для текущего потока. Переменная currentPos будет использоваться в функции потока для определения начальной позиции каждого потока.

**hThreadArray[i] = CreateThread(NULL, 0, MyThreadFunction, &position[i], 0, NULL);** – создание нового потока с использованием функции **CreateThread**. В аргументах указываются параметры для создаваемого потока, включая указатель на функцию потока **MyThreadFunction**, а также передача адреса начальной позиции через указатель на **position[i]**.

**if (hThreadArray[i] != NULL) SuspendThread(hThreadArray[i]);** – приостановка созданного потока с помощью SuspendThread.

**for (int i = 0; i < numThreads; ++i) ResumeThread(hThreadArray[i]);** – возобновление выполнения всех созданных потоков с помощью ResumeThread.

**WaitForMultipleObjects(numThreads, hThreadArray, TRUE, INFINITE);** – ожидание завершения всех созданных потоков. Функция **WaitForMultipleObjects** блокирует вызывающий поток до тех пор, пока не завершатся все указанные потоки.

**DeleteCriticalSection(section);** – удаление критической секции section после окончания расчета числа Пи.

При создании создании потоков с помощью функции **CreateThread** используется функция **MyThreadFunction** (типа DWORD WINAPI) **–** функция потока, предназначенная для вычисления числа ПИ (Рис. 3).

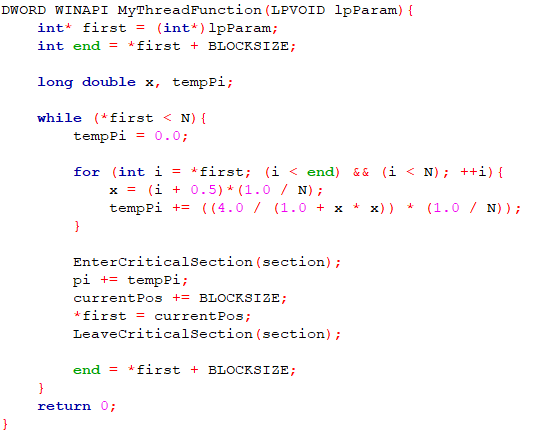


Рис. 3

**int\* first = (int\*)lpParam;** – преобразование указателя lpParam в указатель на целое число first. Параметр lpParam содержит начальную позицию для расчета числа Пи в текущем потоке.

**int end = \*first + BLOCKSIZE;** – определение конечной позиции для расчета числа Пи в текущем потоке. Значение BLOCKSIZE указывает, сколько итераций должен выполнить каждый поток.

**while (\*first < N)** – цикл, который будет выполняться до тех пор, пока текущая позиция first меньше общего количества итераций N.

**for (int i = \*first; (i < end) && (i < N); ++i)** – внутренний цикл, который будет выполняться для каждой итерации от текущей позиции first до конечной позиции end или до достижения общего количества итераций N. Этот цикл используется для вычисления суммы для расчета числа Пи.

**EnterCriticalSection(section);** – вход в критическую секцию section. Критическая секция используется для синхронизации доступа к общим данным (переменным pi и currentPos) между потоками.

**pi += tempPi;** – аккумуляция локальной суммы tempPi в общую переменную pi.

**currentPos += BLOCKSIZE;** – инкрементирование текущей позиции на значение BLOCKSIZE (размер блока обработки).

**\*first = currentPos;** – обновление начальной позиции текущего потока на новое значение текущей позиции.

**LeaveCriticalSection(section);** – выход из критической секции section.

**end = \*first + BLOCKSIZE;** – обновление конечной позиции текущего потока на новое значение начальной позиции плюс размер блока обработки.

**Пункт 2** – замер времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12, 16). Построение графика и определение числа потоков, при котором достигается наибольшая скорость выполнения по результатам измерений и сравнение результатов с **п. 3.1**.

Для измерения времени в миллисекундах использовалась библиотека **#include <chrono>**.

Перед началом основного цикла **for**, в котором создаются потоки, оставляем временную метку **startTime –** обозначающую начало замера времени работы, а после цикла **endTime –** обозначающую конец. Далее находим разницу между временными метками (**auto duration = chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime);**), складываем времена выполнения от каждого запуска функции – **averageTime += duration.count();** и выводим кол-во потоков на котором производились вычисления, среднее время вычисления (**averageTime / TIMES**) и сам объект вычисления – число ПИ.

Далее для удобства работы с графиками записываем кол-во потоков и времена в файл формата .csv (Рис. 4-5) и строим графики в любом удобном редакторе таблиц, например – Excel (Рис. 6).

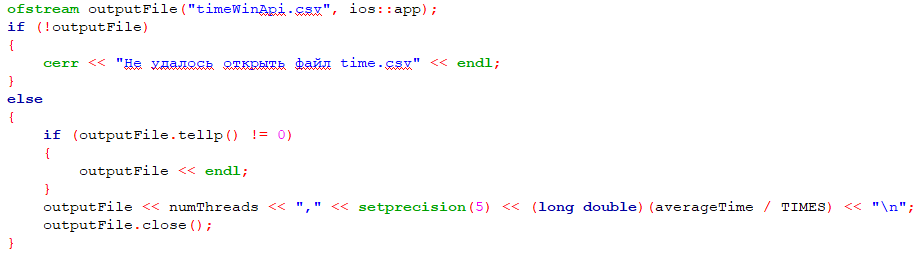


Рис. 4

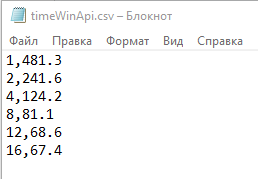
****

Рис. 5

****

Рис. 6

Таблица №1

| Кол-во потоков | 1 | 2 | 4 | 8 | 12 | 16 |
| --- | --- | --- | --- | --- | --- | --- |
| Время, мс | 481.3 | 241.6 | 124.2 | 81.1 | 68.6 | 67.4 |
| Уменьшение времени | - | 1,99 | 1,94 | 1,53 | 1,18 | 1,01 |

**Вывод:** из представленного графика (Рис. 6) можно сделать вывод, что с увеличением количества потоков время выполнения расчета числа ПИ сокращается. Однако эффективность увеличения количества потоков начинает снижаться после достижения значения в 12. Сначала наблюдается уменьшение времени выполнения от 1 до 12 потоков, однако после 12 разница в скорости выполнения не большая, поэтому и самое оптимальное время достигается при использовании не менее 12 потоков. Реализация с использованием WinAPI показывает уменьшение времени выполнения с увеличением количества потоков до определенного значения (в данном случае - 12), а затем стабилизируется. Увеличение количества потоков с 1 до 12 приводит к более чем шестикратному сокращению времени выполнения (Таблица №1). Однако при дальнейшем увеличении количества потоков более 12 не наблюдается существенного улучшения производительности. Это объясняется наличием накладных расходов на координацию и синхронизацию потоков, которые начинают превышать выигрыш от параллельной обработки. Суммируя все выше описанное, использование нескольких потоков позволяет существенно повысить скорость выполнения программы, но только до определенного числа потоков.

Пример работы программы (Рис. 7):

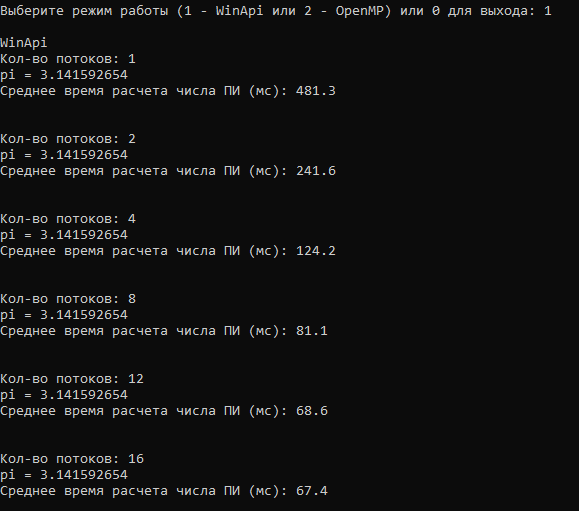
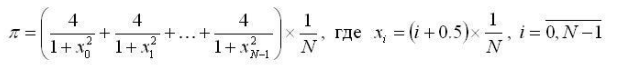
****

Рис. 7

**Задание 3.2.** Реализация многопоточного приложения с использованием технологии OpenMP.

**Ход работы**

**Пункт 1** – создание программы, которая вычисляет число pi с точностью N знаков после запятой по следующей формуле



,

где N=100000000. Для распределения работы программы по потокам была использована OpenMP-директива **for** и динамическое планирование блоками итераций, где **Размер блока = 10 \* Nстудбилета. (Nстудбилета = 130725)** (Рис. 8)**.**

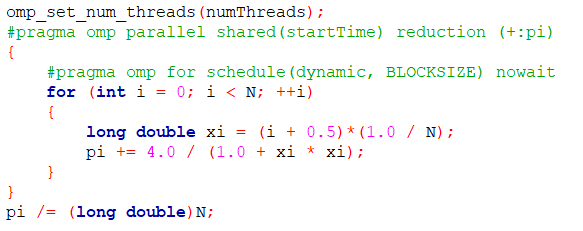


Рис. 8

В данном отрезке кода представлено:

**omp\_set\_num\_threads(numThreads);** – установка кол-ва потоков

**#pragma omp parallel shared(startTime) reduction (+:pi)** – создание параллельной области, в которой каждый поток будет выполнять итерации цикла for. Директива **shared(startTime)** указывает, что переменная startTime общая для всех потоков. Директива **reduction(+:pi)** гарантирует, что переменная pi будет суммироваться в конце выполнения цикла, чтобы получить итоговое значение числа ПИ. Директива **#pragma omp for** используется внутри параллельной области (**#pragma omp parallel**) для распределения итераций цикла **for** между потоками. Она позволяет выполнить итерации цикла в параллельном режиме. Аргументы **schedule(dynamic, BLOCKSIZE)** определяют способ распределения итераций. В конкретном случае, dynamic означает, что итерации цикла будут динамически распределены между потоками во время выполнения. **BLOCKSIZE** указывает размер блока итераций, которые будут выделены каждому потоку. Если размер блока не указан, система по умолчанию выберет оптимальный размер. Ключевое слово **nowait** указывает, что после завершения выполнения итераций цикла каждым потоком, потоки могут продолжать свое выполнение без явной синхронизации. Таким образом, эта директива распределяет итерации цикла for между потоками, используя динамическое распределение с размером блока, равным **BLOCKSIZE**, и позволяет потокам продолжить выполнение без ожидания других потоков при завершении итераций. Данный код реализует функцию нахождения числа ПИ по формуле представленной в задании.

**Пункт 2** – замер времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12, 16). Построение графика и определение числа потоков, при котором достигается наибольшая скорость выполнения по результатам измерений и сравнение результатов с **п. 3.1** (Рис. 9).

Для измерения времени в миллисекундах использовалась библиотека **#include <chrono>**.

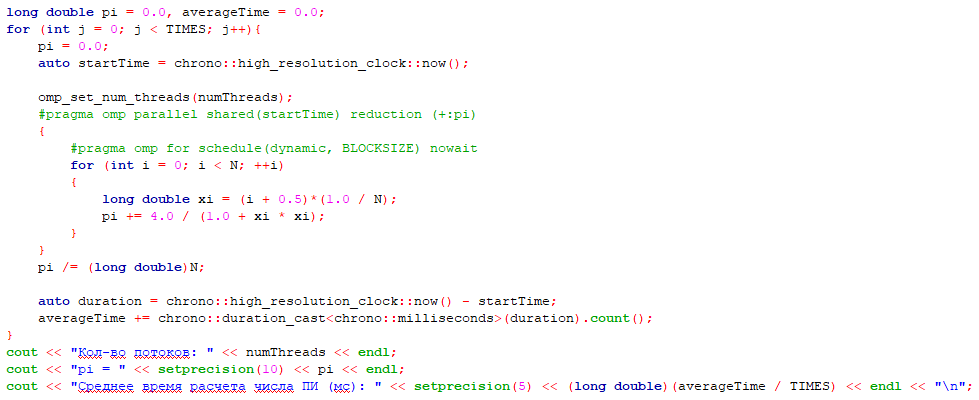


Рис. 9

Перед началом основного цикла **for** оставляем временную метку **startTime –** обозначающую начало замера времени работы. После работы цикла находим разницу времени между стартовой меткой и текущим временем. (**auto duration = chrono::high\_resolution\_clock::now() - startTime;**) и выводим кол-во потоков на котором производились вычисления, среднее время вычисления (**averageTime / TIMES**) и сам объект вычисления – число ПИ.

Далее для удобства работы с графиками записываем кол-во потоков и времена в файл формата .csv (Рис. 10-11) и строим графики в любом удобном редакторе таблиц, например – Excel (Рис. 12).

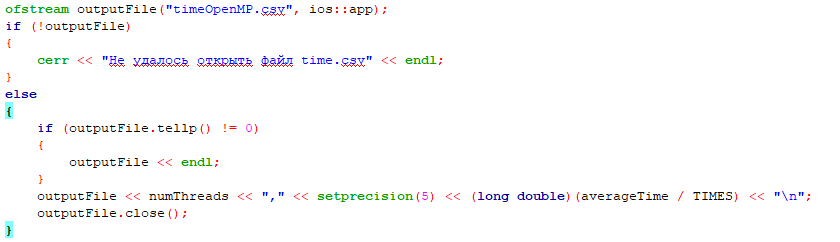


Рис. 10

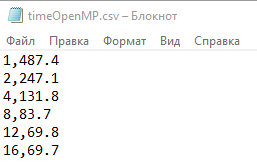


Рис. 11

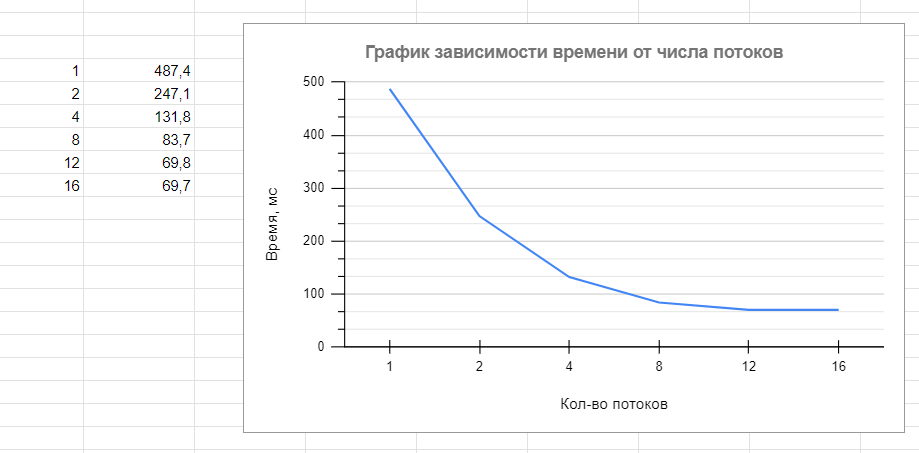


Рис. 12

Таблица №2

| Кол-во потоков | 1 | 2 | 4 | 8 | 12 | 16 |
| --- | --- | --- | --- | --- | --- | --- |
| Время, мс | 487,4 | 247,1 | 131,8 | 83,7 | 69,8 | 69,7 |
| Уменьшение времени | - | 1,97 | 1,88 | 1,58 | 1,2 | 1 |

**Вывод:** Из представленного графика (Рис. 12) можно сделать вывод, что с увеличением количества потоков время выполнения расчета числа ПИ сокращается. Однако эффективность увеличения количества потоков начинает снижаться после достижения значения в 12. Сначала наблюдается уменьшение времени выполнения от 1 до 12 потоков, однако после 12 разница в скорости выполнения не большая, поэтому и самое оптимальное время достигается при использовании не менее 12 потоков. Реализация с использованием WinAPI показывает уменьшение времени выполнения с увеличением количества потоков до определенного значения (в данном случае - 12), а затем стабилизируется. Увеличение количества потоков с 1 до 12 приводит почти к семикратному (Таблица №2) сокращению времени выполнения. Однако при дальнейшем увеличении количества потоков более 12 не наблюдается существенного улучшения производительности. Это объясняется наличием накладных расходов на координацию и синхронизацию потоков, которые начинают превышать выигрыш от параллельной обработки.

Суммируя все выше описанное, использование OpenMp позволяет использовать преимущество многопоточности, минуя сложность её программной реализации и экономя время программисту.

Пример работы программы (Рис. 13):

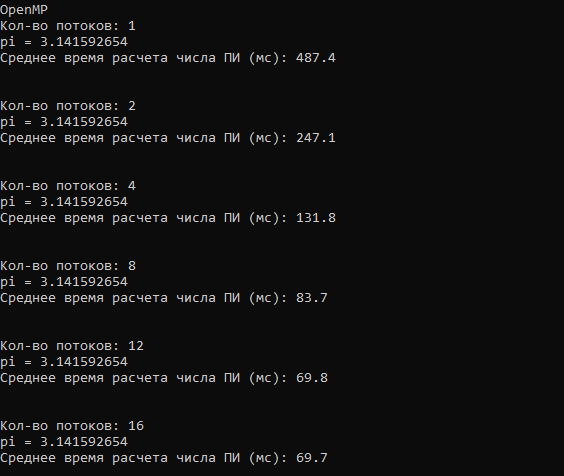


Рис. 13

**Общий вывод по работе**

Многозадачность – техника, используемая операционной системой, когда потоков создается больше, чем доступно физических ядер. Это позволяет распределить ресурсы процессора между потоками, так что каждый поток выполняется в течение некоторого времени до переключения на другой поток.

Техника многозадачности, или планирование потоков, позволяет максимально эффективно использовать ресурсы процессора и предоставлять каждому потоку порцию времени для выполнения своих задач. Операционная система решает, каким образом и в каком порядке будут выполняться потоки, исходя из их приоритета, времени выполнения и других факторов.

Таким образом, даже если количество созданных потоков превышает количество доступных физических ядер, операционная система все равно может эффективно управлять выполнением потоков и обеспечить параллельную обработку задач. Однако стоит иметь в виду, что при создании большого количества потоков возникают накладные расходы на переключение контекста и синхронизацию, поэтому не всегда целесообразно создавать гораздо больше потоков, чем доступных физических ядер.

Когда количество потоков превышает количество физических ядер, возникают дополнительные нагрузки на систему, связанные с конкуренцией за доступ к вычислительным ресурсам. Физическим ядрам приходится переключаться между потоками, что влечет затраты на переключение контекста и ожидание освобождения ядра. Это может объяснить, почему увеличение количества потоков после определенного порога не приводит к существенному улучшению производительности.

Результирующая производительность системы зависит от нескольких факторов, включая архитектуру процессора, характер задачи и эффективность использования параллельных вычислений. В некоторых случаях, когда задачи не могут быть эффективно распараллелены или они зависят от внешних факторов, увеличение числа потоков может не привести к улучшению производительности. Однако в других случаях, когда задачи можно эффективно распараллелить и система поддерживает соответствующую архитектуру, дополнительные потоки могут привести к улучшению производительности.

Применение многопоточности для улучшения производительности программы требует тщательного анализа и проектирования. Возможно, есть ограничения по количеству потоков, которые могут использоваться для параллельного выполнения задачи, такие как размер системы или характеристики самой задачи. Если нужно, можно провести оптимизацию количества потоков и распределение работы между ними для достижения наилучшего результата. Также стоит учитывать, что время выполнения программы может различаться из-за множества факторов, включая аппаратные характеристики компьютера, загрузку системы и другие внешние факторы, которые могут повлиять на общую производительность программы.

При разработке приложений с использованием параллельного программирования есть два эффективных подхода, которые можно использовать: WinAPI и OpenMP. В обоих случаях возможно распараллеливание выполнения кода на разных уровнях и использование множества ядер процессора для повышения производительности.

WinAPI предоставляет разработчикам Windows широкие возможности для создания многопоточных приложений. С помощью функций и инструментов WinAPI можно управлять потоками, синхронизировать их работу, а также осуществлять управление вводом-выводом и другими аспектами параллельного выполнения кода на уровне операционной системы.

OpenMP, в свою очередь, является стандартом для параллельного программирования на языках C и C++. Он предлагает набор директив, которые позволяют компилятору автоматически распараллеливать циклы, задачи и другие участки кода для эффективного использования многопроцессорных систем.

Оба подхода имеют свои особенности и преимущества. WinAPI предоставляет разработчикам больше гибкости и контроля в управлении потоками, позволяя точно настроить создание, синхронизацию и взаимодействие потоков. Он также предоставляет широкий набор функций для работы с многопоточностью и другими аспектами параллельного программирования на платформе Windows. Однако, использование WinAPI требует более низкоуровневых знаний и может потребовать больше усилий для реализации.

OpenMP, с другой стороны, предоставляет более высокоуровневую и декларативную абстракцию для параллельного программирования. Он позволяет разработчикам указывать директивы для автоматического распараллеливания циклов, задач и других участков кода. Это делает использование OpenMP более простым и доступным, особенно для разработчиков с ограниченным опытом в параллельном программировании.

Выбор между WinAPI и OpenMP зависит от требований вашего проекта. Если вам необходим полный контроль над созданием и управлением потоков, а также доступ к специфическим функциям операционной системы, то WinAPI может быть предпочтительным вариантом. Если же вам нужна более простая и удобная абстракция для распараллеливания кода без необходимости работы с низкоуровневыми функциями, то OpenMP может быть более удобным выбором.

В итоге, оба подхода могут быть эффективными в параллельном программировании, и выбор зависит от ваших предпочтений и требований проекта.

**Код программы**

#include <iostream>

#include <windows.h>

#include <time.h>

#include <omp.h>

#include <chrono>

#include <iomanip>

#include <locale>

#include <fstream>

using namespace std;

const int Nstud = 130725,

N = 100000000,

BLOCKSIZE = Nstud \* 10,

TIMES = 10;

int currentPos = 0;

double pi = 0.0;

LPCRITICAL\_SECTION section = new CRITICAL\_SECTION;

DWORD WINAPI MyThreadFunction(LPVOID lpParam){

int\* first = (int\*)lpParam;

int end = \*first + BLOCKSIZE;

long double x, tempPi;

while (\*first < N){

tempPi = 0.0;

for (int i = \*first; (i < end) && (i < N); ++i){

x = (i + 0.5)\*(1.0 / N);

tempPi += ((4.0 / (1.0 + x \* x)) \* (1.0 / N));

}

EnterCriticalSection(section);

pi += tempPi;

currentPos += BLOCKSIZE;

\*first = currentPos;

LeaveCriticalSection(section);

end = \*first + BLOCKSIZE;

}

return 0;

}

void winApi(int numThreads)

{

HANDLE hThreadArray[numThreads];

int position[numThreads];

double averageTime = 0.0;

for (int j = 0; j < TIMES; j++)

{

auto startTime = chrono::steady\_clock::now();

pi = 0.0;

InitializeCriticalSection(section);

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

{

position[i] = BLOCKSIZE \* i;

currentPos = position[i];

hThreadArray[i] = CreateThread(NULL, 0, MyThreadFunction, &position[i], 0, NULL);

if (hThreadArray[i] != NULL)

SuspendThread(hThreadArray[i]);

}

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

ResumeThread(hThreadArray[i]);

WaitForMultipleObjects(numThreads, hThreadArray, TRUE, INFINITE);

DeleteCriticalSection(section);

auto endTime = chrono::steady\_clock::now();

auto duration = chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime);

averageTime += duration.count();

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

CloseHandle(hThreadArray[i]);

}

cout << "Кол-во потоков: " << numThreads << endl;

cout << "pi = " << setprecision(10) << pi << endl;

cout << "Среднее время расчета числа ПИ (мс): " << setprecision(5) << (long double)(averageTime / TIMES) << endl << "\n";

ofstream outputFile("timeWinApi.csv", ios::app);

if (!outputFile)

{

cerr << "Не удалось открыть файл time.csv" << endl;

}

else

{

if (outputFile.tellp() != 0)

{

outputFile << endl;

}

outputFile << numThreads << "," << setprecision(5) << (long double)(averageTime / TIMES) << "\n";

outputFile.close();

}

}

void OpenMP(int numThreads)

{

long double pi = 0.0, averageTime = 0.0;

for (int j = 0; j < TIMES; j++){

pi = 0.0;

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

omp\_set\_num\_threads(numThreads);

#pragma omp parallel shared(startTime) reduction (+:pi)

{

#pragma omp for schedule(dynamic, BLOCKSIZE) nowait

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

{

long double xi = (i + 0.5)\*(1.0 / N);

pi += 4.0 / (1.0 + xi \* xi);

}

}

pi /= (long double)N;

auto duration = chrono::high\_resolution\_clock::now() - startTime;

averageTime += chrono::duration\_cast<chrono::milliseconds>(duration).count();

}

cout << "Кол-во потоков: " << numThreads << endl;

cout << "pi = " << setprecision(10) << pi << endl;

cout << "Среднее время расчета числа ПИ (мс): " << setprecision(5) << (long double)(averageTime / TIMES) << endl << "\n";

ofstream outputFile("timeOpenMP.csv", ios::app);

if (!outputFile)

{

cerr << "Не удалось открыть файл time.csv" << endl;

}

else

{

if (outputFile.tellp() != 0)

{

outputFile << endl;

}

outputFile << numThreads << "," << setprecision(5) << (long double)(averageTime / TIMES) << "\n";

outputFile.close();

}

}

int main() {

setlocale(LC\_ALL, "ru\_RU.UTF-8");

SetConsoleOutputCP(65001);

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

int numThreads = sizeof(threadCounts) / sizeof(threadCounts[0]);

int choice;

bool running = true;

while (running) {

cout << "Выберите режим работы (1 - WinApi или 2 - OpenMP) или 0 для выхода: ";

cin >> choice;

switch (choice) {

case 1:

cout << "\nWinApi" << endl;

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

{

winApi(threadCounts[i]);

cout << endl;

}

break;

case 2:

cout << "\nOpenMP" << endl;

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

{

OpenMP(threadCounts[i]);

cout << endl;

}

break;

case 0:

running = false;

break;

default:

cout << "Неверный выбор!" << endl;

break;

}

}

return 0;

}