**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: Потоки и процессы**

|  |  |  |
| --- | --- | --- |
| Студен гр. 2307 |  | Стукен В.А |
| Преподаватель |  | Тимофеев А.В |

Санкт-Петербург

2024

Оглавление

[Цель работы. 3](#_Toc165411362)

[Задание 3.1. Реализация многопоточного приложения с использованием функций Win32 API. 3](#_Toc165411363)

[Указания к выполнению. 3](#_Toc165411364)

[Демонстрация работы программы 4](#_Toc165411365)

[Код программы 5](#_Toc165411366)

[Выводы по первой части. 8](#_Toc165411367)

[Задание 3.2. Реализация многопоточного приложения с использованием технологии OpenMP. 9](#_Toc165411368)

[**Указания к выполнению.** 9](#_Toc165411369)

[Демонстрация работы программы 9](#_Toc165411370)

[Код программы 10](#_Toc165411371)

[Выводы по второй части 11](#_Toc165411372)

[**Источники** 12](#_Toc165411373)

# ****Цель работы.****

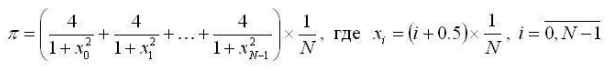
Исследовать механизмы создания и управления процессами и потоками в ОС Windows.

# ****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=100000000.



* 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, 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.

2. Произведите замеры времени выполнения приложения для

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

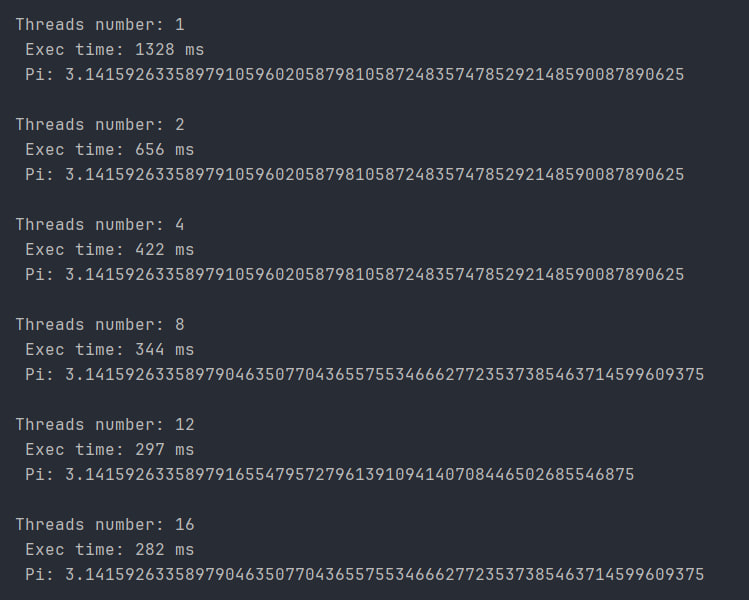
2 постройте график и определите число потоков, при котором достигается

наибольшая скорость выполнения. Запротоколируйте результаты в

отчет.

3. Подготовьте итоговый отчет с развернутыми выводами по заданию.

## Демонстрация работы программы

* 1. 

## Код программы

#include <iostream>  
#include <windows.h>  
#include <iomanip>  
#include <list>  
#include <numeric>  
  
using namespace std;  
  
DWORD beginTme = 0;  
DWORD endTime = 0;  
DWORD executionTime = -1;  
size\_t blockIndex;  
size\_t totalBlocks;  
HANDLE iterationMutex;  
HANDLE summaryMutex;  
const size\_t BLOCKSIZE = 10 \* 230720; //930824  
const size\_t N = 100000000;  
list<long double> resultList;  
long double finalResult = 0.0;  
  
  
DWORD WINAPI countingByBlock(LPVOID threadData);  
  
void prepareThreads(int numThreads);  
  
void rangeSetup(size\_t &start, size\_t &end);  
  
void countingPI(size\_t start, size\_t end);  
  
int main() {  
 int numberOfThreads[] = {1, 2, 4, 8, 12, 16};  
 int lengthOfList = sizeof(numberOfThreads) / sizeof(numberOfThreads[0]);  
  
 for (int i = 0; i < lengthOfList; i++) {  
 resultList.clear();  
 finalResult = 0.0;  
 prepareThreads(numberOfThreads[i]);  
 cout << "\n Threads number: " << numberOfThreads[i] << "\n " << " Exec time: " << executionTime << " ms"  
 << setprecision(N) << "\n " << " Pi: " << finalResult << "\n";  
 }  
 return 0;  
}  
  
  
void countingPI(size\_t start, size\_t end) {  
 long double localResult = 0.0;  
 int i;  
  
 localResult = 0.0;  
  
 for (i = start; i++ <= end;) {  
 localResult = localResult + (4 / (1 + (((long double) i + 0.5) / (long double) N) \*  
 (((long double) i + 0.5) / (long double) N)));  
 }  
  
 WaitForSingleObject(summaryMutex, INFINITE);  
  
 resultList.push\_back(localResult);  
  
 ReleaseMutex(summaryMutex);  
}  
  
void rangeSetup(size\_t &start, size\_t &end) {  
 WaitForSingleObject(iterationMutex, INFINITE);  
  
 if (blockIndex < totalBlocks) {  
 start = blockIndex \* BLOCKSIZE;  
 end = (blockIndex + 1) \* BLOCKSIZE - 1;  
 if (end > N - 1) {  
 end = N - 1;  
 }  
 blockIndex = blockIndex + 1;  
 } else {  
 start = 2;  
 end = 1;  
 }  
  
 ReleaseMutex(iterationMutex);  
}  
  
  
DWORD WINAPI countingByBlock(LPVOID threadData) {  
 int loopFlag = 1;  
 size\_t start = 2;  
 size\_t end = 1;  
 long double localResult = 0.0;  
  
 while (loopFlag != 0) {  
  
 rangeSetup(start, end);  
  
 if (start <= end) {  
 countingPI(start, end);  
 } else {  
 loopFlag = 0;  
 }  
 }  
  
 return 0;  
}  
  
void prepareThreads(int numThreads) {  
 blockIndex = 0;  
 totalBlocks = (N / BLOCKSIZE + 1);  
 int j;  
 auto \*threads = new HANDLE[numThreads];  
 iterationMutex = CreateMutex(nullptr, FALSE, nullptr);  
 summaryMutex = CreateMutex(nullptr, FALSE, nullptr);  
  
 for (j = 0; j < numThreads; j++) {  
 threads[j] = CreateThread(nullptr, 0, countingByBlock, nullptr, CREATE\_SUSPENDED, nullptr);  
 }  
  
 beginTme = GetTickCount();  
  
 for (unsigned i = 0; i < numThreads; i++) {  
 ResumeThread(threads[i]);  
 }  
  
 WaitForMultipleObjects(numThreads, threads, true, INFINITE);  
  
 finalResult = accumulate(std::begin(resultList), std::end(resultList), 0.0);  
 finalResult = finalResult / N;  
  
 endTime = GetTickCount();  
  
 executionTime = endTime - beginTme;  
  
 for (j = 0; j < numThreads; ++j) {  
 CloseHandle(threads[j]);  
 }  
 CloseHandle(iterationMutex);  
 CloseHandle(summaryMutex);  
 delete[] threads;  
}

## ****Выводы по первой части.****

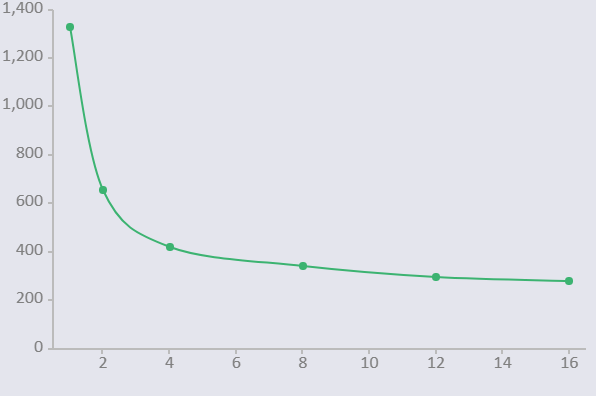


Рисунок 1 Зависимость времени выполнения от кол-ва потоков

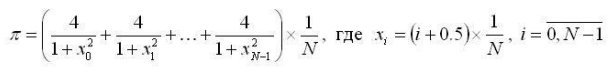
Для распределения итераций блоками по потокам была использована функция Win32 API CreateThread. Приложение создавало необходимое количество потоков в приостановленном состоянии в начале программы, а затем потоки по очереди получали свои блоки итераций. После завершения обработки текущего блока итераций потоки приостанавливались и ожидали выделения следующего свободного блока. Как видно по графику, время выполнения сильно уменьшается в промежутке от 1 до 8 потоков, а дальше почти переходит в параллельную прямую. Это говорит о том, что оптимальное количество потоков для данной задачи может быть меньше, чем количество доступных ядер процессора, из-за оверхеда создания и управления потоками.

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

**Указания к выполнению.**

1. Создайте приложение, которое вычисляет число пи с точностью

N знаков после запятой по следующей формуле



где N=100000000.

* Распределите работу по потокам с помощью OpenMP-директивы

for.

* Используйте динамическое планирование блоками итераций

(размер блока = 10 \* N\*студбилета).

2. Произведите замеры времени выполнения приложения для

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

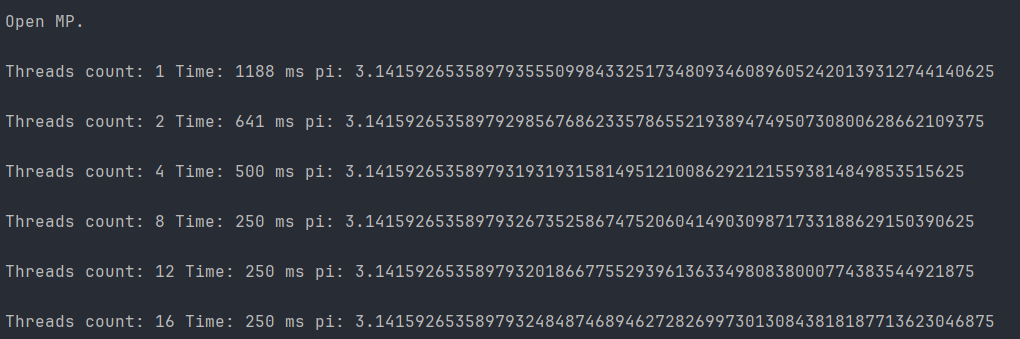
постройте график и определите число потоков, при котором достигается

наибольшая скорость выполнения. Запротоколируйте результаты в

отчет, сравните с результатами прошлой работы.

3. Подготовьте итоговый отчет с развернутыми выводами по заданию.

## Демонстрация работы программы



## Код программы

#include <iostream>  
#include <windows.h>  
#include <string>  
#include <iomanip>  
#include <omp.h>  
  
using namespace std;  
  
DWORD startTimer = 0;  
DWORD endTimer = 0;  
DWORD totalTime = -1;  
const size\_t BLOCKSIZE = 10 \* 230720;  
const size\_t N = 100000000;  
  
long double calculatePI(size\_t localIterations, size\_t localBlocksize, int localThreads);  
  
int main()  
{  
 int threadCounts[] = {1, 2, 4, 8, 12, 16};  
 int arraySize = sizeof(threadCounts) / sizeof(threadCounts[0]);  
 long double piValue;  
 cout << "\nOpen MP.\n";  
  
 for (int i = 0; i < arraySize; i++)  
 {  
 startTimer = 0;  
 endTimer = 0;  
 totalTime = 0;  
 piValue = calculatePI(N, BLOCKSIZE, threadCounts[i]);  
 cout << "\nThreads count: " << threadCounts[i] << " Time: " << totalTime << " ms" << setprecision(N) << " pi: " << piValue << "\n";  
 }  
 return 0;  
}  
  
long double calculatePI(size\_t localIterations, size\_t localBlocksize, int localThreads)  
{  
 startTimer = GetTickCount();  
  
 long double summaryResult = 0.0;  
#pragma omp parallel shared(startTimer, endTimer, totalTime) reduction(+: summaryResult) num\_threads(localThreads)  
 {  
#pragma omp for schedule(dynamic, localBlocksize) nowait  
 for (int i = 0; i < localIterations; i++)  
 {  
 summaryResult = summaryResult + (4 / (1 + (((long double)i + 0.5) / (long double)localIterations)\*(((long double)i + 0.5) / (long double)localIterations)));  
 }  
 }  
  
 summaryResult = summaryResult / localIterations;  
 endTimer = GetTickCount();  
  
 totalTime = endTimer - startTimer;  
  
 return summaryResult;  
}

## Выводы по второй части

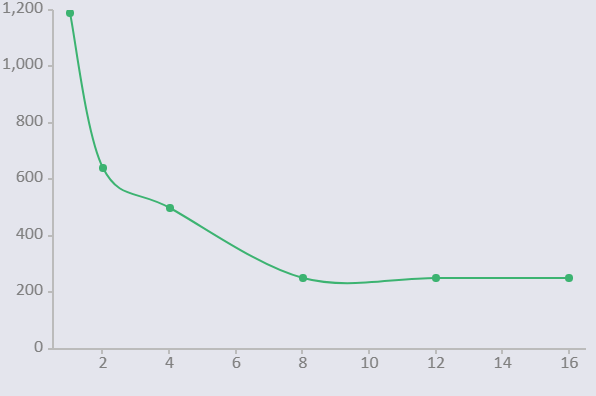


Рисунок 2 Зависимость времени работы от кол-ва потоков

В результате выполнения лабораторной работы было создано приложение для вычисления числа π с точностью до N знаков после запятой, где N=100000000, используя формулу. Работа была распределена по потокам с использованием OpenMP-директивы for, при этом применялось динамическое планирование блоками итераций. При анализе скорости выполнения программы видно, что реализация с использованием OpenMP показывает примерно те же результаты, что говорит об отсутствии ощутимых отличий реализации через OpenMP в плане скорости, однако в остальном этот подход кажется лучше. Во-первых, простота использования, меньше кода и меньше мест, где можно «выстрелить себе в ногу». Во-вторых, OpenMP реализует множество оптимизаций, таких как автоматическое распределение нагрузки между потоками и согласование доступа к общим данным, что может улучшить производительность параллельного кода. А также Многие современные компиляторы имеют встроенную поддержку OpenMP, что позволяет автоматически встраивать директивы OpenMP в код при компиляции. Это упрощает процесс сборки и оптимизации параллельного приложения.

**Источники**

1. Операционные системы: электронные методические указания к лабораторным работам / Сост.: А. В. Тимофеев. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2016.

2. Курс «Операционные системы» в образовательной онлайн-системе Google Класс. Материалы лекций.

3. <https://www.openmp.org>

4. <https://passlab.github.io/OpenMPProgrammingBook/openmp_c/1_IntroductionOfOpenMP.html>

5. <https://learn.microsoft.com/ru-ru/cpp/parallel/openmp/1-introduction?view=msvc-170>