МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Операционные системы»
Тема: Процессы и потоки

Студент гр. 9308 _____ Яловега Н.В.

Преподаватель _____ Тимофеев А.В.

Санкт-Петербург
2021

Введение

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

Задание:

1. Реализация многопоточного приложения с использованием функций Win32 API:

$$\pi = \left(\frac{4}{1+x_0^2} + \frac{4}{1+x_1^2} + \ldots + \frac{4}{1+x_{N-1}^2} \right) \times \frac{1}{N}, \text{ где } x_i = (i+0.5) \times \frac{1}{N}, \; i = \overline{0, N-1}$$

- Создать приложение, которое вычисляет число pi с точностью N знаков после запятой по следующей формуле (N = 100000000):
- Использовать распределение итераций блоками (размер блока = 10 * N студбилета) 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;
- Создать потоки 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, 32). По результатам измерений построить график и определить число потоков, при котором достигается наибольшая скорость выполнения.

2. Реализация многопоточного приложения с использованием технологии OpenMP:

- Создать приложение, которое вычисляет число pi с точностью N знаков после запятой по следующей формуле (N = 100000000):

$$\pi = \left(\frac{4}{1+x_0^2} + \frac{4}{1+x_1^2} + \ldots + \frac{4}{1+x_{N-1}^2} \right) \times \frac{1}{N}, \text{ где } x_i = (i+0.5) \times \frac{1}{N}, \; i = \overline{0, N-1}$$

- Распределить работу по потокам с помощью OpenMP-директивы for;
- Использовать динамическое планирование блоками итераций (размер блока = 10 * N студбилета);
- Произвести замеры времени выполнения приложения для разного числа потоков (1, 2, 4, 8, 12, 16, 32). По результатам измерений построить график и определить число потоков, при котором достигается наибольшая скорость выполнения.

Задание 1. Реализация многопоточного приложения с использованием функций Win32 API

Чтобы память не испортилась в случае если два потока будут одновременно изменять одну ячейку нам надо как-то гарантировать что в один момент времени только один поток получает доступ к переменной, которая хранит число Пи.

В Win32 API существует несколько методов синхронизации потоков. Самый простой из методов синхронизации потоков состоит в использовании критических секций. Критическая секция — это некоторый участок кода, который в каждый момент времени может выполняться только одним из потоков. Если код, поместить в критическую секцию, то другие потоки не смогут войти в этот участок кода до тех пор, пока первый поток не завершит его выполнение. Это единственный метод синхронизации потоков, который не требует привлечения ядра Windows. (Критическая секция не является объектом ядра). Однако этот метод может использоваться только для синхронизации потоков одного процесса.

Условия проведения эксперимента

ПК с Intel Core i7-8750H (6 физических ядер или 12 логических) и DDR4.

При проведении эксперимента возникают случайные погрешности, неизбежно возникающие в работающей операционной системе, которая может начать процесс обновления или оптимизации, не уведомляя пользователя. Для уменьшения влияния случайных факторов, будем проводить не один, а сразу K экспериментов (для всех экспериментов использовалось значение 10) с параллельной программой, не меняя исходные данные. Получается K замеров времени, которые в общем случае будут различными вследствие различных случайных факторов, влияющих на проводимый эксперимент. Будем принимать результирующее время как среднее арифметическое K замеров времени. Кроме того, система минимально нагружена дополнительными процессами.

Результат эксперимента

В результате проведения эксперимента можем построить график зависимости времени от количества потоков. График приведен на рисунке 1.

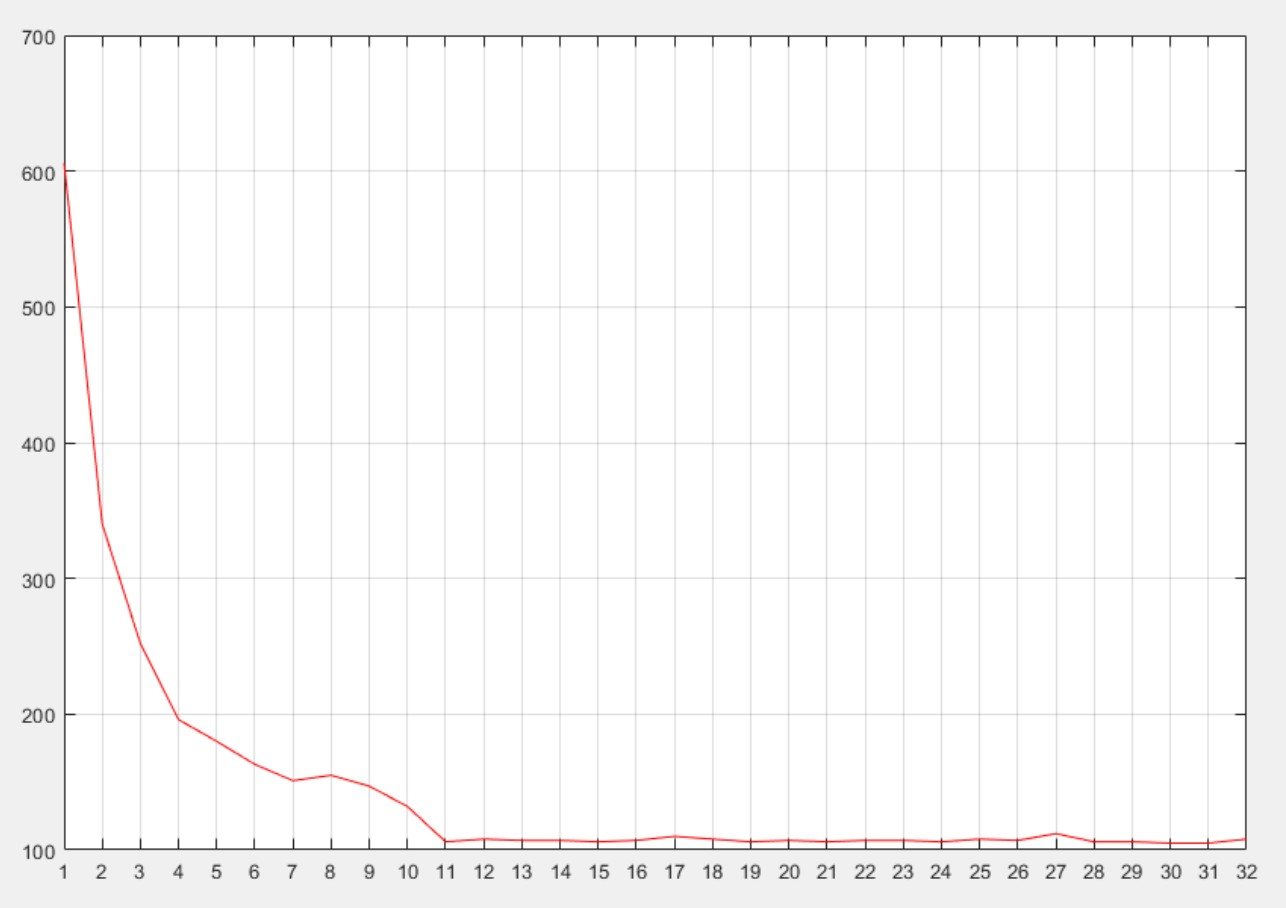

Рисунок 1: График зависимости времени от количества потоков для задания 1

Из графика сделать вывод, что время почти не будет меняться, начиная с количества потоков, равное примерно количеству логических ядер процессора: начиная с примерно 12 потоков время уже почти не меняется и остаётся минимальным.

Выводы по заданию

Исследованы механизмы создания и управления процессами и потоками в ОС Windows. Изучены методы синхронизации потоков средствами Win32 API

Оптимальное количество потоков – 12.

Задание 2. Реализация многопоточного приложения с использованием технологии OpenMP

OpenMP — открытый стандарт для распараллеливания программ на языках C/C++/Fortran. Он позволяет делать примерно то же самое, что мы делали с помощью API потоков операционной системы — но гораздо проще и лаконичней, так как всю работу по запуску и уничтожению потоков, по распределению работы между потоками и т.п берет на себя OpenMP. Для реализации параллельного выполнения блоков приложения нужно просто добавить в код директивы pragma.

Для приложения с OpenMP использовались следующие директивы препроцессора:

#pragma omp parallel reduction (+: pi) num_threads(threadNum) – означает, что блок нужно выполнять параллельно и что нужно задать локальную переменную pi, которая потом "просуммируется" к итоговой pi.

#pragma omp for schedule(dynamic, BLOCKSIZE) nowait - означает, что планирование будет динамическим, где каждый поток будет получать blocksize итераций, а после выполнения запросит ещё; nowait означает, что барьерной синхронизации в конце цикла не будет.

Условия проведения эксперимента

ПК с Intel Core i7-8750H (6 физических ядер или 12 логических) и DDR4.

При проведении эксперимента возникают случайные погрешности, неизбежно возникающие в работающей операционной системе, которая может начать процесс обновления или оптимизации, не уведомляя пользователя. Для уменьшения влияния случайных факторов, будем проводить не один, а сразу K экспериментов (для всех экспериментов использовалось значение 10) с параллельной программой, не меняя исходные данные. Получается K замеров времени, которые в общем случае будут различными вследствие различных случайных факторов, влияющих на проводимый эксперимент. Будем принимать результирующее время как среднее арифметическое K замеров времени. Кроме того, система минимально нагружена дополнительными процессами.

Результат эксперимента

В результате проведения эксперимента можем построить график зависимости времени от количества потоков. График приведен на рисунке 2.

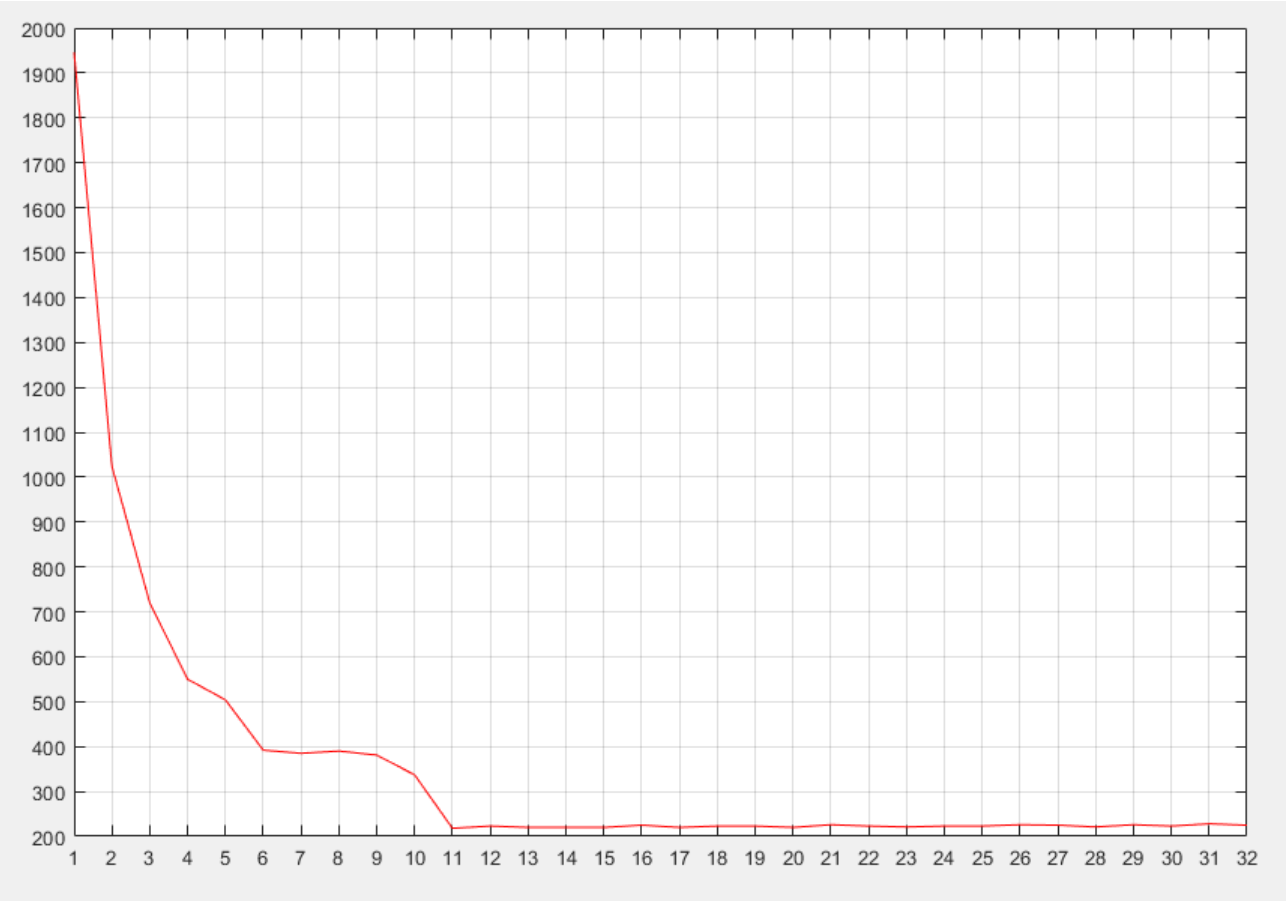

Рисунок 2: График зависимости времени от количества потоков для задания 2

Из графика сделать вывод, что время почти не будет меняться, начиная с количества потоков, равное примерно количеству логических ядер процессора: начиная с примерно 12 потоков время уже почти не меняется и остаётся минимальным.

Выводы по заданию

Использование OpenMP значительно сократило код, так как вся логика, отвечающая за запуск и уничтожение потоков, за распределение работы между потоками и т.п берет на себя OpenMP. Кроме того, при использовании OpenMP мы получили обратную совместимость и кроссплатформенность.

Оптимальное количество потоков – 12.

Сравнение реализаций

Построим результаты экспериментов на одном графике, чтоб наглядно сравнить их. Общий график приведен на рисунке 3.

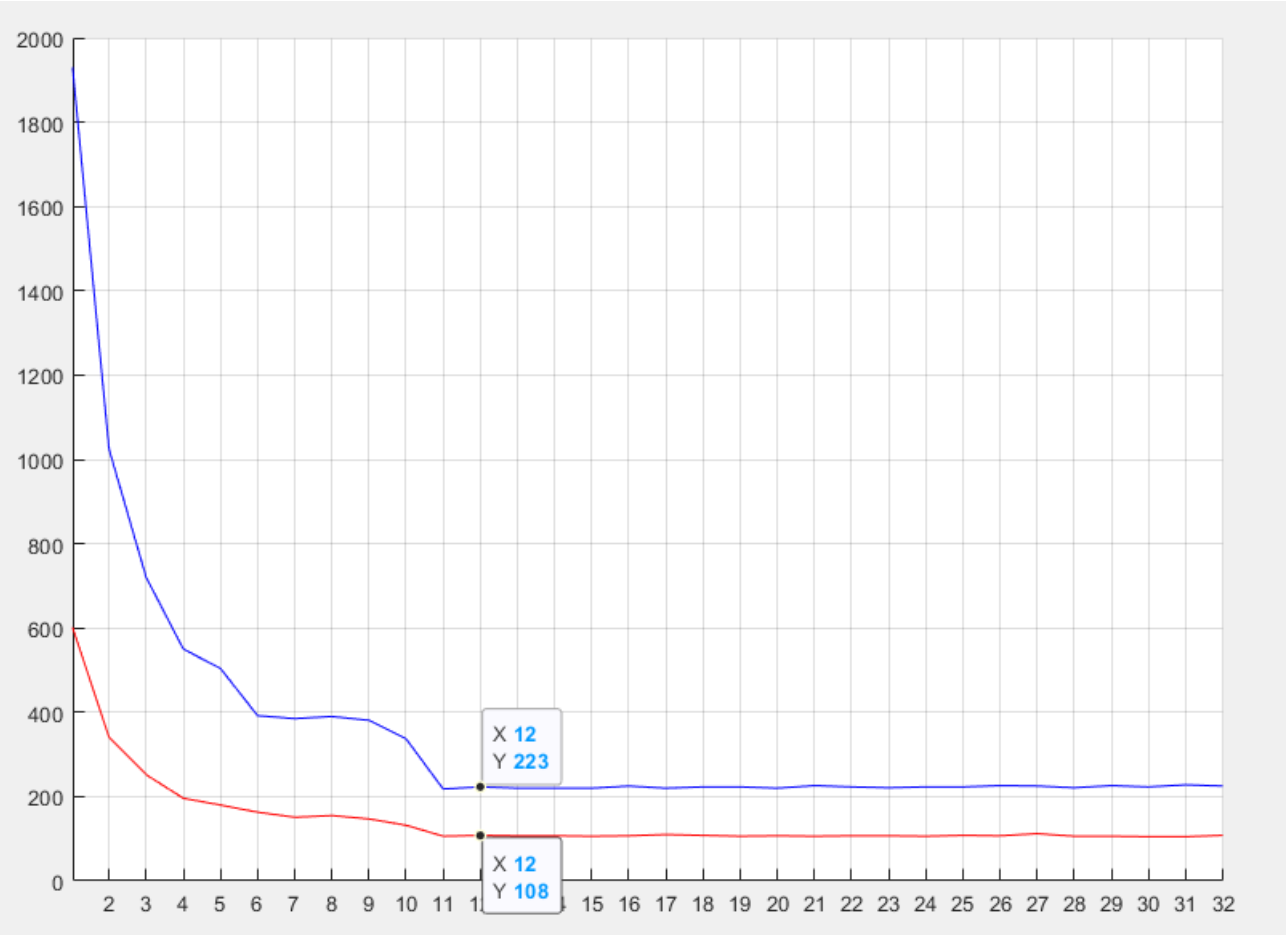

Рисунок 3: Сравнение результатов экспериментов

Красная зависимость - Win32 API.

Синяя зависимость – OpenMP.

Как видно из графиков, использование Win32 API дает лучшую производительность сравнению с кроссплатформенной технологией OpenMP. Разница времени выполнения одинаковой задачи при значении количества потоков 12 составляет 115 мс.

Вывод

Исследованы механизмы создания и управления процессами и потоками в ОС Windows. Выявлено, что с увеличением числа потоков прирост производительности постоянно падает, а после значения, равного примерно количеству логических ядер процессора практически перестает изменяться. Это происходит потому что логические ядра определяют число одновременно выполняемых потоков. А так как при увеличении числа потоков больше этого числа не осталось никакой вычислительной мощности для выполнения других потоков, пока один из них не будет завершен или получен, что не поможет быстрее закончить вычисления.

Приложения

task1.cpp

```
#include <iomanip>
#include <iostream>
#include <windows.h>

const size_t N = 100000000;
const size_t BLOCKSIZE = 9308060;

int currentPos = 0;
long double pi = 0.0;
LPCRITICAL_SECTION section = new CRITICAL_SECTION;

DWORD WINAPI calculatingFunc(LPVOID lpParam)
{
   int* first = (int*)lpParam;
   int end = *first + BLOCKSIZE;
   long double x = 0.0, sl = 0.0;

   do
   {
      sl = 0.0;
      for (int i = *first; (i < end) && (i < N); ++i)
      {
         x = (i + 0.5) / N;
         sl += (4 / (1 + x * x));
      }
      EnterCriticalSection(section);
      pi += sl;
      currentPos += BLOCKSIZE;
      *first = currentPos;
      LeaveCriticalSection(section);
      end = *first + BLOCKSIZE;
   }
   while (*first < N);

   return 0;
}


long double calculatePI (const unsigned threadNum, DWORD *millisec)
{
   DWORD begin;
   DWORD end;

   HANDLE* Threads = new HANDLE[threadNum];
   int* position = new int[threadNum];
   InitializeCriticalSection(section);

   for (int i = 0; i < threadNum; ++i)
   {
      position[i] = BLOCKSIZE * i;
```

```
        currentPos = position[i];
        Threads[i] = CreateThread(NULL, 0, calculatingFunc, &position[i], CREATE_SUSPENDED,
NULL);
    }

    begin = clock();

    for (int i = 0; i < threadNum; ++i)
        ResumeThread(Threads[i]);

    WaitForMultipleObjects(threadNum, Threads, TRUE, INFINITE);

    pi /= (long double)N;

    end = clock();

    *millisec = end - begin;
    DeleteCriticalSection(section);
    return pi;
}

void printArray(unsigned* a, size_t n)
{
    std::cout << "[";
    for(size_t i = 0; i < n; ++i)
    {
        std::cout << std::to_string(a[i]);
        if(i != n-1)
            std::cout << ", ";
    }
    std::cout << "];\n";
}

int main(int argc, char **argv)
{
    if(argc == 1) // Для замеров
    {
        const unsigned maxThreads = 32;
        const unsigned attempts = 10;
        DWORD currentMillis;
        unsigned *results = new unsigned[maxThreads];
        unsigned *x = new unsigned[maxThreads];
        for(unsigned i = 1; i <= maxThreads; ++i)
        {
            DWORD averageMillis = 0;
            std::cout << "Number of threads: " << i << std::endl;
            for(unsigned j = 0; j < attempts; ++j)
            {
                std::cout << std::setprecision(5) << "Result of pi calculations (attempt " << (j+1) << "): "
<< calculatePI(i, &currentMillis) << std::endl;
                averageMillis += currentMillis;
            }
            averageMillis /= attempts;
```

```
            results[i-1] = averageMillis;
            x[i-1] = i;
            std::cout << "Time of pi calculations " << averageMillis << " millisec (" << (long double)
averageMillis / 1000 << " sec)." << std::endl;
        }
        printArray(x, maxThreads);
        printArray(results, maxThreads);
        delete results;
        delete x;
    }
    else if(argc == 2) // для тестов
    {
        unsigned threadNum = atoi(argv[1]);
        DWORD millisec = -1;
        long double pi = calculatePI(threadNum, &millisec);
        std::cout << std::setprecision(N) << "Result of pi calculations: " << pi << std::endl;
        std::cout << "Time of pi calculations " << millisec << " millisec (" << (long double)millisec /
1000 << " sec)." << std::endl;
    }
    else
    {
        return -1;
    }
    return 0;
}
```

task2.cpp

```
#include <iomanip>
#include <iostream>
#include <windows.h>
#include <omp.h>

const size_t N = 100000000;
const size_t BLOCKSIZE = 9308060;

long double calculatePI(const unsigned threadNum, DWORD *millisec)
{
   DWORD begin;
   DWORD end;
   long double pi = 0;

   begin = GetTickCount();

   #pragma omp parallel shared(begin, end) reduction (+: pi) num_threads(threadNum)
   {
      #pragma omp for schedule(dynamic, BLOCKSIZE) nowait
      for (size_t i = 0; i < N; ++i)
      {
         long double x_i;
         x_i = (i + 0.5) * 1.0 / N;
         pi += (long double) 4.0 / (1.0 + x_i * x_i);
      }
   }

   pi /= (long double)N;

   end = GetTickCount();

   *millisec = end - begin;
   return pi;
}

void printArray(unsigned* a, size_t n)
{
   std::cout << "[";
   for(size_t i = 0; i < n; ++i)
   {
      std::cout << std::to_string(a[i]);
      if(i != n-1)
         std::cout << ", ";
   }
   std::cout << "];\n";
}

int main(int argc, char **argv)
{
   if(argc == 1) // Для замеров
   {
```

```
        const unsigned maxThreads = 32; // 72
        const unsigned attempts = 10;
        DWORD currentMillis;
        unsigned *results = new unsigned[maxThreads];
        unsigned *x = new unsigned[maxThreads];
        for(unsigned i = 1; i <= maxThreads; ++i)
        {
            DWORD averageMillis = 0;
            std::cout << "Number of threads: " << i << std::endl;
            for(unsigned j = 0; j < attempts; ++j)
            {
                std::cout << std::setprecision(5) << "Result of pi calculations (attempt " << (j+1) << "): "
<< calculatePI(i, &currentMillis) << std::endl;
                averageMillis += currentMillis;
            }
            averageMillis /= attempts;
            results[i-1] = averageMillis;
            x[i-1] = i;
            std::cout << "Time of pi calculations " << averageMillis << " millisec (" << (long double)
averageMillis / 1000 << " sec)." << std::endl;
        }
        printArray(x, maxThreads);
        printArray(results, maxThreads);
        delete results;
        delete x;
    }
    else if(argc == 2) // для тестов
    {
        unsigned threadNum = atoi(argv[1]);
        DWORD millisec = -1;
        std::cout << "Number of threads: " << threadNum << std::endl;
        long double pi = calculatePI(threadNum, &millisec);
        std::cout << std::setprecision(N) << "Result of pi calculations: " << pi << std::endl;
        std::cout << "Time of pi calculations " << millisec << " millisec (" << (long double)millisec /
1000 << " sec)." << std::endl;
    }
    else
    {
        return -1;
    }
    return 0;
}
```