**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчет

**по лабораторной работе №5**

**по дисциплине «Операционные системы»**

Тема: «Процессы и потоки»

|  |  |  |
| --- | --- | --- |
| Студент гр. 7307 |  | Торопов. В.А. |
| Преподаватель |  | Тимофеев А.В. |

Санкт-Петербург

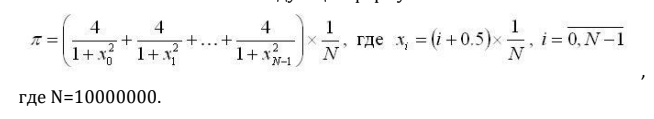
2019

**Цель работы.**

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

**Задание 1.** Реализация многопоточного приложения с использованием функций Win32 API.

Создать многопоточное консольное приложение, которое вычисляет число Пи с точностью до N знаков по формуле:



Использовать распределение итераций блоками размером 730720 \* 10 по потокам. Так для N = 10000000 будет 2 блока. Выявить число потоков, при котором достигается наибольшая скорость выполнения программы.

Проведем эксперимент по выявлению самого эффективного числа потоков.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N потоков | 1 | 2 | 4 | 8 | 12 | 16 |
| Время выполнения, мс | 343 | 234 | 250 | 250 | 250 | 250 |

Как видно на графике и по данным максимальная эффективность выполнения подсчета достигается при использовании 2 потоков, что логично для подсчета 2 блоков. Так будут работать ток 2 потока и остальные влиять на работу программы не будут.

Попробуем увеличить размер N в 10 раз и повторить эксперимент. Так получим размер блока и их количество больше 10.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N потоков | 1 | 2 | 4 | 8 | 12 | 16 |
| Время выполнения, мс | 3326 | 1868 | 692 | 400 | 379 | 380 |

При данной точности 12 потоков наблюдается заметная эффективность числом потоков, так как далее время работы уже уменьшается незначительно.

Теперь запустим приложение с 6 потоками и выведем информацию о появившемся процессе и его потоках.

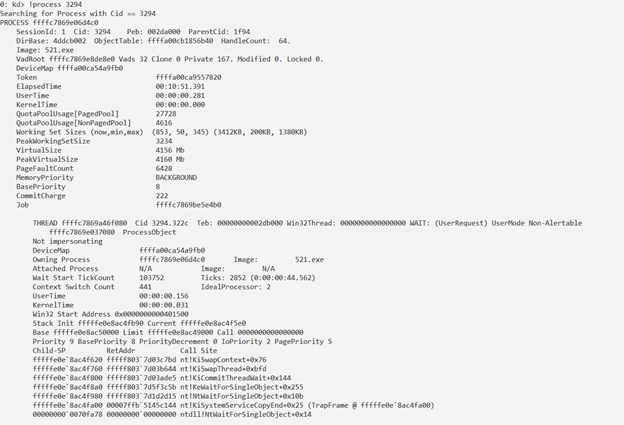


Рисунок 1.1. Информация о процессе

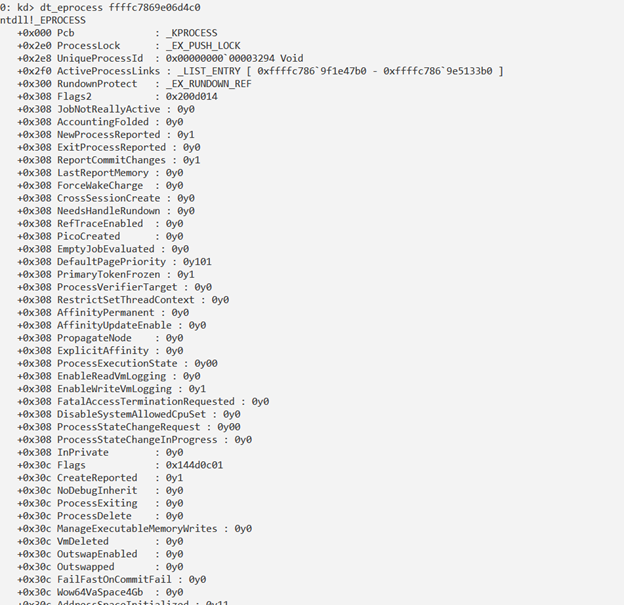


Рисунок 1.2a. Подробная информация процесса

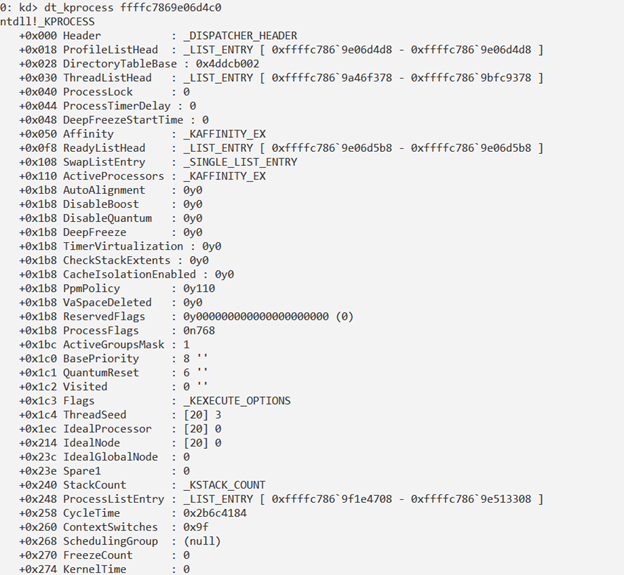


Рисунок 1.2б. Подробная информация процесса

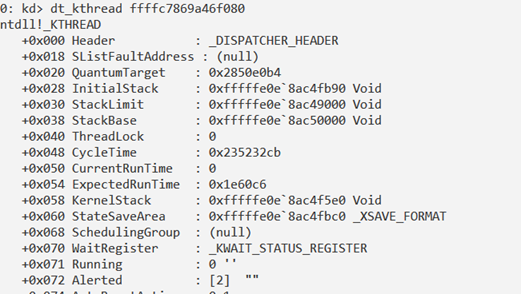


Рисунок 1.3. Информация об одном из потоков

Потоки находятся в состоянии WAIT

Выведем информацию о 3 ядре процессора.

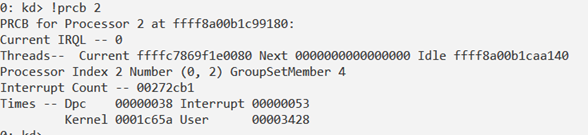


Рисунок 1.4. Информация 3 ядра

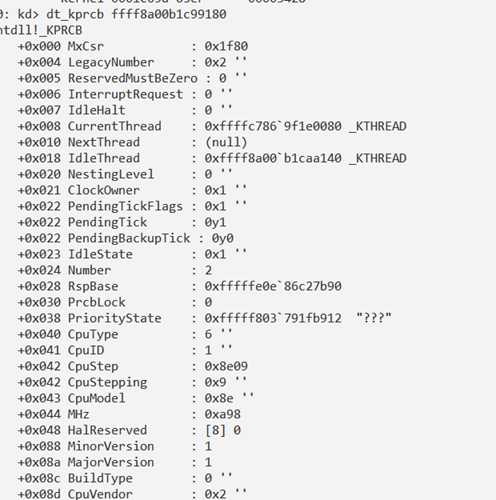


Рисунок 1.5. Информация о потоках ядра

Как видно, на ядре в данный момент выполняется поток простоя и нет следующего в очереди. Выведем информацию о потоке простоя.

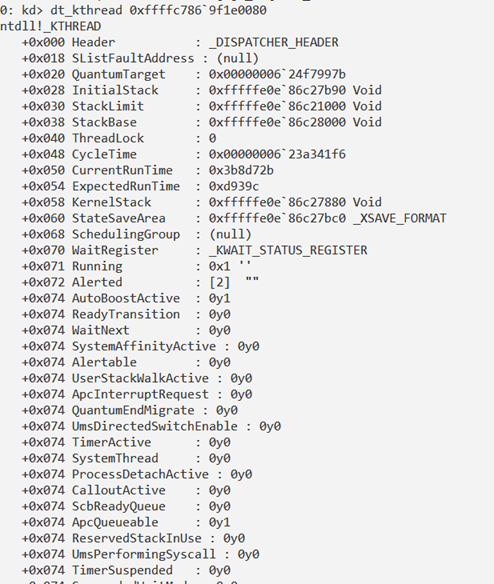


Рисунок 1.5.

**Вывод**: Использование нескольких параллельных потоков в определенных случаях позволяет ускорить выполнение программ. При написании программ нужно учитывать, что не все алгоритмы можно реализовать параллельно (например, не всех операций хватает для распределения по потокам). Так же стоит учитывать, что изменение производительности нелинейно, а более того ограничено по закону Амдала. Поэтому не всегда стоит использовать все ядра и все потоки, которые имеются. Стоит оптимально продумывать архитектуру и логику программ для конкретных систем и задач, для уменьшения возникновения ошибок и поддержки потокобезопасности данных, а также лучшей синхронизации потоков. В некоторых ситуациях оптимальное число потоков может быть приблизительно просто на основе исходных данных.

**Задание 2**. Реализация многопоточного приложения с использованием технологии OpenMP.

Написать программу, выполняющую идентичную задачу из предыдущего задания, используя OpenMP, и выявить число потоков, при котором достигается наибольшая скорость выполнения программы.

Проведем эксперимент, идентичный эксперименту из предыдущего задания при N = 10000000 и размере блока 730720 \* 10.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N потоков | 1 | 2 | 4 | 8 | 12 | 16 |
| Время выполнения, мс | 230 | 159 | 163 | 161 | 167 | 162 |

Очевидно, что уже при 2 потоках эффективность от роста числа потоков не растет, как и в задании 1.

Так же попробуем снова изменить размер N в 10 раз и повторить эксперимент. Так получим размер блока N = 100000000 и количество станет больше 10.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N потоков | 1 | 2 | 4 | 8 | 12 | 16 |
| Время выполнения, мс | 2190 | 1123 | 656 | 344 | 330 | 328 |

В данном случае из-за большого числа блоков, которые необходимо обработать, количество потоков играет более значимую роль, однако на приблизительно 12 потоках почти достигается максимум эффективности, затем прирост эффективности становится незначительным и колеблется.

Теперь запустим программу с 5 потоками и выведем информацию о процессе и его потоках.

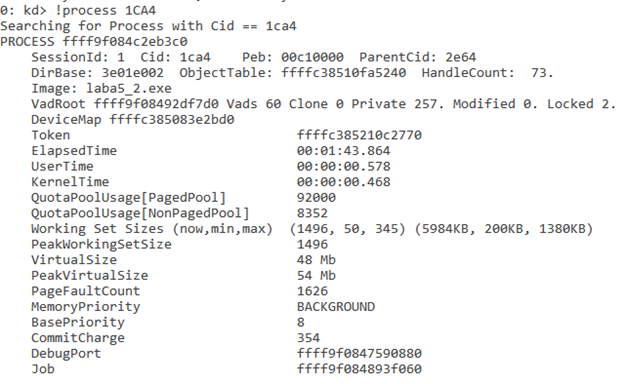
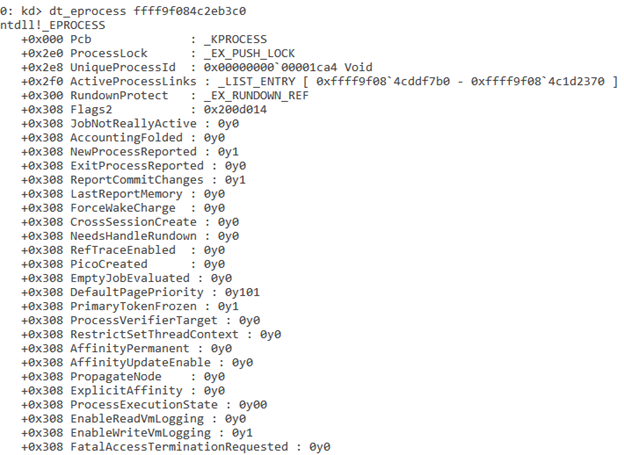


Рисунок 2.1. Информация о процессе



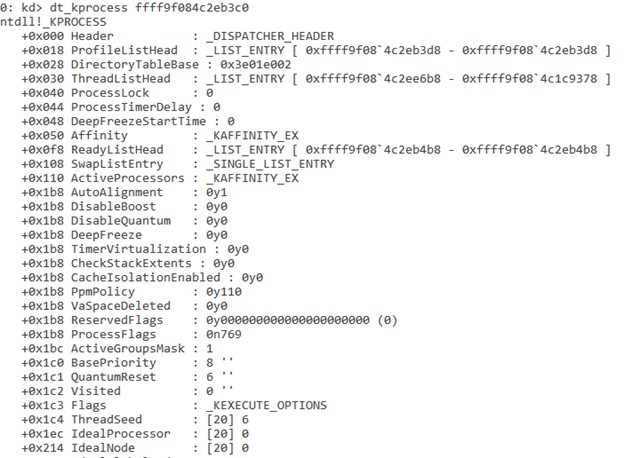


Рисунок 2.2. Подробная информация процесса

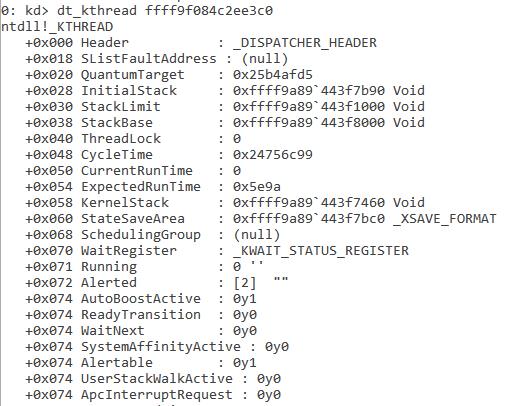


Рисунок 2.3. Поток 1

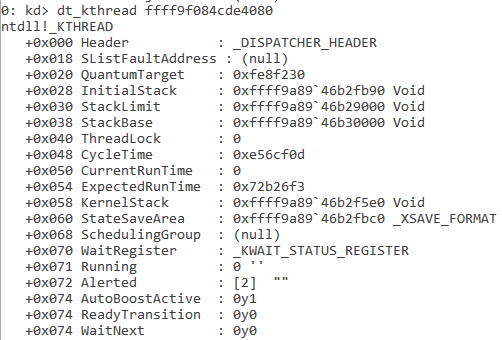


Рисунок 2.4. Поток 2

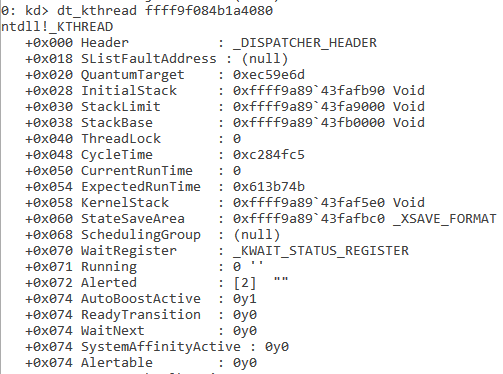


Рисунок 2.5. Поток 3

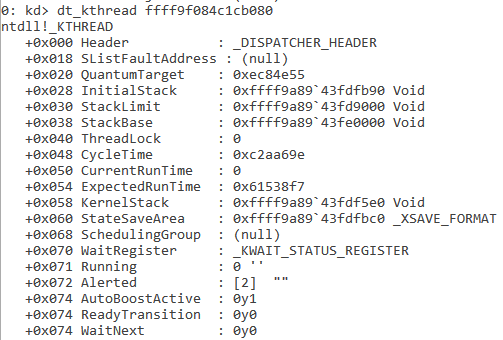


Рисунок 2.6. Поток 4

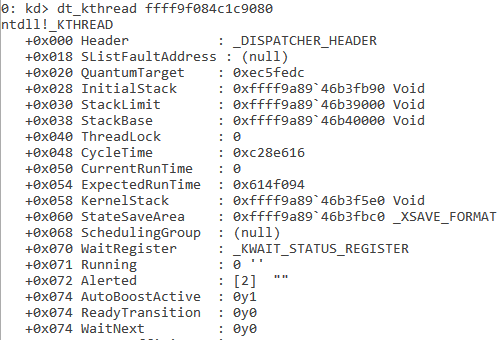


Рисунок 2.7. Поток 5

Все потоки находятся в состоянии WAIT.

Теперь выведем информацию о 2 ядре процессора.

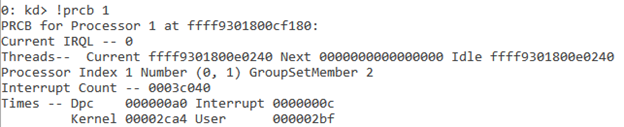


Рисунок 2.8. Информация о 2 ядре процессора

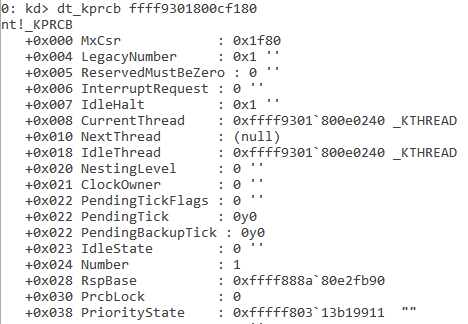


Рисунок 2.9. Информация о работе ядра

Как видно, на ядре в данный момент выполняется поток простоя и нет

следующего в очереди.

Теперь выведем информацию о потоке простоя.

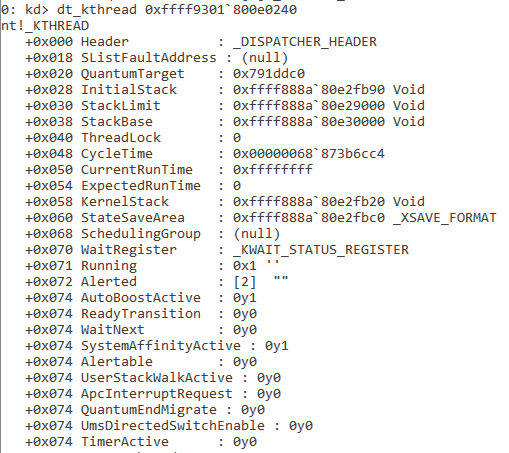


Рисунок 2.10. Информация о потоке

**Вывод:** OpenMP сильно упрощает разработку многопоточных приложений, облегчая синхронизацию и планирование потоков, обращение к общим переменным среди потоков. Также можно заметить, что программа, написанная с помощью OpenMP, выполняется быстрее, чем предыдущее приложение. Стоит отметить, что при использовании OpenMP можно добиться результатов лучше, чем при написании без него, поскольку OpenMP упрощает синхронизацию потоков, уменьшает количество организационных операций в коде, более продуманно распределяет итерации между потоками. Главным преимуществом OpenMP является спасение от неправильной организации работы потоков программистом.

Приложение 1

#include <iostream>  
#include<windows.h>  
#include<iomanip>  
#include <chrono>  
  
HANDLE \*threads;  
HANDLE \*events;  
const int N = 100000000; //100M  
const int blocksize = 730720 \* 10;  
const int numBlocks = N / blocksize + 1;  
int \*blockArray;  
volatile double pi = 0;  
volatile long int numBlock;  
using namespace std;  
  
DWORD WINAPI ThreadFunc(LPVOID lpParam) {  
 double x;  
 double threadPi = 0;  
 int currentThreadBlock = (intptr\_t) lpParam;  
 if (currentThreadBlock >= numBlocks) {  
 SetEvent(events[(intptr\_t) lpParam]);  
 SuspendThread(threads[(intptr\_t) lpParam]);  
 } else {  
 while (currentThreadBlock < numBlocks) {  
 int startIteration = currentThreadBlock \* blocksize;  
 int endIterarion = (currentThreadBlock + 1) \* blocksize;  
 if (endIterarion > N)  
 endIterarion = N;  
 for (int i = startIteration; i < endIterarion; i++) {  
 x = (i + 0.5) / N;  
 threadPi += 4 / (1 + x \* x);  
 }  
 SetEvent(events[(intptr\_t) lpParam]);  
 SuspendThread(threads[(intptr\_t) lpParam]);  
 currentThreadBlock = blockArray[(intptr\_t) lpParam];  
  
 }  
 pi += threadPi;  
 }  
 return 0;  
}  
  
  
int main() {  
 int numThread;  
 do {  
 cout << "Enter the number of threads" << endl;  
 std::cin >> numThread;  
 if (numThread != 0) {  
 threads = new HANDLE[numThread];  
 events = new HANDLE[numThread];  
 blockArray = new int[numThread];  
 for (int i = 0; i < numThread; i++) {  
 blockArray[i] = i;  
 threads[i] = CreateThread(NULL, 0, ThreadFunc, (LPVOID) i, CREATE\_SUSPENDED, NULL);  
 if (!threads[i])  
 std::cout << "Couldn't create a thread " << i << ". Error " << GetLastError() << std::endl;  
 events[i] = CreateEventA(NULL, true, 0, NULL);  
 }  
 numBlock = numThread;  
 auto start = chrono::high\_resolution\_clock::now();  
 for (int i = 0; i < numThread; i++)  
 ResumeThread(threads[i]);  
  
 while (numBlock < numBlocks) {  
 int suspendedThreadIndex = WaitForMultipleObjects(numThread, events, false, INFINITE) - WAIT\_OBJECT\_0;  
 blockArray[suspendedThreadIndex] = numBlock++;  
 Sleep(1); // resume after suspend  
 ResetEvent(events[suspendedThreadIndex]);  
 ResumeThread(threads[suspendedThreadIndex]);  
 }  
  
 int countFinalThread = 0;  
 while (countFinalThread != numThread) {  
 int suspendedThreadIndex = WaitForMultipleObjects(numThread, events, false, INFINITE);  
 blockArray[suspendedThreadIndex] = numBlocks + 1;  
 countFinalThread++;  
 Sleep(1);  
 ResetEvent(events[suspendedThreadIndex]);  
 ResumeThread(threads[suspendedThreadIndex]);  
 }  
  
 WaitForMultipleObjects(numThread, threads, true, INFINITE);  
  
 auto end = chrono::high\_resolution\_clock::now();  
  
 auto time = chrono::duration\_cast<chrono::milliseconds>(end - start).count();  
 cout << "Time (in milliseconds): " << time << endl;  
 pi = pi / N;  
 std::cout << "Pi = " << std::setprecision(N) << pi << endl;  
  
 for (int i = 0; i < numThread; i++) {  
 CloseHandle(threads[i]);  
 CloseHandle(events[i]);  
 }  
 system("pause");  
 pi = 0;  
 }  
 } while (numThread != 0);  
 return 0;  
}

Приложение 2

#include <iostream>  
#include <omp.h>  
#include <Windows.h>  
#include <iomanip>  
#include <chrono>  
#include <ctime>  
  
using namespace std;  
static const DWORD BLOCK\_SIZE = 730734 \* 10;  
static const DWORD N = 100000000;  
#pragma comment(lib, "winmm.lib")  
  
std::pair<DWORD, double> OMP(int q)  
{  
 DWORD t1 = 0, t2 = 0;  
 double pi = 0, x = 0;  
 omp\_set\_dynamic(0);  
 auto start = std::chrono::system\_clock::now();  
  
#pragma omp parallel for schedule(dynamic, BLOCK\_SIZE) private(x) reduction(+:pi) num\_threads(q)  
 for (int i = 0; i < N; i++)  
 {  
 x = (i + 0.5) / N;  
 pi += 4 / (1 + x \* x);  
 }  
  
 pi = pi / N;  
 auto end = std::chrono::system\_clock::now();  
 std::chrono::duration<double> elapsed\_seconds = end - start;  
 std::time\_t end\_time = std::chrono::system\_clock::to\_time\_t(end);  
 std::cout << "finished computation at " << std::ctime(&end\_time)  
 << "elapsed time: " << elapsed\_seconds.count() << "s\n";  
 return make\_pair(0, pi);  
  
}  
  
int main()  
{  
 setlocale(0, "rus");  
  
 int q;  
 do  
 {  
 cout << "To exit enter 0" << endl;  
 cout << "Enter the number of threads: " << endl;  
 cin >> q;  
 if (q) {  
 DWORD timeStart = timeGetTime();  
 auto temp = OMP(q);  
 cout << "Pi: " << setprecision(50) << temp.second << endl;  
 cout << temp.first<< " ms" << endl << endl;  
 DWORD timeEnd = timeGetTime();  
 cout << endl << "Time for copy: " << timeEnd - timeStart << " ms" << endl;  
 }  
  
 } while (q);  
  
 return 0;  
}