Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЁТ

к лабораторной работе №1

на тему

УПРАВЛЕНИЕ ПРОЦЕССАМИ, ПОТОКАМИ, НИТЯМИ

|  |
| --- |
| Выполнил: студент гр. 253503  Котова К.А. |
| Проверил: ассистент кафедры информатики Гриценко Н.Ю. |

МИНСК 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 5](#_Toc177985959)

[2. Описание основных фунцкций программы 6](#_Toc177985960)

[2.1 Использование пула потоков 6](#_Toc177985961)

[2.2 Мониторинг загруженности CPU 6](#_Toc177985962)

[2.3 Многопоточная сортировка и слияние массива 6](#_Toc177985963)

[3 Результаты выполнения работы программы 7](#_Toc177985964)

[3.1 Работа программы при многопоточной сортировке 7](#_Toc177985965)

[3.2 Работа программы при однопоточной сортировке 8](#_Toc177985966)

[3.3 Обработка ошибок 9](#_Toc177985967)

[Вывод 10](#_Toc177985968)

[Список использованных источников 11](#_Toc177985969)

[Приложение А 12](#_Toc177985970)

# 

# 1 ФОРМУЛИРОВКА ЗАДАЧИ

Цель данной лабораторной работы многопоточная обработка – сортировка массива.

Лабораторная работа включает в себя разбиение массива данных на несколько частей (фрагментов), обработку каждого отдельным потоком (или процессом), окончательную «сборку» результата. Количество потоков (процессов), в т.ч. единственный поток/процесс, размер массива и другие необходимые параметры задаются пользователем. Количество потоков (процессов) не слишком большое, чтобы оставалось удобным для отображения.

Вывод программы состоит из отображения прогресса выполнения (достаточно готовности/неготовности каждого фрагмента), времени выполнения для сравнения и анализа зависимости, уровня загруженности системы (использования ЦП).

# 2. ОПИСАНИЕ ОСНОВНЫХ ФУНКЦИЙ ПРОГРАММЫ

### **2.1 Использование пула потоков**

Пул потоков управляет многопоточными задачами, избегая накладных расходов на создание и завершение потоков для каждой операции. Он позволяет эффективно использовать многопоточность для выполнения задач, таких как сортировка частей массива и их слияние. Вместо создания нового потока для каждой задачи, программа использует уже существующие потоки, что сокращает время выполнения и повышает производительность.

Основные функции, которые используются для работы с пулом потоков – это CreateThreadpoolWork, которая создаёт задачу, SubmitThreadpoolWork, которая отправляет задачу на выполнение, и WaitForThreadpoolWorkCallbacks, которая ожидает завершения задачи.

### **2.2 Мониторинг загруженности CPU**

Мониторинг загрузки процессора реализован в виде отдельного класса CpuMonitor. Он позволяет отслеживать текущее состояние системы во время выполнения программы. Класс использует библиотеку PDH для сбора данных о загрузке процессора, а функция getCpuUsage собирает данные с помощью вызова PdhCollectQueryData и возвращает текущее значение загрузки процессора. Эта часть программы запускается в отдельном потоке, что позволяет выполнять мониторинг параллельно с основными вычислениями.

### **2.3 Многопоточная сортировка и слияние массива**

Программа реализует эффективную многопоточную сортировку и слияние массивов, что значительно сокращает общее время обработки данных за счет использования параллельных вычислений.

Массив делится на части, каждая из которых сортируется независимо в отдельных потоках, после чего происходит слияние отсортированных частей в единый массив. Процесс сортировки выполняется стандартной функцией std::sort. а для слияния используется алгоритм, который объединяет части в несколько этапов. Каждый этап слияния также выполняется параллельно, что ускоряет процесс обработки больших объемов данных.

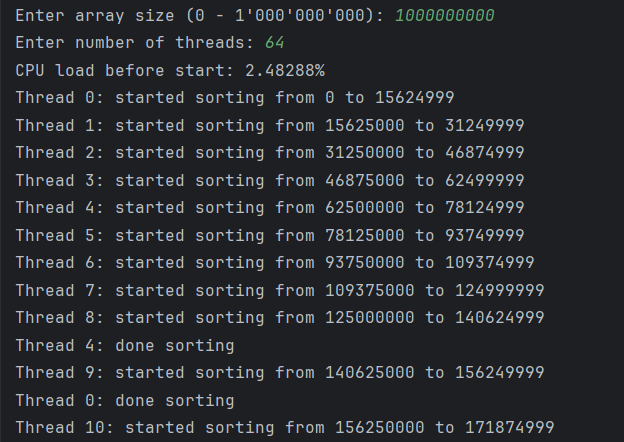
Основные функции, которые обеспечивают этот процесс – это parallelSort, выполняющая сортировку частей массива, и parallelMerge, которая отвечает за слияние отсортированных частей.

# 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ПРОГРАММЫ

## 3.1 Работа программы при многопоточной сортировке

Сортировка массива из 1'000'000'000 элементов была проведена с использованием 64 потоков. Как показано на рисунке 3.1, каждый поток отвечает за сортировку своей части массива, что обеспечивает параллельное выполнение задачи. Среднее время выполнения программы при использовании 64 потоков составило **200.45 секунд**. Во время выполнения загрузка процессора составляла **41.7842%**, что указывает на активное использование многопоточности для распределения нагрузки между ядрами процессора.

После выполнения программы загрузка процессора снизилась до **18.5609%**, что свидетельствует о завершении всех потоков. Однако при использовании большого числа потоков могут возникать накладные расходы, связанные с синхронизацией и управлением потоками, что может влиять на время выполнения и производительность.



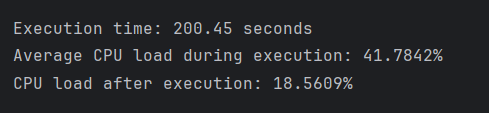


Рисунок 3.1 – Результат выполнения многопоточной сортировки

Сортировка массива из 1’000’000’000 элементов 4 потоками.

Результат выполнения показан на рисунке 3.2.

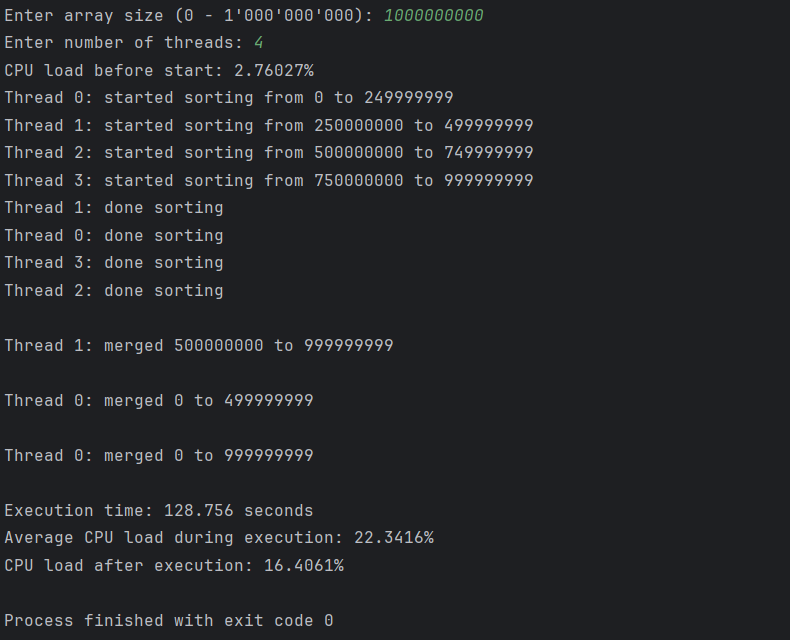


Рисунок 3.2 – Результат выполнения многопоточной сортировки

## 3.2 Работа программы при однопоточной сортировке

Сортировка массива из 1’000’000’000 элементов 1 потоком.

Результат выполнения на рисунке 3.2.

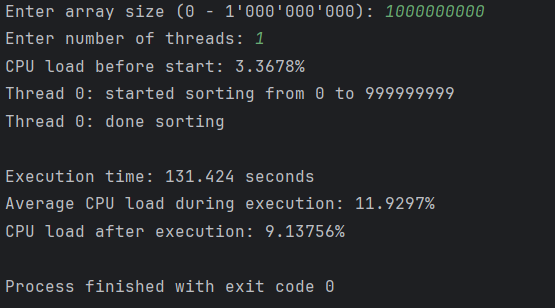


Рисунок 3.3 – Результат выполнения однопоточной сортировки

## 3.3 Обработка ошибок

Ошибки при вводе букв вместо значений или отрицательных значений. Результат выполнения показан на рисунке 3.4.

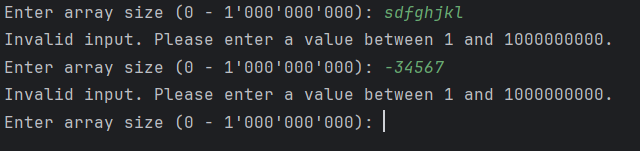


Рисунок 3.4 – Результат обработки ошибок

Ошибки при вводе значений, выходящих за заданные границы. Результат выполнения показан на рисунке 3.5.

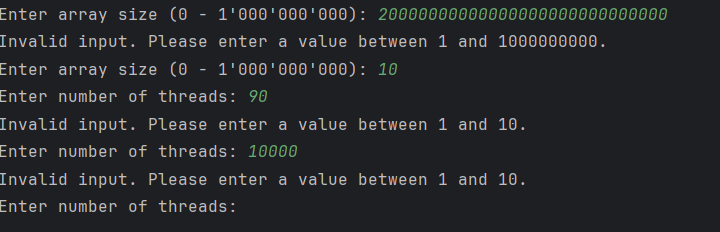


Рисунок 3.4 – Результат обработки ошибок

# ВЫВОД

В ходе выполнения лабораторной работы были освоены навыки многопоточной работы и сортировки данных. Была разработана программа, которая выполняет параллельную сортировку массива с использованием потоков. Программа принимает на вход размер массива и количество потоков, с помощью которых будет происходить сортировка. Для многопоточной сортировки массив разбивается на части, и каждая часть обрабатывается отдельным потоком.

Сначала программа использует пул потоков, что позволяет управлять потоками более эффективно. Потоки выполняют сортировку каждой части массива, а затем, после сортировки, происходит параллельное слияние отсортированных частей с использованием многопоточности.

Для синхронизации доступа к консоли используется мьютекс, который гарантирует корректный вывод информации о прогрессе каждого потока. В каждом потоке выводится информация о начале и завершении сортировки и слияния фрагментов массива.

Кроме того, программа осуществляет мониторинг загрузки процессора во время выполнения сортировки. Для этого применяется библиотека PDH (Performance Data Helper), которая собирает информацию о загрузке процессора в режиме реального времени. В ходе выполнения программа вычисляет среднюю загрузку процессора, что позволяет оценить влияние многопоточности на использование ресурсов системы.

В результате лабораторной работы можно сделать выводы, что, когда основная часть времени уходит на вычисления (CPU-bound процессы), ресурсы процессора загружены практически полностью. В таких задачах процессор активно выполняет математические операции и обрабатывает данные. Использование небольшого количества потоков приводит к тому, что задача занимает больше времени на выполнение, но нагрузка на процессор при этом остается оптимальной. Если же потоков слишком много, это создает значительную нагрузку на процессор из-за накладных расходов на управление потоками и частых контекстных переключений. В этом случае увеличение числа потоков не всегда сокращает время выполнения задачи, а иногда даже может его увеличить.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Документация Microsoft [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/api/> –Дата доступа: 16.09.2024.

[2] Работа с mutex [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/sync/using-mutex-objects> –Дата доступа: 15.09.2024.

[3] Создание потоков [Электронный ресурс]. – Электронные данные. – Режим доступа https://learn.microsoft.com/en-us/windows/win32/api/process threadsapi/nf-processthreadsapi-createthread – Дата доступа: 15.09.2024.

[4] Collecting Perfomance Data [Электронный ресурс]. – Электронные данные. – Режим доступа <https://learn.microsoft.com/en-us/windows/win32/perfctrs/collecting-performance-data> –Дата доступа: 14.09.2024.

[5] Handles and objects [Электронный ресурс]. – Электронные данные. – Режим доступа: [https://learn.microsoft.com/en-us/windows/win32/sysinfo/hand les-and-objects](https://learn.microsoft.com/en-us/windows/win32/sysinfo/hand%20les-and-objects) –Дата доступа: 15.09.2024.

# ПРИЛОЖЕНИЕ А (обязательное) Код программы

Содержимое файла sort.cpp

#include <windows.h>

#include <iostream>

#include <vector>

#include <algorithm>

#include <mutex>

#include <pdh.h>

#include <numeric>

#include <atomic>

std::mutex coutMutex;

std::atomic<bool> monitoringCpu(true);

class CpuMonitor {

public:

CpuMonitor() {

PdhOpenQuery(NULL, 0, &cpuQuery);

PdhAddEnglishCounterW(cpuQuery, L"\\Processor(\_Total)\\% Processor Time", 0, &cpuTotal);

PdhCollectQueryData(cpuQuery);

Sleep(1000);

}

~CpuMonitor() {

PdhCloseQuery(cpuQuery);

}

double getCpuUsage() {

PDH\_FMT\_COUNTERVALUE counterVal;

PdhCollectQueryData(cpuQuery);

PdhGetFormattedCounterValue(cpuTotal, PDH\_FMT\_DOUBLE, NULL, &counterVal);

return counterVal.doubleValue;

}

private:

PDH\_HQUERY cpuQuery{};

PDH\_HCOUNTER cpuTotal{};

};

DWORD WINAPI monitorCpuUsage(LPVOID param) {

auto \*cpuMonitor = static\_cast<CpuMonitor \*>(param);

std::vector<double> cpuUsageValues;

while (monitoringCpu) {

double cpuUsage = cpuMonitor->getCpuUsage();

{

std::lock\_guard<std::mutex> lock(coutMutex);

cpuUsageValues.push\_back(cpuUsage);

}

Sleep(1000);

}

double averageCpuUsage = std::accumulate(cpuUsageValues.begin(), cpuUsageValues.end(), 0.0) / cpuUsageValues.size();

std::lock\_guard<std::mutex> lock(coutMutex);

std::cout << "Average CPU load during execution: " << averageCpuUsage << "%\n";

return 0;

}

void merge(size\_t \*arr, size\_t left, size\_t mid, size\_t right) {

size\_t leftSize = mid – left + 1;

size\_t rightSize = right – mid;

std::vector<size\_t> leftArray(leftSize), rightArray(rightSize);

for (size\_t i = 0; i < leftSize; ++i) leftArray[i] = arr[left + i];

for (size\_t i = 0; i < rightSize; ++i) rightArray[i] = arr[mid + 1 + i];

size\_t i = 0, j = 0, k = left;

while (i < leftSize && j < rightSize) {

if (leftArray[i] <= rightArray[j]) arr[k++] = leftArray[i++];

else arr[k++] = rightArray[j++];

}

while (i < leftSize) arr[k++] = leftArray[i++];

while (j < rightSize) arr[k++] = rightArray[j++];

}

struct TaskData {

size\_t \*array;

size\_t left;

size\_t mid;

size\_t right;

size\_t threadId;

};

VOID CALLBACK SortCallback(PTP\_CALLBACK\_INSTANCE, PVOID param, PTP\_WORK) {

auto \*data = static\_cast<TaskData \*>(param);

{

std::lock\_guard<std::mutex> lock(coutMutex);

std::cout << "Thread " << data->threadId << ": started sorting from "

<< data->left << " to " << data->right << "\n";

}

std::sort(data->array + data->left, data->array + data->right + 1);

{

std::lock\_guard<std::mutex> lock(coutMutex);

std::cout << "Thread " << data->threadId << ": done sorting\n";

}

}

VOID CALLBACK MergeCallback(PTP\_CALLBACK\_INSTANCE, PVOID param, PTP\_WORK) {

auto \*data = static\_cast<TaskData \*>(param);

merge(data->array, data->left, data->mid, data->right);

std::lock\_guard<std::mutex> lock(coutMutex);

std::cout << "\nThread " << data->threadId << ": merged " << data->left << " to " << data->right << "\n";

}

void parallelSort(size\_t \*arr, size\_t size, size\_t numThreads, PTP\_POOL pool, PTP\_CLEANUP\_GROUP cleanupGroup,

PTP\_CALLBACK\_ENVIRON callbackEnviron) {

size\_t chunkSize = size / numThreads;

std::vector<TaskData> threadData(numThreads);

std::vector<PTP\_WORK> workItems(numThreads);

for (size\_t i = 0; i < numThreads; ++i) {

size\_t left = i \* chunkSize;

size\_t right = (i == numThreads – 1) ? size – 1 : (left + chunkSize – 1);

threadData[i] = {arr, left, 0, right, i};

workItems[i] = CreateThreadpoolWork(SortCallback, &threadData[i], callbackEnviron);

SubmitThreadpoolWork(workItems[i]);

}

for (size\_t i = 0; i < numThreads; ++i) {

WaitForThreadpoolWorkCallbacks(workItems[i], FALSE);

CloseThreadpoolWork(workItems[i]);

}

}

void parallelMerge(size\_t \*arr, size\_t size, size\_t numThreads, PTP\_POOL pool, PTP\_CLEANUP\_GROUP cleanupGroup,

PTP\_CALLBACK\_ENVIRON callbackEnviron) {

size\_t chunkSize = size / numThreads;

std::vector<TaskData> threadData(numThreads);

std::vector<PTP\_WORK> workItems(numThreads);

for (size\_t mergeSize = chunkSize; mergeSize < size; mergeSize \*= 2) {

size\_t currentThreads = numThreads / 2;

for (size\_t i = 0; i < currentThreads; ++i) {

size\_t left = i \* 2 \* mergeSize;

size\_t mid = std::min(left + mergeSize – 1, size – 1);

size\_t right = std::min(left + 2 \* mergeSize – 1, size – 1);

threadData[i] = {arr, left, mid, right, i};

workItems[i] = CreateThreadpoolWork(MergeCallback, &threadData[i], callbackEnviron);

SubmitThreadpoolWork(workItems[i]);

}

for (size\_t i = 0; i < currentThreads; ++i) {

WaitForThreadpoolWorkCallbacks(workItems[i], FALSE);

CloseThreadpoolWork(workItems[i]);

}

numThreads /= 2;

}

}

void sortAndMerge(size\_t \*arr, size\_t size, size\_t numThreads, PTP\_POOL pool, PTP\_CLEANUP\_GROUP cleanupGroup,

PTP\_CALLBACK\_ENVIRON callbackEnviron) {

if (numThreads == 1){

}

parallelSort(arr, size, numThreads, pool, cleanupGroup, callbackEnviron);

parallelMerge(arr, size, numThreads, pool, cleanupGroup, callbackEnviron);

}

void getInput(size\_t &variable, const std::string &prompt, size\_t minValue, size\_t maxValue) {

while (true) {

std::cout << prompt;

std::cin >> variable;

if (std::cin.fail() || variable < minValue || variable > maxValue) {

std::cout << "Invalid input. Please enter a value between " << minValue << " and " << maxValue << ".\n";

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

} else {

break;

}

}

}

int main() {

size\_t size = 0, numThreads = 0;

getInput(size, "Enter array size (0 – 1'000'000'000): ", 1, 1'000'000'000);

size\_t maxNumThreads = size > 64 ? 64 : size;

getInput(numThreads, "Enter number of threads: ", 1, maxNumThreads);

std::vector<size\_t> array(size);

for (size\_t j = 0; j < size; ++j) array[j] = j % 10;

CpuMonitor cpuMonitor;

std::cout << "CPU load before start: " << cpuMonitor.getCpuUsage() << "%\n";

PTP\_POOL pool = CreateThreadpool(NULL);

SetThreadpoolThreadMaximum(pool, numThreads);

SetThreadpoolThreadMinimum(pool, 0);

PTP\_CLEANUP\_GROUP cleanupGroup = CreateThreadpoolCleanupGroup();

TP\_CALLBACK\_ENVIRON callbackEnviron;

InitializeThreadpoolEnvironment(&callbackEnviron);

SetThreadpoolCallbackPool(&callbackEnviron, pool);

SetThreadpoolCallbackCleanupGroup(&callbackEnviron, cleanupGroup, NULL);

HANDLE cpuMonitorThread = CreateThread(NULL, 0, monitorCpuUsage, &cpuMonitor, 0, NULL);

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

sortAndMerge(array.data(), size, numThreads, pool, cleanupGroup, &callbackEnviron);

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

std::chrono::duration<double> duration = end – start;

std::cout << "\nExecution time: " << duration.count() << " seconds\n";

monitoringCpu = false;

WaitForSingleObject(cpuMonitorThread, INFINITE);

CloseHandle(cpuMonitorThread);

CloseThreadpoolCleanupGroupMembers(cleanupGroup, FALSE, NULL);

CloseThreadpoolCleanupGroup(cleanupGroup);

CloseThreadpool(pool);

DestroyThreadpoolEnvironment(&callbackEnviron);

// for (size\_t i = 0; i < size; ++i) {

// std::cout << array[i];

// }

std::cout << "CPU load after execution: " << cpuMonitor.getCpuUsage() << "%\n";

return 0;

}