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 №5

на тему

**УПРАВЛЕНИЕ ПОТОКАМИ, СРЕДСТВА СИНХРОНИЗАЦИИ**

Выполнил: студент гр.253505 Снежко М.А.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

[1 Формулировка задачи 3](#_Toc179742261)

[2 Краткие теоритические сведения 4](#_Toc179742262)

[3 Описание функций программы 5](#_Toc179742263)

[Заключение 6](#_Toc179742264)

[Список использованных источников 7](#_Toc179742265)

[Приложение А (обязательное) 8](#_Toc179742266)

# 1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью данной лабораторной работы является изучение подсистемы потоков (*pthread*) в операционной системе *Unix*, а также освоение основных механизмов управления потоками и средств синхронизации. Основное внимание уделяется практическому проектированию, реализации и отладке программ с параллельными взаимодействующими (конкурирующими) потоками. Работа направлена на создание многопоточной программы, которая выполняет обработку достаточно большого массива данных, например его сортировку, с использованием эффективного распараллеливания.

Количество потоков должно быть ограничено для удобства отладки и предотвращения перегрузки системы. Программа должна предоставлять сведения о времени выполнения для конкретной конфигурации, а также минимальный протокол выполнения, который позволяет отслеживать этапы работы потоков.

Реализация программы предполагает использование библиотеки *pthread* для создания и управления потоками, а также механизмов синхронизации, таких как мьютексы или условные переменные, для обеспечения корректного взаимодействия потоков. Основной алгоритм сортировки должен быть выбран таким образом, чтобы допускать эффективное распараллеливание, например, сортировка слиянием (*merge sort*).

Особое внимание уделяется созданию программы, которая корректно обрабатывает данные в многопоточной среде, обеспечивая синхронизацию доступа к общим ресурсам и минимизируя накладные расходы на управление потоками. Программа должна быть протестирована на различных конфигурациях (разные размеры массива и количество потоков) для оценки производительности и корректности работы. В ходе лабораторной работы также необходимо проанализировать влияние количества потоков на производительность программы и выявить оптимальное соотношение между размером массива и числом потоков. Это позволит лучше понять, как параллелизация влияет на время выполнения задач и какие ограничения могут возникнуть.

Таким образом, данная лабораторная работа позволяет изучить ключевые аспекты многопоточного программирования в *Unix*, включая создание потоков, управление их выполнением и синхронизацию. Полученные навыки помогут в дальнейшем разрабатывать эффективные и надежные программы для решения задач, требующих параллельной обработки данных.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

*UNIX* – многопользовательская, многозадачная операционная система (ОС), разработанная для гибкости и адаптивности. Первоначально разработанная в 1970-х годах, *Unix* была одной из первых ОС, написанных на языке программирования *C*. С момента своего появления операционная система *Unix* и ее ответвления оказали глубокое влияние на компьютерную и электронную промышленность, предлагая переносимость, стабильность и совместимость в различных гетерогенных средах и типах устройств. [1]

Семафоры являются полезным инструментом для взаимного исключения, но они являются не единственным таким инструментом, предоставляемым ядром. Вместо этого, большинство блокировок осуществляется механизмом, названным спин-блокировкой. В отличие от семафоров, спин-блокировки могут быть использованы в коде, который не может спать, таком, как обработчики прерываний. При правильном применении, спин-блокировки предлагают в целом более высокую производительность, чем семафоры. Они, однако, имеют другой набор ограничений на своё использование.

Спин-блокировки – простая концепция. Спин-блокировка является взаимным исключением устройства, которое может иметь только два значения: "заблокировано" и "разблокировано". Это обычно реализуется как один бит в целом числе. Код, желающий забрать какую-либо определённую блокировку, проверяет соответствующий бит. Если блокировка доступна, "блокирующий" бит устанавливается и этот код продолжает работу в критической секции. Вместо этого, если блокировка уже захвачена кем-то другим, код переходит в короткий цикл, где постоянно проверяет блокировку, пока она не станет доступна. Этот цикл является "спин" частью спин-блокировки. [2]

Поток выполнения – последовательность запрограммированных команд, которые могут независимо управляться планировщиком, являющимся частью операционной системы. Реализация потоков и процессов различается в операционных системах, но в большинстве случаев поток является составной частью процесса. Несколько потоков могут существовать в одном процессе, одновременно выполняться и совместно использовать ресурсы, такие как память, в то время как разные процессы не делят эти ресурсы. В частности, потоки процесса совместно используют свой исполняемый код и значения переменных в любой момент времени. [3]

Таким образом, программирование на языке *C* в *Unix* предоставляет мощные средства для создания приложений, которые могут взаимодействовать с операционной системой, обрабатывать текстовые данные и управлять ресурсами. Полученные знания и навыки могут быть использованы для разработки более сложных программ, таких как утилиты для обработки текста, системные утилиты или инструменты автоматизации.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Разработанная программа представляет собой многопоточную сортировку, которая обрабатывает достаточно большой массив данных, деля его на фрагменты и сортируя каждый фрагмент в отдельном потоке. Основная задача программы заключается в генерации случайного массива, разбиении его на части, сортировке каждой части с использованием потоков и окончательной сборке отсортированного массива.

Программа начинается с проверки аргументов командной строки. Если аргументы не переданы, программа выводит сообщение о неправильном использовании. Если указан один аргумент, он рассматривается как размер массива, а второй – как количество потоков. Программа также проверяет корректность введённых значений, чтобы размер массива и количество потоков были положительными целыми числами.

Основная обработка данных выполняется в функции, которая запускает потоки для сортировки фрагментов массива. Каждый поток получает свои параметры, включая часть массива, которую он должен отсортировать. После завершения работы всех потоков происходит слияние отсортированных фрагментов. Это обеспечивает эффективное использование ресурсов и значительное ускорение процесса сортировки по сравнению с однопоточной реализацией. Итог работы программы представляет собой отсортированный массив, а также выводит на экран время, затраченное на сортировку и слияние, что позволяет оценить производительность многопоточной обработки. Результат работы программы демонстрирует, как эффективная параллелизация может быть использована для оптимизации задач сортировки. Результат работы программы представлен на рисунке 3.1.

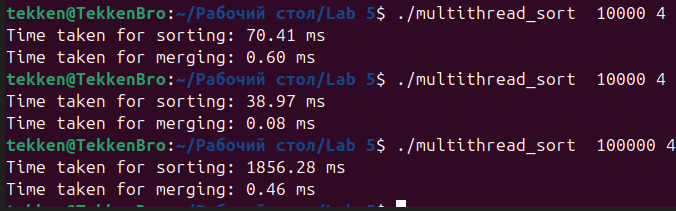


Рисунок 3.1 – Результат работы программы

Таким образом, программа иллюстрирует использование языка C для реализации многопоточных алгоритмов в среде Unix. Она позволяет изучить работу с потоками, синхронизацией и управлением памятью, а также применение функций для обработки массивов.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы была разработана многопоточная программа на языке *C*, реализующая обработку большого массива данных с применением методов параллельного программирования. Основной целью работы являлось изучение механизма управления потоками (*pthread*) в операционной системе *Unix/Linux*, а также освоение различных средств синхронизации, обеспечивающих корректное взаимодействие между потоками в многозадачной среде.

Разработанный программный модуль выполняет многопоточную сортировку массива, что позволило на практике изучить ключевые аспекты работы с потоками, включая их создание, управление и завершение. В процессе работы были реализованы три основные стадии обработки данных: разбиение исходного массива на несколько подмассивов, их сортировка в отдельных потоках и последующая финальная сборка отсортированных частей в единый массив. При этом количество потоков и размер массива задаются пользователем, что обеспечивает гибкость в выборе параметров и позволяет анализировать влияние многопоточности на производительность программы.

Реализация данной лабораторной работы позволила глубже изучить особенности работы с потоками в Unix-среде, а также принципы организации параллельных вычислений. В ходе выполнения лабораторной работы был проведён анализ времени выполнения программы в зависимости от количества потоков, что позволило оценить эффективность параллельного подхода к обработке данных. Эксперименты показали, что при разумном выборе числа потоков можно добиться значительного ускорения выполнения задачи, однако чрезмерное увеличение их количества может привести к перегрузке системы и ухудшению производительности из-за накладных расходов на переключение контекста.

Кроме того, при разработке программы были рассмотрены вопросы синхронизации потоков и управления критическими секциями для предотвращения возможных конфликтов при доступе к общим ресурсам. Были изучены и применены различные механизмы синхронизации, включая мьютексы и барьеры, что позволило обеспечить корректное взаимодействие потоков при объединении отсортированных частей массива.

Таким образом, выполнение данной лабораторной работы позволило не только изучить основы многопоточного программирования и средства синхронизации в *Unix/Linux*, но и углубить понимание работы операционной системы при выполнении параллельных вычислений. Полученные знания и навыки могут быть полезны при разработке более сложных многопоточных приложений, работающих в среде *Unix*, а также при решении задач, связанных с обработкой больших объёмов данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Thecode "Обзор UNIX" – Электронный ресурс. Режим доступа: https://thecode.media/unix.

[2] Dmilvdv "Спин-блокировки" – Электронный ресурс. Режим доступа: https://dmilvdv.narod.ru/Translate/LDD3/ldd\_spinlocks.html.

[3] Kvckr "Вычислительные потоки UNIX" – Электронный ресурс. Режим доступа: https://kvckr.me/mag/sp/9.html.

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <time.h>

#include <sys/time.h>

#include "utils.h"

int main(int argc, char \*argv[]) {

if (argc != 3) {

printf("Usage: %s <array\_size> <num\_threads>\n", argv[0]);

return 1;

}

int array\_size = atoi(argv[1]);

int num\_threads = atoi(argv[2]);

if (array\_size <= 0 || num\_threads <= 0) {

printf("Array size and number of threads must be positive integers.\n");

return 1;

}

// Генерация случайного массива

int \*array = (int \*)malloc(array\_size \* sizeof(int));

srand(time(NULL));

for (int i = 0; i < array\_size; i++) {

array[i] = rand() % 1000; // числа от 0 до 999

}

// printf("Original Array:\n");

// for (int i = 0; i < array\_size; i++) {

// printf("%d ", array[i]);

// }

// printf("\n");

// Разбиение массива на части

int chunk\_size = (array\_size + num\_threads - 1) / num\_threads;

pthread\_t threads[num\_threads];

ThreadData thread\_data[num\_threads];

struct timeval start\_time, end\_time;

// Запуск сортировки в потоках

gettimeofday(&start\_time, NULL);

for (int i = 0; i < num\_threads; i++) {

thread\_data[i].array = array;

thread\_data[i].start = i \* chunk\_size;

thread\_data[i].end = (i + 1) \* chunk\_size;

if (thread\_data[i].end > array\_size) {

thread\_data[i].end = array\_size;

}

pthread\_create(&threads[i], NULL, sort\_thread, &thread\_data[i]);

}

for (int i = 0; i < num\_threads; i++) {

pthread\_join(threads[i], NULL);

}

gettimeofday(&end\_time, NULL);

double sort\_time = (end\_time.tv\_sec - start\_time.tv\_sec) \* 1000.0 +

(end\_time.tv\_usec - start\_time.tv\_usec) / 1000.0;

// Слияние частей

gettimeofday(&start\_time, NULL);

for (int size = chunk\_size; size < array\_size; size \*= 2) {

for (int i = 0; i < array\_size; i += 2 \* size) {

int start1 = i;

int end1 = i + size;

int start2 = end1;

int end2 = start2 + size;

if (end1 > array\_size) end1 = array\_size;

if (end2 > array\_size) end2 = array\_size;

merge(array, start1, end1, start2, end2);

}

}

gettimeofday(&end\_time, NULL);

double merge\_time = (end\_time.tv\_sec - start\_time.tv\_sec) \* 1000.0 +

(end\_time.tv\_usec - start\_time.tv\_usec) / 1000.0;

// printf("Sorted Array:\n");

// for (int i = 0; i < array\_size; i++) {

// printf("%d ", array[i]);

// }

// printf("\n");

printf("Time taken for sorting: %.2f ms\n", sort\_time);

printf("Time taken for merging: %.2f ms\n", merge\_time);

free(array);

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

typedef struct {

int \*array;

int start;

int end;

} ThreadData;

void \*sort\_thread(void \*arg) {

ThreadData \*data = (ThreadData \*)arg;

int \*arr = data->array;

int start = data->start;

int end = data->end;

// Cортировка

for (int i = start; i < end - 1; i++) {

for (int j = start; j < end - 1 - (i - start); j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

pthread\_exit(NULL);

}

void merge(int \*array, int start1, int end1, int start2, int end2) {

int size1 = end1 - start1;

int size2 = end2 - start2;

int \*temp = (int \*)malloc((size1 + size2) \* sizeof(int));

int i = start1, j = start2, k = 0;

while (i < end1 && j < end2) {

if (array[i] < array[j]) {

temp[k++] = array[i++];

} else {

temp[k++] = array[j++];

}

}

while (i < end1) temp[k++] = array[i++];

while (j < end2) temp[k++] = array[j++];

for (i = 0; i < size1 + size2; i++) {

array[start1 + i] = temp[i];

}

free(temp);

}