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

**ОТЧЕТ**

к лабораторной работе № 5

на тему

«Управление потоками, средства синхронизации»

Выполнил:

студент гр. 153504

Сивый А. А.

Проверил:

Гриценко Н. Ю.

Минск 2024

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc146631498)

[2 Краткие теоретические сведения 4](#_Toc146631499)

[3 Полученные результаты 5](#_Toc146631500)

[Выводы 6](#_Toc146631501)

[Список использованных источников 7](#_Toc146631502)

[Приложение А (обязательное) Листинг исходного кода 8](#_Toc146631503)

## 1 ЦЕЛЬ РАБОТЫ

Изучение подсистемы потоков (pthread), основных особенностей функционирования и управления, средств взаимодействия потоков. Практическое проектирование, реализация и отладка программ с параллельными взаимодействующими (конкурирующими) потоками.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Unix обеспечивает механизм управления потоками с использованием библиотеки POSIX Threads (pthread), которая предоставляет API для создания, управления и синхронизации потоков. Потоки позволяют создавать параллельные процессы в рамках одного процесса, что повышает эффективность использование ресурсов многопроцессорных систем [1].

Для создания нового потока используется функция pthread\_create, которая принимает указатель на переменную, в которую будет сохранен идентификатор потока, указатель на функцию, которая будет являться точкой входа для потока, и аргументы для этой функции. Новый поток может начать выполняться немедленно после создания или ожидать своей очереди на выполнение.

Функция pthread\_join используется для ожидания завершения выполнения потока. Она блокирует вызывающий поток до тех пор, пока целевой поток не завершится, и возвращает код завершения потока. Это позволяет контролировать порядок выполнения потоков и собирать результаты их работы.

Поток может быть завершен с помощью функции pthread\_exit, которая принимает указатель на значение, которое будет возвращено в функции pthread\_join. Это позволяет потоку сообщить о своем завершении и передать результат своей работы другим потокам.

Для обеспечения корректной работы параллельных потоков необходимо использовать средства синхронизации, которые позволяют управлять доступом к общим ресурсам и согласовывать выполнение потоков.

Основные средства синхронизации в POSIX Threads:

1 Мьютексы (mutex). Обеспечивают эксклюзивный доступ к общему ресурсу. Поток, захвативший мьютекс, становится его владельцем и имеет эксклюзивное право на доступ к ресурсу.

2 Условные переменные. Позволяют потокам ожидать определенного условия, прежде чем продолжать выполнение.

3 Семафоры (semaphore). Позволяют управлять доступом к общему ресурсу, а также ограничивать количество потоков, имеющих доступ к ресурсу.

4 Барьеры. Позволяют синхронизировать группу потоков, ожидая, пока все потоки достигнут определенной точки в программе.

Управление потоками в Unix с помощью POSIX Threads дает возможность эффективно использовать ресурсы многопроцессорных систем и создавать многопоточные программы с разделением нагрузки и параллельным выполнением задач. Однако использование потоков требует осторожности из-за потенциальных проблем с синхронизацией и доступом к общим ресурсам.

## 3 ПОЛУЧЕННЫЕ РЕЗУЛЬТАТЫ

В результате выполнения лабораторной работы была разработана программа, которая генерирует случайным образом массив из целочисленных значений, после чего сортирует его и выводит результат на экран. Результат работы программы представлен на рисунке 3.1.

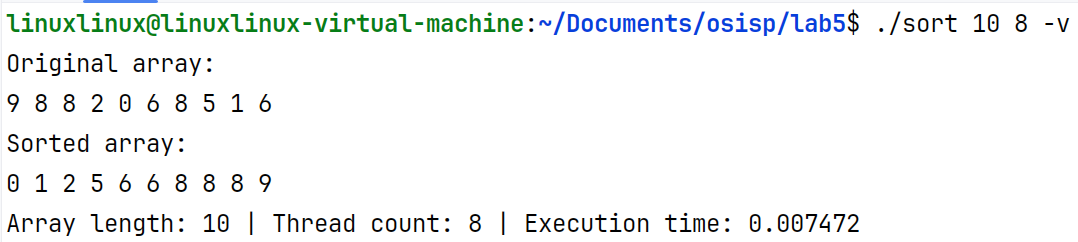


Рисунок 3.1 – Результат выполнения программы

Также программа выводит на экран информацию, такую как длину массива, количество потоков и время выполнения сортировки.

Также в программе предусмотрена возможность скрыть расширенный вывод программы с помощью флага -v или –verbose. В результате на экран не будут выведены массивы, что удобно при больших размерах исходных данных.

## ВЫВОДЫ

В результате выполнения лабораторной работы по управлению потоками и средствам синхронизации в Unix был разработан программный продукт, который позволяет выполнять сортировку параллельно несколькими потоками. Это позволило эффективнее использовать ресурсы системы и ускорить сортировку за счет параллельного выполнения. Данный подход лучше всего подходит для обработки больших объемов данных.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Брайан Керниган, Роб Пайк. UNIX — универсальная среда программирования. – М.: Изд. Дом Вильямс, 1992. – 230 с.

[2] Руководство по Си [Электронный ресурс] – Режим доступа: <https://metanit.com/c/tutorial/> – Дата доступа: 01.03.2024

**ПРИЛОЖЕНИЕ А**

## (обязательное)

## Листинг исходного кода

Листинг 1 – Программный код разработанного приложения

#include "utils.c"

#include "merge\_sort.c"

#include <time.h>

#include <string.h>

#define MAX\_THREAD\_COUNT 8

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

if (argc < 3) {

fprintf(stderr, "Using: %s <array\_length> <thread\_count> [<-v/--verbose>]\n", argv[0]);

exit(1);

}

if (!is\_number(argv[1])) {

fprintf(stderr, "Array length is not a number\n");

exit(1);

}

if (!is\_number(argv[2])) {

fprintf(stderr, "Thread count is not a number\n");

exit(1);

}

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

int thread\_count = atoi(argv[2]);

int\* array = generate\_array(array\_length, 0, array\_length);

if (thread\_count > MAX\_THREAD\_COUNT || thread\_count < 0) {

fprintf(stderr, "Thread count must be betwen 0 and %d\n", MAX\_THREAD\_COUNT);

exit(1);

}

if (argc >= 4 && (!strcmp(argv[3], "-v") || !strcmp(argv[3], "--verbose"))) {

printf("Original array:\n");

print\_array(array, array\_length);

}

clock\_t start = clock();

parallel\_sort(array, array\_length, thread\_count);

clock\_t end = clock();

if (argc >= 4 && (!strcmp(argv[3], "-v") || !strcmp(argv[3], "--verbose"))) {

printf("Sorted array:\n");

print\_array(array, array\_length);

}

double execution\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Array length: %d | Thread count: %d | Execution time: %f\n", array\_length, thread\_count, execution\_time);

free(array);

exit(0);

}

typedef struct {

int \*ptr;

int length;

} ArrayPart;

static int compare(const void \*a, const void \*b) {

return (\*(int\*)a - \*(int\*)b);

}

static void sort(void\* array\_part) {

ArrayPart \*part = (ArrayPart\*)array\_part;

qsort(part->ptr, part->length, sizeof(int), compare);

pthread\_exit(NULL);

}

static void merge(ArrayPart parts[], int thread\_count) {

int\* left = parts[0].ptr;

int leftLength = parts[0].length;

for (int p = 1; p < thread\_count; p++) {

int\* right = parts[p].ptr;

int rightLength = parts[p].length;

int\* buffer = (int\*)malloc((leftLength + rightLength) \* sizeof(int));

int left\_index = 0, right\_index = 0, buffer\_index = 0;

while (left\_index < leftLength && right\_index < rightLength) {

if (left[left\_index] <= right[right\_index]) {

buffer[buffer\_index] = left[left\_index++];

} else {

buffer[buffer\_index] = right[right\_index++];

}

buffer\_index++;

}

while (left\_index < leftLength) {

buffer[buffer\_index++] = left[left\_index++];

}

while (right\_index < rightLength) {

buffer[buffer\_index++] = right[right\_index++];

}

leftLength += rightLength;

int \*array = left;

for (int i = 0; i < leftLength; i++) {

array[i] = buffer[i];

}

free(buffer);

}

}

static void parallel\_sort(int\* array, int array\_length, int thread\_count) {

pthread\_t threads[thread\_count];

ArrayPart parts[thread\_count];

int part\_size = array\_length / thread\_count;

for (int i = 0; i < thread\_count; i++) {

parts[i].ptr = array + i \* part\_size;

parts[i].length = (i == thread\_count - 1) ? (array\_length - i \* part\_size) : part\_size;

pthread\_create(&threads[i], NULL, (void \*(\*)(void \*)) sort, &parts[i]);

}

for (int i = 0; i < thread\_count; i++) {

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

}

merge(parts, thread\_count);

}

static int\* generate\_array(int length, int min, int max) {

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

if (!array) {

exit(1);

}

srand(time(NULL));

for (int i = 0; i < length; i++) {

array[i] = min + rand() % (max - min + 1);

}

return array;

}

static void print\_array(int\* array, int array\_length) {

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

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

}

printf("\n");

}

static int is\_number(const char\* str) {

int len = strlen(str);

int i = 0;

if (\*str == '-')

i++;

while (i < len) {

if (!isdigit(\*(str + i++)))

return 0;

}

return 1;

}