

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, СРЕДСТВАМИ СИНХРОНИЗАЦИИ

Выполнил: студент гр. 253503
Тимошевич К.С.

Проверил: ассистент кафедры
информатики Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

1 Постановка задачи.....	3
2 Описание работы программы.....	4
2.1 Создание потоков.....	4
2.2 Сортировка частей массива.....	4
2.3 Объединение отсортированных частей.....	4
3 Ход выполнения программы.....	5
3.1 Примеры выполнения задания.....	5
Вывод.....	6
Список использованных источников.....	7
Приложение А (справочное) Исходный код.....	8

1 ПОСТАНОВКА ЗАДАЧИ

Целью данной лабораторной работы является изучение механизмов управления потоками и средств их синхронизации в операционных системах семейства *Unix/Linux*. В рамках работы необходимо разработать многопоточное приложение, реализующее обработку массива данных, например, его сортировку. Реализация должна демонстрировать эффективное распараллеливание вычислений и использование механизмов взаимодействия между потоками.

Программа должна быть реализована с использованием библиотеки *POSIX Threads (pthread)* и включать основные аспекты многопоточного программирования: создание потоков с помощью *pthread_create()*; завершение потоков (*pthread_exit()* и *pthread_cancel()*); использование синхронизации, включая мьютексы (*pthread_mutex*), барьеры (*pthread_barrier*) и спин-блокировки (*pthread_spinlock*) [1].

В качестве примера многопоточной обработки предлагается реализация сортировки массива, включающая следующие этапы: разделение массива на несколько частей, сортировка каждой части в отдельном потоке, объединение отсортированных частей в один массив.

Пользователь должен иметь возможность задавать размер массива и количество потоков. Программа должна учитывать ограничение на избыточное количество потоков, чтобы избежать перегрузки системы.

2 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

Данный раздел описывает работу многопоточной программы, выполняющей сортировку массива методом распараллеливания. Программа разделяет массив на несколько частей, каждая из которых обрабатывается отдельным потоком. После завершения сортировки частей выполняется их последовательное объединение. В разделе рассматриваются основные этапы работы программы, включая создание потоков, сортировку фрагментов массива и слияние результатов.

2.1 Создание потоков

Программа начинает свою работу с запроса у пользователя размера массива и количества потоков. После этого выделяется память для массива, который заполняется случайными числами. Для сортировки массив делится на равные части, количество которых соответствует числу потоков.

Создание потоков осуществляется с помощью *pthread_create()*, каждому потоку передается свой участок массива, который он должен отсортировать. Потоки выполняются независимо друг от друга, что позволяет эффективно использовать вычислительные ресурсы.

После завершения сортировки каждый поток завершает свою работу с помощью *pthread_exit()*, а основной процесс ожидает их завершения, используя *pthread_join()* [2].

2.2 Сортировка частей массива

Каждый поток выполняет сортировку своего фрагмента массива с использованием стандартной функции *qsort()* [3]. Данная функция применяет алгоритм быстрой сортировки, обеспечивая эффективную обработку данных. По завершении сортировки поток выводит сообщение о выполненной работе.

Благодаря разбиению массива на части и параллельной обработке, общая производительность программы увеличивается по сравнению с однопоточной сортировкой, особенно на многоядерных системах.

2.3 Объединение отсортированных частей

После завершения работы всех потоков основной процесс выполняет последовательное объединение отсортированных фрагментов массива. Объединение выполняется поочередно с помощью функции *merge()*, которая реализует алгоритм слияния двух отсортированных массивов.

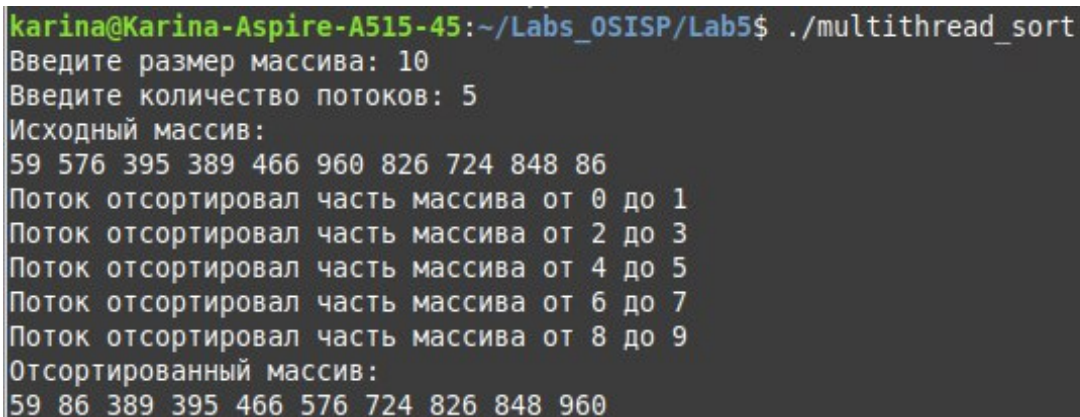
Функция *merge()* сравнивает элементы двух соседних фрагментов, записывая их в новый временный массив в отсортированном порядке. После завершения объединения временный массив копируется обратно в основной массив. Процесс повторяется до полной сортировки массива.

3 ХОД ВЫПОЛНЕНИЯ ПРОГРАММЫ

3.1 Примеры выполнения задания

На рисунке 3.1 продемонстрирована работа программы при размере массива 10 и использовании 5 потоков. Программа запрашивает у пользователя размер массива и количество потоков, после чего генерирует массив случайных чисел и выводит его на экран. Затем происходит разбиение массива на 5 частей, каждая из которых сортируется в отдельном потоке.

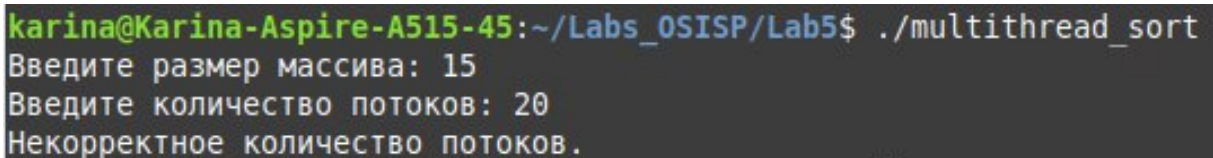
После завершения сортировки программа объединяет отсортированные фрагменты, выполняя последовательное слияние. В конечном итоге на экран выводится отсортированный массив, что подтверждает корректность выполнения алгоритма.



```
karina@Karina-Aspire-A515-45:~/Labs_OSISP/Lab5$ ./multithread_sort
Введите размер массива: 10
Введите количество потоков: 5
Исходный массив:
59 576 395 389 466 960 826 724 848 86
Поток отсортировал часть массива от 0 до 1
Поток отсортировал часть массива от 2 до 3
Поток отсортировал часть массива от 4 до 5
Поток отсортировал часть массива от 6 до 7
Поток отсортировал часть массива от 8 до 9
Отсортированный массив:
59 86 389 395 466 576 724 826 848 960
```

Рисунок 3.1 – Результаты работы

На рисунке 3.2 показан случай, когда пользователь вводит размер массива 15 и количество потоков 20. В этом случае программа выводит сообщение о некорректности ввода, так как число потоков не может превышать количество элементов массива. Этот пример демонстрирует механизм проверки входных данных и предотвращает запуск некорректных конфигураций.



```
karina@Karina-Aspire-A515-45:~/Labs_OSISP/Lab5$ ./multithread_sort
Введите размер массива: 15
Введите количество потоков: 20
Некорректное количество потоков.
```

Рисунок 3.2 – Результаты работы при некорректных входных данных

ВЫВОД

В ходе выполнения лабораторной работы была разработана и реализована программа на языке C, демонстрирующая механизм самовосстанавливающегося процесса. Программа корректно обрабатывает сигналы, которые обычно приводят к завершению процесса (*SIGINT* и *SIGTERM*), и создает свою копию для продолжения выполнения с прерванного места. Это позволяет избежать полного завершения программы при получении критических сигналов, что особенно полезно в системах, требующих высокой отказоустойчивости и непрерывности выполнения задач.

В процессе работы были изучены и применены основные механизмы управления процессами в *Unix/Linux*, такие как создание процессов с помощью *fork*, обработка сигналов с использованием *signal*, а также управление временем выполнения программы с помощью функций *sleep* и *time*. Программа успешно демонстрирует возможность восстановления после получения сигналов завершения, что подтверждается корректным увеличением счетчика и выводом соответствующих сообщений на экран.

Для наглядности работы программы был реализован периодический вывод значения счетчика, который увеличивается каждую секунду. Это позволяет визуально наблюдать за работой процесса и убедиться в его способности восстанавливаться после получения сигналов. Программа также отслеживает время своего выполнения и завершает работу через 10 секунд, что упрощает тестирование и демонстрацию ее функциональности.

В результате выполнения лабораторной работы был получен практический опыт работы с процессами в *Unix*-системах, включая создание дочерних процессов, обработку сигналов и управление временем выполнения. Программа успешно демонстрирует механизм самовосстановления, что может быть полезно при разработке отказоустойчивых систем, требующих непрерывного выполнения задач даже в условиях внешних воздействий, таких как сигналы завершения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] POSIX Threads libraries [Электронный ресурс]. – Режим доступа: <https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>. – Дата доступа: 05.03.2024.

[2] The pthread_join() function [Электронный ресурс]. – Режим доступа: https://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_join.html. – Дата доступа: 05.03.2024.

[3] qsort() Function in C [Электронный ресурс]. – Режим доступа: <https://www.geeksforgeeks.org/qsort-function-in-c/>. – Дата доступа: 01.03.2024.

ПРИЛОЖЕНИЕ А

(справочное)

Исходный код

```
# main.c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

typedef struct {
    int* array;
    int start;
    int end;
} ThreadData;

int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

void* thread_sort(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    qsort(data->array + data->start, data->end - data->start, sizeof(int),
    compare);
    printf("Поток отсортировал часть массива от %d до %d\n", data->start,
    data->end - 1);
    pthread_exit(NULL);
}

void merge(int* array, int start1, int end1, int start2, int end2) {
    int i = start1, j = start2, k = 0;
    int* temp = (int*)malloc((end1 - start1 + end2 - start2) * sizeof(int));

    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 = start1, j = 0; i < end2; i++, j++) {
        array[i] = temp[j];
    }

    free(temp);
}

int main() {
    int n, num_threads;
    printf("Введите размер массива: ");
    scanf("%d", &n);
    printf("Введите количество потоков: ");
```



```

scanf("%d", &num_threads);

if (num_threads <= 0 || num_threads > n) {
    printf("Некорректное количество потоков.\n");
    return 1;
}

int* array = (int*)malloc(n * sizeof(int));
srand(time(NULL));
for (int i = 0; i < n; i++) {
    array[i] = rand() % 1000;
}

printf("Исходный массив:\n");
for (int i = 0; i < n; i++) {
    printf("%d ", array[i]);
}
printf("\n");

pthread_t* threads = (pthread_t*)malloc(num_threads * sizeof(pthread_t));
ThreadData* thread_data = (ThreadData*)malloc(num_threads *
sizeof(ThreadData));

int part_size = n / num_threads;
for (int i = 0; i < num_threads; i++) {
    thread_data[i].array = array;
    thread_data[i].start = i * part_size;
    thread_data[i].end = (i == num_threads - 1) ? n : (i + 1) * part_size;
    pthread_create(&threads[i], NULL, thread_sort, &thread_data[i]);
}

for (int i = 0; i < num_threads; i++) {
    pthread_join(threads[i], NULL);
}

for (int i = 1; i < num_threads; i++) {
    merge(array, 0, i * part_size, i * part_size, (i + 1) * part_size);
}

printf("Отсортированный массив:\n");
for (int i = 0; i < n; i++) {
    printf("%d ", array[i]);
}
printf("\n");

free(array);
free(threads);
free(thread_data);

return 0;
}

```