

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

Н. В. Климкович

Проверил

Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

1 Постановка задачи.....	3
2 Теоретические сведения	4
3 Результат выполнения	5
Выводы	6
Список использованных источников	7
Приложение А (обязательное) Листинг кода.....	8

1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения данной лабораторной работы является изучение подсистемы потоков pthread, основных особенностей функционирования и управления, средств взаимодействия потоков. Также практическое проектирование, реализация и отладка программ с параллельными взаимодействующими (конкурирующими) потоками.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В Unix-системах, особенно при разработке на языке программирования C, управление и координация работы параллельных потоков требует использования механизмов синхронизации.

В Unix потоки представлены в виде легковесных процессов (LWP) или просто потоков, каждый из которых имеет свой собственный стек выполнения, но разделяет другие ресурсы процесса, такие как адресное пространство и файловые дескрипторы.[1]

Создание потоков в Unix выполняется с использованием функции `pthread_create`, которая принимает указатель на функцию, которая будет выполнена в потоке, и ее аргументы.

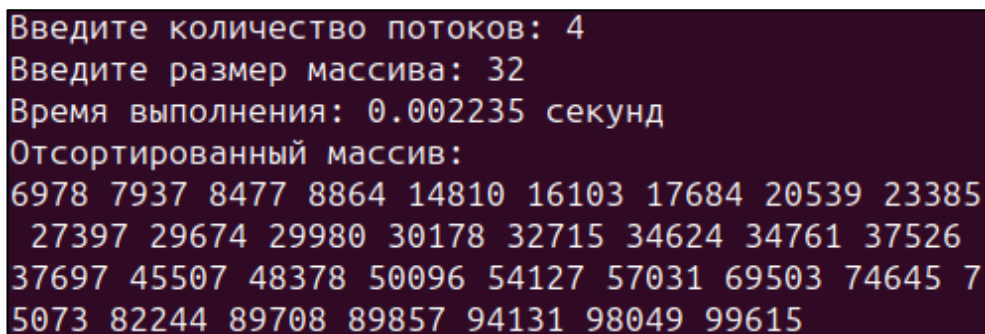
Для синхронизации и координации работы потоков в Unix используются различные механизмы, такие как мьютексы, семафоры, условные переменные и барьеры. Мьютексы используются для защиты общих ресурсов от одновременного доступа нескольких потоков.[2] Семафоры контролируют доступ к общим ресурсам и обеспечивают синхронизацию потоков. Условные переменные позволяют потокам ожидать определенного условия перед продолжением выполнения. Барьеры используются для синхронизации выполнения нескольких потоков, ожидающих друг друга перед продолжением выполнения.

Для завершения потока используется функция `pthread_exit`, а для ожидания его завершения и получения возвращаемого значения функция `pthread_join`. Для отмены выполнения потока используется функция `pthread_cancel`. [3]

При разработке многопоточных приложений важно правильно управлять ресурсами и избегать гонок данных и взаимоблокировок. Необходимо быть осторожным при использовании блокировок, чтобы избежать блокировки всей программы и обеспечить эффективность работы приложения.

3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ

В ходе выполнения лабораторной работы была разработана многопоточная программа, предназначенная для обработки обширных массивов данных, включая их сортировку. Пользователю предоставляется возможность указать количество потоков и размер массива. Результатом работы программы являются данные о времени выполнения для определенной конфигурации и отсортированный массив. Результат работы программы представлен на рисунке 3.1.



```
Введите количество потоков: 4
Введите размер массива: 32
Время выполнения: 0.002235 секунд
Отсортированный массив:
6978 7937 8477 8864 14810 16103 17684 20539 23385
 27397 29674 29980 30178 32715 34624 34761 37526
37697 45507 48378 50096 54127 57031 69503 74645 7
5073 82244 89708 89857 94131 98049 99615
```

Рисунок 3.1 – Результат работы программы

Программа выводит отсортированный массив, который заполняется случайными числами в диапазоне от 0 до 100000, время, затраченное на выполнение программы, то есть на сортировку случайного массива чисел, а также остаются входные данные, которые ввёл пользователь для работы программы.

ВЫВОДЫ

В ходе выполнения данной лабораторной работы мы изучили основные принципы управления потоками и методы взаимодействия между ними в операционной системе Unix, используя язык программирования C и библиотеку pthread.

Разработанная в процессе работы программа представляет собой многопоточную сортировку массива данных. Она осуществляет разбиение исходного массива на фрагменты и сортирует каждый фрагмент с помощью отдельного потока. Затем отсортированные фрагменты собираются в один отсортированный массив. Программа предоставляет пользователю возможность задать размер массива и количество потоков для сортировки, что обеспечивает гибкость и настройку процесса.

Одним из важных аспектов работы было измерение времени выполнения сортировки для различных конфигураций (размера массива и количества потоков) с целью оценки эффективности реализованного алгоритма.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Welcome to Unix [Электронный ресурс]. – Режим доступа: <https://unix.com>. – Дата доступа: 01.04.2024.

[2] Makefile tutorial how to write [Электронный ресурс]. – Режим доступа: <https://www.tutorialspoint.com/makefile>. – Дата доступа: 03.04.2024.

[3] Threads C [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/3.1.php>. – Дата доступа: 02.04.2024.

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг кода

Листинг 1 – Основной файл программы main.c

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

typedef struct {
    int *array;
    int size;
} ThreadData;

void bubble_sort(int *array, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}

void* thread_sort(void *arg) {
    ThreadData *data = (ThreadData*)arg;
    bubble_sort(data->array, data->size);
    pthread_exit(NULL);
}

void merge(int *array, int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = array[l + i];
    for (j = 0; j < n2; j++)
        R[j] = array[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            array[k] = L[i];
            i++;
        } else {
            array[k] = R[j];
            j++;
        }
        k++;
    }
}
```



```

while (i < n1) {
    array[k] = L[i];
    i++;
    k++;
}

while (j < n2) {
    array[k] = R[j];
    j++;
    k++;
}
}

int main() {
    int num_threads;
    int array_size;

    printf("Введите количество потоков: ");
    scanf("%d", &num_threads);

    if (num_threads <= 0) {
        printf("Неверное количество потоков\n");
        return 1;
    }

    printf("Введите размер массива: ");
    scanf("%d", &array_size);

    if (array_size <= 0) {
        printf("Неверный размер массива\n");
        return 1;
    }

    int *array = (int*)malloc(array_size * sizeof(int));

    srand(time(NULL));
    for (int i = 0; i < array_size; i++) {
        array[i] = rand() % 100000;
    }

    pthread_t threads[num_threads];
    ThreadData thread_data[num_threads];

    int chunk_size = array_size / num_threads;

    clock_t start_time = clock();

    for (int i = 0; i < num_threads; i++) {
        thread_data[i].array = &array[i * chunk_size];
        thread_data[i].size = (i == num_threads - 1) ? (array_size - i *
chunk_size) : chunk_size;
        pthread_create(&threads[i], NULL, thread_sort,
(void*)&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 * chunk_size - 1, (i + 1) * chunk_size - 1);
    }
}

```

```

    clock_t end_time = clock();
    double execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Время выполнения: %.6f секунд\n", execution_time);

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

    free(array);

    return 0;
}

```

Листинг 2 – Программная реализация makefile файла

```

CC = gcc
CFLAGS = -Wall

TARGET = lab4

all: $(TARGET)

$(TARGET): main.c
    $(CC) $(CFLAGS) -o $(TARGET) main.c

clean:
    rm -f $(TARGET)

```