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 К. А. Тимофеев

Проверил Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения данной лабораторной работы является изучение подсистемы потоков, основных особенностей функционирования и управления, средств взаимодействия потоков. Кроме этого, необходимо реализовать программу на языке программирования C, которая будет реализовывать параллельную обработку блока данных различными потоками с использованием семафоров.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Поток выполнения – наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса и его значения переменных, которые они имеют в любой момент времени.[1]

В качестве потоков в системе Unix исполняются «облегченные» процессы. Эти облегченные процессы делят общие ресурсы с другими потоками в рамках одного процесса.[2]

Каждый поток имеет свой уникальный целочисленный идентификатор, который называется TID. TID используется для идентификации конкретного потока в системе. Для главного потока процесса значения PID и TID совпадают, то есть фактически в качестве PID процесса выступает TID главного потока. Для всех остальных потоков процесса PID такое же, как и для главного потока, а значение TID уже индивидуально.

В многопоточных приложениях доступны традиционные средства синхронизации и обмена данными, такие как каналы, сокеты, семафоры, мьютексы, разделяемая память.[3]

Для выполнения данной лабораторной работы были использованы следующие сведения и концепции:

1 Разделяемая память: для работы с разделяемой памятью была использована структура данных, к которой обращались процессы. В коде программы отсутствует явная работа с разделяемой память. Программа использует многопоточность для обработки блоков данных различными потоками, а семафоры используются для обеспечения синхронизации доступа к общим данным.

2 Семафоры: для контроля доступа к разделяемой памяти каждого процесса были использованы семафоры, а также функции malloc для выделения определенного размера памяти под семафор, sem_init для инициализации семафора, sem_wait для уменьшения значения семафора на единицу, sem_post для увеличения значения семафора на единицу, sem_destroy для уничтожения семафора, free для освобождения выделенной памяти под семафор.

3 Управление потоками: при помощи функции pthread_create создавался новый поток. При помощи функции pthread_join блокировалось выполнение главного потока до тех пор, пока не завершатся все созданные раннее потоки. При помощи функции pthread_exit завершается выполнение потоков и освобождаются ресурсы, связанные с ними.

3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе лабораторной работы была реализована программа, в которой реализуется создание нескольких потоков, которые параллельно сортируют массив чисел 3.1.

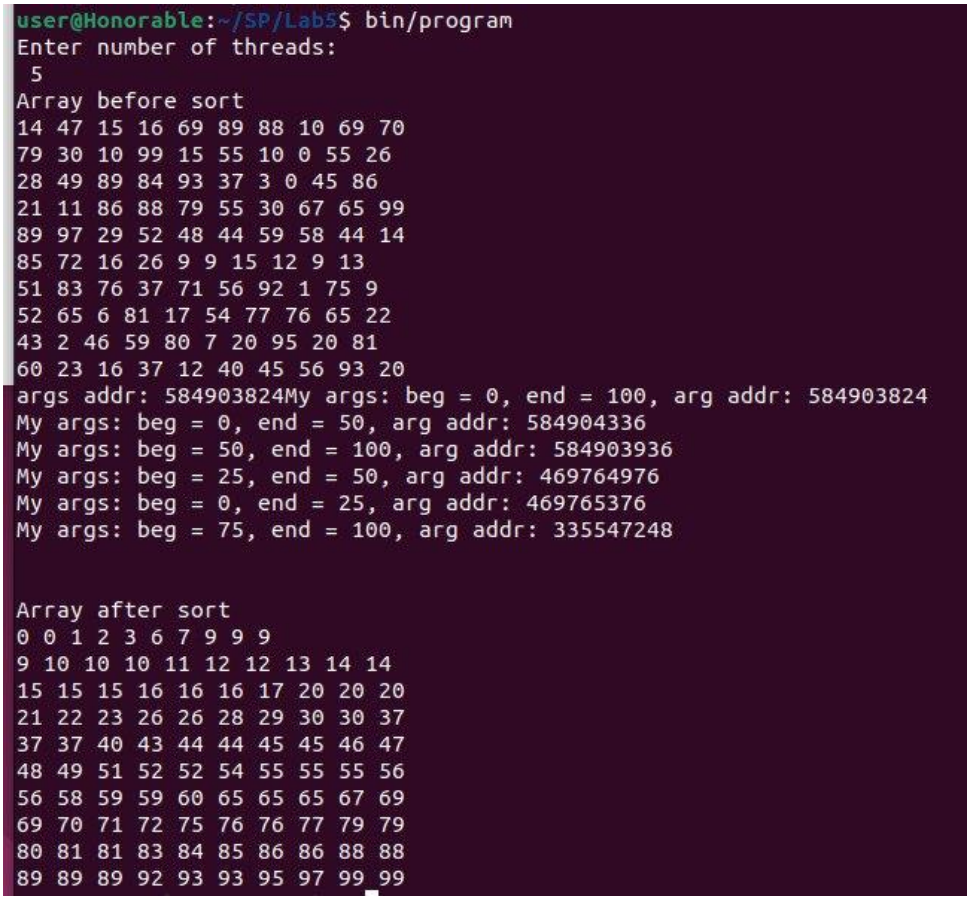

Рисунок 3.1 – Результат работы программы

Таким образом, в ходе лабораторной работы была реализована программа, реализующая сортировку блока данных несколькими потоками.

ВЫВОДЫ

В ходе лабораторной работы были изучены подсистема потоков, основные особенности функционирования и управления, средства взаимодействия потоков. Кроме этого, была реализована программа на языке программирования C, которая параллельно обрабатывает блок данных различными потоками, которые заполняют данный блок данных случайными числами, после чего родительский поток подсчитывает сумму этих значений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Процессы и потоки [Электронный ресурс]. – Режим доступа: https://acm.bsu.by/wiki/Unix2019b/. – Дата доступа: 11.02.2024.

[2] Архитектура Unix. Процессы [Электронный ресурс]. – Режим доступа: https://acm.bsu.by/wiki/Unix2018/. – Дата доступа: 14.02.2024.

[3] Разделяемая память и семафоры [Электронный ресурс]. – Режим доступа: https://debianinstall.ru/razdelyaemaya-pamyat-semafory-i-ocheredi-soobshhenij-v-os-linux/. – Дата доступа: 13.02.2024.

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг исходного кода

Листинг 1 – Программный код lab5.c

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

int FREE_THREAD_COUNT = 0;

int ID_COUNTER = 0;

const int ptr_shift = 0;
const int beg_shift = sizeof(int*);
const int end_shift = sizeof(int*) + sizeof(int);
const int sem_shift = end_shift + sizeof(int);
const int id_shift = sem_shift + sizeof(sem_t*);

void* sort(void* arg)
{
    int* ptr = *((int**)arg);
    int beg = *((int*)(arg + beg_shift));
    int end = *((int*)(arg + end_shift));
    sem_t* sem = *((sem_t**)(arg + sem_shift));
    int id = *((int*)(arg + id_shift));
    printf("My args: beg = %i, end = %i, arg addr: %i\n", beg, end, (int)arg);
    int curr_beg = beg;
    int curr_end = end;

    int* arr = ptr;
    pthread_t tid = 0;
    pthread_t tid2 = 0;
    void* new_arg = NULL;
    void* new_arg2 = NULL;
    if(end - beg > 1)
    {
        sem_wait(sem);
        if(FREE_THREAD_COUNT > 0 )
        {
            --FREE_THREAD_COUNT;
            ++ID_COUNTER;
            int new_id = ID_COUNTER;
            sem_post(sem);
            new_arg = malloc(sizeof(int*) * 3 * sizeof(int) + sizeof(sem_t*));
            // args
            curr_end = (beg + end) / 2;

            *((int**)(new_arg + ptr_shift)) = arr;
            *((int*)(new_arg + beg_shift)) = curr_end; //COUNT
            *((int*)(new_arg + end_shift)) = end; // COUNT
            *((sem_t**)(new_arg + sem_shift)) = sem;
            *((int*)(new_arg + id_shift)) = new_id;

            if(pthread_create(&tid, NULL, sort, new_arg))
            {
                perror("pthread create error");
                exit(1);
            }

        }
        else
            sem_post(sem);

    }

    if(curr_end - curr_beg > 1)
    {
        sem_wait(sem);
        if(FREE_THREAD_COUNT > 0 )
        {
```

```
        --FREE_THREAD_COUNT;
        ++ID_COUNTER;
        int new_id = ID_COUNTER;
        sem_post(sem);
        new_arg2 = malloc(sizeof(int*) * 3 * sizeof(int) + sizeof(sem_t*));
        // args

        *((int**)(new_arg2 + ptr_shift)) = arr;
        *((int*)(new_arg2 + beg_shift)) = curr_beg; //COUNT
        *((int*)(new_arg2 + end_shift)) = curr_end; // COUNT
        *((sem_t**)(new_arg2 + sem_shift)) = sem;
        *((int*)(new_arg2 + id_shift)) = new_id;
        if(pthread_create(&tid, NULL, sort, new_arg2))
        {
            perror("pthread create error");
            exit(1);
        }

    }
    else
        sem_post(sem);
}

// sort
if(tid2 == 0)
{
    for(int i = beg; i < end - 1; ++i)
    {
        for(int j = beg; j < end - (i - beg) - 1; ++j)
        {
            // printf("%i ", arr[j]);
            if(arr[j] > arr[j+1])
            {
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}

if(tid == 0) return NULL;

pthread_join(tid, NULL);
free(new_arg);
if(tid2 != 0)
    pthread_join(tid2, NULL);

new_arg = NULL;
free(new_arg2);
new_arg2 = NULL;

int* tmp = (int*) malloc(sizeof(int) * (end - beg));
for(int i = 0; i < end - beg; ++i)
{
    tmp[i] = arr[i+beg];
}
int p1 = 0;
int p2 = curr_end - beg;

// merge

for(int i = beg; i < end; ++i)
{
    if(p1 < curr_end - beg && p2 < end - beg)
    {
        if(tmp[p1] < tmp[p2])
        {
            arr[i] = tmp[p1];
            ++p1;
        }
        else
        {
            arr[i] = tmp[p2];
            ++p2;
        }
    }
    else if(p1 < curr_end - beg)
```

```
        {
            arr[i] = tmp[p1];
            ++p1;
        }

        else
        {
            arr[i] = tmp[p2];
            ++p2;
        }
    }
    free(tmp);
    return NULL;
}

int main()
{
    printf("Enter number of threads:\n ");
    scanf("%i", &FREE_THREAD_COUNT);
    if(FREE_THREAD_COUNT < 0)
    {
        printf("haha\n");
        return;
    }
    sem_t* sem = (sem_t*)malloc(sizeof(sem_t));
    if(sem_init(sem, 0, 1))
    {
        perror("Sem init error");
    }

    int size = 100;

    int* arr = (int*)malloc(sizeof(int) * size);
    srand(time(NULL));
    for(int i = 0; i < size; ++i)
    {
        arr[i] = rand() % 100;
    }

    printf("Array before sort\n");
    int counter = 0;
    for(int i = 0; i < size; ++i)
    {
        printf("%i ", arr[i]);
        counter = (counter + 1) % 10;
        if(counter == 0) printf("\n");
    }

    void* args = malloc(sizeof(int*) * 3 * sizeof(int) + sizeof(sem_t*));

    *((int**)(args + ptr_shift)) = arr;
    *((int*)(args + beg_shift)) = 0; //COUNT
    *((int*)(args + end_shift)) = size; // COUNT
    *((sem_t**)(args + sem_shift)) = sem;
    *((int*)(args + id_shift)) = 0;
    printf("args addr: %i", (int)args);
    sort(args);

    printf("\n\nArray after sort\n");
    counter = 0;
    for(int i = 0; i < size; ++i)
    {
        printf("%i ", arr[i]);
        counter = (counter + 1) % 10;
        if(counter == 0) printf("\n");
    }

    sem_close(sem);
    sem_destroy(sem);
    free(arr);
    free(sem);
}
```