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 № 4
на тему

УПРАВЛЕНИЕ ПРОЦЕССАМИ И ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ

Выполнил К. А. Тимофеев

Проверил Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

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

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

По терминологии Unix выполняющаяся программа называется процессом.

Ядро Unix является многозадачным, оно может распределять время между несколькими процессами, создавая впечатление, будто все они выполняются одновременно. При этом ядро регулирует доступ к ресурсам. В данном случае ресурсом является время центрального процессора.[1]

В ходе процесса одновременно выполняется только одно действие. То есть в любой момент времени можно точно определить, какая часть кода выполняется.

Регулирование доступа к времени центрального процессора называется планированием. Планирование делит время на кванты, которые называются временными квантами.

Процессы в Unix имеют три различных вида приоритетов:

– статический;

– динамический;

– реального времени.

Приоритет – это целочисленное значение, которое присваивает процессу при определении того, какому процессу должно быть выделено определенное время.

Идентификация процесса – присвоение ему целочисленного идентификатора, уникального в пределах системы.

Каждый процесс в Unix системе имеет свой PID процесса, который используется для идентификации процесса в системе и назначается ядром операционной системы. В операционной системе Windows концепция идентификации процесса существует, но является не такой выраженной, как в Unix-системах. Вместо этого, в Windows каждый процесс имеет свой уникальный дескриптор процесса.[2]

IPC – это механизм обмена данными и синхронизации между процессами в операционной системе.[3]

В Unix-подобных системах существует несколько механизмов IPC, а именно:

1 Каналы: представляют собой однонаправленный поток данных между двумя связанными процессами.

2 Сигналы: базовые и простейшие IPC для управления и взаимодействия процессами.

3 Семафоры: примитивы синхронизации, которые используются для доступа к ресурсам, которые могут быть разделяемыми между несколькими процессами. Представляют собой счетчик, который может быть уменьшен или увеличен, и используется для ограничения доступа процессов к критическим секциям или ресурсам.

4 Разделяемая память: область памяти, которая может быть доступна нескольким процессам.

5 Очереди сообщений: передача данных осуществляется в виде законченных фрагментов с определенной структурой и в определенном порядке.

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, а также функции ftok для создания ключа, shmget для создания нового сегмента разделяемой памяти, shmat для присоединения сегмента разделяемой памяти, shmdt для отсоединения сегмента разделяемой памяти, shmctl для удаления сегмента разделяемой памяти.

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 процессами: при помощи функции fork создавались дочерние процессы, в которых происходила инициализация значениями блока данных. После при помощи функции wait родительский процесс ждет завершения дочерних процессов. Подсчет суммы значений блока данных осуществляется уже в основном родительском процессе.

3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе лабораторной работы была реализована программа, в которой реализуется заполнение родительским процессом матрицы со случайными элементами, после чего дочерние процессы получают матрицу квадратных корней значений оригинальной матрицы. Результат работы программы представлен на рисунке 3.1.

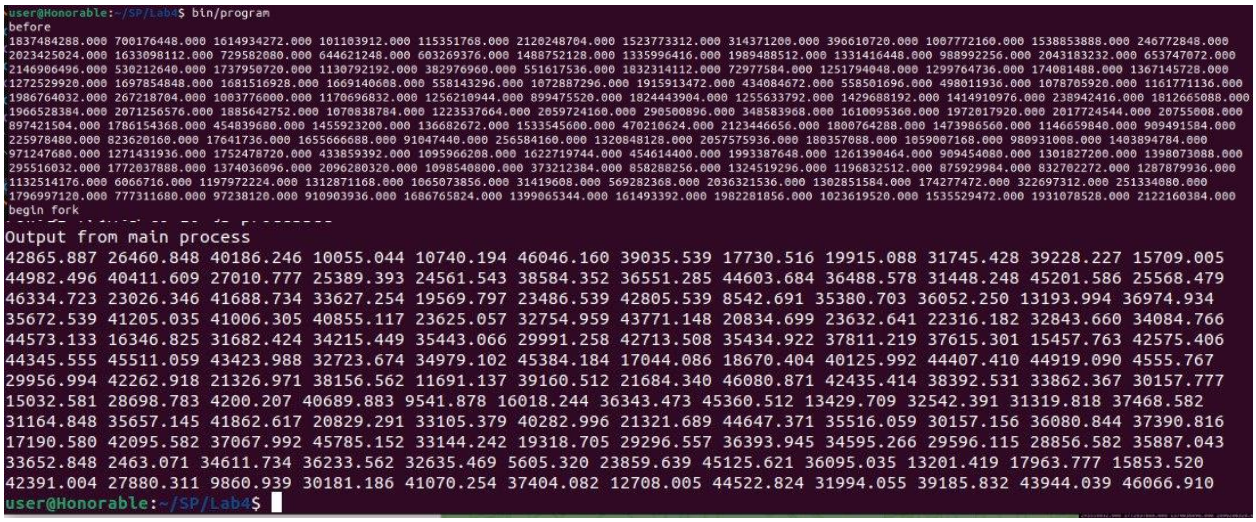

Рисунок 3.1 – Результат работы программы

Таким образом, в ходе лабораторной работы была реализована программа, реализующая заполнение блока данных случайными значениями и подсчет квадратных корней этих значений.

ВЫВОДЫ

В ходе лабораторной работы были изучены основные особенности подсистемы управления процессами и средства взаимодействия процессов в Unix. Кроме этого, была реализована программа на языке программирования C, которая параллельно обрабатывает блок данных различными процессами, которые считают квадратные корни значений оригинального блока данных.

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: http://heap.altlinux.org/tmp/unix_base/ch01s03.html. – Дата доступа: 13.02.2024.

[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 lab4.c

```
// semaphores processes shared memory
# include <sys/types.h>
# include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>
#include <errno.h>
#include <math.h>

#define ROW_PROCESS 6
#define COLUMNS_PROCESS 6

void process(int iBeg, int jBeg, int iEnd, int jEnd, int shmid, int rows, int columns)
{
    void* mem = (void*)shmat(shmid, NULL, 0);

    sem_t* sem = (sem_t*)mem;

    float* arr = (float*)(sem + 1);

    for(int i = iBeg; i < iEnd && i < rows; ++i)
    {
        for(int j = jBeg; j < jEnd && j < columns; ++j)
        {
            arr[i*columns + j] = sqrtf(arr[i*columns + j]);
        }
        sem_wait(sem);
        printf("row %i from %i to %i is processed\n", i, jBeg, jEnd<columns?jEnd:columns);
        sem_post(sem);
    }

    shmdt(mem);
}

int main()
{
    // buffer for shm operations
    int rows = 12;
    int columns = 12;

    int pids[ROW_PROCESS * COLUMNS_PROCESS];

    struct shmid_ds buf;
    // key
    key_t key = ftok(".", 'R');
    if (key == -1)
    {
        perror("ftok error");
        exit(1);
    }
    // semaphore
    int shmid = shmget(key, sizeof(sem_t) + sizeof(float)*rows*columns, IPC_CREAT|IPC_EXCL| 00666);
    if(shmid == -1)
    {
        if(errno == EEXIST)
        {
            perror("Error eexist");
        }
        else
        {
            perror("Not eexist");
        }
        exit(1);
    }
    void* mem = (void*)shmat(shmid, NULL, 0);
    if((long)mem == -1)
    {
        perror("memory at error");
        exit(1);
    }

    sem_t* sem = (sem_t*)mem;
```

```
    if(sem_init(sem, 1, 0) == -1)
    {
        perror("sem_init error");
        exit(1);
    }

    // array init
    float* arr = (float*)(sem + 1);

    srand(time(NULL));
    for(int i = 0; i < rows; ++i)
    {
        for(int j = 0; j < columns; ++j)
        {
            arr[i*columns + j] = rand();
        }
    }

    printf("before\n");
    for(int i = 0; i < rows; ++i)
    {
        for(int j = 0; j < columns; ++j)
        {
            printf("%.3f ", arr[i*columns + j]);
        }
        printf("\n");
    }
    printf("begin fork\n");
    for(int i = 0; i < ROW_PROCESS; ++i)
    {
        for (int j = 0; j < COLUMNS_PROCESS; ++j)
        {
            int res = fork();
            if(res == -1)
            {
                perror("fork error");
                exit(1);
            }
            else if(res == 0)
            {
                process(i * (rows / ROW_PROCESS), (j) * (columns / COLUMNS_PROCESS), (i + 1) *
(rows/ROW_PROCESS), (j + 1) * (columns/ COLUMNS_PROCESS),
                shmid, rows, columns);
                return 0;
            }

            pids[i*COLUMNS_PROCESS + j] = res;
        }
    }

    sem_post(sem);


    for (int i = 0; i <  ROW_PROCESS * COLUMNS_PROCESS; ++i)
    {
        if(waitpid(pids[i], NULL, WUNTRACED) == -1)
        {
            perror("waitpid error");
            exit(1);
        }
    }

    printf("Output from main process\n");

    for(int i = 0; i < rows; ++i)
    {
        for(int j = 0; j < columns; ++j)
        {
            printf("%.3f ", arr[i*columns + j]);
        }
        printf("\n");
    }

    sem_close(sem);
    sem_destroy(sem);
    shmdt(mem);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```