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             А. Д. Филипеня

Проверил                          Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc157722973)

[2 Краткие теоретические сведения 4](#_Toc157722974)

[3 Результаты выполнения лабораторной работы 6](#_Toc157722975)

[Выводы 7](#_Toc157722976)

[Список использованных источников 8](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 9](#_Toc157722978)

# **1 ПОСТАНОВКА ЗАДАЧИ**

Целью выполнения данной лабораторной работы является изучение основных особенностей подсистемы управления процессами и средств взаимодействия процессов в Unix. Также необходимо реализовать программу, которая будет реализовывать параллельную обработку блока данных, совместно обрабатываемые данные размещаются в общей разделяемой памяти. Необходимо решить задачу контроля свободности-занятости блоков и предотвращения «столкновений».

# **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 Unix реализована виртуальная память процессов. Каждый процесс исполняется в собственном адресном пространстве и не может непосредственно обращаться к памяти других процессоров. Такая изоляция процессов друг от друга обеспечивается механизмами операционной системы и поддерживается на аппаратном уровне в современных процессорах.

Разделение виртуальной памяти отдельных процессов необходимо в целях безопасности – процессы не должны иметь возможность по собственной инициативе менять данные других процессов – не только из-за потенциальных возможностей злоумышленников, но и потому, что программы могут совершать ошибки, и аварийное завершение одного процесса не должно отражаться на ходе работы других процессов и операционной системы в целом.

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. Сигнал может быть отправлен процессу операционной системой или другим процессом. Операционная система использует сигналы для доставки процессу уведомлений об ошибках и неправильном поведении.

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: это специальный механизм, с помощью которого средствами операционной системы два процесса могут обращаться к общему участку физической памяти – каждый через свое адресное пространство.

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 shmget для создания нового сегмента разделяемой памяти, shmat для присоединения сегмента разделяемой памяти к адресному пространству процессора, shmdt для отсоединения сегмента разделяемой памяти, shmctl для удаления сегмента разделяемой памяти.

2 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. Также использовалась функция getpid для получения PID процессора.

3 Мьютексы: для контроля доступа к разделяемой памяти каждого процесса были использованы мьютексы, а также функции pthread\_mutex\_init для инициализации мьютекса, pthread\_mutex\_destroy для удаления мьютекса, pthread\_mutex\_lock для того, чтобы занять мьютекс, если он еще не занят, pthread\_mutex\_unlock для освобождения занятого мьютекса.

# **3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ             РАБОТЫ**

В ходе лабораторной работы была создана программа, в которой реализуется создание нескольких дочерних процессов, которые заполняют случайными значениями определенный блок данных. В ходе работы было осуществлено деление общего массива памяти на несколько блоков, каждый из которых может быть в конкретный момент времени занят одним набором данных и использоваться одним процессом. Функцию контроля свободности-занятости блоков осуществляют мьютексы. Результат работы программы представлен на рисунке 3.1.

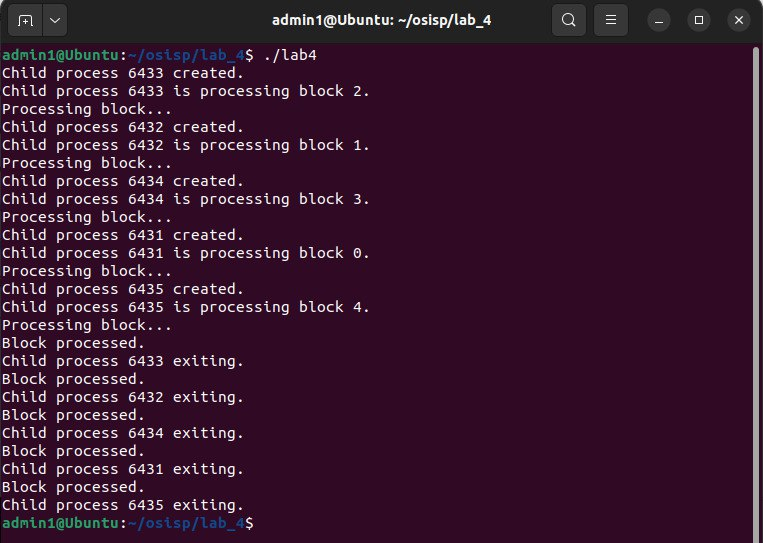


Рисунок 3.1 – Результат работы программы

Таким образом, в ходе лабораторной работы была реализована программа, реализующая параллельную обработку данных, причем совместно обрабатываемые данные размещаются в общей разделяемой памяти и передаются между участниками по номеру блока памяти.

# **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 С, которая реализует параллельную обработку блока данных, при этом совместно обрабатываемые данные размещаются в общей разделяемой памяти. Также была решена задача контроля свободности-занятости блоков и предотвращения «столкновений».

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Процессы и потоки [Электронный ресурс]. – Режим доступа: <https://acm.bsu.by/wiki/Unix2019b/>. – Дата доступа: 19.02.2024.

[2] Архитектура Unix. Процессы [Электронный ресурс]. – Режим доступа: http://heap.altlinux.org/tmp/unix\_base/ch01s03.html. – Дата доступа: 19.02.2024.

[3] Разделяемая память и семафоры [Электронный ресурс]. – Режим доступа: https://debianinstall.ru/razdelyaemaya-pamyat-semafory-i-ocheredi-soobshhenij-v-os-linux/. – Дата доступа: 20.02.2024.

## **ПРИЛОЖЕНИЕ А**

## **(обязательное)**

## **Листинг исходного кода**

Листинг 1 – Программный код lab4.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <pthread.h>

#include <time.h>

#define NUM\_BLOCKS 5

#define BLOCK\_SIZE 100

typedef struct {

int data[BLOCK\_SIZE];

pthread\_mutex\_t mutex;

} BlockData;

void process\_data(BlockData \*block) {

printf("Processing block...\n");

srand(time(NULL));

for (int i = 0; i < BLOCK\_SIZE; i++) {

block->data[i] = rand() % 100;

}

sleep(1);

printf("Block processed.\n");

}

int main() {

int shm\_id;

BlockData \*shared\_mem;

int i;

// Создание сегмента разделяемой памяти

shm\_id = shmget(IPC\_PRIVATE, NUM\_BLOCKS \* sizeof(BlockData), IPC\_CREAT | 0666);

if (shm\_id < 0) {

perror("shmget");

exit(1);

}

// Присоединение сегмента к адресному пространству процесса

shared\_mem = shmat(shm\_id, NULL, 0);

if ((int)shared\_mem == -1) {

perror("shmat");

exit(1);

}

// Инициализация блоков данных и мьютексов

for (i = 0; i < NUM\_BLOCKS; i++) {

pthread\_mutex\_init(&shared\_mem[i].mutex, NULL);

}

for (i = 0; i < NUM\_BLOCKS; i++) {

if (fork() == 0) {

printf("Child process %d created.\n", getpid());

printf("Child process %d is processing block %d.\n", getpid(), i);

pthread\_mutex\_lock(&shared\_mem[i].mutex); // Блокировка блока данных

process\_data(&shared\_mem[i]);

pthread\_mutex\_unlock(&shared\_mem[i].mutex); // Разблокировка блока данных

printf("Child process %d exiting.\n", getpid());

exit(0);

}

}

// Ожидание завершения всех дочерних процессов

for (i = 0; i < NUM\_BLOCKS; i++) {

wait(NULL);

}

// Удаление мьютексов

for (i = 0; i < NUM\_BLOCKS; i++) {

pthread\_mutex\_destroy(&shared\_mem[i].mutex);

}

// Отключение сегмента от адресного пространства процесса

shmdt(shared\_mem);

// Удаление сегмента разделяемой памяти

shmctl(shm\_id, IPC\_RMID, NULL);

return 0;

}

Листинг 2 – Программный код makefile

CC = gcc

CFLAGS = -Wall

all: lab4

lab4: lab4.c

$(CC) $(CFLAGS) lab4.c -o lab4

clean:

rm -f lab4