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             Е. А. Киселёва

Проверил                          Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc157722973)

[2 Краткие теоретические сведения 4](#_Toc157722974)

[3 Результаты выполнения лабораторной работы 5](#_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. Практическое проектирование, реализация и отладка программных комплексов из нескольких взаимодействующих процессов.

# **2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Подсистема управления процессами и средств взаимодействия процессов в операционной системе Unix играет ключевую роль в организации работы приложений и ресурсов компьютерной системы. В Unix каждая запущенная программа представлена в виде процесса, который обладает уникальным идентификатором PID (Process ID).

Основные механизмы управления процессами в Unix включают в себя следующие операции:

1 fork: этот системный вызов используется для создания нового процесса путем дублирования уже существующего. При этом новый процесс получает копию адресного пространства исходного процесса.

2 exec: данный вызов используется для замены адресного пространства текущего процесса новым содержимым, например, другой программой. После выполнения exec() процесс перестает быть тем, что он был, и становится новой программой.

3 wait: механизм ожидания завершения дочернего процесса и получения его статуса. Этот вызов позволяет родительскому процессу координировать выполнение дочерних процессов.

4 exit: выход из процесса с передачей статуса завершения. Этот вызов завершает выполнение текущего процесса и возвращает статус завершения родительскому процессу.

Чтобы обеспечить взаимодействие между процессами, Unix предоставляет следующие средства:

1 Каналы (pipes): позволяют передавать данные между процессами через поток данных.

2 Сигналы (signals): механизм оповещения процессов о различных событиях, таких как завершение другого процесса или запрос на прерывание.

3 Сокеты (sockets): обеспечивают обмен данными между процессами через сеть.

4 Семафоры и мьютексы: используются для синхронизации доступа к общим ресурсам между процессами.

Взаимодействие процессов в Unix осуществляется через стандартные потоки ввода/вывода (stdin, stdout, stderr) и специальные системные вызовы, предназначенные для обмена данными и синхронизации между процессами. Процессы могут создавать цепочки или группы для организации взаимодействия и управления. Общий принцип "все есть файл" в Unix расширяет возможности взаимодействия, предполагая, что процессы могут обмениваться данными через файловую систему, сигналы и другие механизмы, что делает систему Unix гибкой и эффективной в управлении процессами и их взаимодействии.

# **3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ             РАБОТЫ**

В ходе лабораторной работы была реализована программа, в которой реализуется создание нескольких дочерних процессов, которые заполняют случайными значениями определенный блок данных, после чего родительский процесс подсчитывает сумму этих значений. Результат работы программы представлен на рисунке 3.1.

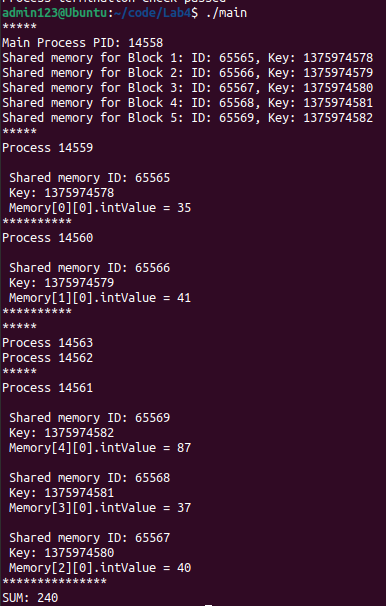


Рисунок 3.1 – Результат работы программы

Также приложение тестируется с помощью makefile. В тесте проверяется, успешно ли завершаются все процессы после отрабатывания приложения. Результат работы теста представлен на рисунке 3.2.

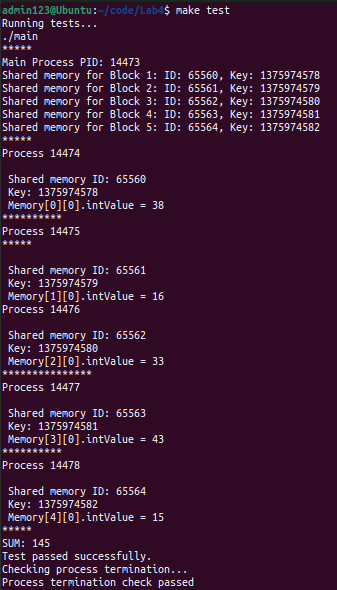


Рисунок 3.2 – Результат работы теста программы

Таким образом, в ходе лабораторной работы была реализована программа, которая заполняет блок данных случайными значениями и считает сумму 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 С, которая параллельно обрабатывает блок данных различными процессами, которые заполняют данный блок данных случайными числами, после чего родительский процесс подсчитывает сумму этих значений.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Процессы и потоки [Электронный ресурс]. – Режим доступа: <https://acm.bsu.by/wiki/Unix2019b/>. – Дата доступа: 10.03.2024.

[2] Архитектура Unix. Процессы [Электронный ресурс]. – Режим доступа: http://heap.altlinux.org/tmp/unix\_base/ch01s03.html. – Дата доступа: 11.03.2024.

[3] Разделяемая память и семафоры [Электронный ресурс]. – Режим доступа: https://debianinstall.ru/razdelyaemaya-pamyat-semafory-i-ocheredi-soobshhenij-v-os-linux/. – Дата доступа: 11.03.2024.

## **ПРИЛОЖЕНИЕ А**

## **(обязательное)**

## **Листинг исходного кода**

Листинг 1 – Программный код makefile

CC=gcc

CFLAGS=-Wall

.PHONY: all clean start test

all: main

main: main.o

$(CC) $(CFLAGS) main.o -o main

main.o: main.c

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

start: main

./main

test: main

@echo "Running tests..."

./main

@echo "Test passed successfully."

@echo "Checking process termination..."

@PROCESS\_COUNT=$$(ps aux | grep "./main" | grep -v "grep" | wc -l); \

if [ "$$PROCESS\_COUNT" -eq 0 ]; then \

echo "Process termination check passed"; \

else \

echo "Process termination check failed"; \

fi

clean:

rm -f main main.o

Листинг 2 – Программный код main.c

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <semaphore.h>

#include <time.h>

#define NUM\_BLOCKS 5

#define BLOCK\_SIZE 1

typedef struct {

int intValue;

} DataBlock;

void create\_agents(key\_t key, int id[], DataBlock \*memory[], sem\_t \*sem) {

int i;

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

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

exit(1);

} else if (pid == 0) {

key\_t process\_key = key + i;

sem\_wait(sem);

printf("\*\*\*\*\*\nProcess %d\n", getpid());

srand(time(NULL) + getpid() + i);

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

memory[i][j].intValue = rand() % 100;

printf("\n Shared memory ID: %d\n Key: %d\n Memory[%d][%d].intValue = %d\n",

id[i], process\_key, i, j, memory[i][j].intValue);

}

printf("\*\*\*\*\*");

sem\_post(sem);

exit(0);

}

}

}

int main() {

int id[NUM\_BLOCKS];

key\_t key;

DataBlock \*memory[NUM\_BLOCKS];

sem\_t sem;

int i;

if (sem\_init(&sem, 1, 1) == -1) {

perror("sem\_init");

exit(1);

}

if ((key = ftok(".", 'R')) == -1) {

perror("ftok");

exit(1);

}

printf("\*\*\*\*\*\nMain Process PID: %d\n", getpid());

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

key\_t process\_key = key + i;

if ((id[i] = shmget(process\_key, BLOCK\_SIZE \* sizeof(DataBlock), IPC\_CREAT | IPC\_EXCL | 0644)) == -1) {

perror("shmget");

exit(1);

}

printf("Shared memory for Block %d: ID: %d, Key: %d\n", i + 1, id[i], process\_key);

memory[i] = (DataBlock \*)shmat(id[i], NULL, 0);

if (memory[i] == (DataBlock \*) -1) {

perror("shmat");

exit(1);

}

}

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

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

memory[i][j].intValue = 0;

}

}

create\_agents(key, id, memory, &sem);

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

wait(NULL);

}

int sum = 0;

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

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

sum += memory[i][j].intValue;

}

}

printf("\nSUM: %d\n", sum);

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

if (shmdt(memory[i]) == -1) {

perror("shmdt");

exit(1);

}

}

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

if (shmctl(id[i], IPC\_RMID, NULL) == -1) {

perror("shmctl");

exit(1);

}

}

if (sem\_destroy(&sem) == -1) {

perror("sem\_destroy");

}

return 0;

}