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

ОТЧЁТ

к лабораторной работе №4

на тему

**УПРАВЛЕНИЕ ПРОЦЕССАМИ И ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ**

Выполнил: студент гр.253505 Снежко М.А.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

[1 Формулировка задачи 3](#_Toc179742261)

[2 Краткие теоритические сведения 4](#_Toc179742262)

[3 Описание функций программы 5](#_Toc179742263)

[Заключение 6](#_Toc179742264)

[Список использованных источников 7](#_Toc179742265)

[Приложение А (обязательное) 8](#_Toc179742266)

# 1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью данной лабораторной работы является изучение подсистемы управления процессами и средств взаимодействия процессов в операционной системе *Unix*, а также освоение основных механизмов создания и управления процессами, таких как системные вызовы *fork*, *exec*, *wait*, и средств межпроцессного взаимодействия (*IPC*), включая каналы, семафоры, очереди сообщений и разделяемую память. Основное внимание уделяется практическому использованию системных вызовов и библиотек для создания многопроцессных приложений, организации взаимодействия между процессами и обработки сигналов. Работа направлена на создание программного комплекса, состоящего из нескольких взаимодействующих процессов, а также на освоение принципов управления процессами и их синхронизации.

В процессе выполнения лабораторной работы необходимо разработать программу, которая создает несколько дочерних процессов, организует между ними взаимодействие с использованием одного из механизмов *IPC*, например, каналов или разделяемой памяти, и реализует обмен данными между процессами. Программа должна поддерживать возможность передачи данных от одного процесса к другому, их обработку и корректное завершение всех процессов с анализом кодов завершения. Для управления сборкой проекта требуется создать *makefile*, включающий цели для компиляции, очистки промежуточных файлов и тестирования программы с заранее подготовленными тестовыми данными.

Для взаимодействия процессов можно использовать каналы, которые позволяют передавать данные между процессами, или разделяемую память для обмена информацией. Программа должна корректно обрабатывать сигналы, такие как *SIGINT* или *SIGTERM*, для управления поведением процессов и их завершения. Особое внимание уделяется созданию makefile, который автоматизирует процесс сборки проекта. Makefile должен включать цели для компиляции каждого модуля программы, сборки исполняемого файла, очистки промежуточных файлов (объектных файлов и временных данных) и тестирования программы. Тестирование может быть реализовано с использованием заранее подготовленных тестовых данных, которые проверяют корректность взаимодействия процессов и обработки данных.

Таким образом, данная лабораторная работа позволяет изучить ключевые аспекты управления процессами и их взаимодействия в *Unix*-средах, включая использование системных вызовов, механизмов *IPC* и обработки сигналов. Полученные навыки помогут в дальнейшем создавать эффективные и надежные многопроцессные приложения для решения широкого круга задач в Unix-средах.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

*UNIX* – многопользовательская, многозадачная операционная система (ОС), разработанная для гибкости и адаптивности. Первоначально разработанная в 1970-х годах, *Unix* была одной из первых ОС, написанных на языке программирования *C*. С момента своего появления операционная система *Unix* и ее ответвления оказали глубокое влияние на компьютерную и электронную промышленность, предлагая переносимость, стабильность и совместимость в различных гетерогенных средах и типах устройств. [1]

В операционных системах семейства *Unix/Linux* программирование на языке C играет ключевую роль в разработке приложений, ориентированных на системные задачи, такие как управление процессами, межпроцессное взаимодействие (*IPC*), обработка сигналов и синхронизация процессов. Основные инструменты разработки включают компилятор *gcc*, который используется для преобразования исходного кода в исполняемые файлы, и систему сборки *make*, которая автоматизирует процесс компиляции и управления проектами.

Компилятор *gcc* поддерживает множество опций, позволяющих настраивать процесс компиляции, включая оптимизацию кода, отладку и генерацию промежуточных файлов. Система сборки *make* использует *makefile*. Это специальный файл, содержащий инструкции для сборки проекта. *Makefile* определяет зависимости между файлами, цели сборки и команды для их выполнения. Это позволяет эффективно управлять многомодульными проектами, где исходный код разделён на несколько файлов. [2]

Основная среда взаимодействия с *UNIX* – командная строка. Суть её в том, что каждая строка, передаваемая пользователем системе, – это команда, которую та должна выполнить. Пока не нажата клавиша *Enter*, строку можно редактировать, затем она отсылается системе. Команды интерпретируются и выполняются специальной программой – командной оболочкой или *shell*. Через командную оболочку производится управление пользовательскими процессами – для этого используются средства межпроцессного обмена. Командная оболочка непосредственно связана с терминалом, через который осуществляется передача управляющих последовательностей и текста. [3]

Таким образом, программирование на языке *C* в *Unix* предоставляет мощные средства для создания приложений, которые могут взаимодействовать с операционной системой, обрабатывать текстовые данные и управлять ресурсами. Полученные знания и навыки могут быть использованы для разработки более сложных программ, таких как утилиты для обработки текста, системные утилиты или инструменты автоматизации.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Программа представляет собой самовосстанавливающийся процесс, который продолжает выполнение даже после получения сигнала, стандартно вызывающего завершение. Основная задача программы заключается в периодическом выполнении действий (например, обновлении счетчика и записи его значения в файл), а также в создании копии процесса при получении сигнала *SIGTERM*. Это позволяет процессу восстановиться и продолжить выполнение с прерванного места, избегая безусловного завершения.

Программа начинается с установки обработчика сигнала *SIGTERM* с помощью функции *signal*. Обработчик *handle\_signal* изменяет значение флага *keep\_running* на 0, что приводит к завершению основного цикла программы. Внутри цикла программа увеличивает значение счетчика и записывает его в файл *counter.txt* с помощью функции *write\_counter*. Для записи данных используется системный вызов *open* для открытия файла и *dprintf* для записи значения счетчика. После каждой итерации программа приостанавливает выполнение на 1 секунду с помощью функции *sleep*.

При получении сигнала *SIGTERM* программа завершает основной цикл и создает копию процесса с помощью системного вызова *fork*. Если вызов *fork* успешен, дочерний процесс запускает новую копию программы с помощью *execlp*. Это позволяет процессу восстановиться и продолжить выполнение с прерванного места. В случае ошибки при вызове *exec* программа завершается с выводом сообщения об ошибке. Результат работы программы представлен на рисунке 3.1.

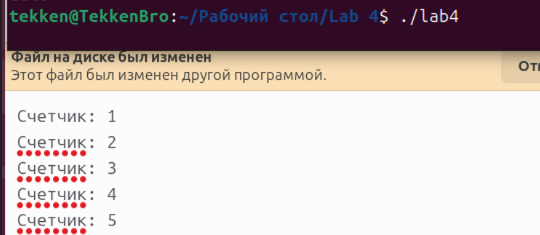


Рисунок 3.1 – Результат работы программы

Таким образом, программа демонстрирует использование языка *C* для управления процессами и обработки сигналов в среде *Unix*. Она позволяет изучить работу с системными вызовами, такими как *fork*, *exec*, *signal*, а также применение функций для работы с файлами. Реализация программы иллюстрирует возможности языка *C* для создания устойчивых приложений, способных восстанавливаться после критических ситуаций.

# ЗАКЛЮЧЕНИЕ

Таким образом, в рамках данной лабораторной работы была разработана программа, реализующая самовосстанавливающийся процесс на языке *C* в среде *Unix*. Основная цель работы заключалась в изучении механизмов управления процессами, обработки сигналов и создания копий процессов для обеспечения устойчивости к завершению. Реализация программы позволила на практике освоить возможности языка *C*, а также методы работы с системными вызовами, такими как *fork*, *exec* и обработка сигналов.

Разработанный самовосстанавливающийся процесс представляет собой устойчивое решение, которое при получении сигнала, стандартно вызывающего завершение, создает свою копию и продолжает выполнение с прерванного места. Это позволяет избежать безусловного завершения процесса и обеспечивает его устойчивость к неперехватываемым сигналам. В качестве демонстрации работы процесса используется периодическое выполнение действий, таких как обновление счетчика и запись его значения в файл, что позволяет визуально отслеживать "живость" процесса.

Выполнение данной лабораторной работы позволило на практике изучить основные принципы управления процессами в *Unix*, включая создание процессов, обработку сигналов и их взаимодействие. Реализация программы помогла закрепить знания о системных вызовах, таких как *fork* и *exec*, а также подчеркнула важность корректной обработки сигналов для обеспечения устойчивости приложений. Полученные навыки могут быть полезны при разработке более сложных многопроцессных приложений, требующих высокой надежности и устойчивости к внешним воздействиям.

Таким образом, лабораторная работа позволила не только изучить основные конструкции программирования на *C*, но и освоить практические аспекты их применения для решения реальных задач. Реализация программы углубила знания о работе с процессами, сигналами и системными вызовами, а также продемонстрировала важность корректного взаимодействия с ресурсами операционной системы. Полученные навыки могут быть использованы для создания более сложных инструментов автоматизации и управления процессами. Также были изучены компилятор *gcc*, система сборки *make* и язык *makefile*, что позволило автоматизировать процесс сборки и тестирования программы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Thecode "Обзор UNIX" – Электронный ресурс. Режим доступа: https://thecode.media/unix.

[2] Skyeng "Что такое Gcc" – Электронный ресурс. Режим доступа: https://skyeng.ru/magazine/wiki/it-industriya/gcc-chto-eto-takoe.

[3] Heap.altlinux "Коммандная строка UNIX" – Электронный ресурс. Режим доступа: http://heap.altlinux.org/modules/unix\_base.dralex/ch02.html.

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

volatile sig\_atomic\_t keep\_running = 1;

void handle\_signal(int signal) {

if (signal == SIGTERM) {

keep\_running = 0;

}

}

void write\_counter(int count) {

// Записываем значение счетчика в файл

int fd = open("counter.txt", O\_WRONLY | O\_CREAT | O\_APPEND, 0644);

if (fd == -1) {

perror("Ошибка открытия файла");

exit(EXIT\_FAILURE);

}

dprintf(fd, "Счетчик: %d\n", count);

close(fd);

}

int main() {

signal(SIGTERM, handle\_signal);

int count = 0;

while (keep\_running) {

count++;

write\_counter(count);

sleep(1); // Задержка в 1 секунду

}

// Создаем копию процесса

pid\_t pid = fork();

if (pid == 0) {

// Код дочернего процесса

execlp("./lab4", "./lab4", NULL);

perror("Ошибка exec");

exit(EXIT\_FAILURE);

}

return 0;

}