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.253504
Фроленко К.Ю.
Проверил: ассистент кафедры информатики
Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью данной лабораторной работы является изучение механизмов управления процессами и средств взаимодействия между ними в операционной системе *Unix*. В процессе выполнения работы осуществляется практическое освоение системных вызовов для порождения процессов, обработки сигналов и организации межпроцессного взаимодействия. Основное внимание уделяется реализации принципа самовосстанавливающегося процесса, который при получении сигнала завершения порождает свою копию, позволяющую продолжить выполнение программы без прерывания работы.

В рамках работы необходимо разработать приложение «Самовосстанавливающийся процесс», которое должно обеспечивать следующие функциональные возможности:

1 Обеспечение непрерывного выполнения: программа должна периодически выполнять определённые действия (например, инкрементировать и выводить значение счётчика, записывать его в файл), демонстрируя «живость» процесса.

2 Обработка сигналов: приложение должно корректно обрабатывать стандартные сигналы завершения (*SIGTERM, SIGINT*) посредством установки пользовательских обработчиков (через *sigaction*). При получении соответствующего сигнала процесс должен инициировать процедуру самовосстановления.

3 Порождение нового процесса: при возникновении сигнала завершения основной процесс должен создать свою копию с сохранением текущего состояния (например, текущего значения счётчика) с помощью вызова *fork*. Родительский процесс завершается, передавая выполнение дочернему, что позволяет избежать полного останова работы программы.

4 Модульность реализации: программа должна быть структурирована на несколько функций – отдельная функция для обработки сигналов и порождения нового процесса, основной модуль для организации ввода-вывода и реализации логики приложения.

5 Автоматизация сборки: необходимо создать *makefile*, содержащий цели для сборки исполняемого файла, очистки промежуточных файлов (*clean*) и, по возможности, тестирования работы приложения.

Результатом выполнения лабораторной работы станет работоспособное приложение, демонстрирующее практическое применение системных вызовов *Unix* для управления процессами, а также навыки разработки многомодульных программ и автоматизации сборки. Это способствует глубокому пониманию возможностей среды *Unix* и приобретению важных навыков для профессиональной разработки системных и отказоустойчивых приложений.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Программирование на языке *C/C++* в операционной системе *Unix* занимает важное место в разработке высокопроизводительных, надёжных и переносимых приложений. Благодаря минималистичному синтаксису и прямому доступу к системным ресурсам, язык *C* позволяет работать «на уровне железа», оптимизируя использование оперативной памяти, процессорного времени и других ресурсов. Такая особенность актуальна для создания системных утилит, серверных приложений и отказоустойчивых систем, где эффективность и надежность имеют первостепенное значение [1].

Одним из ключевых аспектов системного программирования является управление процессами. В *Unix* для создания нового процесса используется системный вызов *fork*, который позволяет порождать точные копии родительского процесса. Вместе с возможностью замены образа процесса через *exec* эта технология служит фундаментом для реализации параллельных вычислений и распределённых систем. В рамках данной лабораторной работы рассматривается концепция самовосстанавливающегося процесса, при которой процесс, получая сигнал завершения (например, *SIGTERM* или *SIGINT*), с помощью механизма обработки сигналов через *sigaction* порождает свою копию и продолжает работу. Такой подход обеспечивает отказоустойчивость и непрерывность выполнения приложений, что критично для систем, требующих высокой доступности и устойчивости к сбоям [2].

При этом управление процессами включает не только их создание, но и мониторинг, синхронизацию и корректное завершение работы. Системные утилиты, такие как *ps* или *top*, позволяют отслеживать состояние запущенных процессов, их приоритеты и использование ресурсов, что является важным инструментом для администрирования и оптимизации работы системы. Возможность управлять процессами на низком уровне даёт разработчику контроль над распределением вычислительной нагрузки и позволяет своевременно реагировать на сбои или перегрузки.

Кроме того, значительную роль играет организация взаимодействия между процессами, реализуемая посредством механизмов межпроцессного взаимодействия (*IPC*). В *Unix* доступны разнообразные средства *IPC*, включая неименованные и именованные каналы (*pipes* и *FIFOs*), очереди сообщений, разделяемую память и семафоры. Каналы используются для передачи потоков байтов между процессами, что удобно при организации конвейерной обработки данных. Разделяемая память обеспечивает высокую скорость обмена информацией, поскольку несколько процессов работают с одним и тем же сегментом памяти, однако требует дополнительных средств синхронизации, таких как семафоры или мьютексы, чтобы избежать конфликтов при одновременном доступе. Очереди сообщений и именованные каналы позволяют передавать структурированные данные между процессами, что особенно важно для распределённых систем и сложных приложений [3].

Обработка сигналов является ещё одним критическим компонентом управления процессами. Сигналы представляют собой механизм

асинхронного уведомления, который позволяет процессам реагировать на различные события, такие как запросы на завершение, ошибки или внешние воздействия. Применение функции *sigaction* позволяет устанавливать надёжные обработчики сигналов, что гарантирует выполнение необходимых процедур даже в условиях высокой нагрузки. В контексте самовосстанавливающегося процесса обработчик сигнала инициирует порождение новой копии, сохраняя текущее состояние, что позволяет избежать полного останова работы приложения и обеспечивает его бесперебойное функционирование.

Кроме того, современные *Unix*-системы поддерживают продвинутые методы управления группами процессов, позволяющие централизованно координировать их выполнение и обмен информацией. Такие механизмы особенно полезны в серверных и многопользовательских системах, где требуется синхронизировать работу большого количества процессов и обеспечить устойчивость при возникновении сбоев.

Неотъемлемой частью разработки системных приложений является автоматизация сборки проекта с использованием системы *make*. *Makefile* позволяет задавать зависимости между исходными файлами, автоматизировать этапы компиляции, линковки и очистки проекта от временных артефактов. Такой подход значительно упрощает процесс разработки, снижает вероятность ошибок, связанных с ручным управлением сборкой, и ускоряет интеграцию изменений в код. Автоматизированная сборка способствует поддержанию единообразного стиля кода, что является важным аспектом при работе в команде и последующем сопровождении программного продукта [3].

Таким образом, теоретические сведения о системных вызовах, управлении процессами и механизмах межпроцессного взаимодействия имеют не только академическую, но и практическую ценность. Понимание принципов работы с процессами, сигналами и *IPC* позволяет разработчикам создавать отказоустойчивые системы, способные динамически реагировать на внешние воздействия и обеспечивать стабильную работу даже в условиях высоких нагрузок и непредвиденных сбоев. Эти знания закладывают прочную основу для разработки надежных системных приложений, удовлетворяющих требованиям современных критически важных и высоконагруженных систем.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

В рамках данной лабораторной работы разработана программа «Самовосстанавливающийся процесс», демонстрирующая практическое применение системных вызовов *Unix* для управления процессами, обработки сигналов и организации межпроцессного взаимодействия. Программа иллюстрирует, как при получении сигнала завершения (например, *SIGTERM* или *SIGINT*) текущий процесс порождает свою копию, а затем корректно завершает работу, что позволяет обеспечить непрерывное выполнение приложения. Ниже приведено подробное описание реализованных функций и используемых механизмов.

3.1 Функция handle_signal

Функция *handle_signal* является основным обработчиком сигналов в программе. Её задача – перехват сигналов, таких как *SIGTERM* и *SIGINT*, и выполнение процедуры самовосстановления. При получении сигнала функция выводит сообщение о том, какой сигнал был получен, и сообщает о запуске нового процесса. Далее происходит вызов системного вызова *fork*, который создает точную копию родительского процесса.

1 Обработка ошибок: если вызов *fork* завершился неудачно (возвращается отрицательное значение), функция выводит сообщение об ошибке и завершает выполнение программы с кодом ошибки.

2 Разделение ролей: если возвращаемое значение положительно, это означает, что текущий процесс является родительским. Родитель выводит сообщение о завершении своей работы и завершает выполнение, что позволяет новому (дочернему) процессу продолжить работу с сохраненным состоянием (например, текущим значением счетчика). В случае, когда *fork* возвращает 0, выполняется дочерний процесс, который выводит сообщение о продолжении работы с указанием текущего значения счетчика.

Таким образом, функция *handle_signal* реализует принцип отказоустойчивости: при поступлении сигнала завершения программа не останавливается, а переходит управление на новую копию процесса, что особенно важно для систем, где требуется высокая доступность.

3.2 Функция main

Функция *main* является точкой входа в программу и объединяет все компоненты в единое целое. В ней выполняется настройка обработки сигналов и реализуется основной цикл выполнения приложения.

1 Настройка обработчиков сигналов: в начале функции создается структура *sigaction*, которой присваивается функция-обработчик *handle_signal*. Флаг *SA_RESTART* гарантирует, что прерванные системные вызовы будут автоматически перезапущены после обработки сигнала. Затем с помощью вызова *sigaction* устанавливаются обработчики для сигналов

SIGTERM и *SIGINT*. При возникновении ошибки при установке обработчика выводится сообщение об ошибке, и программа завершается с соответствующим кодом.

2 Основной цикл: после настройки обработчиков запускается бесконечный цикл, в котором происходит инкрементирование глобального счетчика, вывод его значения на экран и запись в файл *counter.txt*. Для записи в файл используется потоковый вывод через *ofstream* с режимом добавления, что позволяет сохранять историю значений счетчика. Функция *sleep* обеспечивает задержку в 1 секунду между итерациями цикла, демонстрируя непрерывное выполнение программы.

3 Демонстрация непрерывности работы: благодаря такому устройству, даже при попытке завершения процесса, обработчик сигнала обеспечивает запуск нового процесса, и программа продолжает выполнение, сохраняя информацию о своей работе как на экране, так и в файле.

Таким образом, функция *main* не только организует основной рабочий цикл приложения, но и демонстрирует корректную работу с системными вызовами *Unix*, обеспечивая отказоустойчивость и непрерывность выполнения.

3.3 Makefile

Для автоматизации сборки и управления проектом используется makefile. Его структура позволяет быстро и эффективно компилировать проект, а также осуществлять очистку временных файлов.

1 Переменные компиляции: в *makefile* определены переменные, такие как *CXXFLAGS* (флаги компиляции, включающие предупреждения и оптимизацию) и *TARGET* (имя создаваемого исполняемого файла). Например, используются флаги *-Wall -O2 -std=c++11*, что обеспечивает строгую проверку кода и оптимизацию исполнения.

2 Цель all: основная цель *makefile* – сборка исполняемого файла. В ней исходный файл *main.cpp* компилируется с использованием заданных флагов, а затем формируется единый исполняемый файл.

3 Цель clean: цель *clean* предназначена для удаления скомпилированного файла и файла с результатами работы. Это позволяет поддерживать чистоту проекта и предотвращает накопление ненужных артефактов сборки.

Использование *makefile* значительно упрощает процесс сборки, тестирования и дальнейшего сопровождения проекта. Автоматизация компиляции помогает снизить вероятность ошибок, возникающих при ручном управлении сборкой, и обеспечивает единообразие в структуре проекта.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения

Разработанная программа «Самовосстанавливающийся процесс» предназначена для демонстрации отказоустойчивости приложения, которое непрерывно выполняет свою основную задачу – подсчёт, вывод текущего значения счётчика в консоль и запись данных в файл *counter.txt*. Программа запускается в терминале и работает в бесконечном цикле, что позволяет наблюдать динамическое обновление значений в реальном времени. В начальном состоянии, сразу после запуска, в консоли отображается последовательное увеличение счётчика с интервалом в 1 секунду.

На рисунке 4.1 приведён пример начального вывода программы в терминале, где видно, как начинается инкрементирование счётчика.

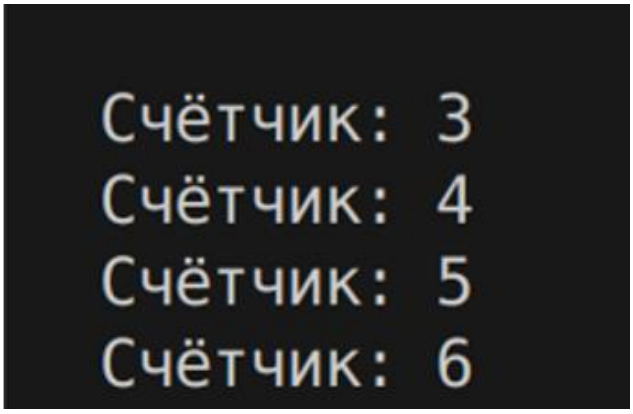

Рисунок 4.1 – Начальное состояние программы с выводом значений счётчика

После запуска программы пользователь может наблюдать стабильное и предсказуемое выполнение основного цикла. Это свидетельствует о корректной настройке среды выполнения и подтверждает, что приложение готово к дальнейшему тестированию механизмов обработки сигналов.

4.2 Обработка сигнала завершения и создание нового процесса

При получении сигнала завершения (например, *SIGTERM* или *SIGINT*) активируется обработчик сигнала, который выводит сообщение о получении сигнала и инициирует вызов системного вызова *fork* для создания нового процесса. Родительский процесс корректно завершает своё выполнение, а дочерний процесс продолжает работу с сохранённым состоянием, включая текущее значение счётчика.

На рисунке 4.2 показан процесс обработки сигнала: видно, как выводятся сообщения о получении сигнала, о запуске нового процесса и о завершении родительского процесса, что свидетельствует о корректной работе механизма самовосстановления.

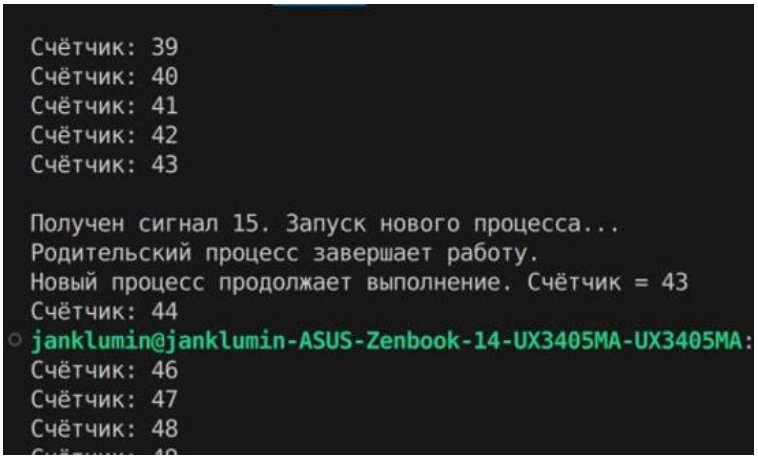

Рисунок 4.2 – Вывод сообщений после получения сигнала и переключения управления на дочерний процесс

Дальнейший анализ подтверждает, что механизм обработки сигналов позволяет приложению продолжать работу без прерывания, обеспечивая отказоустойчивость даже при получении команд на завершение. Это демонстрирует надёжность реализованного подхода к управлению процессами.

4.3 Демонстрация смены PID после убийства процесса

Особое внимание уделяется демонстрации работы механизма самовосстановления в условиях принудительного завершения. На рисунке 4.3 показан пример, где после отправки сигнала завершения (например, командой *kill -TERM <PID>*) видно, что текущий процесс завершается, а система автоматически создаёт новый процесс с новым идентификатором (*PID*). В консольном выводе отражается новый *PID*, что подтверждает, что после убийства процесса запускается новая копия приложения.

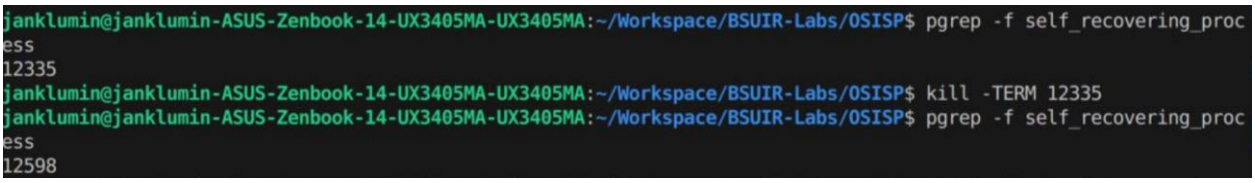

Рисунок 4.3 – Изменение *PID*: новый процесс запускается с новым идентификатором после отправки сигнала завершения

Таким образом, демонстрация смены *PID* иллюстрирует эффективность реализованного механизма самовосстановления. Пользователь получает явное подтверждение того, что даже в условиях принудительного завершения старого процесса система способна немедленно возобновить работу, запуская новый процесс, что гарантирует непрерывность функционирования приложения.

ВЫВОД

В ходе выполнения лабораторной работы была успешно разработана программа «Самовосстанавливающийся процесс», демонстрирующая практическое применение системных вызовов *Unix* для управления процессами, обработки сигналов и организации межпроцессного взаимодействия. Реализация проекта позволила применить и закрепить знания о работе с вызовами *fork* и *sigaction*, что является основополагающим для создания отказоустойчивых системных приложений.

Разработанная программа демонстрирует возможность непрерывного выполнения даже в условиях попыток принудительного завершения процесса. При получении сигналов завершения (*SIGTERM* или *SIGINT*) система автоматически порождает новую копию процесса, сохраняя текущее состояние приложения, что подтверждается изменением *PID* нового процесса. Такой механизм обеспечивает бесперебойное выполнение приложения и является важным элементом для систем, требующих высокой доступности и устойчивости к сбоям.

Особое внимание было уделено модульной организации кода и автоматизации сборки проекта с использованием *makefile*, что значительно упрощает тестирование и дальнейшее сопровождение разработанного решения. Программа корректно обрабатывает возможные ошибки, такие как неудачный вызов *fork* или проблемы с записью в файл, что повышает её надёжность и стабильность работы.

Таким образом, поставленные задачи лабораторной работы были успешно решены. Полученные знания и практические навыки в области системного программирования в *Unix*-среде могут быть успешно применены для разработки более сложных консольных приложений и отказоустойчивых систем автоматизации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] GCC Online Documentation [Электронный ресурс]. – Режим доступа: https://gcc.gnu.org/onlinedocs/. – Дата доступа: 02.03.2025.

[2] man7.org. «sigaction(2)» [Электронный ресурс]. – Режим доступа: https://man7.org/linux/man-pages/man2/sigaction.2.html. – Дата доступа: 03.03.2025.

[3] GNU Make Manual [Электронный ресурс]. – Режим доступа: https://www.gnu.org/software/make/manual/. – Дата доступа: 03.03.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Исходный код программы

```
#include <csignal>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <unistd.h>

volatile sig_atomic_t counter = 0;

void handle_signal(int sig) {
  std::cout << "\nПолучен сигнал " << sig << ". Запуск нового процесса..."
            << std::endl;

  pid_t pid = fork();
  if (pid < 0) {
    perror("Ошибка fork");
    exit(EXIT_FAILURE);
  } else if (pid > 0) {
    std::cout << "Родительский процесс завершает работу." << std::endl;
    exit(EXIT_SUCCESS);
  } else {
    std::cout << "Новый процесс продолжает выполнение. Счётчик = " << counter
              << std::endl;
  }
}

int main() {
  struct sigaction sa;
  sa.sa_handler = handle_signal;
  sa.sa_flags = SA_RESTART;
  sigemptyset(&sa.sa_mask);

  if (sigaction(SIGTERM, &sa, nullptr) == -1) {
    perror("Ошибка установки обработчика SIGTERM");
    exit(EXIT_FAILURE);
  }
  if (sigaction(SIGINT, &sa, nullptr) == -1) {
    perror("Ошибка установки обработчика SIGINT");
    exit(EXIT_FAILURE);
  }

  while (true) {
    counter++;
    std::cout << "Счётчик: " << counter << std::endl;

    std::ofstream out("counter.txt", std::ios::app);
    if (out) {
      out << "Счётчик: " << counter << std::endl;
    } else {
      perror("Не удалось открыть counter.txt");
    }

    sleep(1);
  }

  return 0;
}
```