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 Лавренова А.С.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

Введение3

[1 [Ц](#_agswffz6g6ei)ель работы](#_Toc178067889) 4

[2 Теоретические сведения 5](#_Toc178067890)

[3 Описание и пример выполнения программы 8](#_Toc178067891)

Заключение [9](#_Toc178067899)

[Список использованных источников 10](#_Toc178067900)

[Приложение А (обязательное) 11](#_Toc178067901)

# ВВЕДЕНИЕ

Понимание механизмов управления процессами и обработки сигналов критически важно как для разработки системного программного обеспечения, так и для отладки и администрирования современных операционных систем. Основное внимание в данной лабораторной работе уделяется созданию демон-процесса, способного реагировать на определённые сигналы и вести их протоколирование, что позволяет моделировать поведение фоновых служб, характерных для UNIX-подобных операционных систем.

Демоны являются неотъемлемой частью архитектуры операционной системы, обеспечивая функционирование таких компонентов, как системные службы, сетевые слушатели и фоновая обработка задач. Как правило, они запускаются один раз и продолжают работать бессрочно, реагируя на внешние сигналы и события. В данной работе имитируется реализация подобного процесса, который, после соответствующей демонизации, отслеживает и записывает поступающие к нему сигналы в журнал, обеспечивая тем самым как контроль, так и возможность последующего анализа поведения системы. Это особенно важно при создании отказоустойчивых приложений и систем, которые должны надёжно функционировать в условиях непредсказуемых внешних воздействий.

Важной особенностью данной реализации является использование конфигурационного файла, в котором задаётся список сигналов, подлежащих протоколированию. Дополнительно реализован механизм реактивной реконфигурации, когда демон может перечитать настройки по сигналу SIGHUP, что делает поведение приложения ближе к настоящим системным службам. Также предусмотрен способ корректного завершения работы демона через команду --quit, что позволяет управлять его жизненным циклом из командной строки без принудительного завершения процесса.

Лабораторная работа включает все ключевые этапы системной разработки: от создания и демонизации процесса до проектирования интерфейса взаимодействия с пользователем и построения автоматического тестирования. Через реализацию обработчиков сигналов, работу с PID-файлом, ведение логов, создание конфигурации и управление жизненным циклом процесса, студент получает на практике представление о том, как функционируют внутренние механизмы операционной системы

Таким образом, данная работа направлена не просто на освоение синтаксиса функций fork, signal, kill, setsid и других средств взаимодействия, но на осознание их роли в обеспечении надежности и масштабируемости многопроцессных систем. Полученные знания и навыки могут быть полезны при создании серверов, агентов мониторинга, средств администрирования и других компонентов системного уровня, а также в рамках будущих дисциплин, связанных с многопоточностью, распределёнными системами и программированием в условиях конкурентного доступа к ресурсам.

1 ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является глубокое практическое освоение механизмов управления процессами и обработки сигналов в операционных системах семейства UNIX через разработку полнофункционального демон-процесса, способного к устойчивому фоновому функционированию, динамической реконфигурации и корректному завершению работы. В рамках поставленной задачи необходимо реализовать программу, выполняющую роль демона, которая после отделения от управляющего терминала и преобразования в самостоятельный системный процесс осуществляет приём, обработку и протоколирование заданных сигналов, отражая тем самым одно из фундаментальных понятий в организации взаимодействия между процессами в многозадачной среде.

Решаемая задача направлена на закрепление знаний о жизненном цикле процессов в UNIX-подобных системах, включая создание дочерних процессов при помощи fork(), отделение от управляющего терминала с помощью setsid(), изменение контекста выполнения, настройку прав доступа, а также управление сигнальной системой через установку собственных обработчиков сигналов. Особое внимание уделяется корректной инициализации демона, обеспечению взаимодействия с пользователем через командную строку, использованию конфигурационного файла для определения логики обработки сигналов и реализации надёжного механизма логирования.

Через разработку данной программы студент не только осваивает синтаксис и особенности системных вызовов, но и учится проектировать и реализовывать устойчивые сервисные процессы, обладающие минимальной зависимостью от окружения и способные корректно функционировать в автономном режиме. Также важно отметить, что лабораторная работа демонстрирует принципы взаимодействия между пользователем и демоном через сигналы (например, SIGHUP для перечитывания конфигурации и SIGTERM для завершения), что позволяет выстраивать гибкую архитектуру управления служебными процессами.

Таким образом, выполнение лабораторной работы позволяет в практическом контексте освоить базовые и прикладные аспекты управления процессами и межпроцессного взаимодействия, получить навыки построения надёжных фоновых служб, разобраться в особенностях сигнальной модели UNIX, а также овладеть методами отладки, тестирования и сопровождения системных приложений.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Linux — это свободная операционная система (точнее, семейство систем) с открытым исходным кодом. Само название Linux относится к ядру системы, которое является ее ключевым компонентом, управляющим центральным процессором (ЦП), памятью и периферийными устройствами компьютера. [1]

Процесс в Linux (как и в UNIX) - это программа, которая выполняется в отдельном виртуальном адресном пространстве. Когда пользователь регистрируется в системе, автоматически создается процесс, в котором выполняется оболочка (shell), например, /bin/bash.

В Linux поддерживается классическая схема мультипрограммирования. Linux поддерживает параллельное (или квазипараллельного при наличии только одного процессора) выполнение процессов пользователя. Каждый процесс выполняется в собственном виртуальном адресном пространстве, т.е. процессы защищены друг от друга и крах одного процесса никак не повлияет на другие выполняющиеся процессы и на всю систему в целом. Один процесс не может прочитать что-либо из памяти (или записать в нее) другого процесса без "разрешения" на то другого процесса. Санкционированные взаимодействия между процессами допускаются системой.

Ядро предоставляет системные вызовы для создания новых процессов и для управления порожденными процессами. Любая программа может начать выполняться только если другой процесс ее запустит или произойдет какое-то прерывание (например, прерывание внешнего устройства).

В связи с развитием SMP (Symmetric Multiprocessor Architectures) в ядро Linux был внедрен механизм нитей или потоков управления (threads). Нить - это процесс, который выполняется в виртуальной памяти, используемой вместе с другими нитями процесса, который обладает отдельной виртуальной памятью.

Если интерпретатору (shell) встречается команда, соответствующая выполняемому файлу, интерпретатор выполняет ее, начиная с точки входа (entry point). Для С-программ entry point - это функция main. Запущенная программа тоже может создать процесс, т.е. запустить какую-то программу и ее выполнение тоже начнется с функции main.

Для создания процессов используются два системных вызова: fork() и exec. fork() создает новое адресное пространство, которое полностью идентично адресному пространству основного процесса. После выполнения этого системного вызова мы получаем два абсолютно одинаковых процесса - основной и порожденный. Функция fork() возвращает 0 в порожденном процессе и PID (Process ID - идентификатор порожденного процесса) - в основном. PID - это целое число. [2]

С годами определение и семантика сигналов изменялись и совершенствовались различными реализациями UNIX. В настоящее время существует 2 класса сигналов: ненадежные (unreliable) и надежные (reliable). Ненадежные сигналы - это те, для которых вызванный однажды обработчик сигнала не остается Такие "сигналы-выстрелы" должны перезапускать обработчик внутри самого обработчика, если есть желание сохранить сигнал действующим. Из-за этого возможна ситуация гонок, в которой сигнал может прийти до перезапуска обработчика - и тогда он будет потерян, или придет вовремя - и тогда сработает в соответствии с заданным поведением (например, убьет процесс). Такие сигналы ненадежны, поскольку отлов сигнала и переинсталяция обработчика не являются атомарными операциями.

В семантике ненадежных процессов системные вызовы не повторяются автоматически будучи прерванными поступившим сигналом. Поэтому для обеспечения отработки всех системных вызовов программа должна проверять значение errno после каждого из них и повторять вызовы, если это значение равно EINTR.

По тем же причинам семантика ненадежных сигналов не предоставляет легкого пути реализации атомарных пауз для усыпления процесса до получения сигнала. Ненадежное поведение постоянно перезапускающегося обработчика может привести к неготовности спящего в нужный момент принять сигнал.

Напротив, семантика надежных сигналов оставляет обработчик проинсталированным и ситуация гонок при перезапуске избегается. В то же время определенные сигналы могут быть запущены заново, а атомарная операция паузы доступна через функцию POSIX sigsuspend.

Система Linux IPC (Inter-process communication) предоставляет средства для взаимодействия процессов между собой.

В распоряжении программистов есть несколько методов IPC:

* полудуплексные каналы UNIX;
* FIFO (именованные каналы);
* очереди сообщений в стиле SYSV;
* множества семафоров в стиле SYSV;
* разделяемые сегменты памяти в стиле SYSV;
* сетевые сокеты (в стиле Berkeley) (не охватывается этой статьей);
* полнодуплексные каналы (каналы потоков) (не охватывается этой статьей). [3]

Если эти возможности эффективно используются, то они обеспечивают солидную базу для поддержания идеологии клиент/сервер в любой UNIX-системе, включая Linux.

В итоге, Linux представляет собой мощную операционную систему, поддерживающую мультипрограммирование и защищенное выполнение процессов. Она предоставляет разработчикам широкий спектр инструментов для создания и управления процессами, включая системные вызовы для их создания и взаимодействия. Механизмы потоков управления и различные методы межпроцессного взаимодействия (IPC) позволяют эффективно разрабатывать многопоточные и распределенные приложения. Освоение этих технологий является ключевым для успешной работы в среде Linux.

**3 ОПИСАНИЕ И ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ**

Разработанная в рамках лабораторной работы программа представляет собой демон-процесс, работающий в фоне и выполняющий одну ключевую функцию — приём и протоколирование сигналов, заданных пользователем через конфигурационный файл. Программа запускается в качестве системного демона, отделяется от управляющего терминала и продолжает функционировать в фоновом режиме. Она записывает события, связанные с обработкой сигналов, в лог-файл с привязкой ко времени. Это поведение характерно для большинства сервисных процессов операционной системы UNIX, таких как cron, sshd, syslogd и других.

Запуск программы осуществляется при помощи команды ./build/lab4. После запуска демон переходит в автономный режим работы. В каталоге log/ создаётся лог-файл signal\_daemon.log, в который в дальнейшем поступают записи о полученных сигналах. В каталоге config/ создаётся конфигурационный файл signals.conf, содержащий список имён сигналов, которые необходимо обрабатывать (рисунок 3.1). В этот список по умолчанию входят сигналы SIGHUP, SIGTERM и SIGUSR1.

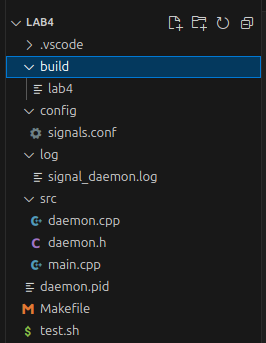


Рисунок 3.1 – Создание каталогов build, log и config

Демон отслеживает указанные сигналы и, при их получении, выполняет соответствующие действия:

1. Если приходит сигнал SIGHUP, происходит повторное чтение конфигурационного файла, что имитирует реконфигурацию демона без его перезапуска.
2. Если приходит SIGTERM, демон корректно завершает работу, удаляя при этом файл с идентификатором процесса (PID).
3. Если приходит SIGUSR1 или другой пользовательский сигнал из конфигурации, он заносится в журнал.

Дополнительно программа поддерживает опцию --quit, которая позволяет пользователю извне инициировать корректное завершение демона. Эта опция реализована в виде интерфейса командной строки и функционирует следующим образом: пользователь вызывает исполняемый файл с аргументом --quit, и в процессе выполнения этой команды в уже работающий демон отправляется сигнал SIGTERM. Таким образом, осуществляется дистанционное завершение процесса без использования прямого kill.

На этапе тестирования было проведено несколько последовательных запусков программы, отправка ей различных сигналов, а также проверка реакции на изменения конфигурационного файла. Результаты работы программы были зафиксированы в лог-файле (рисунок3.2).

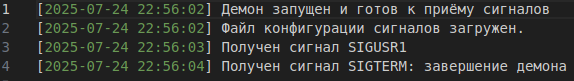


Рисунок 3.2 – Лог-файл

Дополнительно к ручному тестированию взаимодействия с демон-процессом была реализована автоматизированная проверка его поведения с помощью bash-скрипта test.sh, размещённого в корне проекта. Этот скрипт позволяет воспроизвести все основные действия, предусмотренные заданием лабораторной работы, и получить воспроизводимые результаты, пригодные как для отладки, так и для демонстрации.

Скрипт выполняет следующие действия в автоматическом режиме:

До выполнения скрипта все каталоги (build, log, config) и лог-файлы удаляются для чистоты эксперимента. Затем производится сборка проекта через make, запускается демон, считывается его PID из daemon.pid, и в демон отправляются три сигнала: SIGUSR1, SIGHUP и SIGTERM. После каждого сигнала скрипт делает небольшую задержку (в 1–2 секунды), чтобы дать демону время обработать события и записать их в лог. По завершении, скрипт выводит содержимое лог-файла, подтверждая корректную работу всей цепочки сигналов и реакций.

Рисунок 3.3 иллюстрирует автоматизированную демонстрацию всего цикла жизни демона: запуск, реакция на сигналы и корректное завершение. Это подтверждает надёжность реализации, её соответствие требованиям задания и даёт возможность быстро перепроверить работоспособность без ручного вмешательства. Данный тест особенно полезен при изменениях в логике программы или переносе на другую систему — он позволяет сразу убедиться, что демон функционирует должным образом.

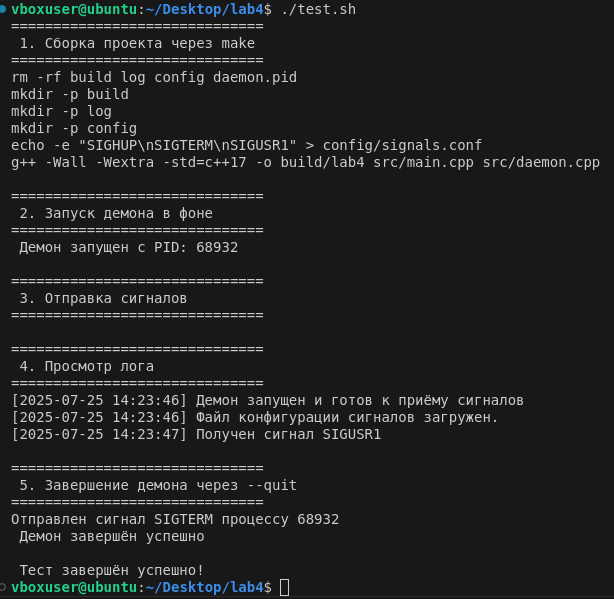


Рисунок 3.3 – Отработка теста

Таким образом, в рамках выполнения лабораторной работы была продемонстрирована полноценная реализация и тестирование фонового демона, способного к приёму, обработке и протоколированию сигналов. Поведение программы соответствует требованиям задания и имитирует реальную работу сервисных процессов в операционных системах UNIX. Тестирование показало корректную работу всех функций, включая реконфигурацию, логирование, завершение и взаимодействие через пользовательский интерфейс.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была достигнута поставленная цель — спроектирован, реализован и протестирован демон-процесс, работающий в фоновом режиме и осуществляющий приём, обработку и протоколирование сигналов, заданных в конфигурационном файле. Программа демонстрирует все характерные особенности системных демонов в UNIX-подобных операционных системах, включая инициализацию с отсоединением от управляющего терминала, перенаправление стандартных потоков, управление PID-файлом, работу с лог-файлом, а также гибкость через динамическую реконфигурацию по сигналу SIGHUP.

Особое внимание в реализации уделено корректному взаимодействию с операционной системой на низком уровне. Были использованы системные вызовы fork, setsid, signal, kill, open, close и другие, что позволило глубже понять внутренние механизмы работы процессов в многозадачной среде. На практике реализован шаблон поведения, характерный для настоящих системных служб. Это формирует представление о структуре и жизненном цикле фоновых сервисов, а также об их роли в организации управляемой, стабильной и отказоустойчивой системы.

Важной особенностью данной работы является взаимодействие между пользователем и процессом при помощи сигналов, которое реализовано не только через стандартную команду kill, но и через пользовательский интерфейс в виде ключей командной строки. Благодаря этому демон может быть остановлен через --quit, что повышает удобство и безопасность управления процессом, особенно в условиях многопользовательской системы. Подобная организация, где внешний интерфейс «оборачивается» в передачу сигналов, приближает поведение программы к тому, как работают настоящие UNIX-сервисы.

В процессе тестирования была не только проверена корректность реакции на сигналы, но и реализован полноценный автоматизированный скрипт, воспроизводящий полный жизненный цикл демона: от сборки и запуска до приёма сигналов и завершения работы. Это даёт возможность в будущем легко повторно проверить работоспособность при переносе или модификации кода.

Таким образом, выполненная работа имеет не только учебную, но и практическую ценность. Она не просто иллюстрирует теоретические знания о сигнальной системе и управлении процессами, но и даёт конкретный, воспроизводимый пример фоновой службы, пригодный для адаптации под реальную задачу. Полученные навыки будут полезны при разработке серверов, мониторинговых агентов, обработчиков событий и других системных компонентов, в которых требуется автономность, стабильность и предсказуемое поведение в условиях взаимодействия с внешними источниками сигналов. Работа закладывает прочную основу для понимания механизмов взаимодействия между процессами и уверенной работы с прикладным и системным программированием в UNIX-среде.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Linux и UNIX: программирование в shell. Руководство разработчика. Дэвид Тейнсли – [Электронный ресурс]. – Режим доступа: https://coollib. com/b/275969/read?ysclid=m6ujdy2lag999562018.

[2] Внутреннее устройство Linux. Дмитрий Кетов – [Электронный ресурс]. – Режим доступа: https://djvu.online/file/wkYDVsl8LdABg?ysclid=mdi of2iooo685605108.

[3] UNIX. Профессиональное программирование. У. Ричард Стивенс, Стивен А. Раго – [Электронный ресурс]. – Режим доступа: http://amplab.ru/bo oks/7.pdf.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код

#include "daemon.h"

#include <iostream>

#include <fstream>

#include <csignal>

#include <unistd.h>

#include <cstring>

#include <cstdlib>

#include <ctime>

#include <sys/stat.h>

#include <set>

#include <map>

std::ofstream logFile;

std::set<int> handledSignals;

std::map<std::string, int> signalMap = {

{"SIGHUP", SIGHUP},

{"SIGTERM", SIGTERM},

{"SIGUSR1", SIGUSR1},

{"SIGUSR2", SIGUSR2}

};

std::string getTime() {

time\_t now = time(nullptr);

char buf[64];

strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S", localtime(&now));

return std::string(buf);

}

void writeLog(const std::string& message) {

logFile << "[" << getTime() << "] " << message << std::endl;

logFile.flush();

}

void loadSignalConfig() {

handledSignals.clear();

std::ifstream conf("/home/vboxuser/Desktop/lab4/config/signals.conf");

if (!conf) {

writeLog("Не удалось открыть config/signals.conf. Обрабатываем все сигналы по умолчанию.");

handledSignals = {SIGHUP, SIGTERM, SIGUSR1, SIGUSR2};

return;

}

std::string line;

while (std::getline(conf, line)) {

if (signalMap.count(line)) {

handledSignals.insert(signalMap[line]);

}

}

writeLog("Файл конфигурации сигналов загружен.");

}

void setupSignalHandlers() {

for (int sig : handledSignals) {

signal(sig, signalHandler);

}

}

void signalHandler(int sig) {

if (!handledSignals.count(sig)) return;

switch (sig) {

case SIGHUP:

writeLog("Получен сигнал SIGHUP: перечитываем конфигурацию");

loadSignalConfig();

setupSignalHandlers();

break;

case SIGTERM:

writeLog("Получен сигнал SIGTERM: завершение демона");

logFile.close();

remove("daemon.pid");

exit(EXIT\_SUCCESS);

break;

case SIGUSR1:

writeLog("Получен сигнал SIGUSR1");

break;

case SIGUSR2:

writeLog("Получен сигнал SIGUSR2");

break;

default:

writeLog("Получен неизвестный сигнал №" + std::to\_string(sig));

break;

}

}

void daemonize() {

pid\_t pid = fork();

if (pid < 0) exit(EXIT\_FAILURE);

if (pid > 0) exit(EXIT\_SUCCESS);

if (setsid() < 0) exit(EXIT\_FAILURE);

signal(SIGCHLD, SIG\_IGN);

signal(SIGHUP, SIG\_IGN);

pid = fork();

if (pid < 0) exit(EXIT\_FAILURE);

if (pid > 0) exit(EXIT\_SUCCESS);

umask(0);

chdir("/");

close(STDIN\_FILENO);

close(STDOUT\_FILENO);

close(STDERR\_FILENO);

logFile.open("/home/vboxuser/Desktop/lab4/log/signal\_daemon.log", std::ios::out | std::ios::trunc);

if (!logFile) exit(EXIT\_FAILURE);

std::ofstream pidFile("/home/vboxuser/Desktop/lab4/daemon.pid");

pidFile << getpid();

pidFile.close();

writeLog("Демон запущен и готов к приёму сигналов");

}