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:

студент гр. 153504

Сивый А. А.

Проверил:

Гриценко Н. Ю.

Минск 2024

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc146631498)

[2 Краткие теоретические сведения 4](#_Toc146631499)

[3 Полученные результаты 5](#_Toc146631500)

[Выводы 6](#_Toc146631501)

[Список использованных источников 7](#_Toc146631502)

[Приложение А (обязательное) Листинг исходного кода 8](#_Toc146631503)

## 1 ЦЕЛЬ РАБОТЫ

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

В Unix процессы играют ключевую роль, поскольку они представляют собой исполняемые программы, которые выполняются в системе. Управление процессами в Unix осуществляется с помощью системных вызовов.

Создание процессов в Unix операционных системах осуществляется с помощью системного вызова fork(). Этот вызов создает точную копию вызывающего процесса, включая его состояние и память. После успешного завершения вызова fork(), родительский процесс получает PID (идентификатор) нового дочернего процесса, а сам дочерний процесс получает значение 0. Таким образом, процессы могут определить, являются ли они родительскими или дочерними, используя возвращаемое значение fork(). Пример использования fork() включает вызов fork() в коде программы, где родительский процесс может использовать возвращенный PID для управления дочерним процессом, а дочерний процесс может начать выполнение своего кода, заменив его с помощью exec().

Сигналы в операционной системе Unix представляют собой механизм асинхронной коммуникации между процессами или ядром операционной системы и процессом. Они используются для уведомления процессов о различных событиях, таких как завершение другого процесса, нажатие пользовательской клавиши или изменение состояния процесса. Каждый сигнал определяется уникальным целым числом и обрабатывается соответствующим образом: процесс может игнорировать сигнал, обработать его стандартным образом или установить собственный обработчик с помощью функции signal(). Некоторые из наиболее распространенных сигналов включают SIGINT (сигнал прерывания, обычно отправляемый при нажатии клавиши прерывания, например, Ctrl+C), SIGKILL (сигнал для немедленного завершения процесса) и SIGTERM (сигнал для завершения процесса). Сигналы являются важным инструментом для обработки событий и управления поведением процессов в Unix.

Процессы-демоны (daemons) в операционной системе Unix представляют собой фоновые процессы, которые работают независимо от конкретного пользователя или терминала. Они обычно запускаются во время загрузки системы и выполняют различные системные задачи, такие как обработка запросов, ведение журналов или обслуживание сетевых соединений. Процессы-демоны не имеют привязки к интерактивной сессии пользователя и обычно запускаются с правами доступа, которые минимизируют возможность злоупотребления. Для создания процесса-демона обычно используется следующий алгоритм: сначала процесс-демон создает дочерний процесс с помощью вызова fork(), а затем дочерний процесс вызывает setsid(), чтобы создать новую сессию и стать лидером новой группы процессов. После этого дочерний процесс закрывает все файловые дескрипторы, кроме необходимых для работы, меняет текущий каталог на корневой и перенаправляет стандартные потоки ввода-вывода на /dev/null.

## 3 ПОЛУЧЕННЫЕ РЕЗУЛЬТАТЫ

В результате выполнения лабораторной работы был создан программный продукт, который представляет собой процесс-демон для протоколирования сигналов. В конфигурационном файле укажем номера сигналов SIGUSR1(10) и SIGUSR2(12) как на рисунке 3.1, после чего запустим программу. После чего убедимся, что процесс-демон запущен, с помощью команды, представленной на рисунке 3.2. Отравим сигналы процессу (рисунок 3.3), после чего получим текст файла логов, представленный на рисунке 3.4.

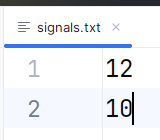


Рисунок 3.1 – Конфигурационный файл

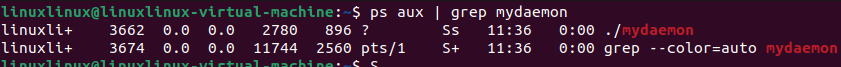


Рисунок 3.2 – Демонстрация работы процесса-демона

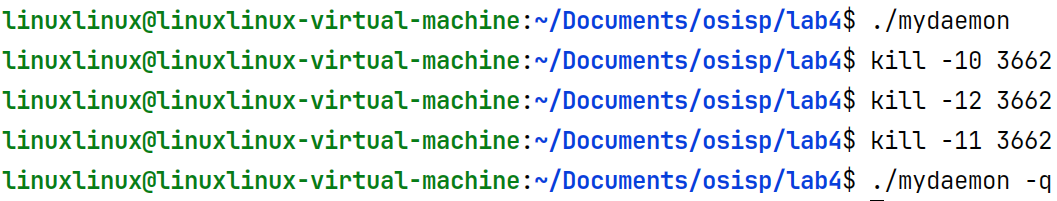


Рисунок 3.3 – Отправка сигналов процессу

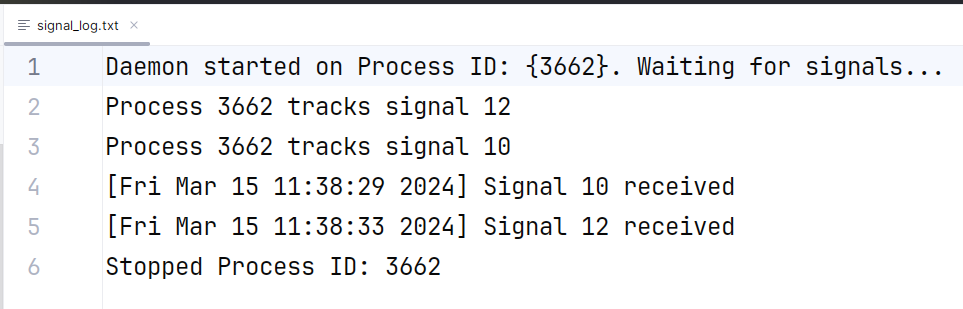


Рисунок 3.4 – Лог процесса

Также предусмотрена возможность выгрузить процесс-демон из памяти, с использованием транзитивного процесса.

## ВЫВОДЫ

В результате выполнения лабораторной работы по управлению процессами и их взаимодействию в Unix были освоены основные принципы работы с процессами. Были изучены способы создания, управления и завершения процессов с использованием системных вызовов, а также использования различных механизмов межпроцессного взаимодействия.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Брайан Керниган, Роб Пайк. UNIX — универсальная среда программирования. – М.: Изд. Дом Вильямс, 1992. – 230 с.

[2] Руководство по Си [Электронный ресурс] – Режим доступа: <https://metanit.com/c/tutorial/> – Дата доступа: 01.03.2024

[3] Makefile Tutorial [Электронный ресурс] – Режим доступа: <https://makefiletutorial.com/> – Дата доступа: 01.03.2024

**ПРИЛОЖЕНИЕ А**

## (обязательное)

## Листинг исходного кода

Листинг 1 – Программный код разработанного приложения

#include "main.h"

int signals[MAX\_SIGNALS];

int num\_signals = 0;

void read\_config\_file() {

FILE\* file = fopen(CONFIG\_FILE, "r");

if (!file) {

exit(1);

}

num\_signals = 0;

char buffer[256];

while(fgets(buffer, sizeof(buffer), file) != NULL && num\_signals < MAX\_SIGNALS) {

signals[num\_signals++] = atoi(buffer);

}

fclose(file);

}

void log\_signal(int sig\_num) {

FILE\* file = fopen(LOG\_FILE, "a");

if (!file) {

exit(1);

}

time\_t current\_time = time(NULL);

char\* c\_time\_string = ctime(&current\_time);

c\_time\_string[strlen(c\_time\_string) - 1] = '\0';

fprintf(file, "[%s] Signal %d received\n", c\_time\_string, sig\_num);

fclose(file);

}

void sighup\_handler(int sig\_num) {

setup\_signal\_handlers();

}

void sigterm\_handler(int sig\_num) {

pid\_t daemon\_pid = getpid();

FILE\* file = fopen(LOG\_FILE, "a");

if (!file) {

exit(1);

}

fprintf(file, "Stopped Process ID: %d", daemon\_pid);

fclose(file);

exit(0);

}

void setup\_signal\_handlers() {

for (int i = 1; i < NSIG; i++) {

signal(i, SIG\_IGN);

}

// Установка обработчика сигнала SIGHUP

signal(SIGHUP, sighup\_handler);

// Установка обработчика сигнала SIGTERM

signal(SIGTERM, sigterm\_handler);

read\_config\_file();

FILE\* file = fopen(LOG\_FILE, "a");

if (!file) {

exit(1);

}

for (int i = 0; i < num\_signals; i++) {

signal(signals[i], log\_signal);

fprintf(file, "Process %d tracks signal %d\n", getpid(), signals[i]);

}

fclose(file);

}

void unload\_daemon() {

pid\_t daemon\_pid = extract\_pid(LOG\_FILE);

kill(daemon\_pid, SIGTERM);

exit(0);

}

int main(int argc, char\* argv[]) {

// Проверка опций командной строки

if (argc > 1 && (strcmp(argv[1], "-q") == 0 || strcmp(argv[1], "--quit") == 0)) {

unload\_daemon();

}

pid\_t pid, sid;

pid = fork(); // создание дочернего процесса

if (pid < 0) { // ошибка при создании процесса

exit(1);

}

if (pid > 0) { // завершение если родительский процесс

exit(0);

}

sid = setsid(); // создание новой сессии

if (sid < 0) { // ошибка при создании сессии

exit(1);

}

// Установка маски создания файлов

umask(0);

// Закрытие стандартных потоков ввода/вывода/ошибок

close(STDIN\_FILENO);

close(STDOUT\_FILENO);

close(STDERR\_FILENO);

write\_start\_message(LOG\_FILE, sid);

setup\_signal\_handlers();

while (1) {

sleep(DELAY);

}

}

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

static char\* read\_file(const char\* filename) {

FILE\* file = fopen(filename, "r");

if (!file) {

fprintf(stderr, "Error reading file\n");

exit(1);

}

fseek(file, 0, SEEK\_END);

long size = ftell(file);

fseek(file, 0, SEEK\_SET);

char\* buffer = (char\*)malloc(size + 1);

fread(buffer, 1, size, file);

buffer[size] = '\0';

fclose(file);

return buffer;

}

static void write\_file(const char\* filename, char\* text, const char\* mode) {

FILE\* file = fopen(filename, mode);

if (!file) {

exit(1);

}

fprintf(file, "%s", text);

fclose(file);

}

static void write\_start\_message(const char\* filename, pid\_t pid) {

FILE\* file = fopen(filename, "w");

if (!file) {

exit(1);

}

fprintf(file, "Daemon started on Process ID: {%d}. Waiting for signals...\n", pid);

fclose(file);

}

static int extract\_pid(const char\* filename) {

char\* line = read\_file(filename);

const char \*start = strstr(line, "{");

if (start != NULL) {

int pid;

if (sscanf(start, "{%d}", &pid) == 1) {

free(line);

return pid;

}

}

free(line);

return -1; // PID не найден или неверный формат

}