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             В. Д. Ключинский

Проверил                          Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc158376840)

[2 Краткие теоретические сведения 4](#_Toc158376841)

[3 Полученные результаты 6](#_Toc158376842)

[Выводы 9](#_Toc158376843)

[Список использованных источников 10](#_Toc158376844)

[Приложение А (обязательное) Листинг исходного кода 11](#_Toc158376845)

**1 ЦЕЛЬ РАБОТЫ**

Целью выполнения лабораторной работы является изучение основных особенностей подсистемы управления процессами и средств взаимодействия процессов в *Linux*. Практическое проектирование, реализация и отладка программных комплексов из нескольких взаимодействующих процессов. В данной работе будет изучена работа процесса-демона, который анализирует файлы системы в многопоточном режиме.

**2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Системное программное обеспечение взаимодействует с ядром системы посредством специальных функций – системных вызовов. В редких случаях существует альтернативный *API*, например, *procfs* или *sysfs*, выполненные в виде виртуальных файловых систем.

Процесс в ядре представляется просто как структура с множеством полей. Только один процесс в системе рождается особенным способом – init. Он порождается непосредственно ядром. Все остальные процессы появляются путём дублирования текущего процесса с помощью системного вызова *fork*.[1]

После выполнения *fork* все ресурсы дочернего процесса – это копия ресурсов родителя. Копировать процесс со всеми выделенными страницами памяти дорого, поэтому в ядре *Linux* используется технология *Copy-On-Write*.

Все страницы памяти родителя помечаются как *read-only* и становятся доступны и родителю, и ребёнку. Как только один из процессов изменяет данные на определённой странице, эта страница не изменяется, а копируется и изменяется уже копия. Оригинал при этом «отвязывается» от данного процесса. Как только *read-only* оригинал остаётся «привязанным» к одному процессу, странице вновь назначается статус *read-write*.[2]

Сразу после выполнения *fork* процесс переходит в состояние «готов». Фактически, процесс стоит в очереди и ждёт, когда планировщик в ядре даст процессу выполняться на процессоре. Как только планировщик поставил процесс на выполнение, началось состояние «выполняется». Процесс может выполняться весь предложенный промежуток (квант) времени, а может уступить место другим процессам, воспользовавшись системным вывозом *sched\_yield*.[3]

Процессы в *Unix* являются основными единицами выполнения программ. Каждый процесс имеет свой адресный простор, стек, код, данные и другие ресурсы. Процессы могут быть созданы с помощью системного вызова *fork()*, который создает копию текущего процесса.

Потоки в *Unix* являются более лёгкими и эффективными единицами выполнения, чем процессы. Потоки разделяют адресный простор, код, данные и другие ресурсы с другими потоками в том же процессе. Потоки могут быть созданы с помощью системного вызова *pthread\_create()*.

Синхронизация процессов и потоков. В *Unix*, процессы и потоки могут синхронизироваться с помощью различных средств, таких как мьютексы, семафоры, барьеры и другие. Мьютексы используются для предотвращения одновременного доступа к разделяемым данным, а семафоры используются для управления доступом к разделяемым ресурсам.

Средства взаимодействия процессов.  В  *Unix*, процессы могут взаимодействовать друг с другом с помощью различных средств, таких как каналы, сокеты, сигналы и другие. Каналы используются для передачи данных между процессами в одном компьютере, а сокеты используются для передачи данных между процессами на разных компьютерах по сети. В *Unix*, процессы и потоки могут выполнять различные системные вызовы для управления процессами, потоками, файлами, сетью и другими ресурсами.

Процессы-демоны (англ. *daemons*) в *Unix*-подобных операционных системах – это процессы, которые работают в фоновом режиме и выполняют определенные задачи, такие как обслуживание сетевых соединений, обработка системных событий и другие.

# **3 ПОЛУЧЕННЫЕ РЕЗУЛЬТАТЫ**

В результате выполнения лабораторной работы была создана программа в качестве демон-процесса, написанная на языке *C++*, которая сканирует содержимое всех директорий, указанных в конфигурационном файле, и записывает информацию в текстовый файл. Данный процесс обновляет список директорий после получения сигнала *SIGHUP*. Информация об очередной процедуре сканирования записывается в системный файл.

Результаты работы программы и исходные файлы представлены на рисунках ниже.

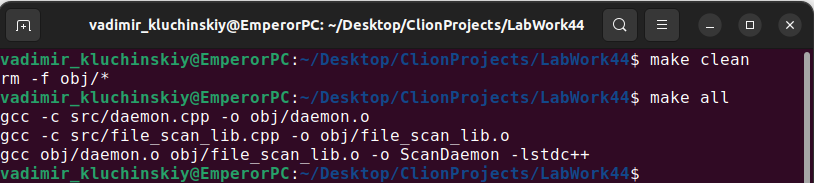


Рисунок 3.1 – Процесс компиляции программы с помощью *makefile*

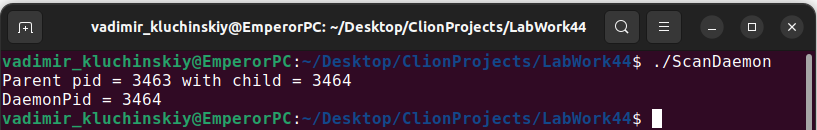


Рисунок 3.2 – Момент запуска демон-процесса

Данный процесс записывает сведения о событиях в файл *sysconf.* Демонстрация приведена на рисунке 3.3:

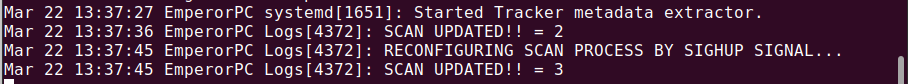


Рисунок 3.3 – Сведения об очередной проверке и переконфигурировании процесса

Данный сканер записывает в текстовый файл информацию о номере выполняющего потока, сигнатуре файла и временной метке. Результат работы приведён на рисунке 3.4:

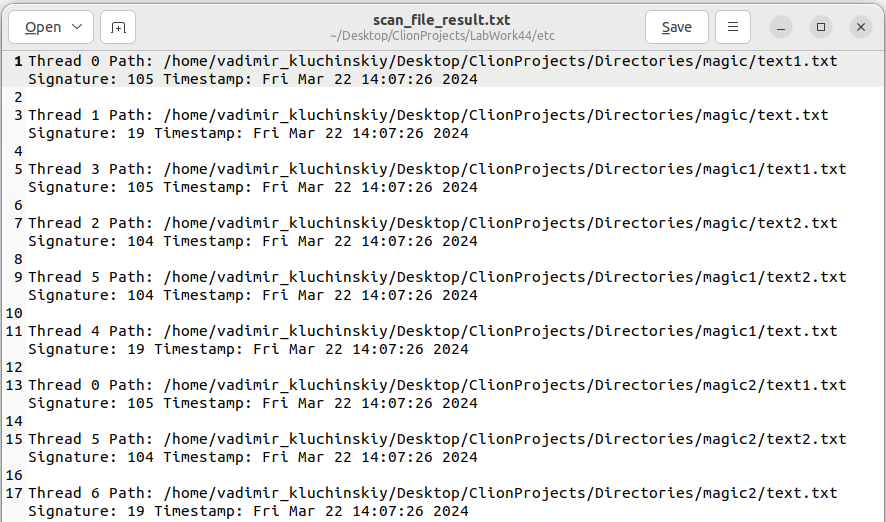


Рисунок 3.4 – Текстовый файл, содержащий информацию о всех файлах в директориях

Через временной интервал в 30 секунд была автоматически проведена повторная проверка. Результаты которой представлены на рисунке 3.5:

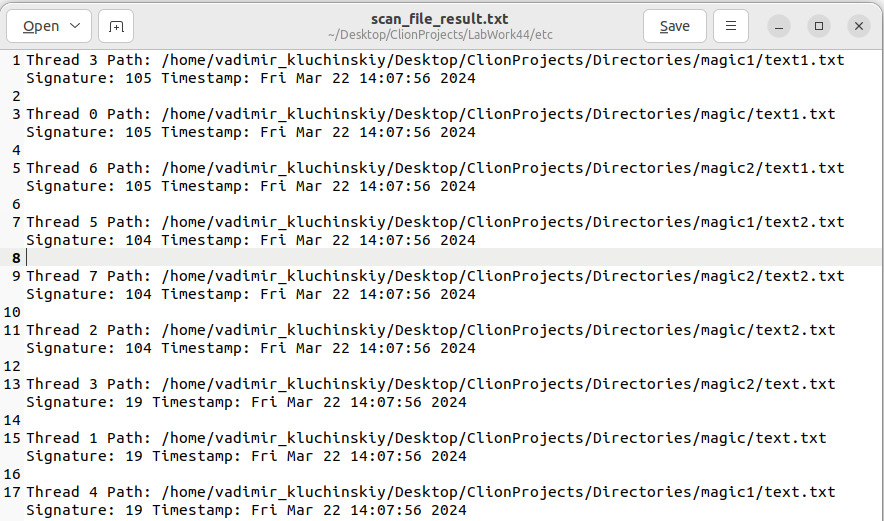


Рисунок 3.5 – Результат повторной проверки

После отправки сигнала *SIGHUP* командой «*kill -HUP 4372*», процесс обновил список требуемых директорий и произвёл новую проверку, результат которой представлен на рисунке 3.6:

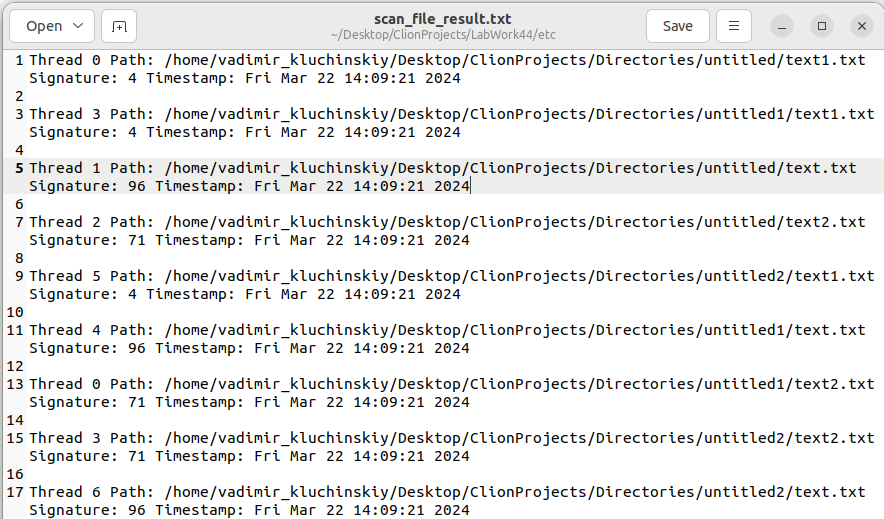


Рисунок 3.6 – Результат проверки после изменения списка директорий и отправки процессу сигнала *SIGHUP*

Ниже представлен текстовый файл со списком директорий:

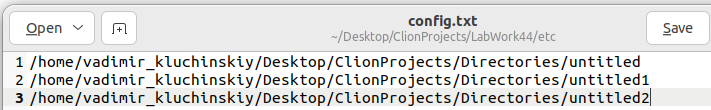


Рисунок 3.7 – Текстовый файл со списком сканируемых директорий

Работу демон-процесса можно безопасно остановить командой *SIGINT*, указав *ID* процесса в *Linux*.



Рисунок 3.8 – Сообщение об остановке демон-процесса

Таким образом, созданная демон-процесс полностью реализует поставленную задачу.

# **ВЫВОДЫ**

В ходе выполнения данной лабораторный работы была создана программа, которая создаёт демон-процесс, параллельно сканирующий файлы в указанных директориях и в их поддиректориях. Для синхронизации потоков к разделяемым блокам кода и файлам был использован объект мьютекс. В демон-процессе были переопределены обработчики сигналов *SIGINT* и *SIGHUP* для управления и переконфигурирования. Был изучен механизм создания дочерних процессов с помощью функции *fork,* а также комбинации прав доступа в файловой системы для пользователей разных категорий.

Для написания программы была выбрана IDE «*Clion*».

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Тейнсли, Д. Linux и другие UNIX-подобные операционные системы: программирование. / Д. Тейнсли. – СПб.: БВХ-Петербург, 2006. – 1056 с.

[2] Щупаков, Ю. Руководство по командам и shell-программированию, операционная система Linux. / Ю. Щупаков. – СПб: Питер, 2008. – 592 с.

[3] UnixTutorials [Электронный ресурс]. – Режим доступа: https://prog /unix-tutorials/. – Дата доступа: 06.02.2024.

# **ПРИЛОЖЕНИЕ А**

# **(обязательное)**

# **Листинг исходного кода**

Листинг 1 – Программный код разработанного приложения:

Файл *file\_scan\_lib.h*:

#ifndef LABWORK4\_FILE\_SCAN\_LIB\_H

#define LABWORK4\_FILE\_SCAN\_LIB\_H

#include <iostream>

#include <fstream>

#include <vector>

#include <cstdint>

#include <cstring>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <queue>

#include <atomic>

#include <chrono>

#include <filesystem>

namespace fs = std::filesystem;

std::uint64\_t calculate\_signature(const fs::path& path);

void worker\_thread(int thread\_id);

int scan\_files(std::vector<std::string>& root\_directories, std::string output\_filename);

#endif //LABWORK4\_FILE\_SCAN\_LIB\_H

Файл *extract\_dictionary\_lib.h*:

#ifndef PROJECT3\_EXTRACT\_DICTIONARY\_LIB\_H

#define PROJECT3\_EXTRACT\_DICTIONARY\_LIB\_H

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "data\_structure.h"

#define MAX\_ARRAY\_LENGTH 1000

void clean\_sources(FILE\* file);

struct Pair\* extract\_text(const char\* filename, int\* array\_size0);

#endif //PROJECT3\_EXTRACT\_DICTIONARY\_LIB\_H

Файл *daemon.cpp*:

#include<stdio.h>

#include<unistd.h> //fork

#include<stdlib.h> //for exit

#include<string.h>

#include<sys/stat.h> //for unmask

#include <sys/types.h>

#include<syslog.h>

#include <signal.h>

#define LOGGING "SCAN UPDATED!! = %d\n"

#include "../headers/file\_scan\_lib.h"

std::string CONFIG\_FILENAME1 = "/home/vadimir\_kluchinskiy/Desktop/ClionProjects/LabWork44/etc/config.txt";

std::string OUTPUT\_FILENAME1 = "/home/vadimir\_kluchinskiy/Desktop/ClionProjects/LabWork44/etc/scan\_file\_result.txt";

std::vector<std::string> RootDirectories;

void set\_directories(){

RootDirectories.clear();

std::string directory;

std::ifstream config\_file(CONFIG\_FILENAME1);

if (config\_file.is\_open()) {

while (getline(config\_file, directory)) {

RootDirectories.push\_back(directory);

}

config\_file.close();

} else {

std::cerr << "Unable to open config";

}

}

void handle\_sigint(int sig) {

printf("SIGINT SIGNAL...\n");

syslog(LOG\_INFO, "FINISH SCAN PROCESS BY SIGINT SIGNAL...");

exit(EXIT\_SUCCESS);

}

void handle\_sighub(int sig){

set\_directories();

syslog(LOG\_INFO, "RECONFIGURING SCAN PROCESS BY SIGHUP SIGNAL...");

}

int main(){

pid\_t pid;

int x\_fd;

//Fork off the parent process

pid = fork();

if(pid<0){

printf("Error occured while creating a child");

exit(EXIT\_FAILURE);

}

else if(pid>0){

printf("Parent pid = %d with child = %d \n",getpid(),pid);

exit(EXIT\_SUCCESS);

}

else {

printf("DaemonPid = %d \n",getpid());

}

if(setsid()<0){

exit(EXIT\_FAILURE);

}

//STEP3

signal(SIGCHLD, SIG\_IGN);

signal(SIGHUP, handle\_sighub);

signal(SIGINT, handle\_sigint);

umask(077);

if ((chdir("/")) < 0) {

printf("Error changing directory!\n");

exit(1);

}

for(x\_fd = sysconf(\_SC\_OPEN\_MAX);x\_fd>=0;--x\_fd){

close(x\_fd);

}

set\_directories();

int count=0;

openlog("Logs", LOG\_PID, LOG\_USER);

while(true){

sleep(10);

scan\_files(RootDirectories,OUTPUT\_FILENAME1);

syslog(LOG\_INFO, LOGGING, ++count);

}

closelog();

return 1;

}