Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Загружаемый модуль ядра, позволяющий скрывать файлы или запрещать их изменение, чтение и удаление»

Студент ИУ7-74Б _____ Д.А. Татаринова
(группа) (Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ Н.Ю.Рязанова
(Подпись, дата) (И.О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Обеспечение безопасного доступа к файлам в операционной системе Linux является одной из актуальных задач. При работе с файлами в Linux необходимо обеспечивать конфиденциальность данных и предоставлять защиту от вредоносных действий.

Данная курсовая работа посвящена разработке модуля, позволяющего ограничивать доступ к определенным файлам.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с задание на курсовую работу необходимо разработать загружаемый модуль ядра для ОС Linux, позволяющий скрывать файлы или запрещать их изменение, чтение и удаление. Предусмотреть возможность ввода пароля для отображения файлов или разрешения операций над ними.

Для достижения поставленной цели необходимо решить следующие задачи:

— изучить принцип работы файловой системы Linux;

— изучить возможности перехвата функций в ядре Linux;

— изучить возможность передачи данных из режима пользователя в режим ядра;

— разработать алгоритмы и структуру программного обеспечения;

— реализовать программное обеспечение;

— протестировать работоспособность разработанного программного обеспечения.

1.2 Файловая система UNIX

Файловая система UNIX представляет собой иерархическую древовидную структуру, состоящую из каталогов и файлов. Начинается она с каталога, который называется корнем (root), а имя этого каталога представлено единственным символом — /.

Каталог представляет собой файл, в котором содержатся каталожные записи. Логически каждую такую запись можно представить в виде структуры, состоящей из имени файла и дополнительной информации, описывающей атрибуты файла. Атрибуты файла — это такие характеристики, как тип файла (обычный файл или каталог), размер файла, владелец файла, права доступа к файлу (есть ли у других пользователей доступ к файлу), время последней модификации файла.

Имена элементов каталога называются именами файлов. Только два символа не могут встречаться в имени файла — это прямой слэш (/) и нулевой символ (\0). Символ слэша разделяет имена файлов, из которых состоит строка пути к файлу, а нулевой символ обозначает конец этой строки. Однако на практике лучше ограничиться подмножеством обычных печатных символов.

Всякий раз, когда создается новый каталог, автоматически создаются два файла: . (называется точка) и .. (называется точка–точка). Под именем «точка» подразумевается текущий каталог, а под именем «точка–точка» — родительский.

В некоторых устаревших версиях UNIX System V длина имени файла ограничена 14 символами. В версиях BSD этот предел был увеличен до 255 символов. Сегодня практически все файловые системы коммерческих версий UNIX поддерживают имена файлов длиной не менее 255 символов.

Сегодня используются самые разные реализации файловых систем UNIX. Например, Solaris поддерживает несколько типов дисковых файловых систем: традиционную для BSD–систем UNIX File System (UFS), DOS–совместимую файловую систему под названием PCFS и файловую систему, предназначенную для компакт–дисков — HSFS.

На рисунке 1.1 представлен диск, поделенный на несколько разделов. Каждый из разделов может содержать файловую систему.

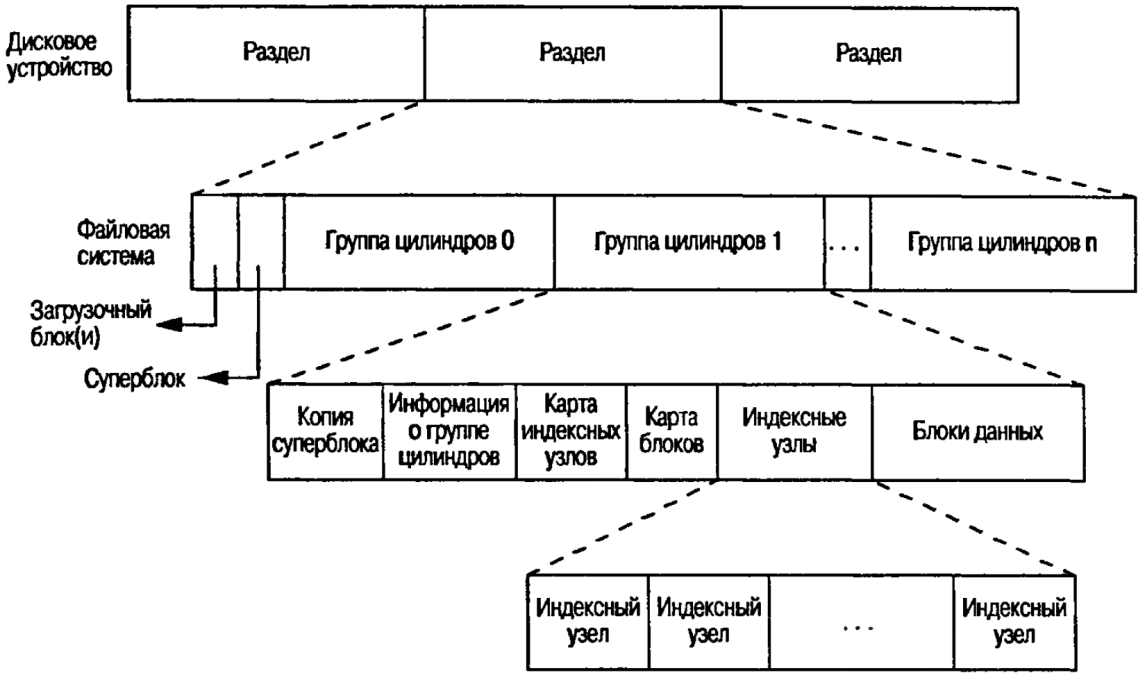

Рисунок 1.1 — Дисковое устройство, разделы и файловая система

Индексные узлы — это записи фиксированной длины, которые содержат большую часть сведений о файлах.

1.3 Файловый ввод–вывод

К операциям файлового ввода–вывода относятся открытие файла, чтение из файла, запись в файл и так далее. Большинство операций файлового ввода–вывода в UNIX можно выполнить с помощью пяти функций: open, read, write, Iseek и close.

Все открытые файлы представлены в ядре файловыми дескрипторами. Файловый дескриптор — это неотрицательное целое число. Когда процесс открывает существующий файл или создает новый, ядро возвращает ему файловый дескриптор. Чтобы выполнить запись в файл или чтение из него, нужно передать функции read или write его файловый дескриптор, полученный в результате вызова функции орел или creat.

В соответствии с принятыми соглашениями командные оболочки UNIX ассоциируют файловый дескриптор 0 со стандартным устройством ввода процесса, 1 — со стандартным устройством вывода и 2 — со стандартным устройством вывода сообщений об ошибках. Это соглашение используется командными оболочками и большинством приложений, но не является особенностью ядра UNIX. Тем не менее многие приложения не смогли бы работать, если это соглашение было бы нарушено.

1.3.1 Функция open

Системный вызов open() открывает файл, указанный в pathname. Если указанный файл не существует, он может (необязательно) (если указан флаг O_CREATE) быть создан open().

Листинг 1.1 — Функция open

```
1 #include <fcntl.h>
2
3 int open (const char *pathname, int flags);
4 int open (const char *pathname, int flags, mode_t mode);
```

Возвращаемое значение open() — дескриптор файла, неотрицательное целое число, которое используется в последующих системных вызовах для работы с файлом.

Параметр flags - это флаги, которые собираются с помощью побитовой операции ИЛИ из таких значений, как:

O_EXEC — открыть только для выполнения (результат не определен, при открытии директории).

O_RDONLY — открыть только на чтение.

O_RDWR — открыть на чтение и запись.

O_SEARCH — открыть директорию только для поиска (результат не определен, при использовании с файлами, не являющимися директорией).

O_WRONLY — открыть только на запись.

O_APPEND — файл открывается в режиме добавления, перед каждой операцией записи файловый указатель будет устанавливаться в конец файла.

O_CLOEXEC — устанавливает флаг close-on-exec для нового файлового дескриптора, указание этого флага позволяет программе избегать дополнительных операций fcntl *F_SETFD* для установки флага *FD_CLOEXEC*.

O_CREAT — если файл не существует, то он будет создан.

O_DIRECTORY — если файл не является каталогом, то open вернёт ошибку.

O_DSYNC — файл открывается в режиме синхронного ввода-вывода (все операции записи для соответствующего дескриптора файла блокируют вызывающий процесс до тех пор, пока данные не будут физически записаны).

O_EXCL — если используется совместно с *O_CREAT*, то при наличии уже созданного файла вызов завершится ошибкой.

O_NOCTTY — если файл указывает на терминальное устройство, то оно не станет терминалом управления процесса, даже при его отсутствии.

O_NOFOLLOW — если файл является символической ссылкой, то open вернёт ошибку.

O_NONBLOCK — файл открывается, по возможности, в режиме non-blocking, то есть никакие последующие операции над дескриптором файла не заставляют в дальнейшем вызывающий процесс ждать.

O_RSYNC — операции записи должны выполняться на том же уровне, что и *O_SYNC*.

O_SYNC — файл открывается в режиме синхронного ввода-вывода (все операции записи для соответствующего дескриптора файла блокируют вызывающий процесс до тех пор, пока данные не будут физически записаны).

O_TRUNC — если файл уже существует, он является обычным файлом и заданный режим позволяет записывать в этот файл, то его длина будет урезана до нуля.

O_LARGEFILE — позволяет открывать файлы, размер которых не может быть представлен типом off_t (long). Для установки должен быть указан макрос _LARGEFILE64_SOURCE

O_TMPFILE — при наличии данного флага создаётся неименованный временный файл.

O_PATH — получить файловый дескриптор, который можно использовать для двух целей: для указания положения в дереве файловой системы и для выполнения операций, работающих исключительно на уровне файловых дескрипторов. Если *O_PATH* указан, то биты флагов, отличные от *O_CLOEXEC*, *O_DIRECTORY* и *O_NOFOLLOW*, игнорируются.

Третий параметр mode всегда должен быть указан при использовании *O_CREAT*; во всех остальных случаях этот параметр игнорируется.

Схема алгоритма работы системного вызова open() представлена на рисунке 1.2.

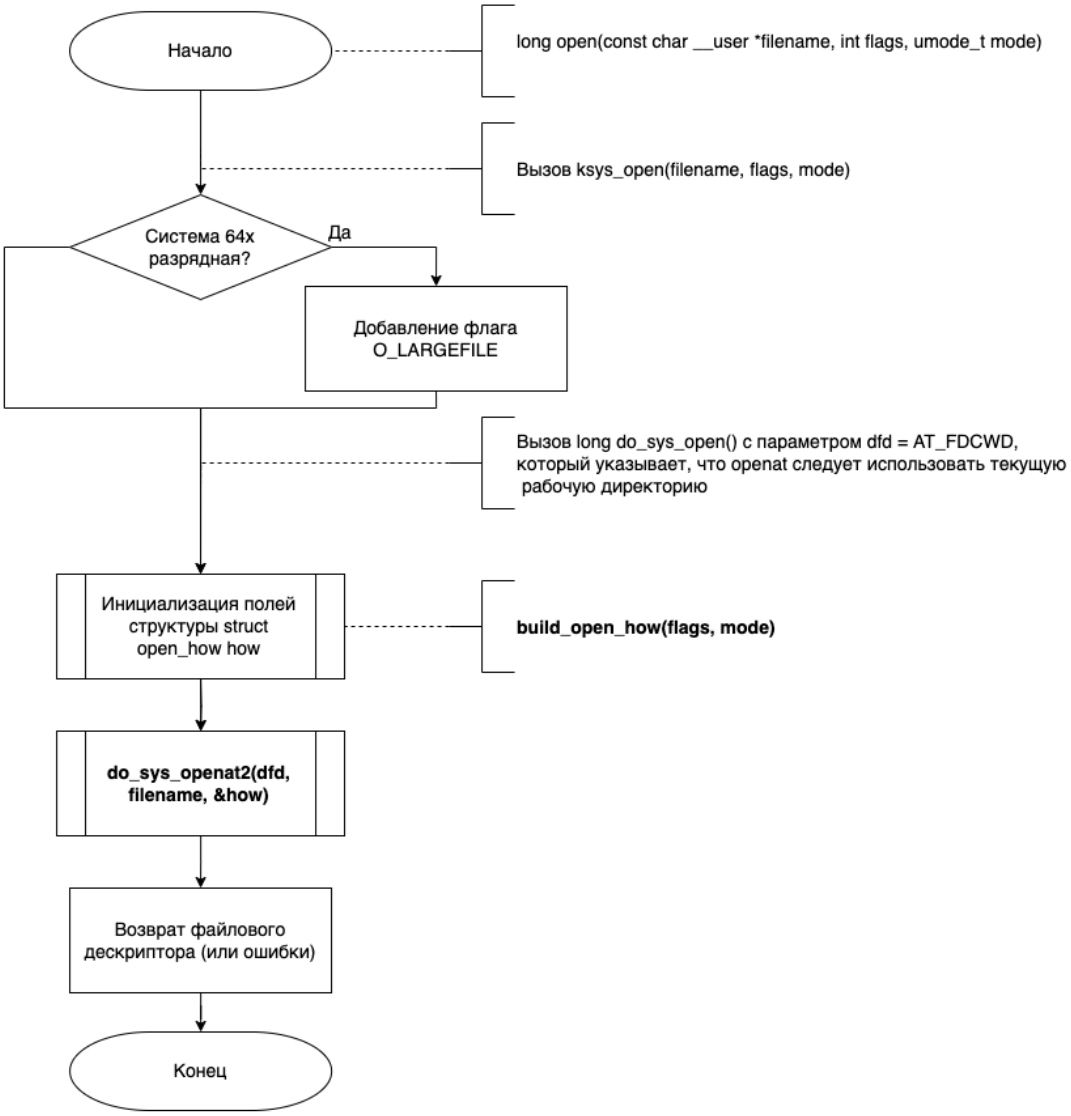

Рисунок 1.2 — Схема алгоритма работы системного вызова open()

Функция open гарантирует, что возвращаемый ею дескриптор файла будет представлять собой наименьшее не используемое в качестве дескриптора положительное число.

1.3.2 Функция write

Запись данных в открытый файл производится функцией write.

Листинг 1.2 — Функция write

```
1 #include <unistd.h>
2
3 ssize_t write(int fd, const void *buf, size_t size);
```

Возвращаемое значение обычно совпадает со значением аргумента size, в противном случае возвращается признак ошибки. Наиболее распространенные случаи, когда возникает ошибка записи, — это переполнение диска или превышение ограничения на размер файла для заданного процесса.

Для обычных файлов запись начинается с текущей позиции файла. Если при открытии файла был указан флаг 0_APPEND, текущая позиция устанавливается в конец файла перед началом каждой операции записи. По окончании записи значение текущей позиции увеличивается на количество фактически записанных байт.

1.3.3 Функции link, unlink

На индексный узел любого файла могут указывать несколько каталожных записей. Такие ссылки создаются с помощью функции link.

Листинг 1.3 — Функция link

```
1 #include <unistd.h>
2
3 int link(const char *existingpath, const char *newpath);
```

Эта функция создает в каталоге новую запись с именем newpath, которая будет указывать на существующий файл existingpath. Если запись с именем newpath уже существует, функция вернет признак ошибки. Создается только последний компонент полного пути newpath, все промежуточные компоненты должны существовать к моменту вызова функции.

Операции создания новой записи в каталоге и увеличения счетчика ссылок должны выполняться атомарно.

Большинство реализаций требуют, чтобы оба пути находились в пределах одной файловой системы, хотя стандарт POSIX.1 допускает возможность создания ссылок на файлы, расположенные в других файловых системах.

Если поддерживается создание жестких ссылок на каталоги, то эта операция может выполняться только суперпользователем. Причина такого ограничения состоит в том ,что создание жесткой ссылки на каталог может привести к появлению замкнутых «петель» в файловой системе, и большинство обслуживающих ее утилит не смогут обработать их надлежащим образом. По этой же причине многие реализации файловых систем вообще не допускают создания жестких ссылок на каталоги.

Удаление записей из каталога производится с помощью функции unlink.

Листинг 1.4 — Функция unlink

```
1 #include <unistd.h>
2
3 int unlink(const char *pathname);
```

Эта функция удаляет запись из файла каталога и уменьшает значение счетчика ссылок на файл pathname. Если на файл указывает несколько ссылок, то его содержимое будет через них по–прежнему доступно. В случае ошибки файл не изменяется.

1.3.4 Функция getdents

Функция getdents6 возвращает записи каталога.

Листинг 1.5 — Функция getdents64

```
1 #include <unistd.h>
2
3 int getdents(unsigned int fd, struct linux_dirent *dirp, unsigned int
      count);
```

Системный вызов getdents() читает несколько структур linux_dirent из каталога, на который указывает открытый файловый дескриптор fd, в буфер, указанный в dirp. В аргументе count задаётся размер этого буфера.

Структура linux_dirent определена следующим образом:

Листинг 1.6 — Структура linux_dirent

```
1 struct linux_dirent {
2     unsigned long   d_ino;
3     unsigned long   d_off;
4     unsigned short  d_reclen;
5     char    d_name[1];
6 };
```

В d_ino указан номер inode. В d_off задается расстояние от начала каталога до начала следующей linux_dirent. В d_reclen указывается размер данного linux_dirent целиком. В d_name задается имя файла, завершающееся null. d_type — тип файла. В нем содержится одно из следующих значений (определённых в <dirent.h>):

— DT_BLK (блочное устройство);

— DT_CHR (символьное устройство);

— DT_DIR (каталог);

— DT_FIFO (именованный канал (FIFO));

— DT_LNK (символическая ссылка);

— DT_REG (обычный файл);

— DT_SOCK (доменный сокет UNIX);

— DT_UNKNOWN (неизвестный тип).

Первоначальный системный вызов Linux getdents() не работал с файловыми системами большого размера и большими смещениями файлов. В связи с этим, в Linux 2.4 была добавлен getdents64().

Системный вызов getdents64() подобен getdents(), за исключением того, что второй аргумент является указателем на буфер, содержащий структуры следующего типа:

Листинг 1.7 — Структура linux_dirent64

```
1 struct linux_dirent64 {
2     ino64_t      d_ino;
3     off64_t      d_off;
4     unsigned short  d_reclen;
5     unsigned char   d_type;
6     char     d_name[];
7 };
```

1.4 Перехват функций в ядре с помощью ftrace

Название ftrace представляет собой сокращение от Function Trace — трассировка функций. Однако возможности этого инструмента гораздо шире: с его помощью можно отслеживать контекстные переключения, измерять время обработки прерываний, высчитывать время на активизацию заданий с высоким приоритетом и многое другое.

Ftrace был разработан Стивеном Ростедтом и добавлен в ядро в 2008 году, начиная с версии 2.6.27. Ftrace — фреймворк, предоставляющий отладочный кольцевой буфер для записи данных. Собирают эти данные встроенные в ядро программы-трассировщики.

Работает ftrace на базе файловой системы debugfs, которая в большинстве современных дистрибутивов Linux смонтирована по умолчанию.

Каждую перехватываемую функцию можно описать следующей структурой:

Листинг 1.8 — Структура ftrace_hook

```
1 struct ftrace_hook {
2     const char *name;
3     void *function;
4     void *original;
5
6     unsigned long address;
7     struct ftrace_ops ops;
8 };
```

Поля структуры:

name — имя перехватываемой функции;

function — адрес функции–обертки, которая будет вызываться вместо перехваченной функции;

original — указатель на место, куда следует записать адрес перехватываемой функции, заполняется при установке;

address — адрес перехватываемой функции, заполняется при установке;

ops — служебная информация ftrace.

Пользователю необходимо заполнить только первые три поля: name, function, original.

Листинг 1.9 — Пример заполнения структуры ftrace_hook

```
1 #define HOOK(_name, _function, _original)       \
2 {                                                 \
3     .name = (_name),                              \
4     .function = (_function),                      \
5     .original = (_original),                      \
6 }
7
8 static struct ftrace_hook hooked_functions[] = {
9     HOOK("sys_clone",   fh_sys_clone,   &real_sys_clone),
10    HOOK("sys_execve",  fh_sys_execve,  &real_sys_execve),
11 };
```

Вывод

В результате проведенного анализа был определен способ перехвата системных вызовов в ядре — путем регистрации функций перехвата с использованием ftrace. Были определены функции, которые необходимо перехватить.

2 Конструкторский раздел

2.1 IDEF0

На рисунке 2.1 приведена диаграмма состояний IDEF0 нулевого уровня, а на рисунке 2.2 — диаграмма состояний IDEF0 первого уровня.

Рисунок 2.1 — Диаграмма состояний IDEF0 нулевого уровня

Рисунок 2.2 — Диаграмма состояний IDEF0 первого уровня

2.2 Алгоритм инициализации модуля

2.3 Алгоритмы функций–оберток

2.4 Структура программного обеспечения

На рисунке 2.3 представлена структура программного обеспечения.

Рисунок 2.3 — Структура программного обеспечения

Вывод

В данном разделе были рассмотрены алгоритмы работы сервера и обработки запроса.

3 Технологический раздел

3.1 Средства и детали реализации

В качестве языка программирования для реализации поставленной задачи был выбран язык Си. Для сборки модуля использовалась утилита make. В качестве среды разработки был выбран VSCode.

3.2 Инициализация модуля

В листинге 3.1 приведена реализация функции инициализации модуля.

Листинг 3.1 — Инициализация модуля

```
1  static int fh_init(void)
2  {
3      struct device *fake_device;
4      int error = 0,err = 0;
5      dev_t devt = 0;
6
7      err = start_hook_resources();
8      if(err)
9          pr_info("Problem in hook functions");
10
11     tidy();
12
13     error = alloc_chrdev_region(&devt, 0, 1, "usb15");
14
15     if (error < 0)
16     {
17         pr_err("Can't get major number\n");
18         return error;
19     }
20
21     major = MAJOR(devt);
22
23     fake_class = class_create(THIS_MODULE, "custom_char_class");
24
25     if (IS_ERR(fake_class)) {
```

```
26          unregister_chrdev_region(MKDEV(major, 0), 1);
27          return PTR_ERR(fake_class);
28      }
29
30      /* Initialize the char device and tie a file_operations to it */
31      cdev_init(&fake_cdev, &fake_fops);
32      fake_cdev.owner = THIS_MODULE;
33      /* Now make the device live for the users to access */
34      cdev_add(&fake_cdev, devt, 1);
35
36      fake_device = device_create(fake_class,
37      NULL,    /* no parent device */
38      devt,     /* associated dev_t */
39      NULL,    /* no additional data */
40      "usb15");  /* device name */
41
42      if (IS_ERR(fake_device))
43      {
44          class_destroy(fake_class);
45          unregister_chrdev_region(devt, 1);
46          return -1;
47      }
48      return 0;
49  }
```

3.3 Инициализация полей структуры ftrace_hook

Инициализация полей структуры ftrace_hook представлена в листинге 3.2.

Листинг 3.2 — Инициализация полей структуры ftrace_hook

```
1  static struct ftrace_hook demo_hooks[] = {
2      HOOK("sys_write", fh_sys_write, &real_sys_write),
3      HOOK("sys_openat", fh_sys_openat, &real_sys_openat),
4      HOOK("sys_unlinkat", fh_sys_unlinkat, &real_sys_unlinkat),
5      HOOK("sys_getdents64", fh_sys_getdents64, &real_sys_getdents64)
6  };
```

3.4 Реализация функций–оберток

Реализация функций–оберток представлена в листингах 3.3–3.6.

Листинг 3.3 — Функция fh_sys_write

```
1  static asmlinkage long fh_sys_write(unsigned int fd, const char __user *buf,
2  size_t count)
3  {
4      long ret;
5      struct task_struct *taskd;
6      struct kernel_siginfo info;
7      int signum = SIGKILL, ret 0;
8      task = current;
9
10     if (task->pid == target_pid)
11     {
12         if (fd == target_fd)
13         {
14             pr_info("write done by process %d to target file.\n",
                   task->pid);
15             memset(&info, 0, sizeof(struct kernel_siginfo));
16             info.si_signo = signum;
17             ret = send_sig_info(signum, &info, task);
18             if (ret < 0)
19             {
20                 printk(KERN_INFO "error sending signal\n");
21             }
22             else
23             {
24                 printk(KERN_INFO "Target has been killed\n");
25                 return 0;
26             }
27         }
28     }
29
30     pr_info("Path debug %s\n", buf);
31     char tmp_path=get_filename(buf);
32     if (check_fs_blocklist(tmp_path))
33     {
34         kfree(tmp_path);
```

```
35          return NULL;
36      }
37      ret = real_sys_write(fd, buf, count);
38
39      return ret;
40  }
```

Листинг 3.4 — Функция fh_sys_openat

```
1  static asmlinkage long fh_sys_openat(int dfd, const char __user *filename,
2  int flags, umode_t mode)
3  {
4       long ret=0;
5       char *kernel_filename;
6       struct task_struct *task;
7       task = current;
8
9       kernel_filename = get_filename(filename);
10
11      if (check_fs_blocklist(kernel_filename))
12      {
13          pr_info("our file is opened by process with id: %d\n", task->pid);
14          pr_info("blocked opened file : %s\n", filename);
15          kfree(kernel_filename);
16          ret = real_sys_openat(dfd, filename, flags, mode);
17          pr_info("fd returned is %ld\n", ret);
18          target_fd = ret;
19          target_pid = task->pid;
20          ret=0;
21          return ret;
22      }
23
24      kfree(kernel_filename);
25      ret = real_sys_openat(filename, flags, mode);
26      return ret;
27  }
```

Листинг 3.5 — Функция fh_sys_unlinkat

```
1  static asmlinkage long fh_sys_unlinkat (int dirfd, const char __user
       *filename, int flags);
2  {
```

```
3       long ret=0;
4       char *kernel_filename = get_filename(filename);
5
6       if (check_fs_blocklist(kernel_filename))
7       {
8           kfree(kernel_filename);
9           pr_info("blocked to not remove file : %s\n", kernel_filename);
10          ret=0;
11          kfree(kernel_filename);
12          return ret;
13      }
14
15      kfree(kernel_filename);
16      ret = real_sys_unlinkat(dirfd,filename, flags);
17      return ret;
18  }
```

Листинг 3.6 — Функция fh_sys_getdents64

```
1  static asmlinkage int fh_sys_getdents64(const struct pt_regs *regs)
2  {
3       struct linux_dirent64 __user *dirent = (struct linux_dirent64
           *)regs->si;
4       struct linux_dirent64 *previous_dir, *current_dir, *dirent_ker = NULL;
5       unsigned long offset = 0;
6       int ret = real_sys_getdents64(regs);
7       dirent_ker = kzalloc(ret, GFP_KERNEL);
8
9       if ( (ret <= 0) || (dirent_ker == NULL) )
10          return ret;
11
12      long error;
13      error = copy_from_user(dirent_ker, dirent, ret);
14
15      if(error)
16          goto done;
17
18      while (offset < ret)
19      {
20          current_dir = (void *)dirent_ker + offset;
21
```

```
22          if (check_fs_hidelist(current_dir->d_name))
23          {
24              if (current_dir == dirent_ker )
25              {
26                  ret -= current_dir->d_reclen;
27                  memmove(current_dir, (void *)current_dir +
                        current_dir->d_reclen, ret);
28                  continue;
29              }
30              previous_dir->d_reclen += current_dir->d_reclen;
31          }
32          else
33          {
34              previous_dir = current_dir;
35          }
36          offset += current_dir->d_reclen;
37      }
38
39      error = copy_to_user(dirent, dirent_ker, ret);
40      if(error)
41      goto done;
42
43  done:
44      kfree(dirent_ker);
45      return ret;
46  }
```

4 Исследовательский раздел

Программное обеспечение было реализовано на дистрибутиве Ubuntu 20.04, ядро версии 5.19.0.

4.1 Пример работы разработанного ПО

Вывод

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы была достигнута цель: разработан статический сервер на языке Си. Были решены следующие задачи:

— проведен анализ предметной области и формализована задача;

— спроектирована структура программного обеспечения;

— реализовано программное обеспечение, которое будет обслуживать контент, хранящийся на диске;

— проведено нагрузочное тестирование и сравнение с NGINX.

В ходе замеров было выявлено, что при увеличении количества клиентов разработанный сервер начинает работать медленнее, чем NGINX.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Среда разработки XCode [Электронный ресурс]. — Режим доступа: https://developer.apple.com/xcode/ (дата обращения: 01.12.2023).

2. Официальный сайт MacOS Ventura [Электронный ресурс]. — Режим доступа: https://www.apple.com/macos/ventura/ (дата обращения: 01.12.2023).

3. Процессор Intel Core i9 7 поколения [Электронный ресурс]. — Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/97539/intel-core-i57260u-processor-4m-cache-up-to-3-40-ghz.html (дата обращения: 13.04.2023).

4. Официальный сайт NGINX [Электронный ресурс]. — Режим доступа: https://nginx.org (дата обращения: 01.12.2023).