Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А

к курсовому проекту на тему:

Мониторинг ОС Linux

Студент _____ Пенской И. С.
(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ Рязанова Н. Ю.
(Подпись, дата) (И.О.Фамилия)

Москва, 2016

Содержание

Введение

Целью проекта является создание программного комплекса, осуществляющего сбор информации о системе Linux, её характеристиках, процессах и запущенных программах. Для демонстрации работы созданного программного комплекса следует также реализовать графический интерфейс, отображающий полученную о системе информацию в удобном для пользователя виде.

Для достижения этой цели необходимо решить следующие задачи: создание загружаемого модуля ядра, задачей которого является сбор информации о запущенных программах, создание приложения, работающего в режиме пользователя, которое получает информацию от модуля ядра и отображает её пользователю, а также занимается сбором информации о системе и процессах.

1 Аналитический раздел

1.1 Обзор технического задания

В соответствии с техническим заданием необходимо разработать программное обеспечение, позволяющее получить информацию о процессах и текущих параметрах системы, а именно:

- имя компьютера;
- имя пользователя;
- время непрерывной работы системы;
- модель процессора;
- частота процессора;
- загруженность процессора;
- объем всей оперативной памяти;
- объем используемой оперативной памяти.

Отдельно получать информацию о запущенных программах, а именно:

- время запуска;
- идентификатор пользователя;
- идентификатор процесса;
- путь к исполняемому файлу;
- аргументы.

1.2 Получение информации о системе в режиме пользователя

Операционная система Linux предоставляет широкие возможности для получения различной системной информации, часто без необходимости при этом обладать правами суперпользователя. Существует множество встроенных утилит и команд, которые позволяют получить подробную информацию о самой операционной системе, а также об аппаратном обеспечении компьютера и т.п.

Утилита *lspci* предназначена для вывода информации обо всех PCI-шинах в системе, а также обо всех устройствах, присоединенных к этим шинам. По умолчанию она показывает краткий список таких устройств. Однако существует возможность использовать многочисленные опции *lspci* для получения более подробной информации или информации, ориентированной на последующую обработку с помощью других программ.

Команда *dmesg* обычно используется в Linux для того, чтобы просмотреть содержимое кольцевого буфера ядра. Она позволяет пользователю вывести содержание сообщений, выдаваемых в процессе загрузки системы. Утилита *lspci* хорошо помогает при обнаружении PCI-устройств, однако нам часто требуется список всех устройств в системе. Используя *dmesg*, мы можем просмотреть характеристики всех устройств, которые обнаружены нашей операционной системой.

Иногда требуется получить информацию об оперативной памяти или центральном процессоре в реальном времени на работающей системе. Для того чтобы сделать это, можно воспользоваться виртуальной файловой системой *procfs* [1]. Также она позволяет получить информацию обо всех процессах, запущенных в системе на данный момент и другую информацию. Выполнив команду *ls* в корневом каталоге *procfs* (обычно */proc*), можно увидеть различные директории и файлы, которые содержат информацию о системе.

Программа *fdisk* - это инструмент для работы с таблицей разбиения диска. Физические диски обычно разбиваются на несколько логических дисков, которые называются разделами диска. Информация о разбиении физического диска на разделы хранится в таблице разбиения диска, которая находится в нулевом секторе физического диска. Если имеется два или более дисков (например, *hda* и *hdb*), и необходимо получить данные о конкретном диске, нужно указать в команде желаемый диск, например *fdisk -l /dev/hda*.

Утилита *dmidecode* выводит содержимое таблицы DMI (Desktop Management Interface) системы в формате, предназначенном для восприятия человеком. Эта таблица содержит информацию, относящуюся к компонентам аппаратного обеспечения системы, а также сведения о версии BIOS и т.д. В выводе dmidecode не только содержится описание текущей конфигурации системы, но и приводятся данные о максимально допустимых значениях параметров, например, о поддерживаемых частотах работы CPU, максимально возможном объеме памяти и так далее.

1.3 Выбор метода получения информации о системе в режиме пользователя

В данной курсовой работе для получения информации о системе и создания первого модуля программы (см. Техническое задание) используется виртуальная файловая система *procfs*, краткое описание которой было дано выше в подразделе 1.2.

Вся необходимая информация (см. 1.1; кроме второго и третьего пунктов, для которых в Linux есть встроенные языковые средства) в удобном виде предоставляется файловой системой *procfs*. Это обосновывает выбор *procfs* как средства получения информации. Важно также отметить, что для получения этой информации не требуются права суперпользователя.

1.4 Получение информации из виртуальной файловой системы *procfs*

Для того чтобы узнать собственное имя компьютера с помощью *procfs*, независимое от сетевых интерфейсов, необходимо прочитать информацию из файла */proc/sys/kernel/hostname*

Для того чтобы узнать модель и частоту центрального процессора, необходимо проанализировать файл */proc/cpuinfo*, в котором хранится информация о процессоре и его состоянии в реальном времени. Пример такого файла представлен на рисунке 1.

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
stepping        : 3
microcode       : 0xffffffff
cpu MHz         : 3491.916
cache size      : 8192 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss
syscall nx lm constant_tsc rep_good nopl eagerfpu pni pclmulqdq ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx
f16c rdrand hypervisor lahf_lm abm fsgsbase bmi1 avx2 smep bmi2 erms xsaveopt
bugs            :
bogomips        : 6983.83
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

Рис. 1. Файл */proc/cpuinfo*.

Модель процессора считывается из поля *model name*, а частота из поля *cpu MHz*.

Для того чтобы вычислить процент загруженности центрального процессора, следует проанализировать файл */proc/stat*, в котором находится информация об активности процессора. Пример такого файла представлен на рисунке 2.

```
cpu  5051 162 861 21898 3990 0 29 0 0 0
cpu0 5051 162 861 21898 3990 0 29 0 0 0
intr 6616 5888 9 0 0 0 0 3 0 0 0 0 0 169 0 0 398 ...
ctxt 665810
btime 1480844493
processes 1932
procs_running 6
procs_blocked 0
softirq 140677 0 27160 3 357 21752 0 58664 0 0 32741
```

Рис. 2. Файл */proc/stat*

Необходимая для вычисления процента загруженности информация хранится в первых четырех полях строчки *cpu*. Перечислим их значение слева направо:

- число процессов, выполняющихся в режиме пользователя;
- число процессов с изменённым приоритетом (*nice*), выполняющихся в режиме пользователя;
- число процессов, выполняющихся в режиме ядра;
- число процессов, выполняющих функцию простоя процессора (*idle*).

Для того чтобы получить информацию об оперативной памяти, следует проанализировать файл */proc/meminfo*. Пример такого файла представлен на рисунке 3.

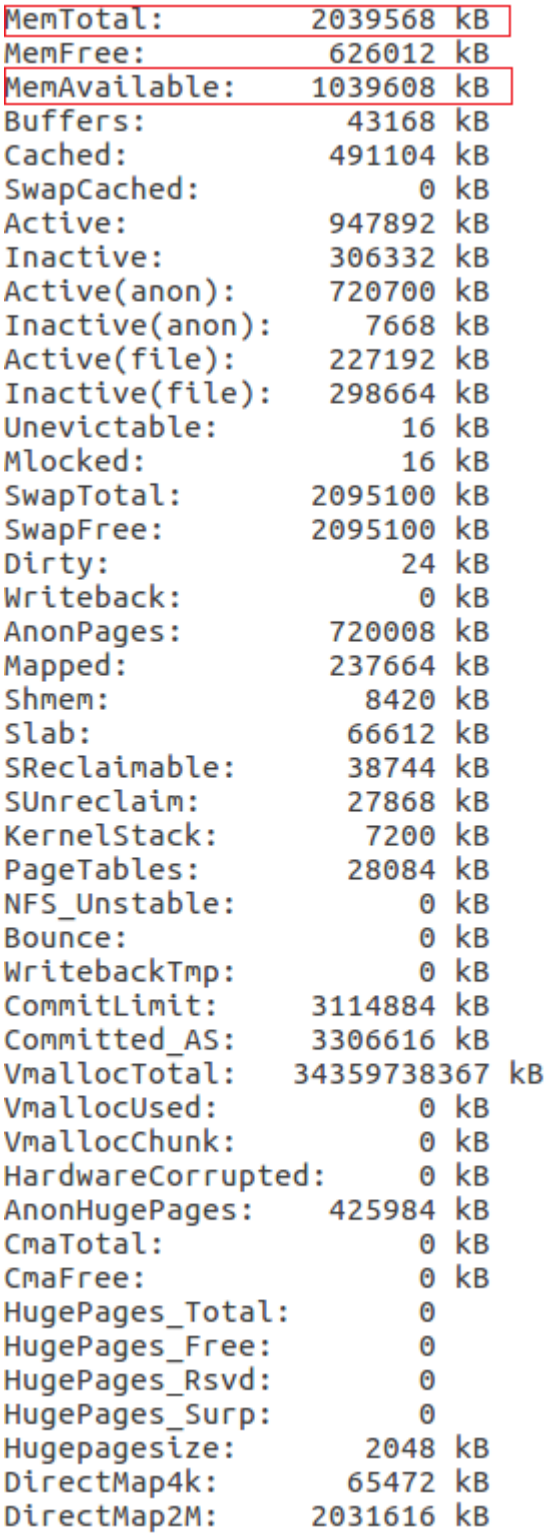

Рис. 3. Файл */proc/meminfo*.

Объем всей оперативной памяти считывается из поля *MemTotal*, а объем памяти, доступной для немедленного её выделения процессам из поля *MemAvailable*. Таким образом, объем используемой оперативной памяти вычисляется из разности этих значений.

Для того чтобы получить информацию о процессах, следует проанализировать содержимое директории */proc*. Все её поддиректории с именем, равным целому числу, обозначают запущенные процессы, а само число — идентификатор (*pid*) процесса. Внутри каждой поддиректории-процессе находится файл *comm*, в котором хранится имя запущенного процесса. Примеры содержимого директории */proc* и директории-процесса представлены на рисунке 4.

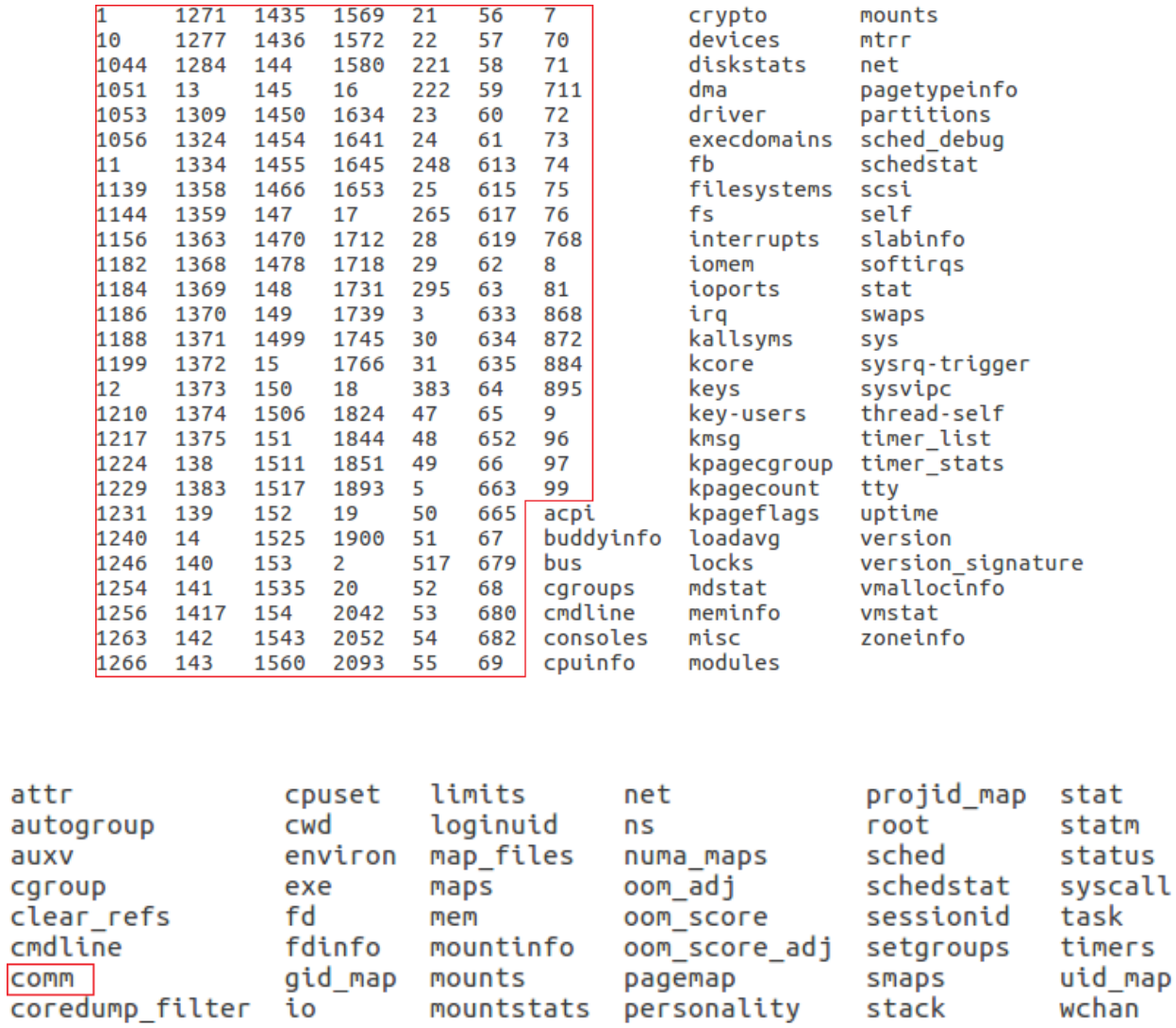

Рис. 4. Директория */proc* и директория-процесс.

1.5 Получение информации о запущенных программах в режиме ядра

Запуск на выполнение любой программы в Linux осуществляется любой из функций семейства системных вызовов *exec()* [3]:

- *execl();*
- *execlp();*
- *execle();*
- *execv();*
- *execvp();*
- *execvpe().*

Следить за вызовом каждого из них отдельно не потребуется, т.к. все они являются обёртками для системного вызова *execve().*

Рассмотрим системный вызов *execve(),* объявление которого представлено в листинге 1.

Листинг 1: Объявление систменого вызова execve

```
1 #include <unistd.h>
2
3 int execve(const char *filename, char *const argv[], char *const envp[]);
```

В результате работы системного вызова *execve()* запускается программа, которая хранится в файле, путь к которому записан в параметре *filename*. Этот файл может быть как бинарным исполняемым файлом, так и скриптом интерпретируемого языка программирования, который начинается со строки вида: *!# <путь к интерпретатору> [<аргументы для интерпретатора>].* Параметр *argv* — массив строк-параметров, переданных программе во время запуска. Параметр *envp* — массив строк, обычно в форме *ключ=значение,* которые передаются программе в момент запуска как параметры среды. Возвращаемое значение отсутствует в случае успеха и равно -1 в случае неудачи, при этом глобальная переменная *errno* принимает значение ошибки.

Первые три пункта необходимой информации (см. 1.1) могут быть получены с помощью языковых средств, четвертый – из параметра *filename*, пятый – из параметра *argv* системного вызова *execve().*

Для того чтобы прочитать данные параметры, следует использовать механизм перехвата системных вызовов в Linux.

1.6 Анализ перехвата системных вызовов в Linux

1.6.1 Перехват с помощью прямого доступа к адресному пространству ядра /dev/kmem

Прямой доступ к адресному пространству ядра обеспечивает файл устройства /dev/kmem. В этом файле отображено все доступное виртуальное адресное пространство, включая раздел подкачки (swap-область). Для работы с файлом kmem используются стандартные системные функции - *open(), read(), write().* Открыв стандартным способом */dev/kmem*, мы можем обратиться к любому адресу в системе, задав его как смещение в этом файле. Обращение к системным функциям осуществляется посредством загрузки параметров функции в регистры процессора и последующим вызовом программного прерывания 0x80. Обработчик этого прерывания, функция *system_call*, помещает параметры вызова в стек, извлекает из таблицы *sys_call_table* адрес вызываемой системной функции и передает управление по этому адресу. Имея полный доступ к адресному пространству ядра, мы можем получить все содержимое таблицы системных вызовов, т.е. адреса всех системных функций. Изменив адрес любого системного вызова, мы, тем самым, осуществим его перехват.

1.6.2 Перехват с помощью загружаемого модуля ядра

Для реализации модуля, перехватывающего системный вызов, предлагается следующий алгоритм:

- сохранить указатель на оригинальный (исходный) вызов для возможности его восстановления;
- создать функцию, реализующую новый системный вызов;
- в таблице системных вызовов *sys_call_table* произвести замену вызовов, т.е. настроить соответствующий указатель на новый системный вызов;
- по окончании работы (при выгрузке модуля) восстановить оригинальный системный вызов, используя ранее сохраненный указатель.

В данной работе используется перехват системного вызова *execve()* с помощью загружаемого модуля ядра.

1.7 Связь модуля ядра с приложением

Для того чтобы приложение могло отобразить информацию о запущенных программах, полученную в загружаемом модуле ядра, необходимо выбрать средство, с помощью которого приложение и модуль ядра смогут передавать друг другу информацию.

Существует несколько возможных вариантов связи режима ядра и режима пользователя:

- писать из модуля ядра напрямую в файл журнала запущенных программ, который будет считываться приложением;
- создать символьное псевдоустройство, в которое модуль ядра будет писать информацию, а приложение — считывать из него.
- использовать механизм сокетов для связи модуля и приложения.

В этой работе используется механизм сокетов.

2 Конструкторский раздел

Программный комплекс в данной курсовой работе состоит из трёх частей:

- приложение с графическим интерфейсом, работающее в режиме пользователя и отображающее информацию о системе;
- программа, работающая в режиме пользователя, связанная с модулем ядра и ведущая журнал запущенных программ;
- загружаемый модуль ядра, непосредственно осуществляющий перехват системного вызова *execve()* и отправляющий информацию в программу-посредника.

2.1 Алгоритм работы приложения с пользовательским интерфейсом

Приложение состоит из четырёх основных процедур:

- обновление информации о состоянии системы;
- обновление информации о процессах;
- обновление журнала запущенных программ;
- завершение процесса по выбору пользователя.

Каждая из этих процедур запускается через определённые промежутки времени. Тогда схема алгоритма работы этого приложения может быть представлена в виде, изображённом на рисунке 5.

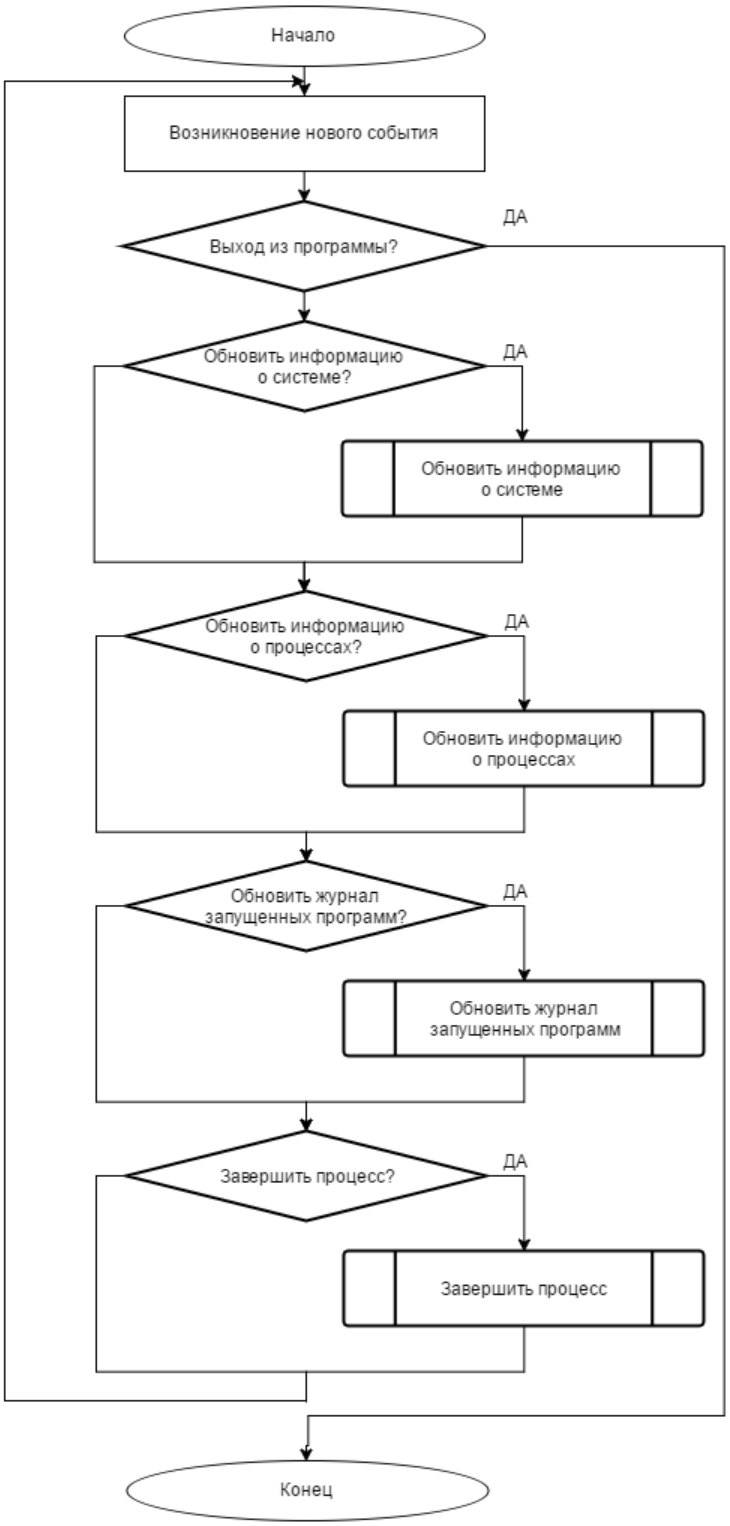

Рис. 5. Схема алгоритма работы приложения с пользовательским интерфейсом.

2.2 Алгоритм работы программы-посредника между приложением и модулем ядра

Для взаимодействия между приложением и модулем ядра создана программа-посредник, принимающая сообщения от модуля ядра и записывающая их в файл журнала запущенных программ. Приложение в свою очередь обновляет информацию о запущенных программах, считывая файл журнала через определённые промежутки времени.

Схема алгоритма работы программы-посредника представлена на рисунке 6.

Рис. 6. Схема алгоритма работы программы-посредника.

2.3 Загружаемый модуль ядра

Загружаемый модуль ядра выполняет функцию перехвата системного вызова *exec()*. При загрузке в ядро модуль заменяет оригинальный системный вызов *exec()* в таблице системных вызовов *sys_call_table* на изменённый системный вызов. Отличие состоит в том, что перед вызовом оригинального системного вызова внутри изменённого вызова происходит формирование и отправка сообщения о запуске новой программы со всеми необходимыми данными. Для полного описания модуля ядра приведём алгоритмы инициализации и деинициализации модуля, а также алгоритм работы изменённого системного вызова *execve()*.

Схема алгоритма инициализации модуля представлена на рисунке 7.

Рис. 7. Схема алгоритма инициализации модуля ядра.

Схема алгоритма работы изменённого системного вызова *execve()* представлена на рисунке 8.

Рис. 8. Схема алгоритма работы изменённого системного вызова *execve().*

Схема алгоритма деинициализации модуля ядра представлена на рисунке 9.

Рис. 9. Схема алгоритма деинициализации модуля ядра.

2.4 Схема взаимодействия модулей программы

Общая схема взаимодействия модулей программы представлена на рисунке 10.

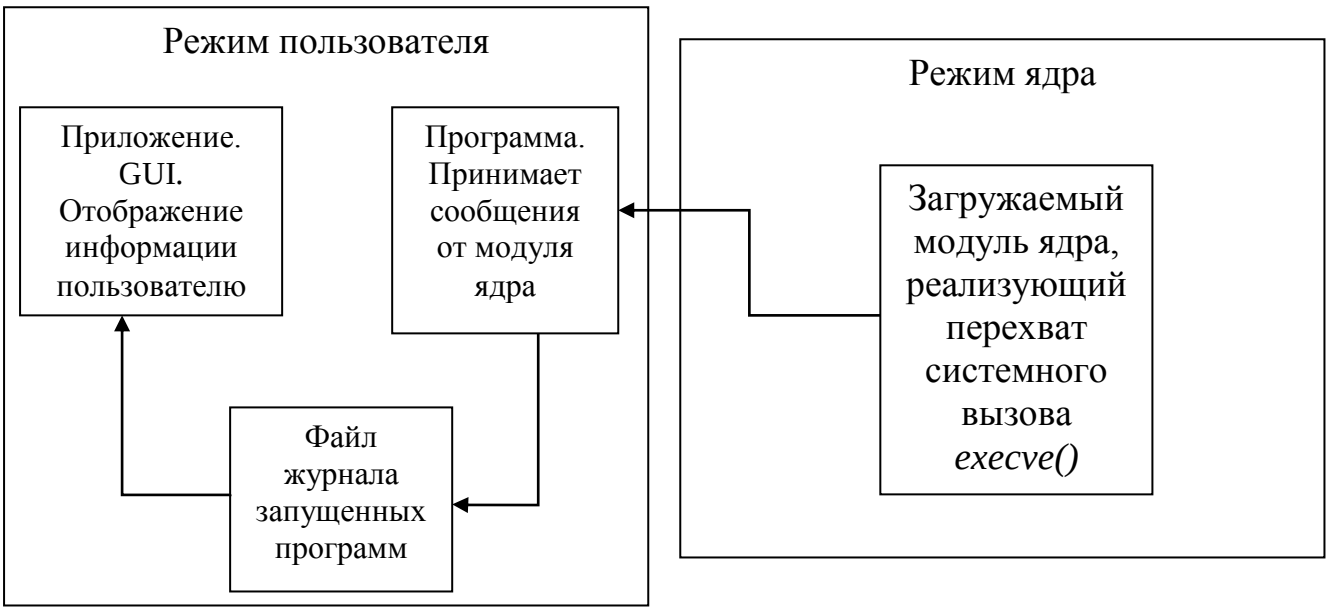

Рис. 10. Общая схема взаимодействия модулей.

3 Технологический раздел

3.1 Средства разработки приложения с пользовательским интерфейсом

Для разработки приложения с графическим интерфейсом был выбран язык C++ совместно с библиотекой Qt. Такой подход предоставляет возможность разработать приложение с графическим интерфейсом на основе событийной модели. Преимущества использования библиотеки Qt:

- простота разработки графического интерфейса;
- кросс-платформенность, независимость от конкретного дистрибутива Linux;
- благодаря системе сигналов и слотов в библиотеке Qt легко устанавливать свои собственные события и интервалы их наступления, что упрощает разработку процедур обновления информации о системе.

Код функции main приложения приведён в листинге 2.

Листинг 2: Функция main приложения с графическим интерфейсом

```
#include "mainwindow.h"
#include <QApplication>
#include <QStyleFactory>
#include <QMessageBox>

int main(int argc, char *argv[])
{
    FILE *f = fopen("exec_log.txt", "w"); // инициализация журнала запущенных программ
    fclose(f);
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    QProcess execmon;
    if (getuid() != 0) { // проверка прав на использование модуля ядра
        QMessageBox msg_box;
        msg_box.setText(" Это приложение должно быть запущено с правами суперпользователя для обеспечения полной функциональности");
        msg_box.exec();
    }
    else {
        QProcess::execute("insmod ../execmon/execmon.ko"); // инициализация модуля ядра
        execmon.start("../execmon/execmon"); // запуск программы посредника
    }
    a.exec(); // запуск событийного цикла приложения
    if (getuid() == 0) {
        execmon.kill(); // завершение программы посредника
        QProcess::execute("rmmod execmon"); // деинициализация модуля ядра
    }
    return 0;
}
```

3.2 Средства разработки программы-посредника

В качестве языка разработки программы-посредника был выбран язык Си, так как он позволяет использовать специальные Netlink [2] сокеты для связи с загружаемым модулем ядра.

Netlink представляет собой особый компонент ядра Linux. С ним можно общаться через сокет передавая и принимая сообщения, сформированные особым образом.

Netlink позволяет:

- получать уведомления об изменении сетевых интерфейсов (название изменившегося интерфейса и что именно произошло), таблиц маршрутизации;
- управлять параметрами сетевых интерфейсов;
- реализовать взаимодействие со своим модулем в ядре.

Последняя указанная возможность Netlink используется в данной курсовой работе. В листингах 3, 4 и 5 приведены исходные коды функций формирования, посылки и приёма сообщений сокетами Netlink.

Листинг 3: Функция формирования сообщения Netlink

```
// comm.c

static void build_nl_msg(struct sockaddr_nl * dest_addr,
            struct nlmsghdr ** nlh,
            struct iovec ** iov,
            struct msghdr * msg,
            size_t data_len) {

    *nlh = (struct nlmsghdr *) malloc(NLMSG_SPACE(data_len));
    /* Инициализация заголовка сообщения */
    memset(*nlh, 0, NLMSG_SPACE(data_len));
    (*nlh)->nlmsg_len = NLMSG_SPACE(data_len);
    (*nlh)->nlmsg_pid = getpid();
    (*nlh)->nlmsg_flags = 0;

    /* Инициализация данных */
    (*iov)->iov_base = (void *) *nlh;
    (*iov)->iov_len = (*nlh)->nlmsg_len;

    /* Инициализация сообщения */
    memset(msg, 0, sizeof(struct msghdr));
    msg->msg_name = (void * ) dest_addr;
    msg->msg_namelen = sizeof(struct sockaddr_nl);
    msg->msg_iov = *iov;
    msg->msg_iovlen = 1;
}
```

Листинг 4: Функция посылки сообщения Netlink

```
// comm.c

struct comm_nl g_comm_nl; // глобальная структура

int COMM_nl_send(void * send_msg, size_t len) {
    int ret = SUCCESS;
    struct nlmsghdr * nlh;
    struct iovec iov;
    struct iovec * iov_p = &iov;
    struct msghdr msg;

    build_nl_msg(&(g_comm_nl.dst_addr), &nlh, &iov_p, &msg, len);
    memcpy(NLMSG_DATA(nlh), send_msg, len);

    ret = sendmsg(g_comm_nl.nl_sock, &msg, 0);

    if (NULL != nlh) {
        free(nlh);
    }

    return ret;

}
```

Листинг 5: Функция приема сообщения Netlink

```
// comm.c

struct comm_nl g_comm_nl; // глобальная структура

int COMM_nl_recv(void * recv_buff, size_t len) {
    int ret = SUCCESS;
    struct nlmsghdr * nlh;
    struct iovec iov;
    struct iovec * iov_p = &iov;
    struct msghdr msg;

    build_nl_msg(&(g_comm_nl.dst_addr), &nlh, &iov_p, &msg, len);

    ret = recvmsg(g_comm_nl.nl_sock, &msg, 0);
    if (0 >= ret) {
        if (NULL != nlh) {
            free(nlh);
        }

        return ret;
    }

    memcpy(recv_buff, NLMSG_DATA(nlh), nlh->nlmsg_len - NLMSG_HDRLEN);

    if (NULL != nlh) {
        free(nlh);
    }

    return ret;
}
```

3.3 Средства разработки загружаемого модуля ядра

В качестве языка разработки загружаемого модуля ядра был выбран язык Си. Перехват системных вызовов производился с помощью дизассемблера udis86. Его функционал предоставляет возможность найти адрес таблицы системных вызовов прямо из кода программы. Следует отметить, что ранние версии ядра Linux (до версии 2.6) экспортировали таблицу системных вызовов, теперь же приходится использовать такие средства, как udis86.

В листинге 6 приведен код функции, получающей адрес таблицы системных вызовов.

Листинг 6: Функция получения адреса таблицы системных вызовов

```
1  static int obtain_sys_call_table_addr(unsigned long * sys_call_table_addr) {
2      int ret = SUCCESS;
3      unsigned long temp_sys_call_table_addr;
4
5      temp_sys_call_table_addr = kallsyms_lookup_name(SYM_SYS_CALL_TABLE);
6
7      /* Return error if the symbol doesn't exist */
8      if (0 == temp_sys_call_table_addr) {
9          ret = ERROR;
10         return ret;
11     }
12
13     *sys_call_table_addr = temp_sys_call_table_addr;
14
15     return ret;
16 }
```

В архитектуре x86 существует специальный защитный механизм, в соответствии с которым попытка записи в защищённые от записи области памяти может приводить к генерации исключения. Поведение процессора в этой ситуации определяется битом WP регистра CR0, а права доступа к странице описываются в соответствующей ей структуре-описателе PTE. При установленном бите WP регистра CR0 попытка записи в защищённые от записи страницы (сброшен бит RW в PTE) ведёт к генерации процессором соответствующего исключения. Решением данной проблемы является временное изменение прав доступа к странице. Это изменение производится

установлением в структуре PTE бита, разрешающего запись для данной страницы. Код функции, разрешающей запись для страницы, приведён в листинге 7.

Листинг 7: Функция, разрешающая запись для страницы

```
1  int MEM_make_rw(unsigned long addr) {
2      int ret = SUCCESS;
3      pte_t * pte;
4      unsigned int level;
5
6      pte = lookup_address(addr, &level);
7      if (NULL == pte) {
8          ret = ERROR;
9          return ret;
10     }
11
12     if (0 == (pte->pte & _PAGE_RW)) {
13         pte->pte |= _PAGE_RW;
14     }
15
16     return ret;
17 }
```

Заключение

Во время выполнения данной курсовой работы был создан загружаемый модуль ядра, осуществляющий сбор информации о запущенных программах, а также создано приложение, связанное с этим модулем и отображающее полученную им информацию пользователю, а также другую информацию о системе и запущенных процессах. Поставленная цель, а именно — создание программного комплекса, осуществляющего мониторинг операционной системы Linux, выполнена успешно.

Список использованных источников

1. Таненбаум Э., Бос Х., Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.: ил.
2. Linux Journal. [Электронный ресурс]. — Режим доступа: http://www.linuxjournal.com/.
3. The Linux man-pages project. [Электронный ресурс]. — Режим доступа: https://www.kernel.org/doc/man-pages/.

Приложение А. Пример работы программы

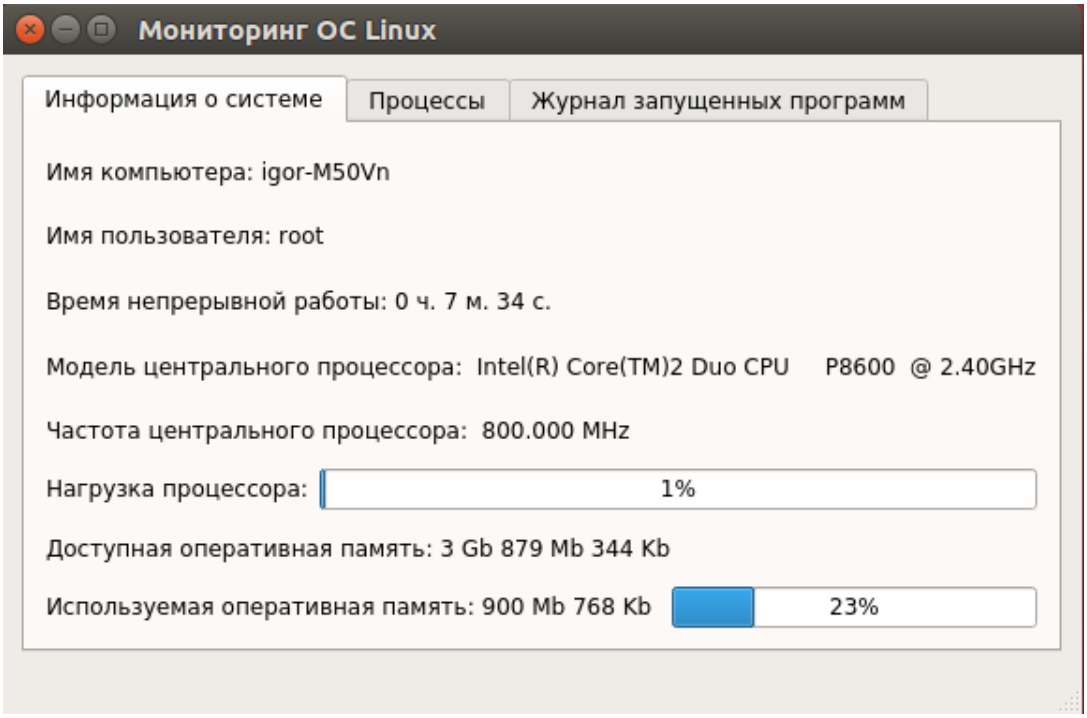

Рис. 11. Пример вывода информации о системе.

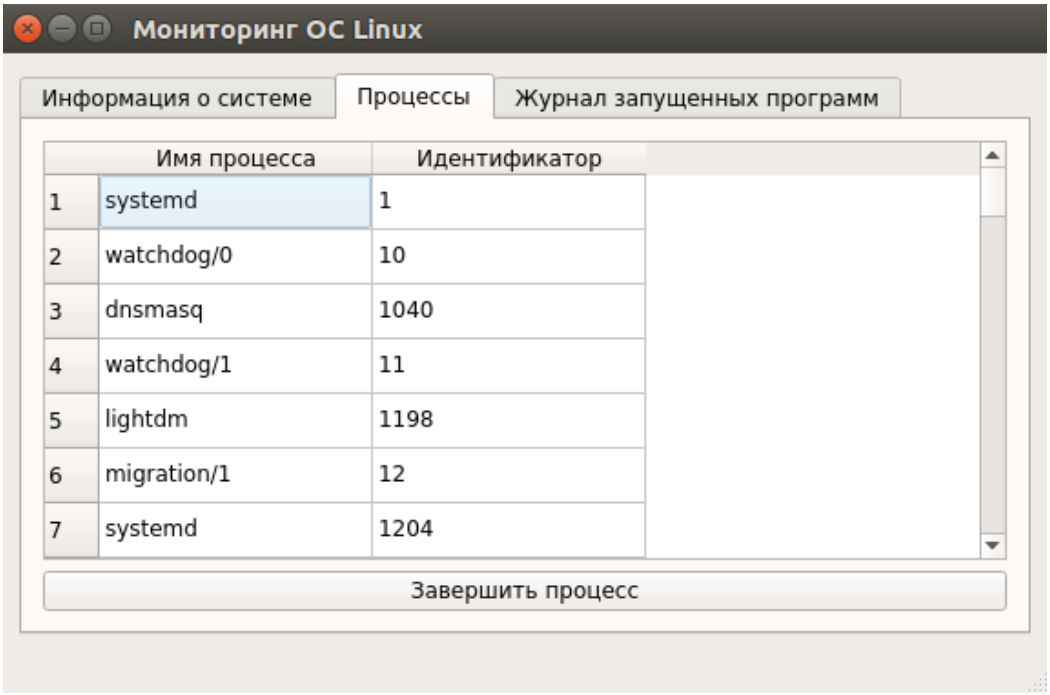

Рис. 12. Пример вывода информации о процессах.

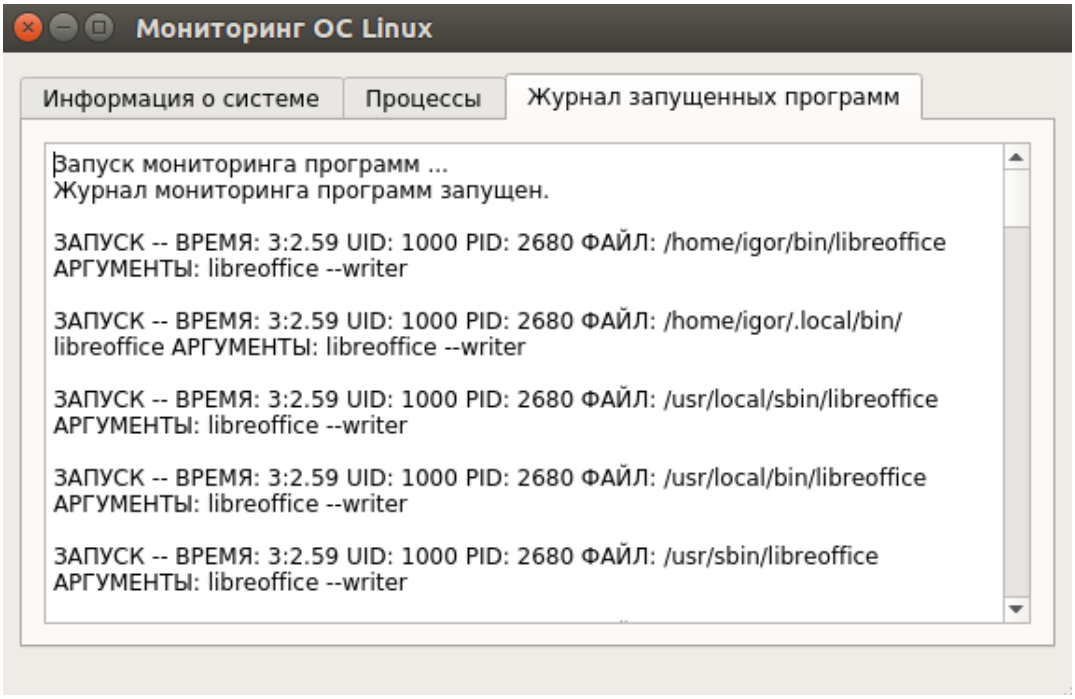

Рис. 13. Пример вывода журнала запущенных программ.

Приложение Б. Листинг измененного системного вызова *execve()*

Листинг 8: Измененный системный вызов *execve()*

```
static asmlinkage long new_sys_execve(const char __user * filename,
                const char __user * const __user * argv,
                const char __user * const __user * envp) {
    size_t exec_line_size = 512;
    char * exec_str = NULL;
    char ** p_argv = (char **) argv;
    char *time_string = NULL;
    int my_uid = (int)current->cred->uid.val;
    int my_pid = (int)current->pid;
    char *time_flag = ВРЕМЯ":";
    char *uid_flag = "UID:";
    char *pid_flag = "PID:";
    char *file_path_flag = ФАЙЛ":";
    char *arguments_flag = АРГУМЕНТЫ":";
    char *uid_str = vmalloc(10);
    char *pid_str = vmalloc(10);
    snprintf(uid_str, 10, "%d", my_uid);
    snprintf(pid_str, 10, "%d", my_pid);

    time_string = get_timestamp();
    exec_str = vmalloc(exec_line_size);
    if (NULL != exec_str) {
        snprintf(exec_str, exec_line_size, "%s %s", time_flag, time_string);
        snprintf(exec_str, exec_line_size, "%s %s %s", exec_str, uid_flag,
    uid_str);
        snprintf(exec_str, exec_line_size, "%s %s %s", exec_str, pid_flag,
    pid_str);
        snprintf(exec_str, exec_line_size, "%s %s %s", exec_str,
    file_path_flag, filename);
        vfree(time_string);
        vfree(uid_str);
        vfree(pid_str);

        snprintf(exec_str, exec_line_size, "%s %s", exec_str, arguments_flag);
        p_argv = (char **) argv;
        while (NULL != *p_argv) {

            snprintf(exec_str, exec_line_size,
                    "%s %s", exec_str, *p_argv);
            (char **) p_argv++;
        }

        COMM_nl_send_exec_msg(exec_str);
    }

    return orig_sys_execve_fn(filename, argv, envp);
}
```