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: «Информатика и системы управления»

Кафедра: «Программное обеспечение ЭВМ и информационные технологии»

Расчетно-пояснительная записка к курсовой работе на тему: «Планирование задач в ядре Linux»

Студент: Левушкин И. К.
Группа: ИУ7-72Б
Научный руководитель: Филиппов М.В.

Москва, 2020 г.

Задание на курсовой проект

Разработать программное обеспечение, состоящее из двух модулей, которое позволяет осуществлять планирование задачи из ядра Linux. Первый модуль - загружаемый модуль ядра, выполняющий функцию запуска второго модуля с указанным интервалом. Второй модуль - приложение, которое, работая в режиме пользователя, выводит в конец файла информацию о последнем загруженном модуле ядра в систему.

Содержание

Введение

Иногда, при работе с Linux-системами, требуется осуществить планирование исполнения задач из ядра системы.

Проект посвящен исследованию способов планирования исполнения задач и получения информации о загруженных модулях ядра.

Целью проекта является разработка программного обеспечения, позволяющего запускать usermode-приложение из ядра системы с указанным интервалом времени. Usermode-приложение пишет в указанный файл информацию о последнем загруженном модуле ядра в систему.

1 Аналитический раздел

В соответствии с заданием на курсовой проект необходимо решить следующие задачи:

1. Провести анализ способов планирования исполнения задач в ядре Linux и выбрать один из них
2. Выбрать способ получения информации о загруженных модулях ядра
3. Разработать загружаемый модуль ядра, выполняющий функцию запуска второго модуля с указанным интервалом
4. Разработать приложение, которое, работая в режиме пользователя, выводит в конец файла информацию о последнем загруженном модуле ядра в систему

1.1 Получение информации о загруженных модулях ядра в систему с помощью файловой системы proc

В Linux получить информацию о процессах можно через файлы и каталоги файловой системы procfs, как правило монтируемой к каталогу /**proc**.

Файловая система proc представляет собой интерфейс к основным структурам данных ядра, которые работают также как и файловая система. Вместо того, чтобы каждый раз обращаться в /dev/kmem и искать путь к определению местонахождения какой-либо информации, все приложения читают файлы и каталоги из /proc.

В частности, файл /**proc/modules** предоставляет информацию о загруженных модулей ядра в систему. Этот файл используется различными утилитами Linux, такими как **lsmod**.

1.2 Способы планирования исполнения задач в ядре Linux

В Linux существует возможность планировки исполнения отложенных задач. Linux предлагает три различных интерфейса для этой цели: очереди задач, тасклеты и таймеры ядра.

Рассмотрим каждый из них поподробней.

1.2.1 Очереди задач

Очереди задач и тасклеты обеспечивают гибкий механизм планировки исполнения в некий дальнейший интервал времени. Они наиболее интересны при написании обработчиков прерываний.

Ситуация, в которой может понадобиться использование очереди задач или тасклетов — это управление физическими устройствами, которые не могут генерировать прерывания, но используют блокировку чтения. Потребуется опрашивать готовность устройства, не обременяя, при этом, процессор ненужными операциями.

Очереди задач представляют собой список задач, где каждая задача представляется указателем на функцию и ее аргументом. Когда задача запускается она получает один аргумент void * и возвращает void. Аргумент указатель может быть использован для передачи произвольной структуры данных в функцию, а может быть игнорирован. Сама очередь представляет список структур (задач), которые принадлежат модулю ядра, в котором они описаны, и который ими управляет. Модуль полностью ответственнен за распределение и уничтожение как динамических, так и статических структур, которые обычно используются для этих целей.

Когда же запускается обработка очередей задач?

Различные очереди запускаются в разное время, но они всегда запускаются тогда, когда у ядра нет другой неотложной работы для выполнения. Наиболее важно то, что они почти наверняка не запустятся пока процесс, который положил задачу в очередь исполняется. Они работают асинхронно. Однако, когда выполняется

очередь задач, то процесс может спать, выполняться на другом процессоре, или, предположительно, пожет быть полностью завершен.

1.2.2 Тасклеты

Тасклеты предоставляют способ безопасного запуска отложенных задач, и всегда исполняются в режиме прерывания. Как и очереди задач, такслеты могут быть запущены только единожды, даже при многократной установке в планировщик, но тасклеты могут быть запущены параллельно с другими тасклетами на системах SMP. Также, на системах SMP существует гарантия, что тасклеты будут запущены на том CPU, который впервые планировал их, что обеспечивает лучшее использование кэша и более высокую производительность.

Каждый тасклет имеет связанную с ним функцию, которыя вызывается тогда, когда тасклет должен быть выполнен. Если тасклет разрешен, то как только он диспетчеризуется, он будет запущен сразу же как только это будет возможно из соображений безопасности. Тасклеты могут перепланировать сами себя много раз, также как и очереди задач. Тасклет может не беспокоиться о запуске своей копии на многопроцессорной системе, так как в ядре предприняты шаги к тому, чтобы гарантировать запуск каждого из тасклетов только в одном месте.

1.2.3 Таймеры ядра

Последним ресурсом управления времени в ядре являются **таймеры**. Таймеры используются для планирования исполнения функции (обработчика таймера) в заданное время. Таким образом, их работа отличается от работы очереди задач и тасклетов тем, что возможно определить время запуска обработчика таймера, что является решающим фактором при выборе инструмента планировки исполнения задач в данной работе.

С другой стороны, таймеры ядра похожи на очереди задач в том, что функция зарегистрированная в таймере ядра выполняется только один раз.

Таймер прост в использовании. Необходимо зарегистрировать функцию единожды, и ядро вызывает ее при истечении счета таймера. Данная функциональность часто используется в самом ядре, но, иногда, требуется и в драйверах, как например, в случае управления мотором флоппи-дисковода.

Таймеры ядра организуются в двунаправленный связанный список. Это означает, что вы можете создавать столько таймеров, сколько захотите. Таймер характеризуется значением таймаута (в джиффисах), и функцией, вызываемой при истечении таймера. Обработчик таймера принимает аргумент, который сохраняется в структуре данных, вместе с указателем на сам обработчик.

Выводы по аналитическому разделу

Был выбран способ получения информации о загруженных модулях ядра в систему с помощью файловой системы proc. Для этого будет необходимо вызвать приложение, работающее в пользовательском пространстве из модуля ядра.

Были рассмотрены основные способы планирования исполнения задач в ядре Linux, и был выбран наиболее предпочтительных из них для данной задачи - таймеры ядра, поскольку появляется возможность определить время запуска обработчика таймера.

2 Конструкторский раздел

2.1 Структура программного обеспечения

В состав программного обеспечения входит загружаемый модуль ядра, запускающий задачи с определенным интервалом времени, которые вызывают приложение, работающее в пользовательском пространстве, и, соответственно, приложение, работающее в пользовательском пространстве, читающая информацию о загруженных модулях ядра из файловой системы proc и выводящая изменения в файл.

2.2 Таймеры ядра

Таймеры задаются с помощью структуры *timer_list*, в которой хранятся все данные, необходимые для реализации таймера. Если рассматривать структуру *timer_list* с точки зрения ее использования, в ней хранится время истечения срока действия таймера, функция обратного вызова, а также контекст, предоставленный пользователем. Затем нужно инициализировать таймер, что можно сделать несколькими способами. Простейшим способом является вызов функции *setup_timer*, которая инициализирует таймер и задает предоставленные пользователями функцию обратного вызова и контекст. Либо пользователь может задать эти значения (функцию и данные) непосредственно в таймере и просто вызвать функцию *init_timer*. Функция *init_time* вызывается внутри функции *setup_timer*.

```
void init_timer( struct timer_list *timer );
void setup_timer( struct timer_list *timer, \
void (*function)(unsigned long), unsigned long data );
```

Теперь, когда таймер инициализирован, необходимо задать время работы таймера, что делается с помощью вызова функции *mod_timer*.

Поскольку обычно указывается время работы таймера, которое относится к будущему, то обычно здесь добавляется значение переменной jiffies к смещению от текущего момента времени. С помощью функции *del_timer* можно можно также удалить таймер (если срок его действия не истек):

```
int mod_timer( struct timer_list *timer, unsigned long expires
void del_timer( struct timer_list *timer );
```

Наконец, с помощью функции *timer_pending* можно определить, работает ли еще таймер (все еще включен) — функция возвращает 1, если таймер еще работает:

```
int timer_pending( const struct timer_list *timer );
```

2.3 Вызов Usermode-приложения из модуля ядра Linux

Интерфейс usermode-helper API представляет собой простое API с хорошо известным набором возможностей. Например, чтобы создать процесс из пользовательского пространства, указывается имя исполняемого модуля и набор переменных среды (execve). То же самое происходит при создании процесса из ядра. Но, поскольку происходит запуск процесса из пространства ядра, есть ряд дополнительных возможностей.

В таблице 1 приведен базовый набор функций ядра, которые есть в usermode-helper API.

Таблица 1

Функция API	**Описание**
call_usermodehelper_setup	Подготовка обработчика handler для вызова функции пользовательского пространства
call_usermodehelper_setkeys	Установка сессионных ключей для helper
call_usermodehelper_setcleanup	Установка функции очистки cleanup для helper
call_usermodehelper_stdinpipe	Создание конвейера *stdin* для helper
call_usermodehelper_exec	Вызов функции пользовательского пространства

В следующей таблице приведены несколько сокращенных функций, объединяющие функции ядра, приведенные в таблице 1 (требуется один вызов вместо нескольких).

Таблица 2

Функция API	**Описание**
call_usermodehelper	Осуществляет вызов функции пользовательского пространства
call_usermodehelper_pipe	Осуществляет вызов функции пользовательского пространства с использованием конвейера *stdin*
call_usermodehelper_keys	Осуществляет вызов функции пользовательского пространства с использованием сессионных ключей

Базовое API в своей работе использует ссылку на обработчик (handler), который является структурой *subprocess_info*. В этой структуре собраны все элементы, необходимые для данного экземпляра usermode-helper. Ссылка на структуру возвращается в вызове *call_usermodehelper_setup*. Дальнейшее конфигурирование структуры (и последующих вызовов) осуществляется с помощью вызовов *call_usermodehelper_setkeys* (работа с учетными данными), *call_usermodehelper_setcleanup* и *call_usermodehelper_stdinpipe*. Наконец, после того, как конфигурирование будет завершено, с помощью функции *call_usermodehelper_exec* вызывается сконфигурированное приложение, работающее в пользовательском режиме.

Базовые функции предоставляют максимальную возможность управления, причем функции helper выполнят большую часть работы за один вызов. Вызовы, использующие конвейеры, создают соответствующий конвейер для использования его helper-ом. В частности, создается конвейер pipe (файловая структура в ядре). Приложение пользовательского пространства читает данные из pipe, а со стороны ядра осуществляется запись в pipe. А что касается записи, то выдача дампа памяти является единственным приложением, которое может использовать конвейер совместно с usermode-helper.

Соотношение между этими функциями и *sub_processinfo* вместе с деталями структуры *subprocess_info* показано на рисунке 1.

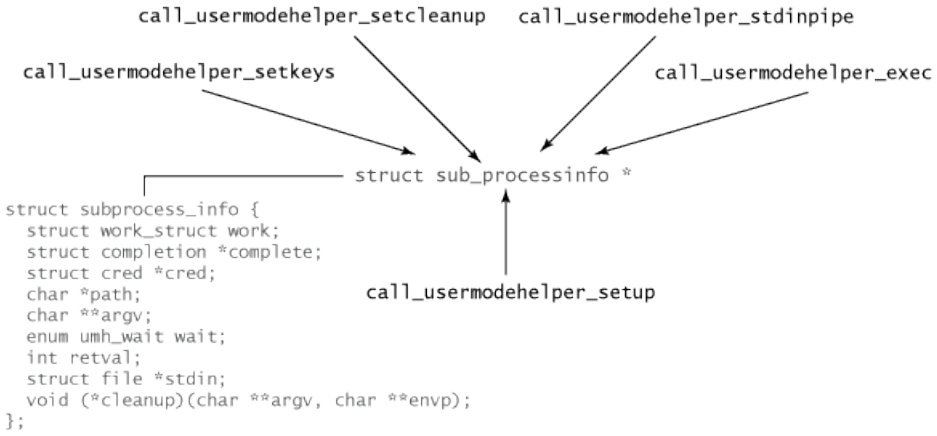

Рисунок 1: Элементы интерфейса usermode-helper API.

Сокращенные функции из таблицы 2 внутри себя вызывают функцию *call_usermodehelper_setup* и *call_usermodehelper_exec*. Последние два вызова, указанные в таблице 2, вызывают, соответственно, *call_usermodehelper_setkeys* и *call_usermodehelper_stdinpipe*.

2.4 Чтение из файла /proc/modules

Ниже приведен пример вывода информации из файла */proc/modules* с последующим объяснением значений.

```
nfs      170109  0 -          Live 0x129b0000
lockd    51593   1 nfs,       Live 0x128b0000
nls_utf8 1729    0 -          Live 0x12830000
vfat     12097   0 -          Live 0x12823000
fat      38881   1 vfat,      Live 0x1287b000
autofs4  20293   2 -          Live 0x1284f000
sunrpc   140453  3 nfs,lockd, Live 0x12954000
3c59x    33257   0 -          Live 0x12871000
uhci_hcd 28377   0 -          Live 0x12869000
md5      3777    1 -          Live 0x1282c000
ipv6     211845 16 -          Live 0x128de000
ext3     92585   2 -          Live 0x12886000
jbd      65625   1 ext3,      Live 0x12857000
dm_mod   46677   3 -          Live 0x12833000
```

Рисунок 2: Пример вывода */proc/modules*.

Где

1. Имя модуля
2. Размер (в байтах) занимаемой модулем памяти
3. Количество загруженных экземпляров модуля. ноль — модуль выгружен
4. Список модулей, зависящих от данного
5. Текущее состояние модуля: Live — загружен, Loading — загружается и Unloading — выгружается
6. Текущее смещение в памяти программы linux, по которому загружен модуль

2.5 Схема алгоритма работы модуля ядра

Ниже приведена схема алгоритма работы модуля ядра.

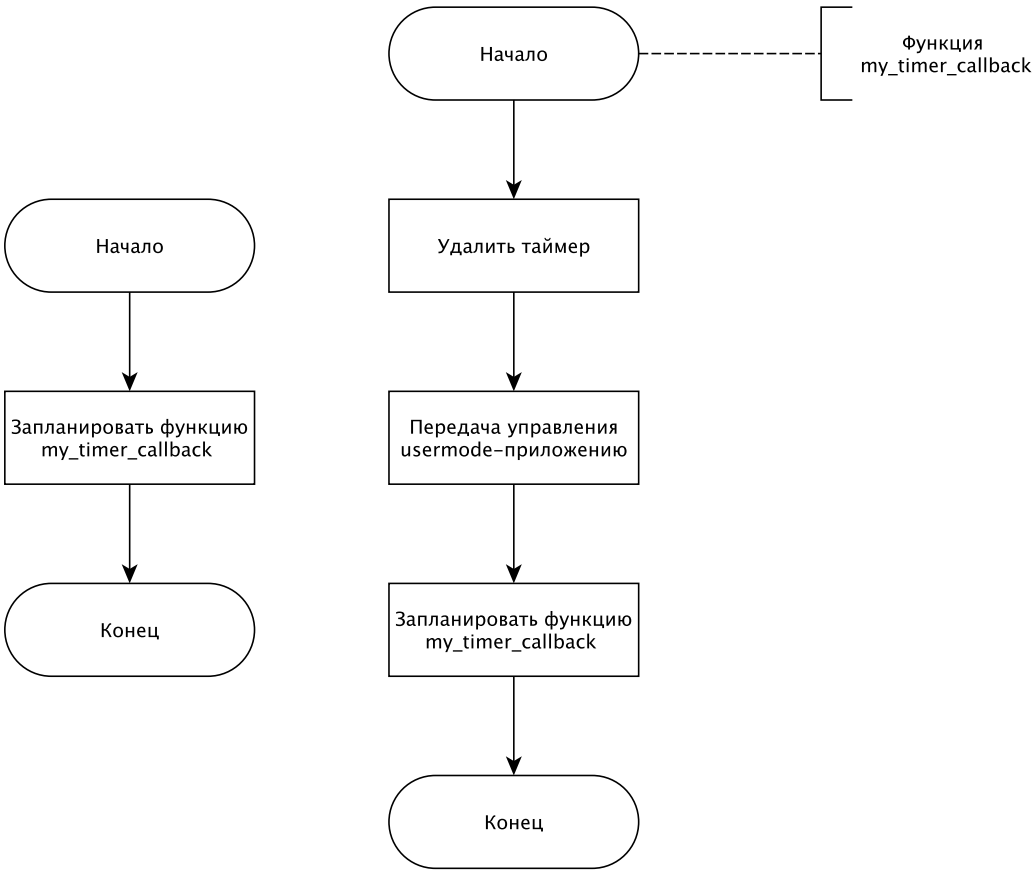

Рисунок 3: Алгоритм работы модуля ядра.

2.6 Схема алгоритма работы usermode-приложения

Ниже приведена схема алгоритма работы приложения, работающего в пользовательском режиме.

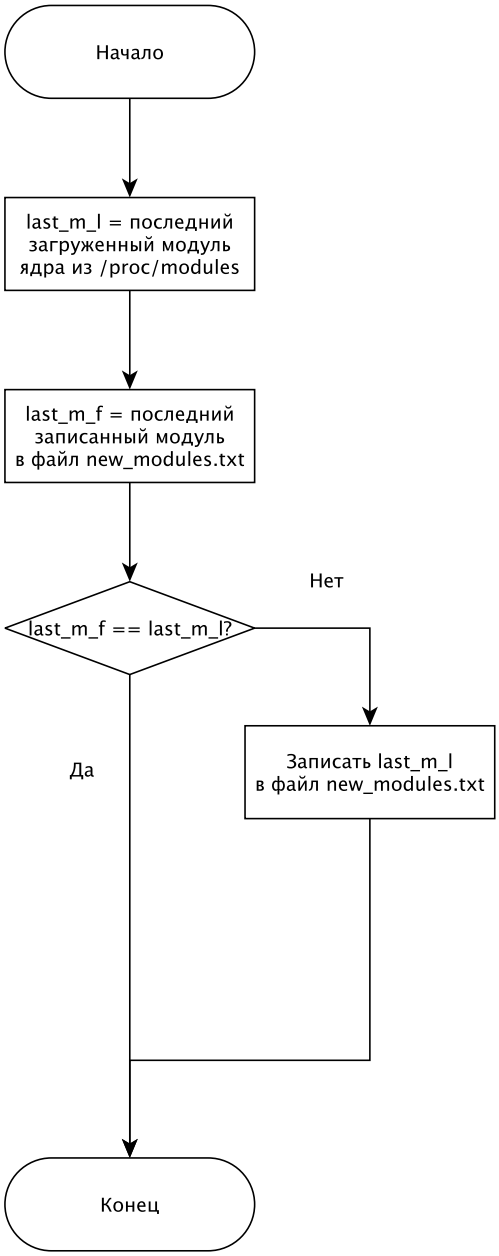

Рисунок 4: Алгоритм работы usermode-приложения.

Выводы по конструкторскому разделу

Были проанализированы и изучены выбранные технологии для реализации поставленной задачи, в частности, API ядра Linux: API системных таймеров и usermode-helper API; а также структура файла /proc/modules.

Были разработаны схемы алгоритмов работы обоих модулей программы.

3 Технологический раздел

3.1 Выбор языка программирования

Для реализации загружаемого модуля был выбран язык C с использованием встроенного в ОС Linux компилятора GCC. Выбор языка основан на том, что исходный код ядра, предо ставляемый системой, написан на C, и использование другого языка программиро вания в данном случае было бы нецелесообразным.

Для реализации приложения, работающего в пользовательском режиме, был выбран язык C++, поскольку он предоставляет удобный интерфейс для работы с файлами

3.2 Выбор среды разработки

Для написания программы, был выбран текстовый редактор Sublime text 3.

- Огромное количество плагинов, которые позволяют делать работу быстрее
- Возможность гибкой настройки под себя
- Много встроенных команд и комбинаций

3.3 Модуль ядра

Ниже приведен листинг кода модуля ядра для данной работы.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/timer.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/uaccess.h>

void my_timer_callback(struct timer_list * my_timer);


#define TIMER 5000
#define USERMODE_PROGRAM \
"/home/ilalevuskin/Desktop/project/proc_modules"
#define FILE_NAME \
"/home/ilalevuskin/Desktop/project/new_modules.txt"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Levushkin Ilya");


static struct timer_list my_timer;

int init_timer(void)
{
        int ret;

        timer_setup(&my_timer, my_timer_callback, 0);

        ret = mod_timer(&my_timer, jiffies + \
         msecs_to_jiffies(TIMER));
```

```
        if (ret)
        {
                printk("Error in mod_timer\n");
        }
        return ret;
}

void delete_timer(void)
{
        int ret;

        ret = del_timer(&my_timer);

        if (ret)
        {
                printk("The timer is still in use...\n");
        }
}

int print_new_load_proc_module_info(void)
{
        char *argv[] = {USERMODE_PROGRAM, FILE_NAME, NULL};
        static char *envp[] = {
        FILE_NAME,
        "TERM=linux",
        "PATH=/home/ilalevuskin/Desktop/project", NULL};
        call_usermodehelper(argv[0], argv, envp, UMH_NO_WAIT);
        return 0;
}


void my_timer_callback(struct timer_list * my_timer)
{
        delete_timer();
```

```
        print_new_load_proc_module_info();
        init_timer();
}

static int __init monitoring_init(void)
{
        int ret;
        printk("Monitoring module was installed\n");
        printk("Kernel modules info update (%d)ms (%ld)\n", \
        TIMER, jiffies);

        ret = init_timer();
        return ret;
}

static void __exit monitoring_exit(void)
{
        delete_timer();
        printk("Monitoring module uninstalled\n");
}

module_init(monitoring_init);
module_exit(monitoring_exit);
```

3.4 Приложение, работающее в режиме пользователя

Ниже приведен листинг кода приложения, работающего в режиме пользователя, для данной работы.

```
#include <iostream>
#include <fstream>
#include <string>
#include <unistd.h>
#include <cstring>
```

```
using namespace std;

#define READ_PROC_ERR 1
#define WRITE_MODULE_ERR 2
#define READ_MODULE_ERR 3
#define OK 0


int get_last_module_to_kernel(string &buff)
{
        ifstream rfile;
        rfile.open("/proc/modules", fstream::in);
        getline(rfile, buff);
        rfile.close();
        return 0;
}

int get_last_added_module_to_kernel(string &res, \
string file_name)
{
        ifstream rfile;
        rfile.open(file_name.c_str(), fstream::in);

        if (!rfile)
        {
                return READ_MODULE_ERR;
        }

        string buf;
        while (getline(rfile, buf))
        {
                res = buf;
        }
        rfile.close();
```

```
        return 0;
}

int new_module_added(const char *last_module, \
const char* before_module)
{
        if (strcmp(last_module, before_module) != 0)
        {
                return 1;
        }
        return 0;
}

int write_added_module_in_file(string buff, string file_name)
{
        ofstream wfile;
        wfile.open(file_name.c_str(), ios::out | ios::app);

        if (!wfile)
        {
                return WRITE_MODULE_ERR;
        }

        wfile << buff << endl;

        wfile.close();
        return OK;
}

int main(int argc, char *argv[])
{
string buff_1, buff_2;
if (argv[1])
{
```

```
        string file_name = argv[1];
        if (get_last_module_to_kernel(buff_1))
        {
                cerr << "Cannot read /proc/modules" \
                << endl;
                return READ_PROC_ERR;
        }
        if (get_last_added_module_to_kernel(buff_2, file_name))
        {
                cerr << "Cannot read file " \
                << file_name.c_str() << endl;
                return READ_MODULE_ERR;
        }
        if (new_module_added(buff_1.c_str(), buff_2.c_str()))
        {
                if (write_added_module_in_file(buff_1, \
                file_name))
                {
                        cerr << "Cannot write in file " \
                        << file_name.c_str() << endl;
                        return WRITE_MODULE_ERR;
                }
        }
        return OK;
}
else
{
        cerr << "No arguments were transfered" << endl;
}
}
```

Выводы по технологическому разделу

Были даны обоснования по выбору языка программирования и среды разработки, приведены листинги кода двух модулей программного обеспечения.

Заключение

В результате выполнения данной курсовой работы были изучены основные способы планирования исполнения задач в ядре Linux.

Была исследована структура файловой системы proc.

Разработано программное обеспечение, позволяющее осуществлять вывод информации о загруженных модулях ядра Linux.

Реализован модуль ядра, позволяющий вызывать через заданные промежутки времени приложение, работающее в пользовательском режиме.

Реализовано приложение, выводящее в конец файла информацию о последнем загруженном модуле ядра в систему.

Список литературы

[1] Вызов приложений, работающих в пользовательском пространстве, из ядра системы [Электронный ресурс]. – Режим доступа: http://rus-linux.net/nlib.php?name=/MyLDP/kernel/api/kernelapi1.html, свободный – (02.12.2020)

[2] Модули Linux ядра, Олег Цилюрик [Электронный ресурс]. – Режим доступа: http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/KERN-modul-4.95.pdf, свободный – (02.12.2020)

[3] Файловая система proc [Электронный ресурс]. – Режим доступа: https://www.opennet.ru/man.shtml?topic=proc&category=5&russian=0 свободный – (02.12.2020)

[4] ФАЙЛОВАЯ СИСТЕМА PROC В LINUX [Электронный ресурс]. – Режим доступа: https://losst.ru/fajlovaya-sistema-proc-v-linux#/proc/modules, свободный – (02.12.2020)

[5] Течение времени в ядре Linux [Электронный ресурс]. – Режим доступа: http://knzsoft.ru/ldd2-ch6/#t3, свободный – (02.12.2020)