Санкт-Петербургский политехнический университет

Институт Компьютерных Наук и Технологий

Кафедра компьютерных систем и программных технологий

Курсовая работа по дисциплине «Проектирование ОС и компонентов»

Выполнил: студент группы № 63501/3

Дедков С.В.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Душутина Е.В.

Санкт-Петербург

2016

# Оглавление

[Оглавление 2](#_Toc445804533)

[Введение 4](#_Toc445804534)

[Системные вызовы 5](#_Toc445804535)

[Цель работы 5](#_Toc445804536)

[Принцип работы системных вызовов 6](#_Toc445804537)

[Назначение и функциональность вызовов sys\_stat, sys\_rename, sys\_mkdir, sys\_rmdir, sys\_fstat 10](#_Toc445804538)

[Введение 10](#_Toc445804539)

[Системные вызовы sys\_stat, sys\_fstat 11](#_Toc445804540)

[Системные вызовы sys\_mkdir, sys\_rmdir 15](#_Toc445804541)

[Системный вызов sys\_rename 18](#_Toc445804542)

[Разбор исходного кода системных вызовов ядра 20](#_Toc445804543)

[Введение 20](#_Toc445804544)

[Исходный код вызова mkdir 20](#_Toc445804545)

[Перехват системных вызовов 23](#_Toc445804546)

[Введение 23](#_Toc445804547)

[Перехват системного вызова mkdir 24](#_Toc445804548)

[Вывод 30](#_Toc445804549)

[Разработка драйверов ОС Linux. Драйвер сетевой карты. 31](#_Toc445804550)

[Цель работы 31](#_Toc445804551)

[Введение 32](#_Toc445804552)

[Разработка каркаса драйвера – загружаемый модуль 33](#_Toc445804553)

[Работа с устройствами. Символьный драйвер 37](#_Toc445804554)

[Драйвер сетевой карты 44](#_Toc445804555)

[Выводы 52](#_Toc445804556)

[Платформа сборки дистрибутивов Linux Для встраиваемых устройств «yocto» 53](#_Toc445804557)

[Цель работы 53](#_Toc445804558)

[Описание проекта, инструментов и возможностей 54](#_Toc445804559)

[Описание процесса сборки дистрибутива 56](#_Toc445804560)

[Процесс сборки с использованием web оболочки Toaster 59](#_Toc445804561)

[Вывод 62](#_Toc445804562)

[Средства тестирования ОС 63](#_Toc445804563)

[Цель работы 63](#_Toc445804564)

[Описание видов тестирования ОС 64](#_Toc445804565)

[Функциональное тестирование 64](#_Toc445804566)

[Тестирование обратной совместимости 65](#_Toc445804567)

[Обнаружение специфических видов ошибок 65](#_Toc445804568)

[Тестирование производительности ОС 66](#_Toc445804569)

[Средства тестирования ОС 67](#_Toc445804570)

[LAVA (Linaro Automated Validation Architecture) 67](#_Toc445804571)

[Opentest 67](#_Toc445804572)

[Autotest 68](#_Toc445804573)

[Jenkins 68](#_Toc445804574)

[Ktest.pl 69](#_Toc445804575)

[Вывод 70](#_Toc445804576)

[Список литературы 71](#_Toc445804577)

# Введение

Данная работа является совокупностью лабораторных работ по курсу «Проектирование ОС и компонентов».

Рассматриваются следующие вопросы:

* Системные вызовы
* Драйвера устройств
* Linux для встраиваемых систем. Yocto Project
* Средства тестирования ОС и компонентов

# Системные вызовы

## Цель работы

* Изучить принцип работы системных вызовов в целом.
* Рассмотреть назначение и функциональность вызовов
  + sys\_stat
  + sys\_rename
  + sys\_mkdir
  + sys\_rmdir
  + sys\_fstat
* Осуществить перехват системного вызова, написать свою реакцию.

## Принцип работы системных вызовов

**Системные вызовы** – это интерфейс между приложениями и ядром ОС, точка входа к функциям ядра.

Чтобы обеспечить безопасную и надежную работу программ с системными ресурсами (жесткие диски, видеокарта, сетевые устройства и т. д.) и безопасное межпроцессное взаимодействие *архитектура современных процессоров*, (Системный вызов, 2016).

Традиционно семейство микропроцессоров x86 обеспечивает четыре кольца защиты(*аппаратное* разделение системного и пользовательского уровней привилегий).

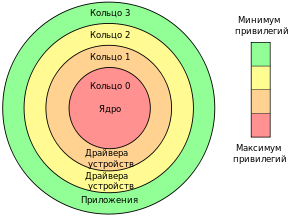
 (Кольца защиты, 2016)

Рисунок 1. Кольца привилегий архитектуры x86 в защищённом режиме

Ядро ОС работает в максимально привилегированном режиме – 0. Приложения в 3.

Для *взаимодействия* с системными ресурсами приложения используют *системные вызовы*, которые дают возможность операционной системе обеспечить безопасный доступ к ним.

Системные вызовы передают управление *ядру* операционной системы, при этом происходит переключение режима работы процессора с 3 на 0, и при возврате управления приложению наоборот.

Ядро ОС обеспечивает приложениям координированный доступ к ресурсам компьютера, таким как (Ядро операционной системы, 2016):

* процессорное время
* память
* внешнее аппаратное обеспечение
* внешнее устройство ввода и вывода информации
* сервисы файловой системы
* сетевые протоколы.

Детали этого механизма в *разных архитектурах* немного *различаются*. Например, в процессорах *i386* пользовательское приложение выполняет инструкцию программного прерывания *int* со значением *0x80*. Эта инструкция осуществляет переключение на работу с пространством ядра - защищенной областью, - где ядром выполняется обработчик программного прерывания. Обработчик прерывания 0x80 - обработчик системного вызова. Приложение сообщает ядру, какой системный вызов требуется выполнить и с какими параметрами. Это делается посредством *аппаратных регистров*. Системные вызовы обозначаются по *номерам*, начиная с 0. В архитектуре i386, чтобы запросить системный вызов 5 (обычно это вызов open()), пользовательское приложение записывает 5 в регистр eax, после чего выдает инструкцию int. Передача параметров обрабатывается схожим образом. Так, в архитектуре i386 регистр применяется для всех возможных параметров — например, регистры ebx, ecx, edx, esi и edi в таком же порядке содержат первые пять параметров. (Лав, 2014)

Рассмотрим простой пример системного вызова, для ввода на экран сообщения.

Для современных архитектур процессоров вместо прерывания int 0x80 используется функция sysenter – она более быстрая, чем прерывание.

Листинг – *scalls/asm/main.s* :

*Примечание*. Здесь и далее исходные коды программ можно найти в репозитории: [*https://github.com/dsvgit/os\_lab\_syscalls*](https://github.com/dsvgit/os_lab_syscalls)

1. .text                           # section declaration
3. # we must export the entry point to the ELF linker or
4. .global \_start              # loader. They conventionally recognize \_start as their
5. # entry point. Use ld -e foo to override the default.
7. \_start:
9. # write our string to stdout
11. movl    $len,%edx         # third argument: message length
12. movl    $msg,%ecx         # second argument: pointer to message to write
13. movl    $1,%ebx           # first argument: file handle (stdout)
14. movl    $4,%eax           # system call number (sys\_write)
15. **int**     $0x80             # call kernel
17. # and exit
19. movl    $0,%ebx           # first argument: exit code
20. movl    $1,%eax           # system call number (sys\_exit)
21. **int**     $0x80             # call kernel
23. .data                           # section declaration
25. msg:
26. .ascii    "Hello, world!\n" # our dear string
27. len = . - msg               # length of our dear string

Makefile:

1. all: main.s
2. gcc -c main.s
3. ld main.o
5. clean:
6. -rm \*.out
7. -rm \*.o
8. -rm \*~

Результат выполнения:

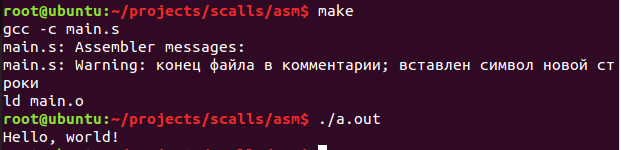


Рисунок 2. Сборка и выполнение примера системного прерывания на языке ассемблер

Лог:

root@ubuntu:~/projects/scalls/asm$ make

gcc -c main.s

main.s: Assembler messages:

main.s: Warning: конец файла в комментарии; вставлен символ новой строки

ld main.o

root@ubuntu:~/projects/scalls/asm$ ./a.out

Hello, world!

root@ubuntu:~/projects/scalls/asm$

Здесь можно увидеть вызов прерывания 0x80 и установку параметров, как это было описано выше:

1. movl    $len,%edx           # third argument: message length
2. movl    $msg,%ecx           # second argument: pointer to message to write
3. movl    $1,%ebx             # first argument: file handle (stdout)
4. movl    $4,%eax             # system call number (sys\_write)
5. **int**     $0x80               # call kernel

В данном случае номер системного вызова - 4 (sys\_write). Таблица системных вызовов Linux - <http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html>, здесь же можно найти путь к исходным кодам. Соответствующие константы можно найти в файлах /usr/include/asm/unistd.h, /usr/include/asm/unistd\_32.c, /usr/include/asm/unistd\_64.c.

Для просмотра системных вызовов программ, можно использовать утилиту strace. Полезна утилита тем, что ничего не зная о программе, имея только исполняемый файл можно увидеть какие системные вызовы делает программа.

Ниже приведен результат отслеживания системных вызовов приведенной выше программы с помощью утилиты strace.

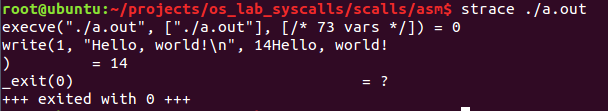


Рисунок 3. strace примера системного прерывания на языке ассемблер

Лог:

root@ubuntu:~/projects/os\_lab\_syscalls/scalls/asm$ strace ./a.out

execve("./a.out", ["./a.out"], [/\* 73 vars \*/]) = 0

write(1, "Hello, world!\n", 14Hello, world!

) = 14

\_exit(0) = ?

+++ exited with 0 +++

Как видно из лога strace данной программой осуществляется вызов write.

В следующей строке можно увидеть:

* куда осуществляется запись: 1(stdout)
* сообщение: Hello world\n
* количество символов:14

write(1, "Hello, world!\n", 14Hello, world!

) = 14

Далее в работе будут рассмотрены вызовы:

* sys\_stat - 18 - fs/stat.c,
* sys\_rename - 38 - fs/namei.c,
* sys\_mkdir - 39 - fs/namei.c,
* sys\_rmdir - 40 - fs/namei.c,
* sys\_fstat - 28 - fs/stat.c;

## Назначение и функциональность вызовов sys\_stat, sys\_rename, sys\_mkdir, sys\_rmdir, sys\_fstat

### Введение

Зачастую, системные вызовы не вызываются непосредственно, они имеют соответствующие функции обертки в системной библиотеке (например, glibc). С точки зрения программиста, системный вызов обычно выглядит как вызов функции из glibc.

Функции обертки, обычно, не делают ничего, кроме (Kerrisk, SYSCALL(2). Linux Programmer's Manual, 2015):

* копирования параметров в соответствующие регистры
* передачи управление ядру ОС, которое выполняет непосредственно в режиме ядра нужные действия
* установки errno(номер последней ошибки), если системный вызов возвращает ошибку, когда процессор переключается в пользовательский режим.

Однако системный вызов, как частный случай вызова такой функции или подпрограммы, следует отличать от более общего обращения к системной библиотеке, поскольку последнее может и не требовать выполнения привилегированных операций.

При программировании на C и C++ для системных вызовов подключаются соответствующие заголовочные файлы системной библиотеки glibc.

* sys/stat.h - sys\_stat, sys\_mkdir, sys\_fstat
* stdio.h - sys\_rename
* unistd.h - sys\_rmdir

Далее рассмотрим каждый системный вызов по отдельности.

### Системные вызовы sys\_stat, sys\_fstat

#### Сигнатура stat

1. **int** stat(**const** **char** \*pathname, **struct** stat \*buf);

#### Описание stat

Системный вызов, предназначен для получения информации о файле.

Функция *stat()* вносит в структуру, на которую указывает *buf*, информацию, содержащуюся в файле, связанном с указателем *pathname* - путь до файла.

При успешном заполнении структуры *stat* возвращается 0. В случае неудачи возвращается —1. (Kerrisk, STAT(2). Linux Programmer's Manual, 2014)

Структура *stat* включает следующие поля:

1. **struct** stat {
2. dev\_t     st\_dev;         /\* ID of device containing file \*/
3. ino\_t     st\_ino;         /\* inode number \*/
4. mode\_t    st\_mode;        /\* protection \*/
5. nlink\_t   st\_nlink;       /\* number of hard links \*/
6. uid\_t     st\_uid;         /\* user ID of owner \*/
7. gid\_t     st\_gid;         /\* group ID of owner \*/
8. dev\_t     st\_rdev;        /\* device ID (if special file) \*/
9. off\_t     st\_size;        /\* total size, in bytes \*/
10. blksize\_t st\_blksize;     /\* blocksize for filesystem I/O \*/
11. blkcnt\_t  st\_blocks;      /\* number of 512B blocks allocated \*/
13. /\* Since Linux 2.6, the kernel supports nanosecond
14. precision for the following timestamp fields.
15. For the details before Linux 2.6, see NOTES. \*/
17. **struct** timespec st\_atim;  /\* time of last access \*/
18. **struct** timespec st\_mtim;  /\* time of last modification \*/
19. **struct** timespec st\_ctim;  /\* time of last status change \*/
21. #define st\_atime st\_atim.tv\_sec      /\* Backward compatibility \*/
22. #define st\_mtime st\_mtim.tv\_sec
23. #define st\_ctime st\_ctim.tv\_sec
24. };

#### Пример stat

Рассмотрим на примере, информацию, которою можно получить воспользовавшись данной функцией. Запустим следующую программу.

Листинг – *scalls/stat/main.cpp*:

1. #include <sys/stat.h>
2. #include <stdlib.h>
3. #include <iostream>
4. #include <fstream>
6. **int** main() {
7. std::string fpath = "test.txt";
8. std::ifstream f ( fpath.c\_str() );
9. **struct** stat finfo;
11. **if**( !f.good() ) {
12. std::cout << "Cannot open file.\n";
13. f.close();
14. exit(1);
15. }
16. f.close();
18. stat( fpath.c\_str(), &finfo );
20. std::cout << "ID of device containing file\t\t" << finfo.st\_dev << "\n";
21. std::cout << "inode number\t\t\t\t" << finfo.st\_ino << "\n";
22. std::cout << "protection\t\t\t\t" << finfo.st\_mode << "\n";
23. std::cout << "number of hard links\t\t\t" << finfo.st\_nlink << "\n";
24. std::cout << "user ID of owner\t\t\t" << finfo.st\_uid << "\n";
25. std::cout << "group ID of owner\t\t\t" << finfo.st\_gid << "\n";
26. std::cout << "device ID (if special file)\t\t" << finfo.st\_rdev << "\n";
27. std::cout << "total size, in bytes\t\t\t" << finfo.st\_size << "\n";
28. std::cout << "blocksize for filesystem I/O\t\t" << finfo.st\_blksize << "\n";
29. std::cout << "number of 512B blocks allocated\t\t" << finfo.st\_blocks << "\n";
31. exit(0);
32. }

Содержание файла test.txt:

This is a sample file.

Результат выполнения:

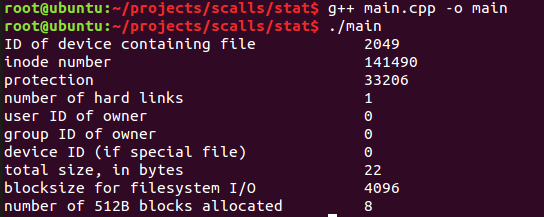


Рисунок 4. Сборка и выполнение примера программы, syscall - stat

Лог:

root@ubuntu:~/projects/scalls/stat$ ./main

ID of device containing file 2049

inode number 141490

protection 33206

number of hard links 1

user ID of owner 0

group ID of owner 0

device ID (if special file) 0

total size, in bytes 22

blocksize for filesystem I/O 4096

number of 512B blocks allocated 8

#### Просмотр вызовов strace stat

Утилита strace выводит все системные вызовы, которые осуществляет программа. Часто удобно отфильтровать вывод утилиты по конкретному вызову. Для этого можно воспользоваться ключом –e. Ниже приведен лог отслеживания системных вызовов stat утилитой strace приведенной выше программы. Однако, нужно знать, что на самом деле здесь вызывается stat64.



Рисунок 5. strace stat64

Лог:

root@ubuntu:~/projects/os\_lab\_syscalls/scalls/stat$ strace -e stat64 ./main

stat64("test.txt", {st\_mode=S\_IFREG|0666, st\_size=22, ...}) = 0

Первый параметр test.txt – файл, о котором запрашивается информация.

#### Пример fstat

Аналогично работает функция *fstat*, единственное отличие - она принимает первым параметром не путь до файла, а дескриптор. Пример аналогичной программы с использованием *fstat*.

Листинг – *scalls/fstat/main.cpp*:

1. #include <sys/stat.h>
2. #include <stdlib.h>
3. #include <iostream>
4. #include <fstream>
6. **int** main() {
7. std::string fpath = "test.txt";
8. **struct** stat finfo;
9. **FILE** \* pf = fopen( fpath.c\_str(), "r" );
11. **if**( pf == NULL ) {
12. std::cout << "Cannot open file.\n";
13. fclose( pf );
14. exit( 1 );
15. }
17. fstat( fileno(pf), &finfo );
18. fclose( pf );
20. std::cout << "ID of device containing file\t\t" << finfo.st\_dev << "\n";
21. std::cout << "inode number\t\t\t\t" << finfo.st\_ino << "\n";
22. std::cout << "protection\t\t\t\t" << finfo.st\_mode << "\n";
23. std::cout << "number of hard links\t\t\t" << finfo.st\_nlink << "\n";
24. std::cout << "user ID of owner\t\t\t" << finfo.st\_uid << "\n";
25. std::cout << "group ID of owner\t\t\t" << finfo.st\_gid << "\n";
26. std::cout << "device ID (if special file)\t\t" << finfo.st\_rdev << "\n";
27. std::cout << "total size, in bytes\t\t\t" << finfo.st\_size << "\n";
28. std::cout << "blocksize for filesystem I/O\t\t" << finfo.st\_blksize << "\n";
29. std::cout << "number of 512B blocks allocated\t\t" << finfo.st\_blocks << "\n";
31. exit( 0 );
32. }

Результат выполнения аналогичен предыдущему.

#### Просмотр вызовов strace fstat

Ниже приведен лог выполнения *strace –e fstat64 ./main*. Как можно заметить данная программа делает 7 вызовов *fstat*. Но только последний соответствует строке программы:

1. fstat( fileno(pf), &finfo );

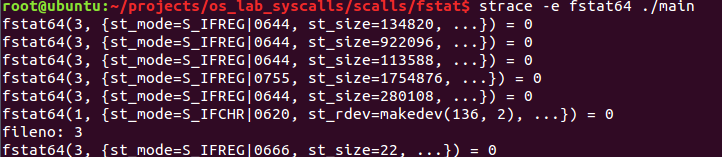


Рисунок 6. strace fstat64

Лог:

root@ubuntu:~/projects/os\_lab\_syscalls/scalls/fstat$ strace -e fstat64 ./main

fstat64(3, {st\_mode=S\_IFREG|0644, st\_size=134820, ...}) = 0

fstat64(3, {st\_mode=S\_IFREG|0644, st\_size=922096, ...}) = 0

fstat64(3, {st\_mode=S\_IFREG|0644, st\_size=113588, ...}) = 0

fstat64(3, {st\_mode=S\_IFREG|0755, st\_size=1754876, ...}) = 0

fstat64(3, {st\_mode=S\_IFREG|0644, st\_size=280108, ...}) = 0

fstat64(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(136, 2), ...}) = 0

fileno: 3

fstat64(3, {st\_mode=S\_IFREG|0666, st\_size=22, ...}) = 0

### Системные вызовы sys\_mkdir, sys\_rmdir

#### Сигнатура mkdir

1. **int** mkdir(**const** **char** \*path, mode\_t mode);

#### Описание mkdir

Функция *mkdir()* создает каталог, используя путь, на который указывает параметр *path*. В случае успеха функция *mkdir()* возвращает 0, в противном случае возвращается -1. Для задания прав используется второй параметр. Описание типов - <http://pubs.opengroup.org/onlinepubs/007908799/xsh/sysstat.h.html>.

#### Пример mkdir

В следующем примере рассматривается использование данной функции.

Листинг – *scalls/mkdir/main.cpp*:

1. #include <sys/stat.h>
3. **int** main() {
4. mkdir("test\_dir", S\_IRWXU | S\_IRWXG | S\_IROTH | S\_IXOTH);
5. **return** 0;
6. }

Результат выполнения:

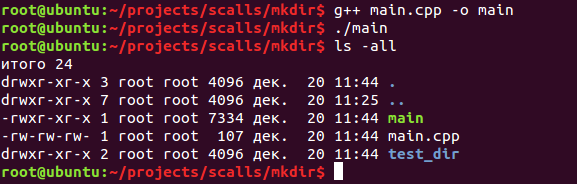


Рисунок 7. Сборка и выполнение примера программы, syscall - mkdir

Лог:

root@ubuntu:~/projects/scalls/mkdir$ g++ main.cpp -o main

root@ubuntu:~/projects/scalls/mkdir$ ./main

root@ubuntu:~/projects/scalls/mkdir$ ls -all

итого 24

drwxr-xr-x 3 root root 4096 дек. 20 11:44 .

drwxr-xr-x 7 root root 4096 дек. 20 11:25 ..

-rwxr-xr-x 1 root root 7334 дек. 20 11:44 main

-rw-rw-rw- 1 root root 107 дек. 20 11:44 main.cpp

drwxr-xr-x 2 root root 4096 дек. 20 11:44 test\_dir

#### Просмотр вызовов strace mkdir

Ниже приведены примеры отслеживания системных вызов утилитой strace, из которых можно увидеть различные результаты завершения вызовов mkdir рассмотренной выше программы.

Первый вызов, когда папка еще не была создана. Можно увидеть, что результат выполнения: 0.



Рисунок 8. strace mkdir

Лог:

root@ubuntu:~/projects/os\_lab\_syscalls/scalls/mkdir$ strace -e mkdir ./main

mkdir("test\_dir", 0775) = 0

+++ exited with 0 +++

И второй вызов, когда папка уже создана. Результат выполнения: -1.



Рисунок 9. strace mkdir

Лог:

root@ubuntu:~/projects/os\_lab\_syscalls/scalls/mkdir$ strace -e mkdir ./main

mkdir("test\_dir", 0775) = -1 EEXIST (File exists)

+++ exited with 0 +++

#### Сигнатура rmdir

1. **int** rmdir(**const** **char** \*path);

#### Описание rmdir

Функция *rmdir()* удаляет каталог, путь к которому определен параметром *path*. Для того, что­бы каталог можно было удалить, он должен быть пустым, но не должен быть текущим или корне­вым каталогом.

В случае успеха функция *rmdir()* возвращает значение 0. В противном случае возвращается - 1.

#### Пример rmdir

Листинг – *scalls/rmdir/main.cpp*:

1. #include <iostream>
2. #include <unistd.h>
4. **int** main() {
5. **if** ( !rmdir( "test\_dir" ) ) {
6. std::cout << "test\_dir removed\n";
7. }
8. **return** 0;
9. }

Результат выполнения:

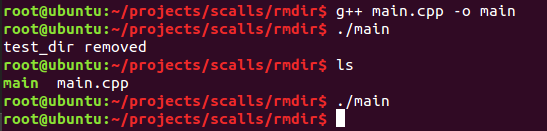


Рисунок 10. Сборка и выполнение примера программы, syscall - rmdir

Лог:

root@ubuntu:~/projects/scalls/rmdir$ g++ main.cpp -o main

root@ubuntu:~/projects/scalls/rmdir$ ./main

test\_dir removed

root@ubuntu:~/projects/scalls/rmdir$ ls

main main.cpp

root@ubuntu:~/projects/scalls/rmdir$ ./main

root@ubuntu:~/projects/scalls/rmdir$

#### Просмотр вызовов strace rmdir

Вызов rmdir, также как и mkdir, имеет разный результат выполнения. Это может зависеть от того, можно ли удалить папку.

Например, если папка есть: 0.

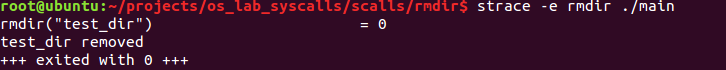


Рисунок 11. strace rmdir

Лог:

root@ubuntu:~/projects/os\_lab\_syscalls/scalls/rmdir$ strace -e rmdir ./main

rmdir("test\_dir") = 0

test\_dir removed

+++ exited with 0 +++

Если нет: -1.



Рисунок 12. strace rmdir

Лог:

root@ubuntu:~/projects/os\_lab\_syscalls/scalls/rmdir$ strace -e rmdir ./main

rmdir("test\_dir") = -1 ENOENT (No such file or directory)

+++ exited with 0 +++

### Системный вызов sys\_rename

#### Сигнатура rename

1. **int** rename(**const** **char** \*oldfname, **const** **char** \*newfname)

#### Описание rename

Функция *rename()* изменяет название файла со старого *oldfname* на новое *newfname*. Новое имя *newfname* не должно совпадать с именами, имеющимися в каталоге. Функция *rename()* возвращает 0 в случае успеха и ненулевую величину - в случае ошибки (Kerrisk, RENAME(2). Linux Programmer's Manual, 2015).

Пример программы.

Листинг *– scalls/rename/main.cpp*:

1. #include <stdio.h>
3. **int** main () {
4. **int** result;
5. **char** oldname[] = "oldname.txt";
6. **char** newname[] = "newname.txt";
7. result = rename( oldname , newname );
8. **if** ( result == 0 )
9. puts ( "File successfully renamed" );
10. **else**
11. perror( "Error renaming file" );
12. **return** 0;
13. }

Результат выполнения:

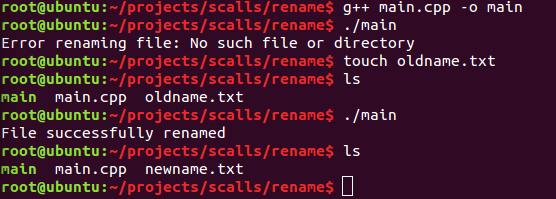


Рисунок 13. Сборка и выполнение примера программы, syscall - rename

Лог:

root@ubuntu:~/projects/scalls/rename$ g++ main.cpp -o main

root@ubuntu:~/projects/scalls/rename$ ./main

Error renaming file: No such file or directory

root@ubuntu:~/projects/scalls/rename$ touch oldname.txt

root@ubuntu:~/projects/scalls/rename$ ls

main main.cpp oldname.txt

root@ubuntu:~/projects/scalls/rename$ ./main

File successfully renamed

root@ubuntu:~/projects/scalls/rename$ ls

main main.cpp newname.txt

#### Просмотр вызовов strace rename

Просмотр вызовов если файл есть:

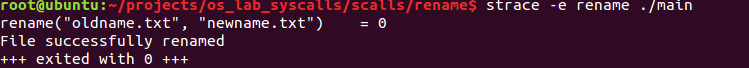


Рисунок 14. strace rename

Лог:

root@ubuntu:~/projects/os\_lab\_syscalls/scalls/rename$ strace -e rename ./main

rename("oldname.txt", "newname.txt") = 0

File successfully renamed

+++ exited with 0 +++

Просмотр вызовов если файла нет:

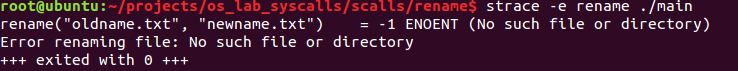


Рисунок 15. strace rename

Лог:

root@ubuntu:~/projects/os\_lab\_syscalls/scalls/rename$ strace -e rename ./main

rename("oldname.txt", "newname.txt") = -1 ENOENT (No such file or directory)

Error renaming file: No such file or directory

+++ exited with 0 +++

Как видно результат выполнения разный.

## Разбор исходного кода системных вызовов ядра

### Введение

Исходный код системных вызовов в ядре находится в зависимости от того к какой подсистеме относится вызов. Все рассмотренные в данной работе системные вызовы относятся к подсистеме fs (файловая система). Соответственно лежат в файлах:

* *fs/stat.c – sys\_stat, sys\_fstat*
* *fs/namei.c - sys\_rename, sys\_mkdir, sys\_rmdir*

В файле *include/linux/syscalls.h* можно увидеть сигнатуры системных вызовов:

1. asmlinkage **long** sys\_stat64(**const** **char** \_\_user \*filename, **struct** stat64 \_\_user \*statbuf);
2. asmlinkage **long** sys\_fstat64(unsigned **long** fd, **struct** stat64 \_\_user \*statbuf);
3. asmlinkage **long** sys\_mkdir(**const** **char** \_\_user \*pathname, umode\_t mode);
4. asmlinkage **long** sys\_rename(**const** **char** \_\_user \*oldname, **const** **char** \_\_user \*newname);
5. asmlinkage **long** sys\_rmdir(**const** **char** \_\_user \*pathname);

А так же макросы объявления системных вызовов:

1. #define SYSCALL\_DEFINE1(name, ...) SYSCALL\_DEFINEx(1, \_##name, \_\_VA\_ARGS\_\_)
2. #define SYSCALL\_DEFINE2(name, ...) SYSCALL\_DEFINEx(2, \_##name, \_\_VA\_ARGS\_\_)
3. #define SYSCALL\_DEFINE3(name, ...) SYSCALL\_DEFINEx(3, \_##name, \_\_VA\_ARGS\_\_)
4. #define SYSCALL\_DEFINE4(name, ...) SYSCALL\_DEFINEx(4, \_##name, \_\_VA\_ARGS\_\_)
5. #define SYSCALL\_DEFINE5(name, ...) SYSCALL\_DEFINEx(5, \_##name, \_\_VA\_ARGS\_\_)
6. #define SYSCALL\_DEFINE6(name, ...) SYSCALL\_DEFINEx(6, \_##name, \_\_VA\_ARGS\_\_)
8. #define SYSCALL\_DEFINEx(x, sname, ...)              \
9. SYSCALL\_METADATA(sname, x, \_\_VA\_ARGS\_\_)         \
10. \_\_SYSCALL\_DEFINEx(x, sname, \_\_VA\_ARGS\_\_)

, так для объявления системного вызова *mkdir* в файле *fs/namei.c* используется следующий код:

1. SYSCALL\_DEFINE2(mkdir, **const** **char** \_\_user \*, pathname, umode\_t, mode)

В файлах *fs/stat.c* и *fs/namei.c* можно увидеть экспортные функции(в дальнейшем при перехвате именно они и будут подменяться):

1. EXPORT\_SYMBOL(vfs\_mkdir);
2. EXPORT\_SYMBOL(vfs\_rename);
3. EXPORT\_SYMBOL(vfs\_rmdir);
4. EXPORT\_SYMBOL(vfs\_stat);
5. EXPORT\_SYMBOL(vfs\_fstat);

Далее рассмотрим исходный код системных вызовов по отдельности.

### Исходный код вызова mkdir

Объявление вызова *mkdir*:

1. SYSCALL\_DEFINE2(mkdir, **const** **char** \_\_user \*, pathname, umode\_t, mode)
2. {
3. **return** sys\_mkdirat(AT\_FDCWD, pathname, mode);
4. }

Как можно увидеть это эквивалентно вызову *mkdirat*, за исключением первого параметра, который подменяется на файловый дескриптор текущей рабочей директории *AT\_FDCWD*.

1. SYSCALL\_DEFINE3(mkdirat, **int**, dfd, **const** **char** \_\_user \*, pathname, umode\_t, mode)
2. {
3. **struct** dentry \*dentry;
4. **struct** path path;
5. **int** error;
6. unsigned **int** lookup\_flags = LOOKUP\_DIRECTORY;
8. retry:
9. dentry = user\_path\_create(dfd, pathname, &path, lookup\_flags);
10. **if** (IS\_ERR(dentry))
11. **return** PTR\_ERR(dentry);
13. **if** (!IS\_POSIXACL(path.dentry->d\_inode))
14. mode &= ~current\_umask();
15. error = security\_path\_mkdir(&path, dentry, mode);
16. **if** (!error)
17. error = vfs\_mkdir(path.dentry->d\_inode, dentry, mode);
18. done\_path\_create(&path, dentry);
19. **if** (retry\_estale(error, lookup\_flags)) {
20. lookup\_flags |= LOOKUP\_REVAL;
21. **goto** retry;
22. }
23. **return** error;
24. }

В данной функции проводятся базовые проверки, формируются параметры и далее вызывается экспортируемая из модуля функция vfs\_mkdir() – которая и будет перехвачена.

1. **int** vfs\_mkdir(**struct** inode \*dir, **struct** dentry \*dentry, umode\_t mode)
2. {
3. **int** error = may\_create(dir, dentry);
4. unsigned max\_links = dir->i\_sb->s\_max\_links;
6. **if** (error)
7. **return** error;
9. **if** (!dir->i\_op->mkdir)
10. **return** -EPERM;
12. mode &= (S\_IRWXUGO|S\_ISVTX);
13. error = security\_inode\_mkdir(dir, dentry, mode);
14. **if** (error)
15. **return** error;
17. **if** (max\_links && dir->i\_nlink >= max\_links)
18. **return** -EMLINK;
20. error = dir->i\_op->mkdir(dir, dentry, mode);
21. **if** (!error)
22. fsnotify\_mkdir(dir, dentry);
23. **return** error;
24. }

В функции vfs\_mkdir() вызывается функция dir->i\_op->mkdir(dir, dentry, mode);

Данная функция находится в структуре i\_op – операции которые можно делать со структурой inode, данные функции определены для каждой конкретной файловой системы. Например, для файловой системы ext4 в файле fs/ext4/namei.c, можно найти определение структуры inode\_operations.

1. **const** **struct** inode\_operations ext4\_dir\_inode\_operations = {
2. .create = ext4\_create,
3. .lookup = ext4\_lookup,
4. .link = ext4\_link,
5. .unlink = ext4\_unlink,
6. .symlink = ext4\_symlink,
7. .mkdir = ext4\_mkdir,
8. .rmdir = ext4\_rmdir,
9. .mknod = ext4\_mknod,
10. .tmpfile = ext4\_tmpfile,
11. .rename = ext4\_rename,
12. .setattr = ext4\_setattr,
13. .setxattr = generic\_setxattr,
14. .getxattr = generic\_getxattr,
15. .listxattr = ext4\_listxattr,
16. .removexattr = generic\_removexattr,
17. .get\_acl = ext4\_get\_acl,
18. .fiemap = ext4\_fiemap,
19. };

А также соответствующую функцию ext4\_mkdir.

1. **static** **int** ext4\_mkdir(**struct** inode \*dir, **struct** dentry \*dentry, umode\_t mode)
2. ...

Для других системных вызовов аналогично можно найти в исходных кодах соответствующие функции.

## Перехват системных вызовов

### Введение

Перехват системных вызовов позволяет осуществлять контроль ключевого компонента ядра ОС — интерфейса системных вызовов, что, в свою очередь, даёт возможность инспектировать запросы прикладного ПО к сервисам ядра.

Перехват будем осуществлять одним из наиболее распространенных способов - на основе LKM (Loadable Kernel Module).

Загружаемый модуль ядра (общепринятое сокращение LKM - Loadable Kernel Module) - программный код, выполняемый в пространстве ядра. Главной особенностью LKM является возможность динамической загрузки и выгрузки (Мешков, 2003).

Каждый LKM состоит из двух основных функций (минимум):

* функция инициализации модуля. Вызывается при загрузке LKM в память:

1. **int** init\_module(**void**) { ... }

* функция выгрузки модуля:

1. **void** cleanup\_module(**void**) { ... }

Для реализации модуля, перехватывающего системный вызов, необходимо определить алгоритм перехвата. Алгоритм следующий:

* 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

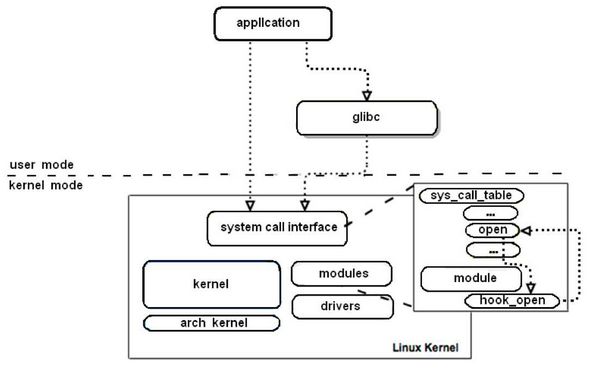
(A. М. Каннер, 2012)

Рисунок 16. Перехват системного вызова с помощью модуля ядра Linux

На рисунке 16 приведена графическая схема перехвата системного вызова, подменой

### Перехват системного вызова mkdir

В данном разделе рассмотрен перехват системного вызова mkdir. Для этого используется установка нового модуля ядра mkdir\_module. При установке модуля передается uid юзера, которому впоследствии должно быть запрещено создание директорий. Процесс сборки и установки модуля описан ниже.

Файл модуля *mkdir\_module.c.*

Листинг – *scalls/lsm\_mkdir/mkdir\_module.c* :

1. #include <linux/init.h>
2. #include <linux/module.h>
4. #include <linux/moduleparam.h>
5. #include <linux/unistd.h>
7. #include <linux/kernel.h>
8. #include <linux/syscalls.h>
9. #include <linux/delay.h>
10. #include <asm/paravirt.h>
11. #include <linux/sched.h>
12. #include <linux/cred.h>
13. #include <linux/uidgid.h>
15. MODULE\_LICENSE( "GPL" );
16. MODULE\_AUTHOR( "Dedkov Sergey" );
18. **static** **int** uid;
19. module\_param(uid, **int**, 0644);
21. unsigned **long** \*\*sys\_call\_table;
22. unsigned **long** original\_cr0;
24. asmlinkage **int** (\* ref\_sys\_mkdir) (**struct** inode \*dir, **struct** dentry \*dentry, **int** mode);
25. asmlinkage **int** new\_sys\_mkdir (**struct** inode \*dir, **struct** dentry \*dentry, **int** mode) {
26. **if** (uid == get\_current\_user()->uid.val)
27. **return** -1;
28. printk ("mkdir hijacked! uid: %d \n", get\_current\_user()->uid.val);
29. printk ("mkdir hijacked! passed uid: %d \n", uid);
30. **return** ref\_sys\_mkdir(dir, dentry, mode);
31. }
33. asmlinkage **int** (\* ref\_sys\_fstat) (unsigned **int** fd, **struct** kstat \*stat);
35. **static** unsigned **long** \*\*aquire\_sys\_call\_table(**void**)
36. {
37. unsigned **long** **int** offset = PAGE\_OFFSET;
38. unsigned **long** \*\*sct;
40. **while** (offset < ULLONG\_MAX) {
41. sct = (unsigned **long** \*\*)offset;
43. **if** (sct[\_\_NR\_close] == (unsigned **long** \*) sys\_close)
44. **return** sct;
46. offset += **sizeof**(**void** \*);
47. }
49. **return** NULL;
50. }
52. **static** **int** \_\_init md\_init(**void**)
53. {
54. **if**(!(sys\_call\_table = aquire\_sys\_call\_table()))
55. **return** -1;
57. original\_cr0 = read\_cr0();
59. write\_cr0(original\_cr0 & ~0x00010000);
60. ref\_sys\_mkdir = (**void** \*)sys\_call\_table[\_\_NR\_mkdir];
61. sys\_call\_table[\_\_NR\_mkdir] = (unsigned **long** \*)new\_sys\_mkdir;
62. write\_cr0(original\_cr0);
64. **return** 0;
65. }
67. **static** **void** \_\_exit md\_exit(**void**)
68. {
69. **if**(!sys\_call\_table) {
70. **return**;
71. }
73. write\_cr0(original\_cr0 & ~0x00010000);
74. sys\_call\_table[\_\_NR\_mkdir] = (unsigned **long** \*)ref\_sys\_mkdir;
75. write\_cr0(original\_cr0);
77. msleep(2000);
78. }
80. module\_init(md\_init);
81. module\_exit(md\_exit);

Makefile.

Листинг – *scalls/lsm\_mkdir/Makefile*:

1. CURRENT = $(shell uname -r)
2. KDIR = /lib/modules/$(CURRENT)/build
3. PWD = $(shell pwd)
4. TARGET1 = mkdir\_module
5. obj-m := $(TARGET1).o
6. **default**:
7. $(MAKE) -C $(KDIR) M=$(PWD) modules
8. clean:
9. @rm -f \*.o .\*.cmd .\*.flags \*.mod.c \*.order
10. @rm -f .\*.\*.cmd \*~ \*.\*~ TODO.\*
11. @rm -fR .tmp\*
12. @rm -rf .tmp\_versions
13. disclean: clean
14. @rm \*.ko \*.symvers

Сборка модуля:

Сборка модуля осуществляется командой *make* в соответствии файлом - *Makefile*.

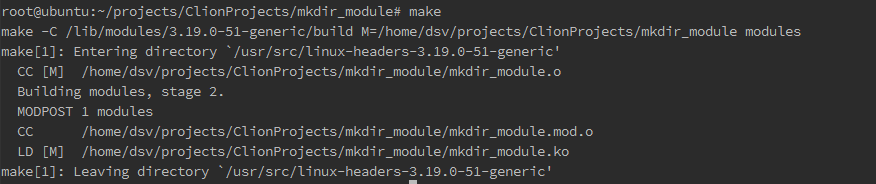


Рисунок 17. Сборка модуля с перехватом системного вызова mkdir

Лог:

root@ubuntu:~/projects/ClionProjects/mkdir\_module# make

make -C /lib/modules/3.19.0-51-generic/build M=/home/dsv/projects/ClionProjects/mkdir\_module modules

make[1]: Entering directory `/usr/src/linux-headers-3.19.0-51-generic'

CC [M] /home/dsv/projects/ClionProjects/mkdir\_module/mkdir\_module.o

Building modules, stage 2.

MODPOST 1 modules

CC /home/dsv/projects/ClionProjects/mkdir\_module/mkdir\_module.mod.o

LD [M] /home/dsv/projects/ClionProjects/mkdir\_module/mkdir\_module.ko

make[1]: Leaving directory `/usr/src/linux-headers-3.19.0-51-generic'

Установка модуля:



Рисунок 18. Установка модуля

Лог:

root@ubuntu:~/projects/ClionProjects/mkdir\_module# insmod mkdir\_module.ko uid=1000

Проверка того, что модуль установлен:



Рисунок 19. Проверка установки модуля

Лог:

root@ubuntu:~/projects/ClionProjects/mkdir\_module# lsmod | grep mkdir

mkdir\_module 16384 0

Проверка работы:

Рассмотрим двух пользователей:

* dsv, uid – 1000
* sergey, uid – 1001

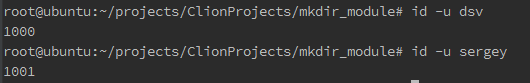


Рисунок 20. Идентификаторы пользователей для примера

Лог:

root@ubuntu:~/projects/ClionProjects/mkdir\_module# id -u dsv

1000

root@ubuntu:~/projects/ClionProjects/mkdir\_module# id -u sergey

1001

Для проверки попробуем создать папку каждым из этих пользователей.

Первый пользователь.



Рисунок 21. Создание папки после установки модуля с перехватом, uid пользователя - 1000

Лог:

dsv@ubuntu:~$ mkdir 1234

mkdir: cannot create directory ‘1234’: No such file or directory

Второй пользователь.

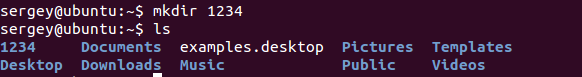


Рисунок 22. Создание папки после установки модуля с перехватом, uid пользователя - 1001

Лог:

sergey@ubuntu:~$ mkdir 1234

sergey@ubuntu:~$ ls

1234 Documents examples.desktop Pictures Templates

Desktop Downloads Music Public Videos

Как видно из лога содавть директорию пользователю, у которого uid – 1000 не удалось. Так же для пользователей у которых uid отличный от 1000 можно увидеть записи в dmesg.

Частичный вывод результата *dmesg | grep mkdir*:



Рисунок 23. Вывод dmesg после создания папки

Лог:

[10640.945324] mkdir hijacked! uid: 1001

[10640.945332] mkdir hijacked! passed uid: 1000

Как видим последнее сообщение - результат перехвата системного вызова *mkdir*.

Для восстановления стандартного системного вызова воспользуемся командой *rmmod*:

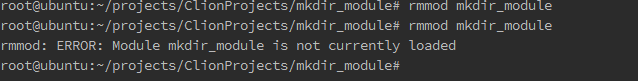


Рисунок 24. Удаления модуля

Лог:

root@ubuntu:~/projects/ClionProjects/mkdir\_module# rmmod mkdir\_module

root@ubuntu:~/projects/ClionProjects/mkdir\_module# rmmod mkdir\_module

rmmod: ERROR: Module mkdir\_module is not currently loaded

root@ubuntu:~/projects/ClionProjects/mkdir\_module#

Рассмотрим код модуля подробнее:

Объявление нового и старого системных вызовов *mkdir*:

1. asmlinkage **int** (\* ref\_sys\_mkdir) (**struct** inode \*dir, **struct** dentry \*dentry, **int** mode);
2. asmlinkage **int** new\_sys\_mkdir (**struct** inode \*dir, **struct** dentry \*dentry, **int** mode) {
3. **if** (uid == get\_current\_user()->uid.val)
4. **return** -1;
5. printk ("mkdir hijacked! uid: %d \n", get\_current\_user()->uid.val);
6. printk ("mkdir hijacked! passed uid: %d \n", uid);
7. **return** ref\_sys\_mkdir(dir, dentry, mode);
8. }

Сигнатуры этих функций можно найти в файле *fs/namei.c*:

1. **int** vfs\_mkdir(**struct** inode \*dir, **struct** dentry \*dentry, umode\_t mode)

Заметим, что новая функция вызывает старую, таким образом сохраняя функциональность для юзера у которого uid отличен от переданного.

Поиск таблицы осуществляется в начале функции инициализации модуля:

1. sys\_call\_table = aquire\_sys\_call\_table()

Функция:

1. **static** unsigned **long** \*\*aquire\_sys\_call\_table(**void**)
2. {
3. unsigned **long** **int** offset = PAGE\_OFFSET;
4. unsigned **long** \*\*sct;
6. **while** (offset < ULLONG\_MAX) {
7. sct = (unsigned **long** \*\*)offset;
9. **if** (sct[\_\_NR\_close] == (unsigned **long** \*) sys\_close)
10. **return** sct;
12. offset += **sizeof**(**void** \*);
13. }
15. **return** NULL;
16. }

Возвращает ссылку на таблицу системных вызовов или *NULL*.

Регистрация функций инициализации и удаления модуля:

1. module\_init(md\_init);
2. module\_exit(md\_exit);

Стоит отметить, что есть несколько вариантов перехвата системных вызовов (A. М. Каннер, 2012):

* найти адрес *sys\_call\_table* в файле */boot/System.map*, содержащем все используемые ядром символы и их адреса в памяти ядра (такой файл создается при каждой компиляции ядра), например так - *grep* *sys\_call\_table/ /boot/System.map | awk ’{print \$$1}’\** и передать в модуль ядра через *Makefile*. Таким образом, данный метод фактически не позволяет распространять модуль ядра в собранном виде (в ви­де бинарного файла с расширением .ко) - его необходимо будет компилировать на каждой новой ОС;
* найти адрес *sys\_call\_table* в файле */boot/System.map* с помощью функции модуля ядра — в данном случае пересобирать модуль под конкретную систему не обязательно;
* найти адрес *sys\_call\_table* в памяти ядра ОС с помощью разбора адресов соседних структур (например *«loops\_per\_jiffy»* и *«cpu\_boot\_data»*). В данном случае необходимо учитывать, что не во всех версиях ядра ОС таблица системных вызовов будет располагаться именно между указанными структурами;
* найти адрес *sys\_call\_table* в памяти ядра ОС, для примера это можно сделать обычным перебором со сравнением определенного элемента таблицы системных вызовов с тем значением, которое должно быть в качестве этого элемента

## Вывод

В данной работе был рассмотрен принцип работы системных вызовов в первой части и приведен пример на языке ассемблер. Из которого видно, как на нижнем уровне осуществляются системные вызовы.

Затем рассмотрена функциональность системных вызовов - *sys\_stat, sys\_rename, sys\_mkdir, sys\_rmdir, sys\_fstat*. Приведены примеры принципов работы с нимим на языке c++.

В завершающей части рассмотрен механизм перехвата системных вызовов путем загрузки модуля ядра.

Это может потребоваться, например, для внедрения собственных ме­ханизмов безопасности в ОС Linux. Фактически требуется написать собственные реализации основ­ных системных вызовов, которые в зависимости от успешности/неуспешности определенных прове­рок будут вызывать/не вызывать выполнение эта­лонных системных вызовов. Таким образом стано­вится возможным реализовать собственные механизмы разграниче­ния доступа, которые будут работать непосредственно до отработки всех штат­ных подсистем разграничения доступа ОС. Сами функции, «переопределяющие» работу системных вызовов логично описать в загружаемом модуле ядра, а при инициализации этого модуля необходимо заменять адреса системных вызовов в таблице системных вызовов на адреса переопределяющих их функций

# Разработка драйверов ОС Linux. Драйвер сетевой карты.

## Цель работы

* Изучить общие принципы разработки драйверов ОС Linux. Написать тестовый модуль – каркас, для последующей разработки драйвера.
* Изучить принципы написания драйверов символьных устройств. Написать драйвер, для общего понимания работы драйверов в ОС Linux.
* Изучить принципы разработки драйверов для PCI устройств. Разработать драйвер сетевой карты.

## Введение

Драйвер - программное обеспечение, с помощью которого другое программное обеспечение (операционная система) получает доступ к аппаратному обеспечению некоторого устройства. (Драйвер, 2016)

Таким образом драйвер – программное обеспечение, которое реализует программный интерфейс аппаратуры, скрывая детали работы устройства. Таким образом, пользователю не приходиться вдаваться в детали аппаратных средств, достаточно выполнять стандартизированные вызовы, которые не зависят от специфики драйвера. Задача драйвера – перевести данные вызовы в специфичный для данного устройства операции.

В ОС Linux драйверы могут быть собраны отдельно от ядра и подгружены в процессе работы, когда это необходимо. Для этой цели используются LKM (Loadable Kernel Module) – загружаемые модули. (Jonathan Corbet, 2005)

В ОС Linux устройства разделяются на 3 типа:

* Символьные
* Блочные
* Сетевые

В данной работе рассматриваются драйвера для:

* символьного устройства – для понимания общих принципов разработки драйверов
* сетевого устройства – из задания к лабораторной работе. Используемая сетевая карта – TP-Link TF-3200. Версия ядра Linux – 2.6.38.8.

В ходе данной работы использовались Ubuntu 14.04.4 LTS и 11.04, которые основаны на Debian GNU/Linux. Соответственно использовался пакетный менеджер APT(Advanced Packaging Tool).

К примерам кода, приводимого в работе, указываются файлы листингов, а также версия ядра на котором модуль запускался.

Полные листинги находятся в репозитории по ссылке - <https://github.com/dsvgit/os_lab_syscalls>.

## Разработка каркаса драйвера – загружаемый модуль

Загружаемый модуль ядра (loadable kernel module, LKM) - в информатике - объектный файл, содержащий код, расширяющий возможности ядра операционной системы. Модули используются, чтобы добавить поддержку нового оборудования или файловых систем или для добавления новых системных вызовов. Когда функциональность, предоставляемая модулем, больше не требуется, он может быть выгружен, чтобы освободить память и другие ресурсы. (Загружаемый модуль ядра, 2016)

Модули ядра могут быть собраны и подгружены/выгружены во время работы ОС.

Реализация загрузки/выгрузки модуля ядра похожа на конструктор/деструктор объекта в терминах объектно-ориентированного программирования. Иными словами – модуль ядра должен реализовывать функцию загрузки и функцию выгрузки (что не всегда обязательно, например, для целей отладки). Функция регистрации модуля просто подготавливает работу модуля, для дальнейшей работы (регистрирует функции для последующего вызова, когда это будет необходимо).

Важно понимать, что код модуля ядра выполняется в пространстве ядра, привилегированном режиме работы процессора. Переход в этот режим осуществляется посредством системных вызовов (Jonathan Corbet, 2005).

Для работы с модулями потребуются заголовочные файлы текущей версии ядра, для их установки используется команда:

# sudo apt-get install linux-headers-$(uname -r)

Команда uname –r возвращает версию текущего ядра ОС.

При программировании в пространстве ядра не используются библиотеки, которые обычно подключаются при работе в пользовательском режиме. Таким образом программирование в режиме ядра предполагает ознакомление основными функциями и типами ядра ОС Linux.

Так же потребуются команды (Драйверы устройств в Linux. Часть 2, 2012):

* lsmod — список модулей, загруженных в текущий момент
* insmod <module\_file> — добавление / загрузка указанного файла модуля
* modprobe <module> — добавление / загрузка модуля вместе со всеми его зависимостями
* rmmod <module> — удаление / выгрузка модуля

Простейший модуль может содержать в себе функции регистрации и удаления модуля.

Ниже приводится пример простейшего модуля:

Листинг: *kernel\_module\_ex1/module\_ex1.c*

Версия ядра: *3.19.0-51*

1. #include <linux/init.h>
2. #include <linux/module.h>
4. MODULE\_LICENSE("GPL");
5. MODULE\_AUTHOR("Sergey Dedkov dsv.mail@yandex.ru");
6. MODULE\_DESCRIPTION("driver skeleton");
8. **static** **int** \_\_init md\_init(**void**) {
9. printk("+++ driver skeleton: md\_init()\n");
10. **return** 0;
11. }
13. **static** **void** \_\_exit md\_exit(**void**) {
14. printk("+++ driver skeleton: md\_exit()\n");
15. }
17. module\_init(md\_init);
18. module\_exit(md\_exit);

Для сборки используется утилита make и следующий Makefile

Листинг: *kernel\_module\_ex1/Makefile*

Версия ядра: *3.19.0-51*

1. ifeq ($(KERNELRELEASE),)
2. KERNELDIR ?= /lib/modules/$(shell uname -r)/build
3. PWD := $(shell pwd)
4. .PHONY: build clean
5. build:
6. $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
7. clean:
8. rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c
9. **else**
10. $(info Building with KERNELRELEASE = ${KERNELRELEASE})
11. obj-m := module\_ex1.o
12. endif

Рассмотренный модуль не делает ничего коме вывода сообщений при регистрации и удалении модуля в системный лог.

Ниже приводится лог сборки, установки, удаления модуля и системного вывода.

Сборка:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ make  
make -C /lib/modules/3.19.0-51-generic/build M=/home/dsv/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1 modules  
make[1]: Entering directory `/usr/src/linux-headers-3.19.0-51-generic'  
Building with KERNELRELEASE = 3.19.0-51-generic  
Building modules, stage 2.  
Building with KERNELRELEASE = 3.19.0-51-generic  
MODPOST 1 modules  
make[1]: Leaving directory `/usr/src/linux-headers-3.19.0-51-generic'

Результат сборки:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ ls  
Makefile module\_ex1.ko module\_ex1.mod.o modules.order  
module\_ex1.c module\_ex1.mod.c module\_ex1.o Module.symvers

Информация о модуле:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ modinfo module\_ex1.ko  
filename: /home/dsv/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1/module\_ex1.ko  
description: driver skeleton  
author: Sergey Dedkov dsv.mail@yandex.ru  
license: GPL  
srcversion: 9BAF37AF4A25F9CA7603D3B  
depends:   
vermagic: 3.19.0-51-generic SMP mod\_unload modversions

Установка модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ sudo insmod module\_ex1.ko

Проверка, того, что модуль установлен:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ lsmod | grep module  
module\_ex1 16384 0

Удаление модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ sudo rmmod module\_ex1

Чтения лога:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ dmesg | grep +++  
[ 6313.243085] +++ driver skeleton: md\_init()  
[ 6328.301227] +++ driver skeleton: md\_exit()

Далее следует пояснить некоторые части исходного кода модуля.

Самая важная часть – макросы module\_init и module\_exit(*linux/init.h*).

В них передаются функции инициализации и удаления модуля.

Со следующими сигнатурами:

**static** **int** md\_init(**void**)

**static** **void** md\_exit(**void**)

Сигнатуры данных функций включают ключевое слово static, т.к. не экспортируемые и будут использоваться только один раз при загрузке/выгрузке.

Функция md\_init возвращает int.

* 0 – модуль установлен
* отрицательное значение – ошибка при установке модуля

В некоторых случаях полезно возвращать, например -1, тогда модуль не будет установлен, не придется его удалять, но код до возврата выполнится в пространстве ядра. Такой способ удобно использовать для отладки. В таких случаях нет необходимости использовать функцию удаления модуля.

Помимо этого, перед именем функций можно увидеть макросы \_\_init и \_\_exit – они используются для оптимизации(*linux/module.h*). Они актуальны только тогда, когда тот же самый код будет встроен в ядро. Все функции, отмеченные как \_\_init, автоматически помещаются компилятором *GCC* при компиляции в секцию инициализации *init*, а все функции, отмеченные как \_\_exit, помещаются в секцию выхода *exit* образа ядра. Предполагается, что все функции с отметкой \_\_init должны выполняться только один раз во время загрузки системы (и не должны выполняться снова до следующей загрузки системы). Так что, как только они будут выполнены во время загрузки системы, ядро удалит их и высвободит оперативную память (будет удалена вся секция *init*). (Драйверы устройств в Linux. Часть 3, 2012)

Макросы MODULE\_LICENSE, MODULE\_AUTHOR, MODULE\_DESCRIPTION(*linux/module.h*) используются для того, чтобы задать информацию о модуле. Выше в логах было показано как посмотреть информацию собранного модуля, с расширением *.ko*. с помощью команды modinfo.

Для ввода в журнал сообщений ядра используется функция printk. Данная функция похожа на привычную printf, за исключением того, что не может выводить числа с плавающей точкой и не предназначена для выдачи дампа своих данных в консоль.

Все вызовы команды printk помещают свои выходные данные в кольцевой буфер (журнал) ядра. Затем демон *syslog*, работающий в пользовательском пространстве, берет их для окончательной обработки и перенаправляет на различные устройства в соответствие с тем, что задано в файле конфигурации */etc/syslog.conf*.

При использовании printk есть возможность задавать уровни журналирования при помощи макросов(*linux/kernel.h*):

#define KERN\_EMERG "<0>" /\* system is unusable \*/

#define KERN\_ALERT "<1>" /\* action must be taken immediately \*/

#define KERN\_CRIT "<2>" /\* critical conditions \*/

#define KERN\_ERR "<3>" /\* error conditions \*/

#define KERN\_WARNING "<4>" /\* warning conditions \*/

#define KERN\_NOTICE "<5>" /\* normal but significant condition \*/

#define KERN\_INFO "<6>" /\* informational \*/

#define KERN\_DEBUG "<7>" /\* debug-level messages \*/

Теперь, в зависимости от этих уровней журналирования (т. е. первых трех символов в строке формата), демон пользовательского пространства syslog перенаправляет каждое сообщения в соответствие с заданной конфигурацией. Обычно местом, куда перенаправлются сообщения всех уровней журналирования, является журнальный файл */var/log/messages*. Таким образом, все данные, выдаваемые командой printk, по умолчанию находятся в этом файле. Можно изменить настройку - например, пересылать сообщения на последовательный порт (например, */dev/ttyS0*).

Сообщения теперь находятся в буфере */var/log/messages*, причем в нем находятся сообщения не только из ядра, но и от различных демонов, работающих в пользовательском пространстве. К тому же, этот файл обычно нельзя читать от имени обычного пользователя. Поэтому для непосредственного разбора сообщений, находящихся в кольцевом буфере ядра, предоставляется утилита пользовательского пространства dmesg, которая выводит дамп буфера в стандартный выходной поток.

## Работа с устройствами. Символьный драйвер

Как было сказано выше в ОС Linux устройства разделяются на типы: символьные, блочные, сетевые. Для понимания общих принципов написания драйверов этот раздел содержит информацию о написании символьного драйвера.

Для любого приложения пользовательского пространства, предназначенного для работы с байт-ориентированным устройством (в пространстве аппаратных средств), следует использовать соответствующий драйвер символьного устройства (в пространстве ядра). Использование символьных драйверов осуществляется через соответствующие файлы символьных устройств, которые прикомпонованы к виртуальной файловой системе (*VFS*). Это означает, что приложение выполняет обычные файловые операции с файлом символьного устройства. Эти операции будут перетранслированы виртуальной файловой системой *VFS* в соответствующие функции в прикомпонованном драйвере символьного устройства. Затем для того, чтобы получить нужные результаты, с помощью этих функций осуществляется окончательный низкоуровневый доступ к реальному устройству (Драйверы устройств в Linux. Часть 4, 2012).

В этом полном подключении из приложения к устройству участвуют следующие четыре основных компонента:

* Приложение
* Файл символьного устройства
* Драйвер символьного устройства
* Символьное устройство

Приложение подключается к файлу устройства при помощи системного вызова *open*, открывающего файл устройства. Файлы устройств подключаются к драйверу устройства с помощью специального механизма регистрации, что осуществляется драйвером. Драйвер связывается с устройством с помощью специальных низкоуровневых операций, характерных для конкретного устройства. Таким образом, формируется полное соединение. При этом, файл символьного устройства не является реальным устройством, это просто специальная методика (*place-holder*) подключения реального устройства.

Для подключения приложения к файлу устройства используется имя файла устройства. Для подключения файла устройства к драйверу устройства используется номер файла устройства, а не имя файла. Приложение пользовательского пространства может использовать для файла устройства любое имя, а в пространстве ядра для связи между файлом устройства и драйвером устройства можно использовать тривиальный механизм индексации. Таким номером файла обычно является пара *<major, minor>* (старший и младший номера файла устройства). (Драйверы устройств в Linux. Часть 4, 2012)

* *major* – номер драйвера устройства (еще можно ассоциировать с типом устройства)
* *minor* – файл устройства.

Файлы всех устройств можно найти в директории /dev (файловая система dev). Так, например, для того, чтобы вывести весь список символьных устройств можно воспользоваться утилитой ls.

Ниже приводится лог вывода первых 10 символьных устройств.

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ ls -l /dev | grep "^c" | head  
crw-rw--— 1 root video 10, 175 Mar 14 17:26 agpgart  
crw-----— 1 root root 10, 235 Mar 14 17:26 autofs  
crw-----— 1 root root 10, 234 Mar 14 17:26 btrfs-control  
crw-----— 1 root root 5, 1 Mar 14 17:26 console  
crw-----— 1 root root 10, 60 Mar 14 17:26 cpu\_dma\_latency  
crw-----— 1 root root 10, 203 Mar 14 17:26 cuse  
crw-rw----+ 1 root audio 14, 9 Mar 14 17:26 dmmidi  
crw-----— 1 root root 10, 61 Mar 14 17:26 ecryptfs  
crw-rw--— 1 root video 29, 0 Mar 14 17:26 fb0  
crw-rw-rw- 1 root root 1, 7 Mar 14 17:26 full

В каждой строчке между группой и датой последней модификации модно увидеть два номера через запятую. Первый – *major* номер устройства – указывает на то, каким драйвером обслуживается устройство. Второй – *minor* номер файла устройства.

Для отображения драйверов устройств пользуются файловой системой *proc*. В файле */proc/devices* можно найти все драйвера, которые используются в текущей системе, разделенные на символьные и блочные. Для вывода можно воспользоваться утилитой cat. Ниже приводится частичный лог вывода файла */proc/devices*.

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ cat /proc/devices  
Character devices:  
1 mem  
4 /dev/vc/0  
4 tty  
4 ttyS

…  
  
Block devices:  
1 ramdisk  
2 fd  
259 blkext  
7 loop  
8 sd

По этим данным можно сопоставить файл устройства с драйвером, который его обслуживает.

Для понимания, того как регистрируются драйвера символьных устройств, рассмотрим пример модуля драйвера символьного устройства, зарегистрируем драйвер и создадим файл устройства.

Листинг: *chr\_device\_driver\_ex2/chr\_driver\_ex2.c*

Версия ядра: *3.19.0-51*

1. #include <linux/module.h>
2. #include <linux/version.h>
3. #include <linux/kernel.h>
4. #include <linux/types.h>
5. #include <linux/kdev\_t.h>
6. #include <linux/fs.h>
7. #include <linux/device.h>
8. #include <linux/cdev.h>
9. #include <linux/uaccess.h>
11. MODULE\_LICENSE("GPL");
12. MODULE\_AUTHOR("Sergey Dedkov");
13. MODULE\_DESCRIPTION("Test Character Driver");
15. **static** dev\_t first;
16. **static** **struct** cdev c\_dev;
17. **static** **struct** **class** \*cl;
19. **static** **char**   message[256] = {0};
20. **static** **short**  size\_of\_message;
22. **static** **int**      dev\_open(**struct** inode \*, **struct** file \*);
23. **static** **int**      dev\_release(**struct** inode \*, **struct** file \*);
24. **static** ssize\_t  dev\_read(**struct** file \*, **char** \*, **size\_t**, loff\_t \*);
25. **static** ssize\_t  dev\_write(**struct** file \*, **const** **char** \*, **size\_t**, loff\_t \*);
27. **static** **struct** file\_operations fops =
28. {
29. .owner = THIS\_MODULE,
30. .open = dev\_open,
31. .release = dev\_release,
32. .read = dev\_read,
33. .write = dev\_write
34. };
36. **static** **int** \_\_init chr\_init(**void**) /\* Constructor \*/
37. {
38. **if** (alloc\_chrdev\_region(&first, 0, 1, "chr\_test\_driver") < 0)
39. {
40. **return** -1;
41. }
43. **if** ((cl = class\_create(THIS\_MODULE, "char\_test\_drv")) == NULL)
44. {
45. unregister\_chrdev\_region(first, 1);
46. **return** -1;
47. }
49. **if** (device\_create(cl, NULL, first, NULL, "chrnull") == NULL)
50. {
51. class\_destroy(cl);
52. unregister\_chrdev\_region(first, 1);
53. **return** -1;
54. }
56. cdev\_init(&c\_dev, &fops);
58. **if** (cdev\_add(&c\_dev, first, 1) == -1)
59. {
60. device\_destroy(cl, first);
61. class\_destroy(cl);
62. unregister\_chrdev\_region(first, 1);
63. **return** -1;
64. }
66. **return** 0;
67. }
69. **static** **void** \_\_exit chr\_exit(**void**) /\* Destructor \*/
70. {
71. cdev\_del(&c\_dev);
72. device\_destroy(cl, first);
73. class\_destroy(cl);
74. unregister\_chrdev\_region(first, 1);
75. }
77. **static** **int** dev\_open(**struct** inode \*i, **struct** file \*f)
78. {
79. printk(KERN\_INFO "Character driver: open()\n");
80. **return** 0;
81. }
83. **static** **int** dev\_release(**struct** inode \*i, **struct** file \*f)
84. {
85. printk(KERN\_INFO "Character driver: close()\n");
86. **return** 0;
87. }
89. **static** ssize\_t dev\_read(**struct** file \*f, **char** \_\_user \*buffer, **size\_t** len, loff\_t \*offset)
90. {
91. **int** error\_count = 0;
92. **int** size = size\_of\_message;
94. error\_count = copy\_to\_user(buffer, message, size\_of\_message);
96. **if** (error\_count != 0)
97. {
98. printk(KERN\_INFO "Failed to send %d characters to the user\n", error\_count);
99. **return** -EFAULT;
100. }
102. printk(KERN\_INFO "Sent %d characters to the user, buffer: %s, message: %s \n", size\_of\_message, buffer, message);
103. size\_of\_message = 0;
104. **return** size;
105. }
107. **static** ssize\_t dev\_write(**struct** file \*f, **const** **char** \_\_user \*buffer, **size\_t** len, loff\_t \*offset)
108. {
109. sprintf(message, "%s(%d letters)\n", buffer, (**int**)len);   // appending received string with its length
110. size\_of\_message = strlen(message);                 // store the length of the stored message
111. printk(KERN\_INFO "Received %d characters from the user, message: %s \n", (**int**)len, message);
112. **return** len;
113. }
115. module\_init(chr\_init);
116. module\_exit(chr\_exit);

Сборка осуществляется аналогично предыдущему примеру. Makefile используется такой же.

Результат работы:

Сборка модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ make  
make -C /lib/modules/3.19.0-51-generic/build M=/home/dsv/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2 modules  
make[1]: Entering directory `/usr/src/linux-headers-3.19.0-51-generic'  
Building with KERNELRELEASE = 3.19.0-51-generic  
CC [M] /home/dsv/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2/chr\_driver\_ex2.o  
Building modules, stage 2.  
Building with KERNELRELEASE = 3.19.0-51-generic  
MODPOST 1 modules  
CC /home/dsv/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2/chr\_driver\_ex2.mod.o  
LD [M] /home/dsv/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2/chr\_driver\_ex2.ko  
make[1]: Leaving directory `/usr/src/linux-headers-3.19.0-51-generic'

Установка модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ sudo insmod chr\_driver\_ex2.ko

Проверка установки модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ lsmod | grep chr  
chr\_driver\_ex2 16384 0

Проверка добавления класса устройств:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ ls /sys/class/char\_test\_drv/  
chrnull

Проверка добавления драйвера. Частичный лог:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ cat /proc/devices  
Character devices:

…  
226 drm  
250 chr\_test\_driver  
251 hidraw  
…  
Block devices:  
1 ramdisk  
2 fd

…

Изменение прав доступа к файлу устройства, чтобы моно было с им работать:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ sudo chmod 777 /dev/chrnull

Вывод в файл устройства сообщения:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ sudo echo "test message" > /dev/chrnull

Чтение из файла устройства сообщения. Вывод в консоль:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ sudo cat /dev/chrnull   
test message  
(13 letters)

Далее представлен разбор кода модуля символьного драйвера. Опущены части кода, объяснение которых происходило в предыдущем разделе.

Драйвер символьного устройства для работы должен знать *major* и *minor* номера устройств. Для этих целей используется структура dev\_t(*linux/types.h*).

1. **static** dev\_t first;

Для работы с данной структурой используются следующие макросы(*linux/kdev\_t.h*):

* *MAJOR(dev\_t dev)* - из *dev* извлекается старший номер
* *MINOR(dev\_t dev)* - из *dev* извлекается младший номер
* *MKDEV(int major, int minor)* - из старшего и младшего номеров создается *dev*

Подключение файла устройства к драйверу выполняется за 2 шага:

* Выполняется регистрация файлов устройств для номеров *major, minor*
* Подключение операций, выполняемых над файлом устройства, к функциям драйвера устройства.

Первый шаг в модуле выполняет следующий код:

1. **if** (alloc\_chrdev\_region(&first, 0, 1, "chr\_test\_driver") < 0)
2. {
3. **return** -1;
4. }

С помощью функции alloc\_chrdev\_region (*linux/fs.h*) динамически определяется свободный старший номер и регистрируется число *count* среди номеров файлов устройств, начинающиеся с *baseminor*, с заданным именем файла *name.*

1. **int** alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count, **const** **char** \*name);

Первый параметр является выходным и заполняется свободными страшим и младшим номерами.

После этого регистрируется класс устройства (class\_create) и файл устройства (chrdev\_region). В функцию создания файла устройства третьим параметром передается структура dev\_t, ранее заполненная.

1. **if** ((cl = class\_create(THIS\_MODULE, "char\_test\_drv")) == NULL)
2. {
3. unregister\_chrdev\_region(first, 1);
4. **return** -1;
5. }
7. **if** (device\_create(cl, NULL, first, NULL, "chrnull") == NULL)
8. {
9. class\_destroy(cl);
10. unregister\_chrdev\_region(first, 1);
11. **return** -1;
12. }

Затем выполняется второй шаг. Устройство инициализируется операциями. Для этого используется функция cdev\_init (*linux/cdev.h*), в которую передается ссылка на структуру описывающее символьное устройство cdev (*linux/cdev.h*) и ссылка на структуру file\_operations (*linux/fs.h*). В последней реализован интерфейс устройства, а именно операции *open, release, write, read.* Код этих операций не рассматривается, так как прозрачен для понимания.

1. **static** **struct** file\_operations fops =
2. {
3. .owner = THIS\_MODULE,
4. .open = dev\_open,
5. .release = dev\_release,
6. .read = dev\_read,
7. .write = dev\_write
8. };

## Драйвер сетевой карты

В данном разделе рассматривается написание простейшего драйвера для сетевой карты TP-Link TF-3200. При разработке использована Ubuntu 11.04, ядро Linux - 2.6.38.8.

Данная карта подключается к компьютеру посредством PCI.

*PCI* (*Peripheral component interconnect*) — шина ввода-вывода для подключения периферийных устройств к материнской плате компьютера. (PCI, 2015)

Все PCI устройства содержат по крайней мере 256 байт адресного пространства. Первые 64 байт стандартизованы, а остальные зависят от устройства. Рисунок ниже показывает схему не зависящего от устройства конфигурационного пространства (Jonathan Corbet, 2005).

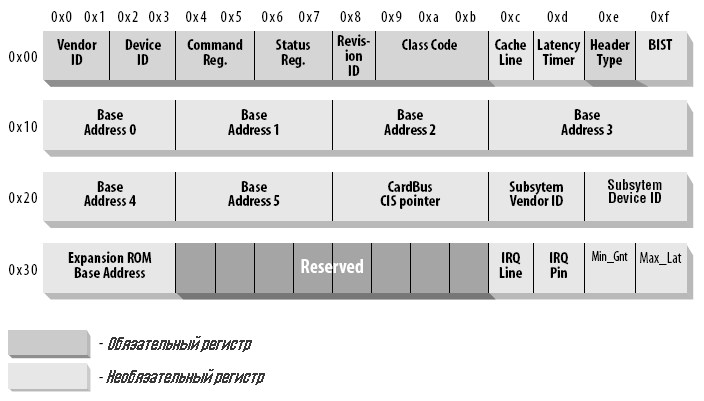


Рисунок 1. Стандартизированные конфигурационные регистры PCI (Jonathan Corbet, 2005)

Устройство идентифицируют три или пять регистров *PCI*: *vendorID, deviceID и class* являются теми тремя, которые используются всегда. Каждый производитель *PCI* присваивает собственные значения этим регистрам, предназначенным только для чтения, и драйвер может использовать их для поиска устройства. Кроме того, с целью дальнейшего различия похожих устройств поставщик иногда устанавливает поля *subsystem vendorID и subsystem deviceID*. (Jonathan Corbet, 2005)

* *vendorID*. Этот 16-ти разрядный регистр идентифицирует изготовителя оборудования. Например, каждое устройство *Intel* отмаркировано одним и тем же числом поставщика, *0x8086*. Существует глобальный реестр таких чисел, который ведёт *PCI Special Interest Group*, и производители должны обратиться туда, чтобы получить уникальный номер.
* *deviceID*. Это другой 16-ти разрядный регистр, выбранный производителем; для *ID* устройства не требуется никакой официальной регистрации. Этот *ID*, как правило, используется в паре с *ID* поставщика, создавая уникальный 32-х разрядный идентификатор аппаратного устройства. Мы используем слово сигнатура для обращения к *ID* поставщика и *ID* устройства в паре. Драйвер устройства обычно полагается на сигнатуру, чтобы определить своё устройство; вы можете найти, какое значение искать, в руководстве оборудования для целевого устройства.
* *class*. Каждое периферийное устройство относится к *class* (классу). Регистр *class* является 24-х разрядным значением, чьи старшие 8 бит идентифицируют "базовый класс" (или группу). Например, "*ethernet*" и "*token ring*" являются двумя классами, принадлежащими к группе "*network*", а классы "*serial*" и "*parallel*" относятся к группе "*communication*". Некоторые драйверы могут поддерживать несколько аналогичных устройств, каждое из них имеет свою сигнатуру, однако все они принадлежат к одному классу; эти драйверы могут рассчитывать на регистр *class* для определения своих периферийных устройства, как показано ниже.

Используя эти разные идентификаторы, драйвер *PCI* может сказать ядру, какие виды устройств он поддерживает. Чтобы определить список различных типов устройств *PCI*, которые поддерживает драйвер, используется структура struct pci\_device\_id (*linux/pci.h*).

Структура *pci\_device\_id* должна быть экспортирована в пользовательское пространство, чтобы позволить системам горячего подключения и загрузки модулей узнать, с какими устройствами работает модуль. Эту задачу решает макрос MODULE\_DEVICE\_TABLE. Пример:

1. MODULE\_DEVICE\_TABLE(pci, ids);

Этот оператор создаёт локальную переменную, называемую \_\_mod\_pci\_device\_table, которая указывает на список struct pci\_device\_id. Позже, в процессе сборки ядра, программа *depmod* просматривает все модули на символ \_\_mod\_pci\_device\_table. Если этот символ найден, она вынимает данные из модуля и добавляет их в файл */lib/modules/KERNEL\_VERSION/modules.pcimap*. После завершения работы *depmod*, все *PCI* устройства, поддерживаемые модулем в ядре, вместе с именами их модулей, перечисляются этом в файле. Когда ядро сообщает системе горячего подключения, что было найдено новое устройство *PCI*, система горячего подключения использует файл *modules*.*pcimap*, чтобы найти для загрузки правильный драйвер.

Основной структурой, которую должны создать все драйверы *PCI* для того, чтобы быть правильно зарегистрированными в ядре, является структура struct pci\_driver (*linux/pci.h*). Эта структура состоит из ряда функций обратного вызова и переменных, описывающих драйвер *PCI* для ядра *PCI*.

Чтобы создать правильную структуру struct pci\_driver должны быть проинициализированы только четыре поля:

1. **static** **struct** pci\_driver pci\_driver = {
2. .name = "dev\_name",
3. .id\_table = ids,
4. .probe = probe,
5. .remove = remove,
6. };

Чтобы зарегистрировать struct pci\_driver в ядре *PCI*, выполняется вызов pci\_register\_driver с указателем на struct pci\_driver. Это традиционно делается в коде инициализации модуля *PCI* драйвера:

1. **static** **int** \_\_init pci\_init(**void**)
2. {
3. **return** pci\_register\_driver(&pci\_driver);
4. }

После этого ядро передаст управление функции pcnet\_probe(), с указателем на *PCI* устройство, для которого совпали значения *Vendor ID* и *Device ID*.

Для карты *TF-3200* *vendorId* - 0x13F0, *deviceId* – 0x0200.

Значит структура pci\_device\_id будет выглядеть следующим образом:

1. **static** **struct** pci\_device\_id ids[] = {
2. { 0x13F0, 0x0200, PCI\_ANY\_ID, PCI\_ANY\_ID, 0, 0, 6 },
3. { 0, }
4. };

Экспорт в пользвательское пространство:

1. MODULE\_DEVICE\_TABLE(pci, ids);

Далее необходимо разрешить устройство *PCI*. В функции probe драйвера *PCI*, прежде чем драйвер сможет получить доступ к любому ресурсу устройства (область ввода/вывода или прерывание) данного *PCI* устройства, драйвер должен вызвать функцию pci\_enable\_device:

1. **int** pci\_enable\_device(**struct** pci\_dev \*dev);

Эта функция фактически разрешает устройство. Она будит устройство и в некоторых случаях также задаёт ему линию прерывания и области ввода/вывода. Это происходит, например, с устройствами *CardBus* (которые были сделаны полностью эквивалентными *PCI* на уровне драйвера). (Jonathan Corbet, 2005)

*Bus mastering* (Управление шиной) это функция шины *PCI* которая может использоваться любым *PCI* устройством (например контроллером диска, графическим контроллером или звуковой картой). *Bus mastering* позволяет устройству управлять шиной и осуществлять (инициировать) любые транзакции чтения / записи к другим устройствам на шине *PCI* или к системной памяти. Эти транзакции осуществляются независимо от главного процессора, поэтому они отнимают время только у шины, но не у процессора. *Bus mastering* транзакции нисколько не быстрее обычных транзакций, осуществляемых процессором. Преимущество *bus mastering* в том, что контроллер запрограммированный на выполнение какой-либо передачи данных или на выполнение последовательности команд больше не требует действий от процессора пока он (контроллер) не завершит свою задачу. Обычно для информирования процессора о том, что хозяин шины (*bus master*) выполнил свои действия используется механизм прерываний. Для включения функции *bus mastering* для *pci* устройства, используется функция: pci\_set\_master(). (pcnet-driver. descrition, 2010)

Далее происходит регистрация устройства. Поскольку для сетевых интерфейсов не существует эквивалента старшего и младшего номеров, сетевой драйвер не запрашивает такой номер. Вместо этого, драйвер помещает структуру данных для каждого вновь обнаруженного интерфейса в глобальный список сетевых устройств.

Каждый интерфейс описывается объектом struct net\_device (*linux/netdevice.h*).

Структура net\_device, как и многие другие структуры ядра, содержит kobject и является, таким образом, учитываемой по ссылке и экспортируемой через sysfs. Как и в случае с другими такими структурами, она должна быть создана динамически. Функцией ядра, предоставленной для выполнения такого выделения памяти, является alloc\_netdev, которая имеет следующий прототип:

1. **struct** net\_device \*alloc\_netdev(**int** sizeof\_priv,
2. **const** **char** \*name,
3. **void** (\*setup)(**struct** net\_device \*));

В функции pcnet\_dummy\_init\_netdev присходит заполнение структуры net\_device, в частности операций

1. ndev->netdev\_ops = &pcnet\_net\_device\_ops;

В данном примере эта структура заполняется заглушками.

Ниже приводится листинг кода и результат регистрации драйвера и устройства.

Листинг: *tplink\_driver\_ex3/tplink\_driver.c*

Версия ядра: *2.6.38.8*

1. #include <linux/types.h>
2. #include <linux/module.h>
3. #include <linux/kernel.h>
4. #include <linux/string.h>
5. #include <linux/timer.h>
6. #include <linux/errno.h>
7. #include <linux/ioport.h>
8. #include <linux/slab.h>
9. #include <linux/interrupt.h>
10. #include <linux/pci.h>
11. #include <linux/proc\_fs.h>
12. #include <linux/netdevice.h>
13. #include <linux/etherdevice.h>
14. #include <linux/skbuff.h>
15. #include <linux/init.h>
16. #include <linux/version.h>
17. #include <asm/uaccess.h>
18. #include <asm/processor.h>
19. #include <asm/bitops.h>
20. #include <asm/io.h>
21. #include <linux/delay.h>
22. #include <linux/spinlock.h>
24. #define DRV\_NAME "tplink\_driver"
25. #define DRV\_VERSION "dev"
26. #define DRV\_DESCRIPTION "IC Plus IP100A Fast Ethernet Adapter"
28. MODULE\_AUTHOR("Sergey Dedkov dsv.mail@yandex.ru");
29. MODULE\_DESCRIPTION(DRV\_DESCRIPTION);
30. MODULE\_VERSION(DRV\_VERSION);
31. MODULE\_LICENSE("GPL");
33. **static** **struct** pci\_device\_id ids[] = {
34. { 0x13F0, 0x0200, PCI\_ANY\_ID, PCI\_ANY\_ID, 0, 0, 6 },
35. { 0, }
36. };
37. MODULE\_DEVICE\_TABLE(pci, ids);
39. **struct** tplink\_private {
40. spinlock\_t lock;
41. **struct** pci\_dev \*pci\_dev;
42. **void** \_\_iomem \*base;
43. };
45. **static** **int** tplink\_dummy\_open(**struct** net\_device \*ndev)
46. {
47. **return** 0;
48. }
50. **static** **int** tplink\_dummy\_stop(**struct** net\_device \*ndev)
51. {
52. **return** 0;
53. }
55. **static** **int** tplink\_dummy\_start\_xmit(**struct** sk\_buff \*skb, **struct** net\_device \*ndev)
56. {
57. **return** 0;
58. }
60. **static** **const** **struct** net\_device\_ops tplink\_net\_device\_ops = {
61. .ndo\_open = tplink\_dummy\_open,
62. .ndo\_stop = tplink\_dummy\_stop,
63. .ndo\_start\_xmit = tplink\_dummy\_start\_xmit
64. };
66. **static** **int** \_\_devinit tplink\_dummy\_init\_netdev(**struct** pci\_dev \*pdev, unsigned **long** ioaddr)
67. {
68. **struct** net\_device \*ndev = pci\_get\_drvdata(pdev);
69. **struct** tplink\_private \*pp;
70. **int** irq;
71. unsigned **long** i;
73. **if** (!ndev)
74. **return** -ENODEV;
76. irq = pdev->irq;
77. pp = netdev\_priv(ndev);
78. ndev->base\_addr = ioaddr;
79. ndev->irq = irq;
80. pp->pci\_dev = pdev;
81. pp->base = (**void** \*)ioaddr;
82. spin\_lock\_init(&pp->lock);
84. /\* read first 6 bytes of PROM \*/
85. **for**(i = 0; i < 6; i++)
86. ndev->dev\_addr[i] = ioread8((**void** \*)ioaddr + i);
87. **if** (!is\_valid\_ether\_addr(ndev->dev\_addr))
88. random\_ether\_addr(ndev->dev\_addr);
90. /\* init net\_dev\_ops \*/
91. ndev->netdev\_ops = &tplink\_net\_device\_ops;
93. **if** (register\_netdev(ndev))
94. **return** -ENODEV;
95. netdev\_info(ndev, "%s %pM\n", DRV\_DESCRIPTION, ndev->dev\_addr);
97. **return** 0;
98. }
100. **static** **int** probe(**struct** pci\_dev \*pdev, **const** **struct** pci\_device\_id \*id)
101. {
102. printk("test probe");
104. **struct** tplink\_private \*pp;
105. **struct** net\_device \*ndev;
106. **void** \_\_iomem \*ioaddr;
107. #ifdef USE\_IO\_OPS
108. **int** bar = 0;
109. #else
110. **int** bar = 1;
111. #endif
113. #ifndef MODULE
114. pr\_info\_once("%s version %s\n", DRV\_DESCRIPTION, DRV\_VERSION);
115. #endif
117. **if** (pci\_enable\_device(pdev))
118. **return** -ENODEV;
119. /\* enables bus-mastering for device pdev \*/
120. pci\_set\_master(pdev);
122. ndev = alloc\_etherdev(**sizeof**(\*pp));
123. **if** (!ndev)
124. **goto** out;
125. /\* register ourself under /sys/class/net/ \*/
126. SET\_NETDEV\_DEV(ndev, &pdev->dev);
128. **if** (pci\_request\_regions(pdev, DRV\_NAME))
129. **goto** out\_netdev;
131. ioaddr = pci\_iomap(pdev, bar, 0x20);
132. **if** (!ioaddr)
133. **goto** out\_res;
134. pci\_set\_drvdata(pdev, ndev);
136. **if** (tplink\_dummy\_init\_netdev(pdev, (unsigned **long**)ioaddr))
137. **goto** out\_res\_unmap;
139. **return** 0;
141. out\_res\_unmap:
142. pci\_iounmap(pdev, ioaddr);
143. out\_res:
144. pci\_release\_regions(pdev);
145. out\_netdev:
146. free\_netdev(ndev);
147. out:
148. pci\_disable\_device(pdev);
149. **return** -ENODEV;
150. }
152. **static** **void** remove(**struct** pci\_dev \*pdev)
153. {
154. **struct** net\_device \*ndev = pci\_get\_drvdata(pdev);
155. **struct** tplink\_private \*pp;
156. pp = netdev\_priv(ndev);
157. unregister\_netdev(ndev);
158. pci\_iounmap(pdev, pp->base);
159. free\_netdev(ndev);
160. pci\_disable\_device(pdev);
161. pci\_release\_regions(pdev);
162. pci\_set\_drvdata(pdev, NULL);
163. }
165. **static** **struct** pci\_driver pci\_driver = {
166. .name = DRV\_NAME,
167. .id\_table = ids,
168. .probe = probe,
169. .remove = remove,
170. };
172. **static** **int** \_\_init pci\_skel\_init(**void**)
173. {
174. **return** pci\_register\_driver(&pci\_driver);
175. }
177. **static** **void** \_\_exit pci\_skel\_exit(**void**)
178. {
179. pci\_unregister\_driver(&pci\_driver);
180. }
182. module\_init(pci\_skel\_init);
183. module\_exit(pci\_skel\_exit);

Сборка осуществляется аналогично предыдущему примеру. Makefile используется такой же.

Результат работы:

Сборка модуля

dsv@dsv-pc:~/driver/projects/pci\_driver$ make

make -C /lib/modules/2.6.38-8-generic/build M=/home/dsv/driver/projects/pci\_driver modules

make[1]: Entering directory `/usr/src/linux-headers-2.6.38-8-generic'

Building with KERNELRELEASE = 2.6.38-8-generic

Building modules, stage 2.

Building with KERNELRELEASE = 2.6.38-8-generic

MODPOST 1 modules

make[1]: Leaving directory `/usr/src/linux-headers-2.6.38-8-generic'

Проверка сетевых адаптеров:

dsv@dsv-pc:~/driver/projects/pci\_driver$ ifconfig

eth0 Link encap:Ethernet HWaddr 60:e3:27:01:be:ae

inet6 addr: fe80::62e3:27ff:fe01:beae/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:22 errors:0 dropped:0 overruns:0 carrier:22

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:5587 (5.5 KB)

Interrupt:17 Base address:0xd400

eth1 Link encap:Ethernet HWaddr 00:13:d4:63:ff:3d

UP BROADCAST MULTICAST MTU:1500 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Interrupt:23 Base address:0x4000

Сейчас видно два устройства.

Удаление стандартного драйвера:

dsv@dsv-pc:~/driver/projects/pci\_driver$ sudo rmmod sundance

Проверка сетевых адаптеров:

dsv@dsv-pc:~/driver/projects/pci\_driver$ ifconfig

eth1 Link encap:Ethernet HWaddr 00:13:d4:63:ff:3d

UP BROADCAST MULTICAST MTU:1500 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Interrupt:23 Base address:0x4000

После удаления осталось только одно устройство

Установка драйвера:

dsv@dsv-pc:~/driver/projects/pci\_driver$ sudo insmod ./tplink\_driver.ko

Проверка сетевых адаптеров:

dsv@dsv-pc:~/driver/projects/pci\_driver$ ifconfig

eth1 Link encap:Ethernet HWaddr 00:13:d4:63:ff:3d

UP BROADCAST MULTICAST MTU:1500 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Interrupt:23 Base address:0x4000

eth2 Link encap:Ethernet HWaddr 00:00:02:00:00:00

inet6 addr: fe80::200:2ff:fe00:0/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Interrupt:17

Как видно после установки драйвера добавился сетевой интерфейс eth2.

## Выводы

В данной работе были рассмотрены основные принципы написания драйверов в ОС Linux. Описаны возможности написания загружаемых модулей, базовые механизмы работы с устройствами, вопросы разработки символьных драйверов и pci драйверов. Описанные в работе драйверы носят учебный характер. Так, например, в PCI реализуется только регистрация драйвера и устройства и опущены вопросы реализации операций сетевого устройства, что может являться предметом дальнейшего изучения.

# Платформа сборки дистрибутивов Linux Для встраиваемых устройств «yocto»

## Цель работы

* Изучить предоставляемые *Yocto* *project* шаблоны, инструменты и методы для создания специальных дистрибутивов Linux для встраиваемых систем на базе различных аппаратных архитектур.
* Описать проект, инструменты и возможности.
* Описать процесс сборки дистрибутива.
* Собрать и запустить дистрибутив Linux для x86.
* Рассмотреть процесс сборки с использованием web оболочки *Toaster*

## Описание проекта, инструментов и возможностей

*Yocto Project* – это совместный *Open Source*-проект разработки шаблонов, инструментов и методов для создания специальных дистрибутивов Linux для встраиваемых систем на базе различных аппаратных архитектур.

Будучи проектом совместного сотрудничества (иногда такие проекты называются "зонтичными"), *Yocto Project* охватывает различные составляющие процесса разработки. Эти составляющие именуются проектами в рамках общего проекта *Yocto Project* и включают в себя инструменты для сборки, метаданные инструкций по сборке (называемые рецептами), библиотеки, утилиты и графические интерфейсы.

Инструменты (Создание специальных дистрибутивов Linux для встраиваемых систем с помощью Yocto Project, 2013):

* *Poky* – это эталонная система сборки в рамках проекта *Yocto Project*. Она включает в себя *BitBake, OpenEmbedded-Core*, пакет поддержки платформы (*Board Support Package, BSP*), а также прочие пакеты и компоненты, объединенные в единую сборку. Название *Poky* также относится к эталонному дистрибутиву Linux, который создается этой системой сборки и может быть чрезвычайно минималистичным (*core-image-minimal*) или же представлять собой полноценную систему Linux с графической оболочкой (*core-image-sato*).
* Набор метаданных разделен на слои, каждый из которых обладает дополнительной функциональностью по отношению к нижележащим слоям. Базовый слой называется *OpenEmbedded*-Core (или *oe-core*) и содержит общие рецепты, классы и связанные с ними функции, необходимые для любой сборки. Эти сборки впоследствии можно настраивать под собственные нужды, добавляя новые слои поверх слоя *oe-core*.
* Пакет поддержки платформы содержит пакеты и драйверы, необходимые для создания Linux-дистрибутива для определенной платформы или архитектуры. Эти пакеты часто поддерживаются производителями компьютерного оборудования. BSP-пакеты являются интерфейсом между операционной системой Linux и аппаратной частью, на которой она запускается. Заметим, что можно также создавать BSP для виртуальных машин.
* *BitBake* – это система сборки. Она считывает рецепты (определенные наборы инструкций) и следует им – скачивает необходимые пакеты, компилирует их и создает результирующие загрузочные образы. *BitBake* совместно поддерживается проектами *Yocto Project* и *OpenEmbedded*.
* *Hob*. Для упрощения процесса разработки Linux для встраиваемых устройств в проекте *Yocto Project* было реализовано несколько различных методов, позволяющих работать с графической средой. Относительно новым дополнением к проекту является *Hob*. Это дополнение предоставляет в распоряжение разработчиков графический пользовательский интерфейс для *BitBake*, т. е. для процесса сборки. Оба этих компонента постоянно развиваются с учетом отзывов пользователей. На данный момент устарел.
* *Toaster* – веб оболочка для сборки дистрибутивов.
* *Embedded GLIBC (EGLIBC)* – это вариант библиотеки *GNU C Library (GLIBC)*, который был разработан для использования во встраиваемых системах. Особенностями *EGLIBC* являются меньший объем, настраиваемые компоненты и улучшенная поддержка кросс-компиляции и кросс-тестирования. *EGLIBC* входит в состав *Yocto Project*, но поддерживается своей собственной руководящей командой.
* Инструментарий для разработки приложений (*Application Development Toolkit, ADT*) позволяет разработчикам систем включать наборы SDK в создаваемые ими дистрибутивы с помощью инструментов *Yocto Project*. Впоследствии эти наборы *SDK* могут использовать сторонние разработчики для создания приложений для этих дистрибутивов. В состав *ADT* входят инструменты кросс-компиляции, утилиты для отладки и анализа производительности, а также сценарии эмуляции и поддержки *QEMU*. Кроме того, в *ADT* включен подключаемый модуль *Eclipse*, который могут использовать те, кто предпочитает работать в интегрированной среде разработки (*IDE*).
* *Autobuilder*: предназначен для автоматизации тестов и оценки качества продуктов, развиваемых на базе *Yocto Project*.
* *Cross-Prelink*: предназначен для предварительной компоновки в средах, использующих кросс-компиляцию, что позволяет повысить производительность программ.
* *Pseudo*: эмулирует доступ от имени пользователя root, что необходимо при создании конечного загрузочного образа.
* *Swabber*: определяет, не содержит ли сборка, выполненная с использованием кросс-компиляции, компоненты хостовой системы.
* *Build Appliance*: это виртуальная машина, в которой запущен *Hob*; позволяет получить представление о *Yocto Project* из первых рук тем, кто создает сборки на компьютерах под управлением операционной системы, отличной от Linux (Примечание: на сегодняшний день инструменты сборки *Yocto Project* можно использовать только в ОС Linux).

## Описание процесса сборки дистрибутива

Для работы с системой сборки *poky*, сначала нужно ее установить на компьютер.

Существует 2 способа получения *poky*:

1. Загрузка *tar* файла с последней протестированной версией *Yocto Project* со страницы загрузки проекта

$ wget \

http://downloads.yoctoproject.org/releases/yocto/yocto-1.2/poky-denzil-7.0.tar.bz2

$ tar xjf poky-denzil-7.0.tar.bz2

$ cd poky-denzil-7.0

1. Получение последней версии (или любой отдельной ветки) при помощи *git*, хотя главная ветка разработки может оказаться менее стабильной по сравнению с протестированной версией, содержащейся в *tar* файле.

$ git clone git://git.yoctoproject.org/poky.git

$ cd poky

Инициализация рабочего окружения:

Сначала нужно инсталлировать все необходимые пакеты из репозитория программного обеспечения (Yocto Project Mega-Manual, 2015).

$ sudo apt-get install sed wget subversion git-core coreutils \

unzip texi2html texinfo libsdl1.2-dev docbook-utils fop gawk \

python-pysqlite2 diffstat make gcc build-essential xsltproc \

g++ desktop-file-utils chrpath libgl1-mesa-dev libglu1-mesa-dev \

autoconf automake groff libtool xterm libxml-parser-perl

Задание значений переменных окружения командной оболочки с помощью входящего в состав поставки сценария:

$ cd poky

$ . ./oe-init-build-env

Создание образа:

После задания значений переменных окружения рабочей директорией станет */build*

### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:

core-image-minimal

core-image-sato

meta-toolchain

adt-installer

meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'

Далее задаются настройки сборки в файле *conf/local.conf*

По умолчанию введены следующие настройки:

MACHINE ??= "qemux86"

DISTRO ?= "poky"

PACKAGE\_CLASSES ?= "package\_rpm"

EXTRA\_IMAGE\_FEATURES = "debug-tweaks"

USER\_CLASSES ?= "buildstats image-mklibs"

PATCHRESOLVE = "noop"

BB\_DISKMON\_DIRS = "\

STOPTASKS,${TMPDIR},1G,100K \

STOPTASKS,${DL\_DIR},1G,100K \

STOPTASKS,${SSTATE\_DIR},1G,100K \

STOPTASKS,/tmp,100M,100K \

ABORT,${TMPDIR},100M,1K \

ABORT,${DL\_DIR},100M,1K \

ABORT,${SSTATE\_DIR},100M,1K \

ABORT,/tmp,10M,1K"

PACKAGECONFIG\_append\_pn-qemu-native = " sdl"

PACKAGECONFIG\_append\_pn-nativesdk-qemu = " sdl"

ASSUME\_PROVIDED += "libsdl-native"

CONF\_VERSION = "1"

Для оптимизации сборки можно задать следующие параметры, связанные с параллельной обработкой, для ускорения процесса сборки. На этом шаге зададим значения этих параметров, равными двукратному числу ядер процессора (например, 8 для 4-ядерного процессора). Для данной работы был использован *VPS (Virtual Private Server*) сервер *DigitalOcean.com*, с характеристиками – 12 CPU, 32 GB RAM, поскольку сборка занимает много времени.

BB\_NUMBER\_THREADS = "24"

PARALLEL\_MAKE = "-j 24"

Для старта сборки используется команда *bitbake*. Это сборка с *GUI SATO*:

bitbake core-image-sato

После этого запустится сборка дистрибутива.

Запуск виртуальной машины QEMU с собранным образом осуществляется следующей командой:

runqemu qemux86

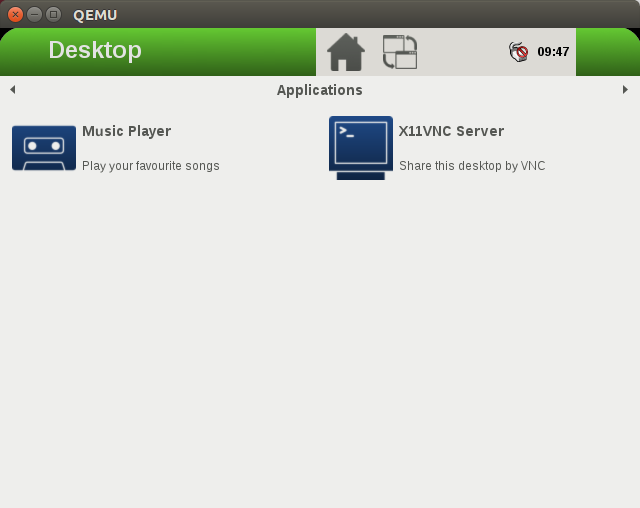


Рисунок 1. QEMU после запуска собранного образа core-image-sato

## Процесс сборки с использованием web оболочки Toaster

Для запуска веб-интерфейса *Toaster* необходимо сделать следующее (Toaster User Manual, 2015):

Установить пакетный менеджер *python - pip*:

sudo apt-get install python-pip

Далее установить необходимые зависимости:

pip install –r /home/user/poky/bitbake/toaster-requirements.txt

И запустить *Toaster*:

. /home/user/poky/bitbake/bin/toaster

Далее запустится web-интерфейс.

Создадим новый проект:

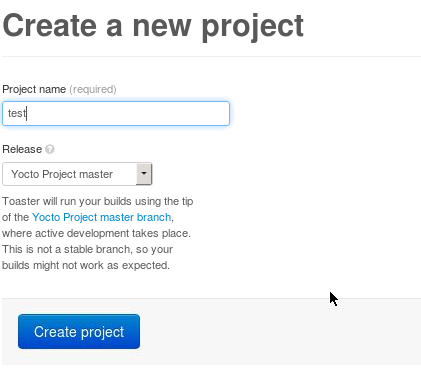


Рисунок 2. Toaster. Создание нового проекта

По умолчанию заданы следующие настройки:

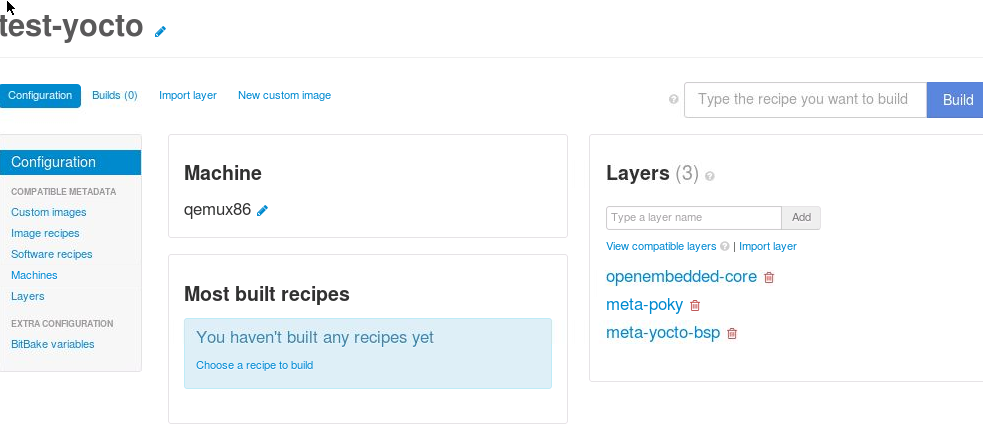


Рисунок 3. Toaster. Начальные натройки проекта

Для того чтобы добавить в дистрибутив нужный пакет нужно его найти в доступных слоях. Например, добавим *apache2*. Выбираем слой *meta-webservers*. Далее следует нажать *Add build*

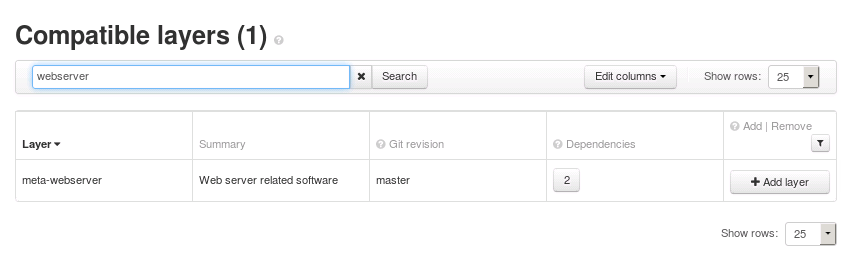


Рисунок 4. Toaster. Доступные слои.

После этого нужно согласиться установить зависимости:

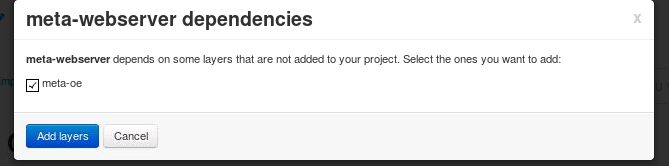


Рисунок 5. Toaster. Диалоговое окно. Предложение установить зависимости

Далее соберем *apache2* рецепт:

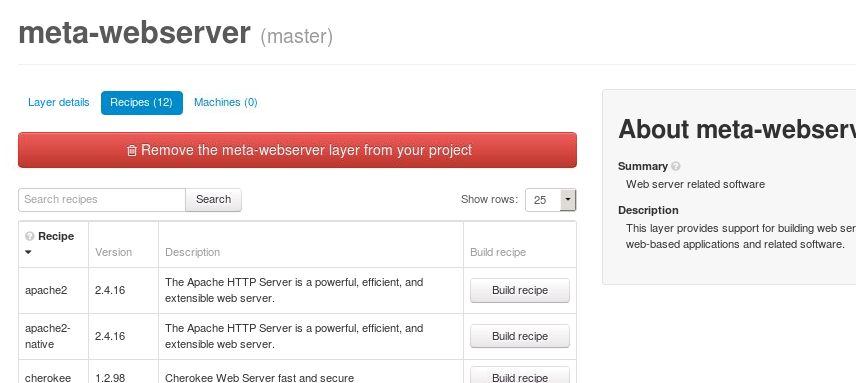


Рисунок 6. Toaster. Сборка рецепта слоя meta-webserver

Для того чтобы образ, который будет собран включал *apache2* нужно задать следующие параметры:

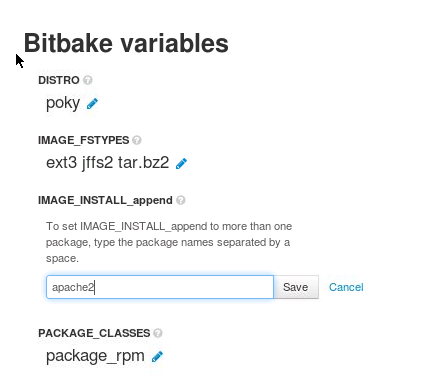


Рисунок 7. Toaster. Настройки BitBake

После этого можно собирать образ, который будет включать *apache2*.

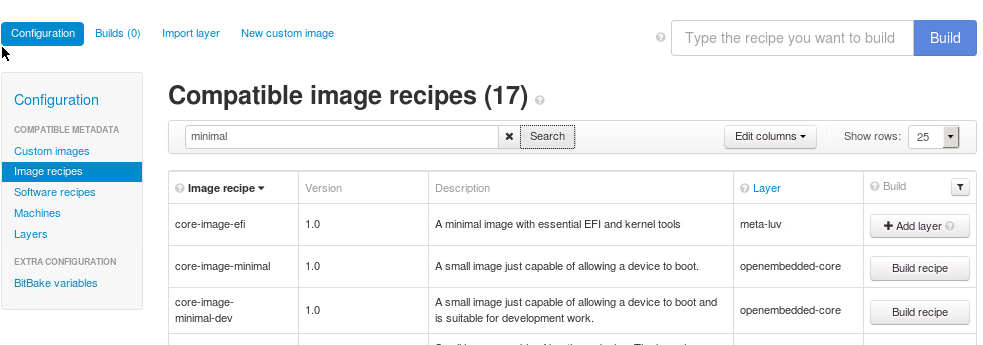


Рисунок 8. Toaster. Сборка образа

Собранный образ запускается, как и в предыдущем примере:

runqemu qemux86

## Вывод

*Yocto Project* – это совместный *Open Source*-проект разработки шаблонов, инструментов и методов для создания специальных дистрибутивов Linux для встраиваемых систем на базе различных аппаратных архитектур.

Будучи проектом совместного сотрудничества (иногда такие проекты называются "зонтичными"), *Yocto Project* охватывает различные составляющие процесса разработки.

Таким образом *yocto* позволяет гибко создавать дистрибутивы для различных архитектур. Множество документации *yocto* оправдывает высокий порог вхождения.

# Средства тестирования ОС

## Цель работы

* Описание видов тестирования ОС, назначения тестирования
* Описание средств тестирования ОС

## Описание видов тестирования ОС

Виды тестирования:

* Функциональное тестирование
* Тестирование обратной совместимости
* Обнаружение специфических видов ошибок
* Тестирование производительности

Рассматривая ОС с точки зрения тестирования, необходимо учитывать следующие особенности.

ОС как программное обеспечение, работающее непосредственно с аппаратурой, обладает:

* значительным внутренним параллелизмом;
* зависимостью от аппаратуры и её конфигураций;
* внутренней активностью.

ОС как основа компьютерной системы в целом и гарант безопасности приложений должна:

* быть устойчивой к различным нестандартным ситуациям, таким как нехватка оперативной памяти, памяти жёсткого диска и т. д.;
* быть устойчивой к атакам и вредоносным действиям со стороны недоверенных приложений, сетевых контрагентов, подключаемых внешних устройств и т. д.
* не допускать утечек ресурсов с целью обеспечения продолжительного функционирования системы без перезагрузки;
* минимизировать накладные расходы на реализацию своих функций.

ОС как среда для работы прикладных программ должна обладать такими свойствами как:

* соответствие стандартам на интерфейсы ОС;
* соответствие документации на интерфейсы ОС;
* совместимость с приложениями как на уровне бинарных интерфейсов, так и на уровне исходного кода.

ОС как платформа, на которой работает тестовая система, является источником дополнительных требований к принципам построения тестовой системы:

* при обнаружении ошибок, ведущих к аварийному прекращению работы ОС, информация, полученная тестом, должна сохраняться;
* тестовая система должна вносить минимальные искажения в поведение тестируемых компонентов ОС, в том числе в их временные характеристики. (Герлиц Е.А., 2015)

### Функциональное тестирование

Одной из ожидаемых характеристик ОС является корректная реализация своей функциональности. Это означает, что реальное поведение компонентов, доступных через публичные интерфейсы ОС, должно соответствовать декларируемым. Среди публичных интерфейсов ОС можно выделить интерфейсы системных библиотек и утилит, а также интерфейсы ядра ОС. Единичным элементом интерфейса библиотеки является функция, например, open(), sin() или fprintf(). Приведённые примеры демонстрируют различные виды реализации библиотечных функций:

* open() является примером функции-обёртки, которая сама по себе мало, что делает, за исключением трансляции запроса к компоненту более низкого уровня, в данном случае, open() обращается к соответствующему системному вызову ядра ОС;
* sin(), напротив, полностью реализуется в математической библиотеке и не требует обращения к другим компонентам;
* fprintf() представляет собой промежуточный вариант, в котором существенная часть функциональности реализуется в библиотеке (форматирование строки), но присутствует также и обращение к внешним компонентам (непосредственный вывод строки).

Функциональное тестирование делится на следующие типы:

* Тестирование обработки внутренних ошибок
* Специализированные тестовые системы

### Тестирование обратной совместимости

Современные операционные системы развиваются независимо от пользовательских приложений. Как следствие, возникает необходимость переноса приложений между различными версиями одной ОС. И так как приложений гораздо больше, чем ОС, то в большинстве случаев задача обеспечения переносимости перекладывается на плечи ОС в виде требования обеспечить обратную совместимость с приложениями. Это означает, что приложение, предназначенное для старой версии ОС, должно беспроблемно работать на новой версии ОС. Обратная совместимость обычно рассматривается на одном из двух уровней: на уровне бинарных файлов или на уровне исходных кодов. В первом случае требуется, чтобы уже скомпилированное приложение можно было установить и использовать на новой версии ОС. Во втором случае речь идёт о возможности перекомпиляции исходного кода приложения под новую версию ОС без каких-либо изменений в коде. Как правило, ОС обеспечивают обратную совместимость при условии, что приложения используют только специфицированные возможности ОС. Также существуют отраслевые и международные стандарты, регламентирующие интерфейс между приложениями и ОС, который позволяет обеспечить переносимость приложений между различными ОС. Примерами таких стандартов являются POSIX и LSB. POSIX описывает интерфейс на уровне исходных кодов для семейства ОС UNIX. LSB нацелен на обеспечение переносимости на бинарном уровне между дистрибутивами ОС Linux. Среди методов тестирования обратной совместимости можно выделить структурные и семантические.

Тестирование обратной совместимости делится на следующие типы:

* Структурное тестирование обратной совместимости
* Семантическое тестирование обратной совместимости

### Обнаружение специфических видов ошибок

Ряд ошибок в программах проявляются не сразу и/или опосредованным образом, поэтому традиционные подходы функционального тестирования не эффективны для их выявления. В этом разделе мы рассмотрим два класса таких ошибок в контексте тестирования ОС: утечки ресурсов и гонки по данным. Также в этом разделе будут рассмотрены ошибки, связанные с некорректной обработкой некорректных входных данных. В качестве интересного факта следует упомянуть, что анализ исправлений ошибок в стабильных версиях ядра ОС Linux за один календарный год показал, что гонки по данным и утечки ресурсов оказались наиболее распространёнными видами исправляемых ошибок: они встречаются в 17% и 16% случаев соответственно.

Этот тип тестирования делится на следующие виды:

* Обнаружение утечек ресурсов
* Обнаружение гонок по данным
* Kernel Strider и OC2000 Data Race Detector
* Race Hound

### Тестирование производительности ОС

Тестирование производительности в инженерии программного обеспечения определяется как тестирование, которое проводится с целью определения, как 98 быстро работает вычислительная система или её часть под определённой нагрузкой. Такое тестирование может также служить для проверки и подтверждения других атрибутов качества системы, таких как масштабируемость, надёжность и потребление ресурсов. Заметим, что в отличии от других видов тестирования, тестирование производительности обычно не ставит своей целью выявление нарушений заранее зафиксированных требований. Когда же речь идёт о производительности, такие требования зачастую отсутствуют. Это вызвано тем, что конкретные значения разнообразных показателей производительности для одной и той же программной системы сильно зависят от возможностей аппаратных платформ, на которых может выполняться эта программная система. Поэтому постановка задачи тестирования производительности сложных программных систем, к которым относятся и операционные системы, требует дополнительной конкретизации. Набор задач тестирования определяется как архитектурой программной системы и ее аппаратной платформы, так и целями тестирования.

Тестируются следующие показатели:

* время обработки прерывания;
* время создания потока управления;
* время переключения между потоками управления (переключение контекстов потоков);
* время переключения процессов (переключение контекстов процессов);
* время создания ресурсов (семафоров, очередей, мьютексов, таймеров и т. п.);
* пропускная способность системы обмена сообщениями между потоками управления и/или процессами.

## Средства тестирования ОС

### LAVA (Linaro Automated Validation Architecture)

LAVA это система непрерывной интеграции для развертывания операционных систем на физических и виртуальных аппаратных средств для выполнения тестов. Виды тестов: тестирование загрузки, тестирование начального загрузчика и тестирование на системном уровне, хотя может потребоваться дополнительное оборудование для некоторых системных тестов. Результаты отслеживаются в течение долгого времени, и данные могут быть экспортированы для дальнейшего анализа.

LAVA архитектурно состоит из двух основных компонентов: сервер и диспетчер. Сервер это Django веб приложение, которое содержит центральный планировщик, панель управления, мониторинг рабочих мест, аутентификацию и авторизацию, а также систему визуализации результата теста. Диспетчер - это подсистема, которая управляет связью с физическими устройствами тестирования.

Доступен для скачивания deb пакет. Может быть установлен на Ubuntu 14.04 следующей командой:

$ sudo apt-get install lava

LAVA предназначена для автоматизации тестирования на реальных и виртуальных системах. Linaro выполняет более 800 тестов через ~ 30 типов устройств каждую неделю.

Документация доступна на сайте validation.linaro.org.

Документация включает руководства по установке LAVA, написания тестов, администрирования LAVA инстанса и разработки кода для LAVA. (Linaro Automated Validation Architecture (LAVA), 2015)

### Opentest

OpenTest - это уже проект с открытым исходным кодом, основанный на другом проекте STAF (Software Testing Automation Framework), и являющийся частью Arago Project. Основу для исполнения тестов на целевой платформе составляет TEE (Test Execution Engine), представляющий собой сервис STAF. Сервис STAF - это обыкновенный процесс ОС, который находится в режиме ожидания и ждет команд от планировщика STAF. Планировщик посылает TEE сервису, который запущен на 8 встраиваемой системе, команду выполнить тест, после чего сервис выполняет тест и отсылает обратно сообщение о результатах в виде XML-файла. Cами тесты создаются через web интерфейс системы TestLink, после чего TestLink экспортирует тесты в XML файлы и делает запрос на исполнение теста планировщику STAF. Такой подход является универсальным, так как пользователь может реализовать что угодно в виде сервиса, однако это и создает определенные сложности. Во-первых, TEE должен быть запущен на целевой платформе и, следовательно, иметь все необходимое окружение STAF. Во-вторых, каждый отдельный тест представляет собой набор входов и выходов, являющихся строками, которые задаются через web интерфейс. Но часто выходной результат может описываться более сложным образом, чем просто строка, например, как результат выполнения некоторой команды bash. И проверка этого результата возложена на TEE, который работает на стороне целевой платформы, что подразумевает, как минимум, наличие некоторых основных утилит (sh, grep, awk, и т.д.), то есть предполагается наличие Linux или какой-то другой ОС с достаточно большими требованиями. (Opentest, 2013)

### Autotest

Autotest является основой для полностью автоматизированного тестирования. Он предназначен в первую очередь для тестирования ядра Linux, хотя может быть полезен и для многих других функций, таких как квалифицировать новое оборудование. Autotest – ПО с открытым исходным кодом с лицензией GPL. Используется и разработан рядом организаций, в том числе Google, IBM, Red Hat, и многими другими.

Autotest состоит из нескольких модулей, которые помогут сделать автономные тесты или установки полностью автоматизированной тестовой сетки.

Список модулей:

* Autotest клиент: Средство, которое выполняет тесты (реж клиента). Каждый тест - каталог (директория client/tests), представляется классом питона, который реализует минимальное количество методов. Клиент в основном предназначен для того, чтобы один разработчик локально мог выполнить тесты. Autotest клиент выполняет “клиентские управляемые файлы“, которые являются программами Python, и использовать API-интерфейс клиента.
* Autotest Сервер: Программа, которая копирует клиента к удаленным компьютерам и контролирует их выполнение. Сервер выполняет Autotest ''клиентских управляемых файлов'', которые также являются регулярные программы Python, но с более высокого уровня API, так как сервер Autotest может контролировать выполнение тестов на нескольких машинах.
* Autotest базы данных: Для тестовых сетей, нам нужен способ для хранения результатов тестирования. Для этого существует данный компонент. Эта БД используется Autotest планировщиком и фронтэндом для хранения и визуализации результатов испытаний.
* Autotest планировщик: Для тестовых сетей, нужна утилита, которая может вызвать и планировать выполнение заданий в тестовых машинах, Autotest планировщик является таковой.
* Autotest Веб-интерфейс: Веб-приложение для тестирования сетей, чей Бэкенд написана в Джанго (http://www.djangoproject.com/) и UI, написан в GWT (http://code.google.com/webtoolkit/~~HEAD=pobj), позволяет пользователям вызвать работу и визуализировать результаты испытаний
* Autotest интерфейс командной строки: Кроме того, пользователи могут также использовать Autotest CLI, написанный на Python

(Autotest: Fully automated tests under the linux platform, 2016)

### Jenkins

Jenkins – это инструмент непрерывной интеграции, написанный на Java. Запускается в контейнере сервлетов, таких как Apache Tomcat или GlassFish. Поддерживает инструментарий для работы с разными системами контроля версий, включая CVS, Subversion, Mercurial, Git и Clearcase, может собирать проекты Apache Ant и Apache Maven, а также исполнять shell-скрипты и команды Windows.

Сборка проектов может быть назначена на разные события, например, производиться по расписанию, используя механизм подобный cron, либо стартовать когда другая сборка уже собрана, либо при запросе определённого URL.

Благодаря возможности создания плагинов для Jenkins, его функциональность как инструмента для сборки проектов можно существенно расширить. Примером может быть Serenity, которая представляет собой написанную на Java библиотеку для составления метрик покрытия кода, его сложности, зависимостей в нём, соединённую с плагином Jenkins для отображения отчётов.

(Hudson, 2015)

### Ktest.pl

Ktest.pl - это скрипт на Perl в исходном дереве ядра, который читает простой конфигурационный файл для выполнения тестирования ядра. Может применятся для удаленных целевых машин. Если есть доступ к консоли удаленного сервера, есть возможность использовать ktest.pl для выполнения автоматических тестов. Пример запуска тестов:

\* git bisecting

\* config bisecting

\* randconfig

\* any .config boot/test

\* patch checking

Поддерживаемые виды тестирования – тестирование сборки, загрузки, модульное тестирование. Хотя ktest написан на Perl, не нужно ничего знать о Perl, чтобы использовать ktest. ktest.pl принимает один конфигурационный файл для его ввода и будет запускать тесты в соответствии с этим конфигурационного файла. ktest.pl был включен в основную ветку ядра Linux в директории tools/testing/ktest. Там лежит образец конфигурации, который охватывает все конфигурации, которые могут быть заданы для использования ktest.

(Ktest, 2011)

## Вывод

В данной работе рассмотрены основные виды и средства тестирования ОС. Как видно средств существует достаточно, для тестирования ОС. В освоении они не очень легки, но существует множество документации по каждому из средств. Все проекты активно развиваются и совершенствуются, позволяя проводить все более качественное тестирование ОС.

# Список литературы

A. М. Каннер, B. П.-р. (05 2012 г.). *Особенности доступа к системным функциям ядра ОС GNU/Linux*. Получено из Сайт компании ОКБ САПР: http://www.okbsapr.ru/kanner\_2012\_5.html

*Autotest: Fully automated tests under the linux platform*. (2016). Получено из github.com: https://github.com/autotest/autotest#readme

*Hudson*. (22 апрель 2015 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/Hudson

Jonathan Corbet, A. R.-H. (2005). *Linux Device Drivers, Third Edition.* Sebastopol: O'Reilly.

Kerrisk, M. (23 07 2014 г.). *STAT(2). Linux Programmer's Manual*. Получено из Michael Kerrisk man7.org: http://man7.org/linux/man-pages/man2/stat.2.html

Kerrisk, M. (23 07 2015 г.). *RENAME(2). Linux Programmer's Manual*. Получено из Michael Kerrisk man7.org: http://man7.org/linux/man-pages/man2/rename.2.html

Kerrisk, M. (29 03 2015 г.). *SYSCALL(2). Linux Programmer's Manual*. Получено из Michael Kerrisk man7.org: http://man7.org/linux/man-pages/man2/syscall.2.html

*Ktest*. (28 октябрь 2011 г.). Получено из elinux.org: http://elinux.org/Ktest

*Linaro Automated Validation Architecture (LAVA)*. (16 сентябрь 2015 г.). Получено из wiki.linaro.org: https://wiki.linaro.org/LAVA

*Opentest*. (6 сентябрь 2013 г.). Получено из http://arago-project.org/: http://arago-project.org/wiki/index.php/Opentest

*PCI*. (16 ноябрь 2015 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/PCI

*pcnet-driver. descrition*. (15 сентябрь 2010 г.). Получено из github.com: https://github.com/pasis/pcnet-driver/blob/master/doc/ru/2-pci-register.txt

*Toaster User Manual*. (апрель 2015 г.). Получено из www.yoctoproject.org: http://www.yoctoproject.org/docs/2.0.1/toaster-manual/toaster-manual.html

*Yocto Project Mega-Manual*. (апрель 2015 г.). Получено из www.yoctoproject.org: http://www.yoctoproject.org/docs/2.0.1/mega-manual/mega-manual.html

Герлиц Е.А., К. В. (2015). *Тестирование операционных систем.*

*Драйвер*. (17 02 2016 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/Драйвер

*Драйверы устройств в Linux. Часть 2*. (июнь 2012 г.). Получено из Linux по-русски. Виртуальная энциклопедия: http://rus-linux.net/MyLDP/BOOKS/drivers/linux-device-drivers-02.html

*Драйверы устройств в Linux. Часть 3*. (июнь 2012 г.). Получено из Linux по-русски. Виртуальная энциклопедия: http://rus-linux.net/MyLDP/BOOKS/drivers/linux-device-drivers-03.html

*Драйверы устройств в Linux. Часть 4*. (июнь 2012 г.). Получено из Linux по-русски. Виртуальная энциклопедия: http://rus-linux.net/MyLDP/BOOKS/drivers/linux-device-drivers-04.html

*Загружаемый модуль ядра*. (11 02 2016 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/Загружаемый\_модуль\_ядра

*Кольца защиты*. (08 03 2016 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/Кольца\_защиты

Лав, Р. (2014). *Linux Системное программирование.* СПб: Питер.

Мешков, В. (24 12 2003 г.). *Перехват системных вызовов в OS Linux (linux kernel module)*. Получено из OpenNET: https://www.opennet.ru/base/dev/intercept\_lnx.txt.html

*Системный вызов*. (28 03 2016 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/Системный\_вызов

*Создание специальных дистрибутивов Linux для встраиваемых систем с помощью Yocto Project*. (16 январь 2013 г.). Получено из http://www.ibm.com/: http://www.ibm.com/developerworks/ru/library/l-yocto-linux/

*Ядро операционной системы*. (08 03 2016 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/Ядро\_операционной\_системы