Санкт-Петербургский политехнический университет

Институт Компьютерных Наук и Технологий

Кафедра компьютерных систем и программных технологий

Лабораторная работа по дисциплине «Проектирование ОС и компонентов» на тему:

Системные вызовы

Выполнил: студент группы № 53501/3

Дедков С.В.

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

Санкт-Петербург

2015

# Оглавление

[Оглавление 2](#_Toc440815044)

[Цель работы 3](#_Toc440815045)

[Принцип работы системных вызовов 4](#_Toc440815046)

[Назначение и функциональность вызовов sys\_stat, sys\_rename, sys\_mkdir, sys\_rmdir, sys\_fstat 7](#_Toc440815047)

[Введение 7](#_Toc440815048)

[Системные вызовы sys\_stat, sys\_fstat 8](#_Toc440815049)

[Сигнатура stat 8](#_Toc440815050)

[Описание stat 8](#_Toc440815051)

[Пример stat 8](#_Toc440815052)

[Просмотр вызовов strace stat 10](#_Toc440815053)

[Пример fstat 10](#_Toc440815054)

[Просмотр вызовов strace fstat 11](#_Toc440815055)

[Системные вызовы sys\_mkdir, sys\_rmdir 12](#_Toc440815056)

[Сигнатура mkdir 12](#_Toc440815057)

[Описание mkdir 12](#_Toc440815058)

[Пример mkdir 12](#_Toc440815059)

[Просмотр вызовов strace mkdir 12](#_Toc440815060)

[Сигнатура rmdir 13](#_Toc440815061)

[Описание rmdir 13](#_Toc440815062)

[Пример rmdir 13](#_Toc440815063)

[Просмотр вызовов strace rmdir 14](#_Toc440815064)

[Системный вызов sys\_rename 15](#_Toc440815065)

[Сигнатура rename 15](#_Toc440815066)

[Описание rename 15](#_Toc440815067)

[Просмотр вызовов strace rename 16](#_Toc440815068)

[Перехват системных вызовов 17](#_Toc440815069)

[Введение 17](#_Toc440815070)

[Перехват системного вызова mkdir 18](#_Toc440815071)

[Вывод 23](#_Toc440815072)

[Список источников 24](#_Toc440815073)

# Цель работы

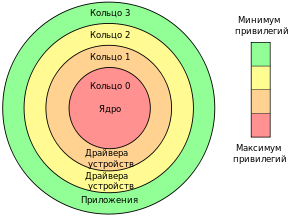
* Изучить принцип работы системных вызовов в целом.
* Рассмотреть назначение и функциональность вызовов
  + sys\_stat
  + sys\_rename
  + sys\_mkdir
  + sys\_rmdir
  + sys\_fstat
* Осуществить перехват системного вызова, написать свою реакцию.

# 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*, *предусматривает* использование защищенного режима с несколькими уровнями привилегий.

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



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:

* 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.

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:

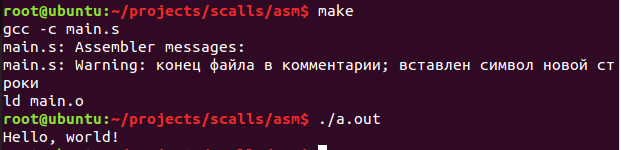


Рисунок 1. Сборка и выполнение примера системного прерывания на языке ассемблер

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$

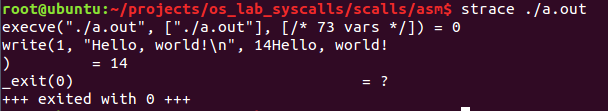
Здесь можно увидеть вызов прерывания 0x80 и установку параметров, как это было описано выше:

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.



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:

* 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.

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:

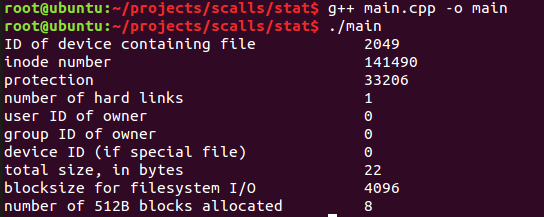


Рисунок 2. Сборка и выполнение примера программы, syscall - stat

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.



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*:

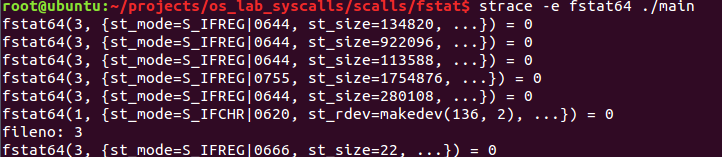
1. #include <sys/stat.h>
2. #include <stdlib.h>
3. #include <iostream>
4. #include <fstream>
6. **int** main() {
7. std::string fpath = "test.txt";
8. **struct** stat finfo;
9. **FILE** \* pf = fopen( fpath.c\_str(), "r" );
11. **if**( pf == NULL ) {
12. std::cout << "Cannot open file.\n";
13. fclose( 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 );



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:

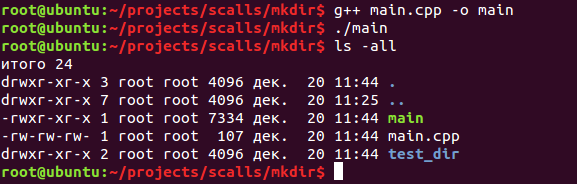


Рисунок 3. Сборка и выполнение примера программы, syscall - mkdir

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.



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.



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:

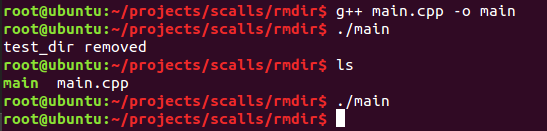


Рисунок 4. Сборка и выполнение примера программы, syscall - rmdir

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

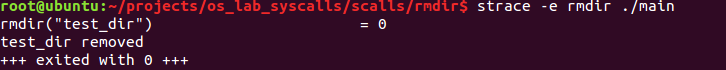
root@ubuntu:~/projects/scalls/rmdir$ ./main

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.



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.



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.

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:

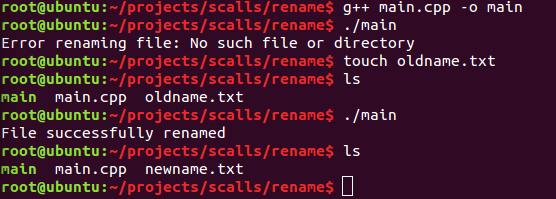


Рисунок 5. Сборка и выполнение примера программы, syscall - rename

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

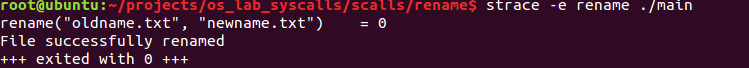
File successfully renamed

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

main main.cpp newname.txt

### Просмотр вызовов strace rename

Просмотр вызовов если файл есть:



Лог:

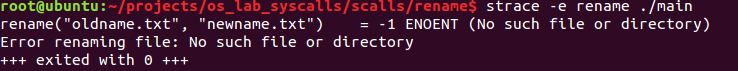
root@ubuntu:~/projects/os\_lab\_syscalls/scalls/rename$ strace -e rename ./main

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

File successfully renamed

+++ exited with 0 +++

Просмотр вызовов если файла нет:



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. }
25. **int** vfs\_mkdir(**struct** inode \*dir, **struct** dentry \*dentry, umode\_t mode)
26. {
27. **int** error = may\_create(dir, dentry);
28. unsigned max\_links = dir->i\_sb->s\_max\_links;
30. **if** (error)
31. **return** error;
33. **if** (!dir->i\_op->mkdir)
34. **return** -EPERM;
36. mode &= (S\_IRWXUGO|S\_ISVTX);
37. error = security\_inode\_mkdir(dir, dentry, mode);
38. **if** (error)
39. **return** error;
41. **if** (max\_links && dir->i\_nlink >= max\_links)
42. **return** -EMLINK;
44. error = dir->i\_op->mkdir(dir, dentry, mode);
45. **if** (!error)
46. fsnotify\_mkdir(dir, dentry);
47. **return** error;
48. }

## Исходный код вызова rename

# 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.

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

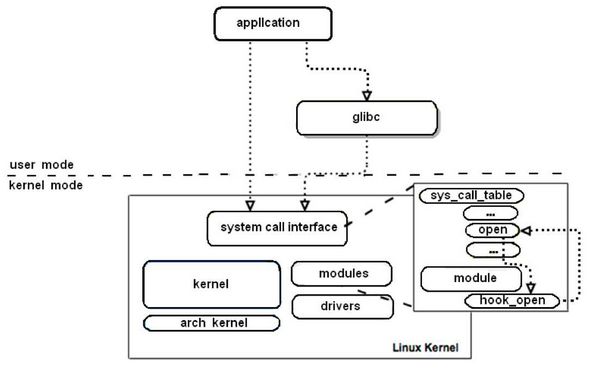


Рисунок 6. Перехват системного вызова с помощью модуля ядра Linux

На рисунке приведена графическая схема перехвата системного вызова, подменой

## Перехват системного вызова mkdir

Файл модуля *intercept\_mkdir.c.*

Листинг – *scalls/lsm/intercept\_mkdir.c* :

1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/syscalls.h>
4. #include <linux/delay.h>
5. #include <asm/paravirt.h>
7. unsigned **long** \*\*sys\_call\_table;
8. unsigned **long** original\_cr0;
10. asmlinkage **int** (\* ref\_sys\_mkdir) (**struct** inode \*dir, **struct** dentry \*dentry, **int** mode);
11. asmlinkage **int** new\_sys\_mkdir (**struct** inode \*dir, **struct** dentry \*dentry, **int** mode) {
12. printk ("mkdir hijacked!\n");
13. **return** ref\_sys\_mkdir(dir, dentry, mode);
14. }
16. // find sys call table
17. **static** unsigned **long** \*\*aquire\_sys\_call\_table(**void**)
18. {
19. unsigned **long** **int** offset = PAGE\_OFFSET;
20. unsigned **long** \*\*sct;
22. **while** (offset < ULLONG\_MAX) {
23. sct = (unsigned **long** \*\*)offset;
25. **if** (sct[\_\_NR\_close] == (unsigned **long** \*) sys\_close)
26. **return** sct;
28. offset += **sizeof**(**void** \*);
29. }
31. **return** NULL;
32. }
34. // change sys call
35. **static** **int** \_\_init interceptor\_start(**void**)
36. {
37. **if**(!(sys\_call\_table = aquire\_sys\_call\_table()))
38. **return** -1;
40. original\_cr0 = read\_cr0();
42. write\_cr0(original\_cr0 & ~0x00010000);
43. ref\_sys\_mkdir = (**void** \*)sys\_call\_table[\_\_NR\_mkdir];
44. sys\_call\_table[\_\_NR\_mkdir] = (unsigned **long** \*)new\_sys\_mkdir;
45. write\_cr0(original\_cr0);
47. **return** 0;
48. }
50. // revert changes
51. **static** **void** \_\_exit interceptor\_end(**void**)
52. {
53. **if**(!sys\_call\_table) {
54. **return**;
55. }
57. write\_cr0(original\_cr0 & ~0x00010000);
58. sys\_call\_table[\_\_NR\_mkdir] = (unsigned **long** \*)ref\_sys\_mkdir;
59. write\_cr0(original\_cr0);
61. msleep(2000);
62. }
64. module\_init(interceptor\_start);
65. module\_exit(interceptor\_end);
67. MODULE\_LICENSE("GPL");

Makefile.

Листинг – *scalls/lsm/Makefile*:

obj-m += intercept\_mkdir.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

Сборка модуля:

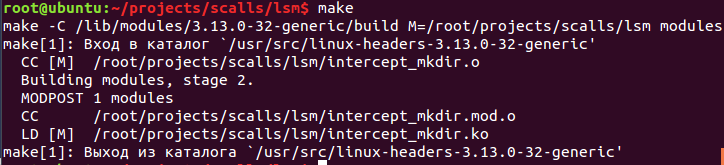


Рисунок 7. Сборка модуля с перехватом системного вызова mkdir

Лог:

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

make -C /lib/modules/3.13.0-32-generic/build M=/root/projects/scalls/lsm modules

make[1]: Вход в каталог `/usr/src/linux-headers-3.13.0-32-generic'

CC [M] /root/projects/scalls/lsm/intercept\_mkdir.o

Building modules, stage 2.

MODPOST 1 modules

CC /root/projects/scalls/lsm/intercept\_mkdir.mod.o

LD [M] /root/projects/scalls/lsm/intercept\_mkdir.ko

make[1]: Выход из каталога `/usr/src/linux-headers-3.13.0-32-generic'

Установка модуля:



Рисунок 8. Установка модуля

Лог:

root@ubuntu:~/projects/scalls/lsm$ insmod intercept\_mkdir.ko

Проверка работы:

Для проверки создадим папку и проверим вывод при помощи утилиты *dmesg*.



Рисунок 9. Создание папки после установки модуля с перехватом

Лог:

root@ubuntu:~/projects/scalls/lsm$ mkdir newdir

Частичный вывод результата *dmesg*:

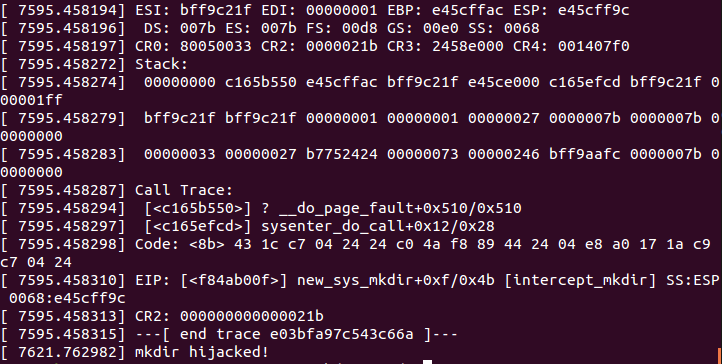


Рисунок 10. Вывод dmesg после создания папки

Лог:

[ 7595.458279] bff9c21f bff9c21f 00000001 00000001 00000027 0000007b 0000007b 00000000

[ 7595.458283] 00000033 00000027 b7752424 00000073 00000246 bff9aafc 0000007b 00000000

[ 7595.458287] Call Trace:

[ 7595.458294] [<c165b550>] ? \_\_do\_page\_fault+0x510/0x510

[ 7595.458297] [<c165efcd>] sysenter\_do\_call+0x12/0x28

[ 7595.458298] Code: <8b> 43 1c c7 04 24 24 c0 4a f8 89 44 24 04 e8 a0 17 1a c9 c7 04 24

[ 7595.458310] EIP: [<f84ab00f>] new\_sys\_mkdir+0xf/0x4b [intercept\_mkdir] SS:ESP 0068:e45cff9c

[ 7595.458313] CR2: 000000000000021b

[ 7595.458315] ---[ end trace e03bfa97c543c66a ]---

[ 7621.762982] mkdir hijacked!

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

Для восстановления стандартного системного вызова воспользуемся командой *rmmod*:

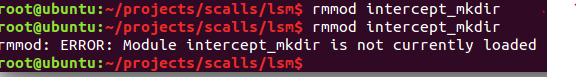


Рисунок 11. Удаления модуля

Лог:

root@ubuntu:~/projects/scalls/lsm$ rmmod intercept\_mkdir

root@ubuntu:~/projects/scalls/lsm$ rmmod intercept\_mkdir

rmmod: ERROR: Module intercept\_mkdir is not currently loaded

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. printk ("mkdir hijacked!\n");
4. **return** ref\_sys\_mkdir(dir, dentry, mode);
5. }

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.

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(interceptor\_start);
2. module\_exit(interceptor\_end);

Стоит отметить, что есть несколько вариантов перехвата системных вызовов:

* 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

# Список источников

1. Особенности доступа к системным функциям ядра ОС GNU/Linux - <http://www.okbsapr.ru/kanner_2012_5.html>
2. Intercepting System Calls - http://books.gigatux.nl/mirror/networksecuritytools/0596007949/networkst-CHP-7-SECT-2.html
3. linux 3.14 sys\_call\_table interception module <https://bbs.archlinux.org/viewtopic.php?id=139406>
4. Перехват системных вызовов в операционной системе Linux http://samag.ru/archive/article/128
5. Перехват системных вызовов в OS Linux (linux kernel module) - <https://www.opennet.ru/base/dev/intercept_lnx.txt.html>
6. Встраивание в ядро Linux: перехват системных вызовов - <http://habrahabr.ru/company/securitycode/blog/245539/>
7. Лав Р. Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014. — 448 с.