**How does the open system call work**

**Introduction**

This is the fifth part of the chapter that describes [system calls](https://en.wikipedia.org/wiki/System_call) mechanism in the Linux kernel. Previous parts of this chapter described this mechanism in general. Now I will try to describe implementation of different system calls in the Linux kernel. Previous parts from this chapter and parts from other chapters of the books describe mostly deep parts of the Linux kernel that are faintly visible or fully invisible from the userspace. But the Linux kernel code is not only about itself. The vast of the Linux kernel code provides ability to our code. Due to the linux kernel our programs can read/write from/to files and don’t know anything about sectors, tracks and other parts of a disk structures, we can send data over network and don’t build encapsulated network packets by hand and etc.

I don’t know how about you, but it is interesting to me not only how an operating system works, but how do my software interacts with it. As you may know, our programs interacts with the kernel through the special mechanism which is called [system call](https://en.wikipedia.org/wiki/System_call). So, I’ve decided to write series of parts which will describe implementation and behavior of system calls which we are using every day like read, write, open, close, dup and etc.

I have decided to start from the description of the [open](http://man7.org/linux/man-pages/man2/open.2.html) system call. if you have written at least one C program, you should know that before we are able to read/write or execute other manipulations with a file we need to open it with the open function:

1. #include <fcntl.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <unistd.h>
5. #include <sys/stat.h>
6. #include <sys/types.h>
7. int main(int argc, char \*argv) {
8. int fd = open("test", O\_RDONLY);
9. if fd < 0 {
10. perror("Opening of the file is failed\n");
11. }
12. else {
13. printf("file sucessfully opened\n");
14. }
15. close(fd);
16. return 0;
17. }

In this case, the open is the function from standard library, but not system call. The standard library will call related system call for us. The open call will return a [file descriptor](https://en.wikipedia.org/wiki/File_descriptor) which is just a unique number within our process which is associated with the opened file. Now as we opened a file and got file descriptor as result of open call, we may start to interact with this file. We can write into, read from it and etc. List of opened file by a process is available via [proc](https://en.wikipedia.org/wiki/Procfs) filesystem:

1. $ sudo ls /proc/1/fd/
2. 0 10 12 14 16 2 21 23 25 27 29 30 32 34 36 38 4 41 43 45 47 49 50 53 55 58 6 61 63 67 8
3. 1 11 13 15 19 20 22 24 26 28 3 31 33 35 37 39 40 42 44 46 48 5 51 54 57 59 60 62 65 7 9

I am not going to describe more details about the open routine from the userspace view in this post, but mostly from the kernel side. if you are not very familiar with, you can get more info in the [man page](http://man7.org/linux/man-pages/man2/open.2.html).

So let’s start.

**Definition of the open system call**

If you have read the [fourth part](https://github.com/0xAX/linux-insides/blob/master/SysCall/syscall-4.md) of the [linux-insides](https://0xax.gitbooks.io/linux-insides/content/index.html) book, you should know that system calls are defined with the help of SYSCALL\_DEFINE macro. So, the open system call is not exception.

Definition of the open system call is located in the [fs/open.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) source code file and looks pretty small for the first view:

1. SYSCALL\_DEFINE3(open, const char \_\_user \*, filename, int, flags, umode\_t, mode)
2. {
3. if (force\_o\_largefile())
4. flags |= O\_LARGEFILE;
5. return do\_sys\_open(AT\_FDCWD, filename, flags, mode);
6. }

As you may guess, the do\_sys\_open function from the [same](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) source code file does the main job. But before this function will be called, let’s consider the if clause from which the implementation of the open system call starts:

1. if (force\_o\_largefile())
2. flags |= O\_LARGEFILE;

Here we apply the O\_LARGEFILE flag to the flags which were passed to open system call in a case when the force\_o\_largefile() will return true.  
What is O\_LARGEFILE? We may read this in the [man page](http://man7.org/linux/man-pages/man2/open.2.html) for the open(2) system call:

O\_LARGEFILE

(LFS) Allow files whose sizes cannot be represented in an off\_t (but can be represented in an off64\_t) to be opened.

As we may read in the [GNU C Library Reference Manual](https://www.gnu.org/software/libc/manual/html_mono/libc.html#File-Position-Primitive):

off\_t

This is a signed integer type used to represent file sizes.  
In the GNU C Library, this type is no narrower than int.  
If the source is compiled with \_FILE\_OFFSET\_BITS == 64 this  
type is transparently replaced by off64\_t.

and

off64\_t

This type is used similar to off\_t. The difference is that  
even on 32 bit machines, where the off\_t type would have 32 bits,  
off64\_t has 64 bits and so is able to address files up to 2^63 bytes  
in length. When compiling with \_FILE\_OFFSET\_BITS == 64 this type  
is available under the name off\_t.

So it is not hard to guess that the off\_t, off64\_t and O\_LARGEFILE are about a file size. In the case of the Linux kernel, the O\_LARGEFILE is used to disallow opening large files on 32bit systems if the caller didn’t specify O\_LARGEFILE flag during opening of a file. On 64bit systems we force on this flag in open system call. And the force\_o\_largefile macro from the [include/linux/fcntl.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/fcntl.h#L7) linux kernel header file confirms this:

1. #ifndef force\_o\_largefile
2. #define force\_o\_largefile() (BITS\_PER\_LONG != 32)
3. #endif

This macro may be architecture-specific as for example for [IA-64](https://en.wikipedia.org/wiki/IA-64) architecture, but in our case the [x86\_64](https://en.wikipedia.org/wiki/X86-64) does not provide definition of the force\_o\_largefile and it will be used from [include/linux/fcntl.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/fcntl.h#L7).

So, as we may see the force\_o\_largefile is just a macro which expands to the true value in our case of [x86\_64](https://en.wikipedia.org/wiki/X86-64) architecture. As we are considering 64-bit architecture, the force\_o\_largefile will be expanded to true and the O\_LARGEFILE flag will be added to the set of flags which were passed to the open system call.

Now as we considered meaning of the O\_LARGEFILE flag and force\_o\_largefile macro, we can proceed to the consideration of the implementation of the do\_sys\_open function. As I wrote above, this function is defined in the [same](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) source code file and looks:

1. long do\_sys\_open(int dfd, const char \_\_user \*filename, int flags, umode\_t mode)
2. {
3. struct open\_flags op;
4. int fd = build\_open\_flags(flags, mode, &op);
5. struct filename \*tmp;
6. if (fd)
7. return fd;
8. tmp = getname(filename);
9. if (IS\_ERR(tmp))
10. return PTR\_ERR(tmp);
11. fd = get\_unused\_fd\_flags(flags);
12. if (fd >= 0) {
13. struct file \*f = do\_filp\_open(dfd, tmp, &op);
14. if (IS\_ERR(f)) {
15. put\_unused\_fd(fd);
16. fd = PTR\_ERR(f);
17. } else {
18. fsnotify\_open(f);
19. fd\_install(fd, f);
20. }
21. }
22. putname(tmp);
23. return fd;
24. }

Let’s try to understand how the do\_sys\_open works step by step.

**open(2) flags**

As you know the open system call takes set of flags as second argument that control opening a file and mode as third argument that specifies permission the permissions of a file if it is created. The do\_sys\_open function starts from the call of the build\_open\_flags function which does some checks that set of the given flags is valid and handles different conditions of flags and mode.

Let’s look at the implementation of the build\_open\_flags. This function is defined in the [same](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) kernel file and takes three arguments:

* flags - flags that control opening of a file;
* mode - permissions for newly created file;

The last argument - op is represented with the open\_flags structure:

1. struct open\_flags {
2. int open\_flag;
3. umode\_t mode;
4. int acc\_mode;
5. int intent;
6. int lookup\_flags;
7. };

which is defined in the [fs/internal.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/internal.h#L99) header file and as we may see it holds information about flags and access mode for internal kernel purposes. As you already may guess the main goal of the build\_open\_flags function is to fill an instance of this structure.

Implementation of the build\_open\_flags function starts from the definition of local variables and one of them is:

1. int acc\_mode = ACC\_MODE(flags);

This local variable represents access mode and its initial value will be equal to the value of expanded ACC\_MODE macro. This macro is defined in the [include/linux/fs.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/fs.h) and looks pretty interesting:

1. #define ACC\_MODE(x) ("\004\002\006\006"[(x)&O\_ACCMODE])
2. #define O\_ACCMODE 00000003

The "\004\002\006\006" is an array of four chars:

1. "\004\002\006\006" == {'\004', '\002', '\006', '\006'}

So, the ACC\_MODE macro just expands to the accession to this array by [(x) & O\_ACCMODE] index. As we just saw, the O\_ACCMODE is 00000003. By applying x & O\_ACCMODE we will take the two least significant bits which are represents read, write or read/write access modes:

1. #define O\_RDONLY 00000000
2. #define O\_WRONLY 00000001
3. #define O\_RDWR 00000002

After getting value from the array by the calculated index, the ACC\_MODE will be expanded to access mode mask of a file which will hold MAY\_WRITE, MAY\_READ and other information.

We may see following condition after we have calculated initial access mode:

1. if (flags & (O\_CREAT | \_\_O\_TMPFILE))
2. op->mode = (mode & S\_IALLUGO) | S\_IFREG;
3. else
4. op->mode = 0;

Here we reset permissions in open\_flags instance if a opened file wasn’t temporary and wasn’t open for creation. This is because:

if neither O\_CREAT nor O\_TMPFILE is specified, then mode is ignored.

In other case if O\_CREAT or O\_TMPFILE were passed we canonicalize it to a regular file because a directory should be created with the [opendir](http://man7.org/linux/man-pages/man3/opendir.3.html) system call.

At the next step we check that a file is not tried to be opened via [fanotify](http://man7.org/linux/man-pages/man7/fanotify.7.html) and without the O\_CLOEXEC flag:

1. flags &= ~FMODE\_NONOTIFY & ~O\_CLOEXEC;

We do this to not leak a [file descriptor](https://en.wikipedia.org/wiki/File_descriptor). By default, the new file descriptor is set to remain open across an execve system call, but the open system call supports O\_CLOEXEC flag that can be used to change this default behaviour. So we do this to prevent leaking of a file descriptor when one thread opens a file to set O\_CLOEXEC flag and in the same time the second process does a [fork](https://en.wikipedia.org/wiki/Fork_/(system_call/)) + [execve](https://en.wikipedia.org/wiki/Exec_/(system_call/)) and as you may remember that child will have copies of the parent’s set of open file descriptors.

At the next step we check that if our flags contains O\_SYNC flag, we apply O\_DSYNC flag too:

1. if (flags & \_\_O\_SYNC)
2. flags |= O\_DSYNC;

The O\_SYNC flag guarantees that the any write call will not return before all data has been transferred to the disk. The O\_DSYNC is like O\_SYNC except that there is no requirement to wait for any metadata (like atime, mtime and etc.) changes will be written. We apply O\_DSYNC in a case of \_\_O\_SYNC because it is implemented as \_\_O\_SYNC|O\_DSYNC in the Linux kernel.

After this we must be sure that if a user wants to create temporary file, the flags should contain O\_TMPFILE\_MASK or in other words it should contain or O\_CREAT or O\_TMPFILE or both and also it should be writeable:

1. if (flags & \_\_O\_TMPFILE) {
2. if ((flags & O\_TMPFILE\_MASK) != O\_TMPFILE)
3. return -EINVAL;
4. if (!(acc\_mode & MAY\_WRITE))
5. return -EINVAL;
6. } else if (flags & O\_PATH) {
7. flags &= O\_DIRECTORY | O\_NOFOLLOW | O\_PATH;
8. acc\_mode = 0;
9. }

as it is written in in the manual page:

O\_TMPFILE must be specified with one of O\_RDWR or O\_WRONLY

If we didn’t pass O\_TMPFILE for creation of a temporary file, we check the O\_PATH flag at the next condition. The O\_PATH flag allows us to obtain a file descriptor that may be used for two following purposes:

* to indicate a location in the filesystem tree;
* to perform operations that act purely at the file descriptor level.

So, in this case the file itself is not opened, but operations like dup, fcntl and other can be used. So, if all file content related operations like read, write and other are permitted, only O\_DIRECTORY | O\_NOFOLLOW | O\_PATH flags can be used. We have finished with flags for this moment in the build\_open\_flags for this moment and we may fill our open\_flags->open\_flag with them:

1. op->open\_flag = flags;

Now we have filled open\_flag field which represents flags that will control opening of a file and mode that will represent umask of a new file if we open file for creation. There are still to fill last flags in the our open\_flags structure. The next is op->acc\_mode which represents access mode to a opened file. We already filled the acc\_mode local variable with the initial value at the beginning of the build\_open\_flags and now we check last two flags related to access mode:

1. if (flags & O\_TRUNC)
2. acc\_mode |= MAY\_WRITE;
3. if (flags & O\_APPEND)
4. acc\_mode |= MAY\_APPEND;
5. op->acc\_mode = acc\_mode;

These flags are - O\_TRUNC that will truncate an opened file to length 0 if it existed before we open it and the O\_APPEND flag allows to open a file in append mode. So the opened file will be appended during write but not overwritten.

The next field of the open\_flags structure is - intent. It allows us to know about our intention or in other words what do we really want to do with file, open it, create, rename it or something else. So we set it to zero if our flags contains the O\_PATH flag as we can’t do anything related to a file content with this flag:

1. op->intent = flags & O\_PATH ? 0 : LOOKUP\_OPEN;

or just to LOOKUP\_OPEN intention. Additionally we set LOOKUP\_CREATE intention if we want to create new file and to be sure that a file didn’t exist before with O\_EXCL flag:

1. if (flags & O\_CREAT) {
2. op->intent |= LOOKUP\_CREATE;
3. if (flags & O\_EXCL)
4. op->intent |= LOOKUP\_EXCL;
5. }

The last flag of the open\_flags structure is the lookup\_flags:

1. if (flags & O\_DIRECTORY)
2. lookup\_flags |= LOOKUP\_DIRECTORY;
3. if (!(flags & O\_NOFOLLOW))
4. lookup\_flags |= LOOKUP\_FOLLOW;
5. op->lookup\_flags = lookup\_flags;
6. return 0;

We fill it with LOOKUP\_DIRECTORY if we want to open a directory and LOOKUP\_FOLLOW if we don’t want to follow (open) [symlink](https://en.wikipedia.org/wiki/Symbolic_link). That’s all. It is the end of the build\_open\_flags function. The open\_flags structure is filled with modes and flags for a file opening and we can return back to the do\_sys\_open.

**Actual opening of a file**

At the next step after build\_open\_flags function is finished and we have formed flags and modes for our file we should get the filename structure with the help of the getname function by name of a file which was passed to the open system call:

1. tmp = getname(filename);
2. if (IS\_ERR(tmp))
3. return PTR\_ERR(tmp);

The getname function is defined in the [fs/namei.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/namei.c) source code file and looks:

1. struct filename \*
2. getname(const char \_\_user \* filename)
3. {
4. return getname\_flags(filename, 0, NULL);
5. }

So, it just calls the getname\_flags function and returns its result. The main goal of the getname\_flags function is to copy a file path given from userland to kernel space. The filename structure is defined in the [include/linux/fs.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/fs.h) linux kernel header file and contains following fields:

* name - pointer to a file path in kernel space;
* uptr - original pointer from userland;
* aname - filename from [audit](https://linux.die.net/man/8/auditd) context;
* refcnt - reference counter;
* iname - a filename in a case when it will be less than PATH\_MAX.

As I already wrote above, the main goal of the getname\_flags function is to copy name of a file which was passed to the open system call from user space to kernel space with the strncpy\_from\_user function. The next step after a filename will be copied to kernel space is getting of new non-busy file descriptor:

1. fd = get\_unused\_fd\_flags(flags);

The get\_unused\_fd\_flags function takes table of open files of the current process, minimum (0) and maximum (RLIMIT\_NOFILE) possible number of a file descriptor in the system and flags that we have passed to the open system call and allocates file descriptor and mark it busy in the file descriptor table of the current process. The get\_unused\_fd\_flags function sets or clears the O\_CLOEXEC flag depends on its state in the passed flags.

The last and main step in the do\_sys\_open is the do\_filp\_open function:

1. struct file \*f = do\_filp\_open(dfd, tmp, &op);
2. if (IS\_ERR(f)) {
3. put\_unused\_fd(fd);
4. fd = PTR\_ERR(f);
5. } else {
6. fsnotify\_open(f);
7. fd\_install(fd, f);
8. }

The main goal of this function is to resolve given path name into file structure which represents an opened file of a process. If something going wrong and execution of the do\_filp\_open function will be failed, we should free new file descriptor with the put\_unused\_fd or in other way the file structure returned by the do\_filp\_open will be stored in the file descriptor table of the current process.

Now let’s take a short look at the implementation of the do\_filp\_open function. This function is defined in the [fs/namei.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/namei.c) linux kernel source code file and starts from initialization of the nameidata structure. This structure will provide a link to a file [inode](https://en.wikipedia.org/wiki/Inode). Actually this is one of the main point of the do\_filp\_open function to acquire an inode by the filename given to open system call. After the nameidata structure will be initialized, the path\_openat function will be called:

1. filp = path\_openat(&nd, op, flags | LOOKUP\_RCU);
2. if (unlikely(filp == ERR\_PTR(-ECHILD)))
3. filp = path\_openat(&nd, op, flags);
4. if (unlikely(filp == ERR\_PTR(-ESTALE)))
5. filp = path\_openat(&nd, op, flags | LOOKUP\_REVAL);

Note that it is called three times. Actually, the Linux kernel will open the file in [RCU](https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt) mode. This is the most efficient way to open a file. If this try will be failed, the kernel enters the normal mode. The third call is relatively rare, only in the [nfs](https://en.wikipedia.org/wiki/Network_File_System) file system is likely to be used. The path\_openat function executes path lookup or in other words it tries to find a dentry (what the Linux kernel uses to keep track of the hierarchy of files in directories) corresponding to a path.

The path\_openat function starts from the call of the get\_empty\_flip() function that allocates a new file structure with some additional checks like do we exceed amount of opened files in the system or not and etc. After we have got allocated new file structure we call the do\_tmpfile or do\_o\_path functions in a case if we have passed O\_TMPFILE | O\_CREATE or O\_PATH flags during call of the open system call. These both cases are quite specific, so let’s consider quite usual case when we want to open already existed file and want to read/write from/to it.

In this case the path\_init function will be called. This function performs some preporatory work before actual path lookup. This includes search of start position of path traversal and its metadata like inode of the path, dentry inode and etc. This can be root directory - / or current directory as in our case, because we use AT\_CWD as starting point (see call of the do\_sys\_open at the beginning of the post).

После инициализации структуры nameidata будет вызвана функция path\_openat: filp = path\_openat (& nd, op, flags | LOOKUP\_RCU); если (маловероятно (filp == ERR\_PTR (-ECHILD))). filp = path\_openat (& nd, op, flags); если (маловероятно (filp == ERR\_PTR (-ESTALE))). filp = path\_openat (& nd, op, flags | LOOKUP\_REVAL);

Обратите внимание, что он вызывается три раза. На самом деле, ядро ​​Linux откроет файл в режиме RCU. Это самый эффективный способ открыть файл. Если эта попытка не удастся, ядро ​​перейдет в нормальный режим. Третий вызов относительно редок, вероятно, будет использоваться только в файловой системе NFS. Функция path\_openat выполняет поиск пути или, другими словами, пытается найти dentry (который ядро ​​Linux использует для отслеживания иерархии файлов в каталогах), соответствующий пути.

Функция path\_openat начинается с вызова функции **get\_empty\_flip** (), которая выделяет новую файловую структуру с некоторыми дополнительными проверками, например, превышаем ли мы количество открытых файлов в системе или нет и т. Д. После того, как мы получили выделенную новую файловую структуру, мы вызываем Функции **do\_tmpfile** или **do\_o\_path** в случае, если мы передали O\_TMPFILE | Флаги O\_CREATE или O\_PATH во время вызова открытого системного вызова. Оба эти случая довольно специфичны, поэтому давайте рассмотрим довольно обычный случай, когда мы хотим открыть уже существующий файл и хотим читать / записывать из него / в него.

В этом случае будет вызвана функция **path\_init**. Эта функция выполняет некоторые подготовительные работы перед фактическим поиском пути. Это включает в себя поиск начальной позиции обхода пути и его метаданных, таких как inode of path, dentry inode и т. Д. Это может быть корневой каталог - или текущий каталог, как в нашем случае, потому что мы используем AT\_CWD в качестве начальной точки (см. Вызов **do\_sys\_open** в начале поста).

The next step after the path\_init is the [loop](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/namei.c#L3457) which executes the link\_path\_walk and do\_last. The first function executes name resolution or in other words this function starts process of walking along a given path. It handles everything step by step except the last component of a file path. This handling includes checking of a permissions and getting a file component. As a file component is gotten, it is passed to walk\_component that updates current directory entry from the dcache or asks underlying filesystem. This repeats before all path’s components will not be handled in such way. After the link\_path\_walk will be executed, the do\_last function will populate a file structure based on the result of the link\_path\_walk. As we reached last component of the given **file path** the **vfs\_open** function from **the do\_last** will be called.

This function is defined in the [fs/open.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/fs/open.c) linux kernel source code file and the main goal of this function is to call an open operation of underlying filesystem.

That’s all for now.

We didn’t consider **full** implementation of the open system call. We skip some parts like handling case when we want to open a file from other filesystem with different mount point, resolving symlinks and etc., but it should be not so hard to follow this stuff. This stuff does not included in **generic** implementation of open system call and depends on underlying filesystem. If you are interested in, you may lookup the file\_operations.open callback function for a certain [filesystem](https://github.com/torvalds/linux/tree/master/fs).

Следующим шагом после **path\_init** является цикл, который выполняет **link\_path\_walk** и **do\_last**. Первая функция выполняет разрешение имен или, другими словами, эта функция запускает процесс прогулки по заданному пути. Он обрабатывает все шаг за шагом, кроме последнего компонента пути к файлу. Эта обработка включает в себя проверку разрешений и получение файлового компонента. При получении файлового компонента он передается в walk\_component, который обновляет текущую запись каталога из dcache или запрашивает базовую файловую систему. Это повторяется до того, как все компоненты пути не будут обработаны таким образом. После выполнения **link\_path\_walk** функция **do\_last** заполняет файловую структуру на основе результата link\_path\_walk. Когда мы добрались до последнего компонента данного пути к файлу, будет вызвана функция vfs\_open из **do\_last**.

Эта функция определена в файле исходного кода ядра Linux fs / open.c, и основная цель этой функции - вызвать операцию открытия базовой файловой системы.

Это все на данный момент. Мы не рассматривали полную реализацию системного вызова open. Мы пропускаем некоторые части, такие как обработка случая, когда мы хотим открыть файл из другой файловой системы с другой точкой монтирования, разрешение символических ссылок и т. Д., Но это не должно быть так сложно. Этот материал не входит в общую реализацию открытого системного вызова и зависит от базовой файловой системы. Если вы заинтересованы, вы можете найти функцию обратного вызова file\_operations.open для определенной файловой системы.

**Conclusion**

This is the end of the fifth part of the implementation of different system calls in the Linux kernel. If you have questions or suggestions, ping me on twitter [0xAX](https://twitter.com/0xAX), drop me an [anotherworldofworld@gmail.com](https://www.bookstack.cn/read/linux-insides/$SysCall-%3Ca%20href=)">email, or just create an [issue](https://github.com/0xAX/linux-internals/issues/new). In the next part, we will continue to dive into system calls in the Linux kernel and see the implementation of the [read](http://man7.org/linux/man-pages/man2/read.2.html) system call.

Это конец пятой части реализации различных системных вызовов в ядре Linux. Если у вас есть вопросы или предложения, напишите мне в твиттер 0xAX, напишите мне по электронной почте anotherworldofworld@gmail.com "> или просто создайте проблему. В следующей части мы продолжим погружаться в системные вызовы в ядре Linux и увидим реализация системного вызова read.

**Please note that English is not my first language and I am really sorry for any inconvenience. If you find any mistakes please send me PR to [linux-insides](https://github.com/0xAX/linux-internals).**