Лабораторная работа «Системный вызов open()»

Пользовательские программы взаимодействуют с ядром операционной системы посредством специальных механизмов, называемых *системными вызовами* (system calls, syscalls). Внешне системные вызовы реализованы в виде обычных функций языка C, однако каждый раз вызывая такую функцию, мы обращаемся непосредственно к ядру операционной системы. Список всех системных вызовов Linux можно найти в файле /usr/include/asm/unistd.h. В этой главе мы рассмотрим основные системные вызовы, осуществляющие ввод-вывод: open(), close(), read(), write(), lseek() и некоторые другие.

Целью данной лабораторной работы является углубленное изучение системных вызовов на примере системного вызова open().

open(2)

Чтобы получить возможность прочитать что-то из файла или записать что-то в файл, его нужно открыть. Это делает системный вызов open().

open – открывает и возможно создает файл.

SYNOPSIS top

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Системный вызов open() открывает файл, указанный в pathname. Если указанный файл не существует, он может (необязательно) (если указан флаг O_CREAT) быть создан open().

Возвращаемое значение open() - это дескриптор файла, небольшое неотрицательное целое число, которое используется в последующих системных вызовах (read (2), write (2), lseek (2), fcntl (2) и т. д.) для ссылки на открытый файл. Дескриптор файла, возвращаемый при успешном вызове, будет дескриптором файла с наименьшим номером, который в данный момент не открыт процессом.

По умолчанию новый дескриптор файла реализован так, чтобы оставаться открытым через execve(2) (при вызове системного вызова exec() файл остается открытым, то есть флаг дескриптора файла FD_CLOEXEC, описанный в fcntl (2), изначально отключен). Описанный ниже флаг O_CLOEXEC можно использовать для изменения этого значения по умолчанию. Смещение файла устанавливается на начало файла (см. lseek(2)).

Вызов open() создает новый дескриптор открытого файла – struct file в системной таблице открытых файлов. Дескриптор открытого файла содержит поля, указывающие смещение в файле и флаги состояния файла. Дескриптор файла это - ссылка на описание открытого файла; она не затрагивается, если впоследствии путь удаляется или изменяется для ссылки на другой файл.

Флаги аргументов должны включать один из следующих режимов доступа: O_RDONLY, O_WRONLY или O_RDWR. Эти запросы открывают файл только для чтения, только для записи или для чтения / записи, соответственно.

Кроме того, ноль или более флагов создания файлов и флагов состояния файлов могут быть побитовыми или определенными в системе флагами.

Флаги создания файла: O_CLOEXEC, O_CREAT, O_DIRECTORY, O_EXCL, O_NOCTTY, O_NOFOLLOW, O_TMPFILE и O_TRUNC. Флаги состояния файла это – достаточно большой набор флагов, которые учитываются при открытии или создании файлов в функциях ядра, предназначенных для этой работы. Различие между этими двумя группами флагов состоит в том, что флаги создания файла влияют на семантику самой операции открытия, в то время как флаги состояния файла влияют на семантику последующих операций ввода-вывода. Флаги состояния файла могут быть извлечены и (в некоторых случаях) изменены системным вызовом fcntl(2).

Если вы хотите, например, открыть файл в режиме чтения и записи, и при этом автоматически создать файл, если такового не существует, то второй аргумент open() будет выглядеть примерно так: **O_RDWR|O_CREAT**. Константы-флаги открытия объявлены в заголовочном файле bits/fcntl.h, однако не стоит включать этот файл в свои программы, поскольку он уже включен в файл fcntl.h.

Список некоторых флагов создания файлов и флагов состояния файлов:

O_APPEND

Файл открывается в режиме добавления. Перед каждой записью (2) смещение файла помещается в конец файла, как будто с lseek (2). Изменение смещения файла и операция записи выполняются как один атомарный шаг.

O_APPEND может привести к повреждению файлов в файловых системах NFS, если несколько файлов одновременно добавляют данные в файл. Это связано с тем, что NFS не поддерживает добавление в файл, поэтому ядро клиента должно имитировать его, что невозможно сделать без условия гонки.

O_CLOEXEC (начиная с Linux 2.6.23)

Включите флаг close-on-exec для нового файлового дескриптора. Указание этого флага позволяет программе избегать дополнительных операций fcntl(2) F_SETFD для установки флага FD_CLOEXEC.

Обратите внимание, что использование этого флага необходимо в некоторых многопоточных программах, потому что использование отдельной операции fcntl(2) F_SETFD для установки флага FD_CLOEXEC недостаточно, чтобы избежать условий гонки, когда один поток открывает дескриптор файла и пытается установить свой флаг close-on-exec с использованием fcntl(2) одновременно с тем, как другой поток выполняет вызвать fork(2), а затем execve(2). В зависимости от порядка выполнения гонка может привести к тому, что дескриптор файла, возвращаемый open(), непреднамеренно попадет в программу, выполняемую дочерним процессом, созданным fork(2). (Этот вид гонки в принципе возможен для любого системного вызова, который создает дескриптор файла, для которого должен быть установлен флаг close-on-exec, а различные другие системные вызовы Linux предоставляют эквивалент флага O_CLOEXEC для решения этой проблемы.)

O_CREAT

Если *имя пути* не существует, то файл создается как обычный файл.

Владельцем (идентификатор пользователя) нового файла устанавливается идентификатор user процесса, который открыл файл.

Идентификатор группы нового файла устанавливается по эффективному групповому идентификатору процесса (семантика System V) или как идентификатор группы родительского каталога (семантика BSD).

В Linux поведение зависит от того, установлен ли бит режима set-group-ID в родительском каталоге: если этот бит установлен, применяется семантика BSD; в противном случае применяется семантика System V. Для некоторых файловых систем поведение также зависит от параметров монтирования bsdgroups и sysvgroups, описанных в mount (8).

Аргумент mode указывает биты режима файла, применяемые при создании нового файла. Этот аргумент должен быть указан, когда флаги O_CREAT или O_TMPFILE; если ни O_CREAT, ни O_TMPFILE не указаны, то режим игнорируется. Эффективный режим модифицируется с

помощью umask процесса обычным способом: при отсутствии ACL[1] по умолчанию, режим создаваемого файла (mode & ~ umask). Обратите внимание, что этот режим применяется к вновь созданному файлу только к будущему доступу; вызов open(), который создает файл только для чтения, вполне может вернуть дескриптор файла для чтения / записи.
Для режима (mode) предусмотрены следующие символические константы:

- Пользователь (владелец файла) имеет права доступа:
  S_IRWXU 00700 – на чтение, на запись, на исполнение;
  S_IRUSR 00400 - на чтение;
  S_IWUSR 00200 - на запись;
  S_IXUSR 00100 - на выполнение;
- Группа пользователя имеет права доступа:
  S_IRWXG 00070 - на чтение, запись и выполнение;
  S_IRGRP 00040 - на чтение;
  S_IWGRP 00020 - на запись;
  S_IXGRP 00010 - на выполнение;
- Остальные имеют права доступа:
  S_IRWXO 00007 - на чтение, запись и выполнение;
  S_IROTH 00004 - на чтение;
  S_IWOTH 00002 - на запись;
  S_IXOTH 00001 - на выполнение,

Если установить флаги следующим образом:
S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH или **0664** это означает для пользователя - чтение/запись, для группы - чтение/запись и для остального мира -чтение.
…
O_EXCL
Убедитесь, что этот вызов создает файл: если этот флаг установлен указывается в сочетании с O_CREAT и pathname уже существует, то open() завершается ошибкой EEXIST.
Когда указаны эти два флага, символические ссылки не отображаются.
В общем случае поведение O_EXCL не определено, если оно используется без O_CREAT. Есть одно исключение: в Linux 2.6 и более поздние версии O_EXCL можно использовать без O_CREAT, если pathname ссылается на блочное устройство. Если блочное устройство используется системой (например, смонтированный), open() завершается ошибкой с ошибкой EBUSY.

В файловой системе NFS флаг O_EXCL поддерживается только при использовании NFSv3 или более поздней версии ядра 2.6. В средах NFS, где нет поддержки O_EXCL, программы, которые предполагают его наличие при выполнении задач блокировки будут возникать условия «гонки».
В переносимых программах, которые выполняют атомарную блокировку файлов, следует использовать link(2). Если link(2) возвращает 0, блокировка выполнена успешно. В противном случае используйте stat(2) на уникальном файле, чтобы проверить, увеличилось ли количество ссылок на него до 2, и в этом случае блокировка также прошла успешно.

O_PATH (since Linux 2.6.39)
Получите файловый дескриптор, который можно использовать для двух целей: указать местоположение в дереве каталогов файловой системы и выполнить операции, которые действуют исключительно на уровне файлового дескриптора. Сам файл не открывается, и другие файловые операции (например, read (2), write (2), fchmod (2), fchown (2), fgetxattr (2), ioctl (2), mmap (2)) завершаются неудачно с ошибкой EBADF.

---

[1] **Access Control List** или **ACL** — список управления доступом, который определяет, кто или что может получать доступ к объекту (программе, процессу или **файлу**), и какие именно операции разрешено или запрещено выполнять субъекту: пользователю, группе пользователей

…

O_TMPFILE (since Linux 3.11)

Создать неназванный временный обычный файл. Аргумент pathname указывает каталог; безымянный индекс будет создан в файловой системе этого каталога. Все, что записано в результирующий файл, будет потеряно при закрытии последнего дескриптора файла, если файлу не присвоено имя.

Флаг O_TMPFILE должен быть указан с одним из O_RDWR или O_WRONLY и, необязательно, O_EXCL. Если O_EXCL не указан, то linkat (2) может использоваться для связывания временного файла с файловой системой.

и т.д.

Рассмотрим пример простой программы, в которой используются системные вызовы open() и close(). Эта программа, которая создает файл с именем, переданным в качестве аргумента (argv[1]) и с правами доступа 0600 (чтение и запись для пользователя). Ниже приведен исходный код программы.

```
/* openclose.c */
#include <fcntl.h>   /* open() and O_XXX flags */
#include <sys/stat.h>       /* S_IXXX flags */
#include <sys/types.h>      /* mode_t */
#include <unistd.h>  /* close() */
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char ** argv)
{
       int fd;
       mode_t mode = S_IRUSR | S_IWUSR;
       int flags = O_WRONLY | O_CREAT | O_EXCL;
       if (argc < 2)
       {
              fprintf (stderr, "openclose: Too few arguments\n");
              fprintf (stderr, "Usage: openclose <filename>\n");
              exit (1);
       }

       fd = open(argv[1], flags, mode);
       if (fd < 0)
       {
       fprintf (stderr, "openclose: Cannot open file '%s'\n",argv[1]);
              exit (1);
       }

       if (close (fd) != 0)
       {
        fprintf (stderr, "Cannot close file (descriptor=%d)\n", fd);
              exit (1);
       }
       exit (0);
}
```

Обратите внимание, если запустить эту программу дважды с одним и тем же аргументом, то на второй раз open() выдаст ошибку. В этом виноват флаг **O_EXCL**, который указывает что можно создать только еще не существующий файл.

Системные вызовы в ОС

Системные вызовы это – сервис, предоставляемый ядром ОС. В программировании на С, часто используются библиотечные функции, которые являются обертками для многих системных вызовов в Linux.

Набор системных вызовов разный в разных операционных системах. Итого в ядре Linux около 310 системных вызовов. С ними можно познакомиться в таблице 1. Для сравнения, в ОС Windows системных вызовов около 460.

Таблица 1

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c
3	close	sys_close	fs/open.c
4	stat	sys_newstat	fs/stat.c
5	fstat	sys_newfstat	fs/stat.c
6	lstat	sys_newlstat	fs/stat.c
7	poll	sys_poll	fs/select.c
8	lseek	sys_lseek	fs/read_write.c

…

На рис.1 схематически показано выполнение С-функции fwrite():

1. fwrite() вместе с остальной частью стандартной библиотеки С реализован в glibc.
2. fwrite() вызывает более низкоуровневую функцию write().
3. write() загружает идентификатор системного вызова, который для write() равен 1, и аргументы в регистры процессора, а затем заставит процессор переключиться на уровень ядра. То, как это делается, зависит от архитектуры процессора, а иногда и от модели процессора. Например, процессоры x86 обычно вызывают прерывание 0x80, а процессоры x86-64 используют инструкцию процессора syscall.
4. Процессор, который теперь работает в режиме ядра, передает идентификатор системного вызова в таблицу системных вызовов, извлекает указатель функции со смещением 1 и вызывает функцию. Эта функция, sys_write(), является реализацией записи в файл.

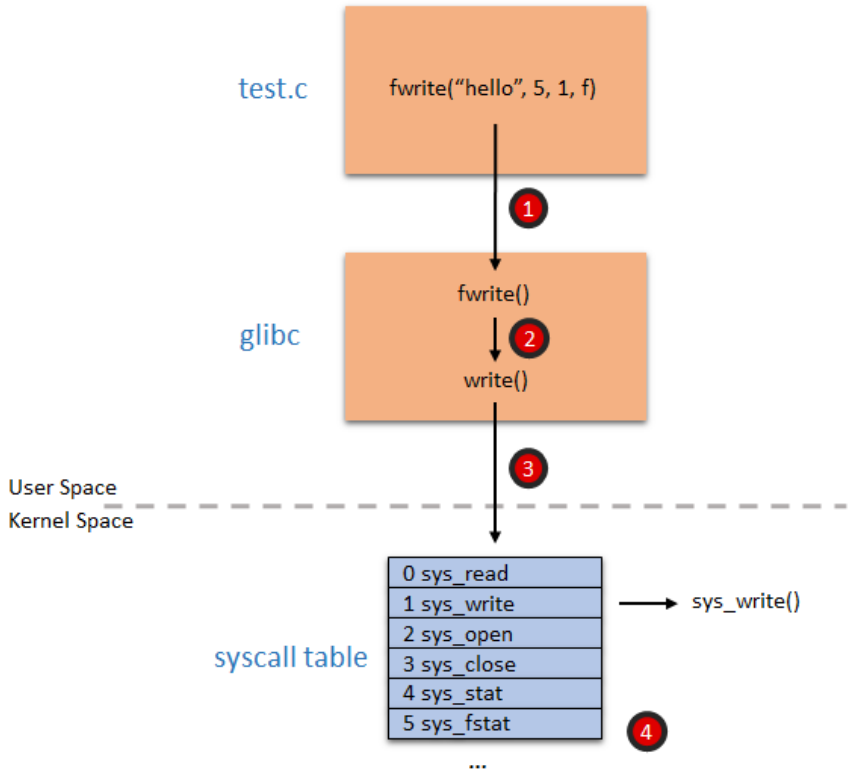


Рис.1

Из пользовательского пространства (кольцо защиты 3) нельзя непосредственно вызвать функцию ядра (кольцо защиты 0), как обычную функцию. На 3-м шаге используется механизм перехода в режим ядра, который зависит от архитектуры компьютера. На компьютерах самой популярной архитектуры x86 системный вызов делается одним из следующих методов:

- через программное прерывание,
- через инструкцию sysenter,
- через инструкцию syscall.

Программное прерывание

Системный вызов или программное прерывание возникает в процессе выполнения программы, когда процессу требуется обслуживание со стороны ОС. В процессорах архитектуры x86 для обращения к ОС за сервисом имеется инструкция int, аргументом которой является номер прерывания (от 0 до 255). В ОС Linux этот номер равен 0x80 . Обработчиком прерывания 0x80 является ядро Linux. Программа перед выполнением прерывания помещает в регистр eax номер системного вызова, который нужно выполнить. Когда управление к ядру, т.е. система переключается на нулевое кольцо защиты, ядро считывает этот номер и вызывает нужную функцию (см. таблицу 1).

Этот метод этот широко применялся на 32-битных системах, на 64-битных считается устаревшим и не применяется, но поддерживается, хотя с целым рядом ограничений (например, нельзя в качестве параметра передать 64-битный указатель).

- Поместить номер системного вызова в eax.
- Поместить аргументы в регистры ebx, ecx, edx, esi, edi, ebp.
- Вызвать инструкцию int 0x80.
- Получить результат из eax.

Пример реализации mygetpid() (получение PID текущего процесса) на ассемблере (для системного вызова getpid() используется номер 20):

```
.intel_syntax noprefix

.globl mygetpid

.text
mygetpid:
   mov eax, 20
   int 0x80
   ret
```

Инструкция sysenter

Для ускорения выполнения системных вызовов, когда ещё не было x86-64, Intel стала применять специальную инструкцию системного вызова, тем самым устраняя некоторые издержки прерывания. Ускорение достигается за счёт того, что на аппаратном уровне при выполнении инструкции sysenter опускается множество проверок на валидность дескрипторов, а также проверок, зависящих от уровня привилегий.

На сегодня эти инструкции (sysenter и sysexit) поддерживаются процессорами Intel в 32- и 64-битных режимах, процессорами AMD — только в 32-битном (на 64-битном приводит к исключению неизвестного типа).

Инструкция syscall

Фирма AMD в архитектуре AMD64 предложила использовать для системных вызовов инструкции syscall и парную ей sysret, которые поддерживаются процессорами Intel только в 64-битном режиме, а процессорами AMD — во всех режимах.

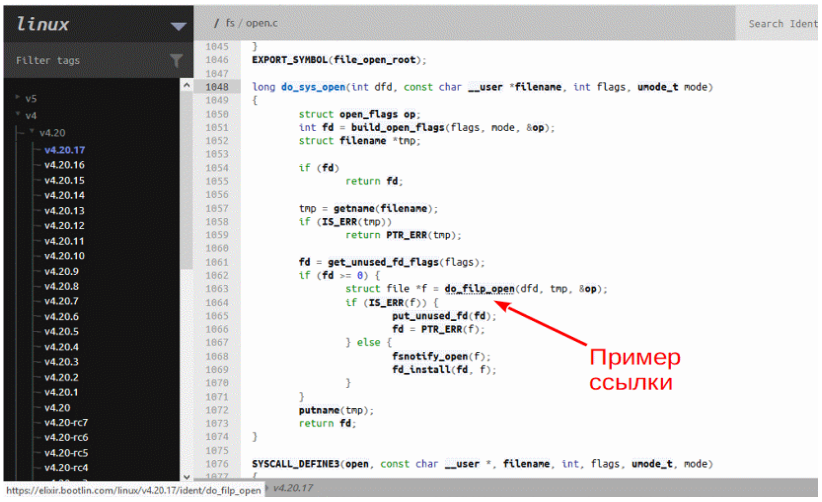
Системные вызовы при помощи этой инструкции делаются в современных версиях 64-битного Linux.

- Номер системного вызова помещается в rax.
- Аргументы записываются в rdi, rsi, rdx, r10, r8 и r9.
- Затем вызывается syscall.
- Когда управление возвращается, результат находится в rax.
- Значения всех регистров, кроме r11, rcx и rax, системным вызовом не изменяются, дополнительно сохранять их не требуется.

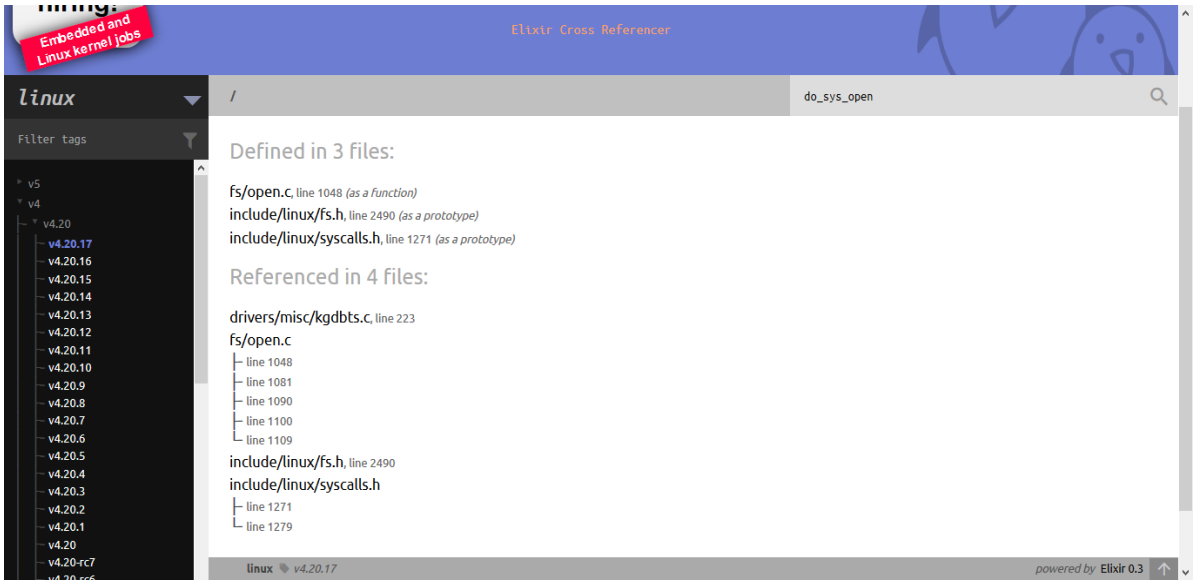
«Анатомия» системного вызова open()

Для просмотра исходного кода рекомендуется использовать сайт elixir.bootlin.com (https://elixir.bootlin.com/linux/v4.20.17/source), который подгружает исходный код из репозитория Linux и строит перекрестные ссылки на функции, структуры и макросы. Сайт выглядит следующим образом:

В левой части располагается выбор нужной версии ядра, в центре отображаются файлы и результаты поиска, строка поиска находится сверху справа. Для навигации по



коду достаточно кликнуть на интересующую функцию, структуру или макрос.



Результаты поиска разбиты на две части: определения (defined) и ссылки (referenced). В части определения отображаются места определения функций, прототипов функции, макросов и структур с соответствующими пометками as function, as prototype, as macro и as struct. Для нахождения исходного кода функции нужно открывать ссылку, помеченную как as function, а для нахождения исходного кода макроса - как as macro.

В схему нужно обязательно включить:

1) копирование названия файла из пространства пользователя в пространство ядра;
2) блокировка/разблокировка (spinlock) структуры files_struct;
3) алгоритм поиска свободного дескриптора открытого файла;
4) работу со структурой nameidata;
5) алгоритм разбора пути (кратко);
6) инициализацию struct file;
7) создание inode в случае отсутствия открываемого файла.

Отчет должен включать: титульный лист и схему алгоритма работы системного вызова open().

Системные вызовы определяются с помощью макроса SYSCALL_DEFINE. Системный вызов open() не является исключением. Определение системного вызова open() находится в файле исходного кода fs/open.c

```
https://elixir.bootlin.com/linux/latest/source/fs/open.c#L1130
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags,
umode_t, mode)
{
        return ksys_open(filename, flags, mode);
}
```

include/linux/syscalls.h, line 1381 *(as a function)*

Макрос **SYSCALL_DEFINE3** определяет системный вызов с тремя параметрами.

Системный вызов **open()** является оберткой функции ядра **ksys_open()**, которая в свою очередь вызывает функцию **do_sys_open():**

```
static inline long ksys_open(const char __user *filename, int flags,
                             umode_t mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;
        return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

Ясно, что основную работу выполняет функция **do_sys_open()** из того же файла исходного кода. Но прежде, чем эта функция будет вызвана, рассмотрим предложение if, с которого начинается реализация системного вызова open():

```
if (force_o_largefile())
   flags |= O_LARGEFILE;
```

Здесь применяется флаг O_LARGEFILE к флагам, которые были переданы для системного вызова open() в случае, когда force_o_largefile() вернет true.

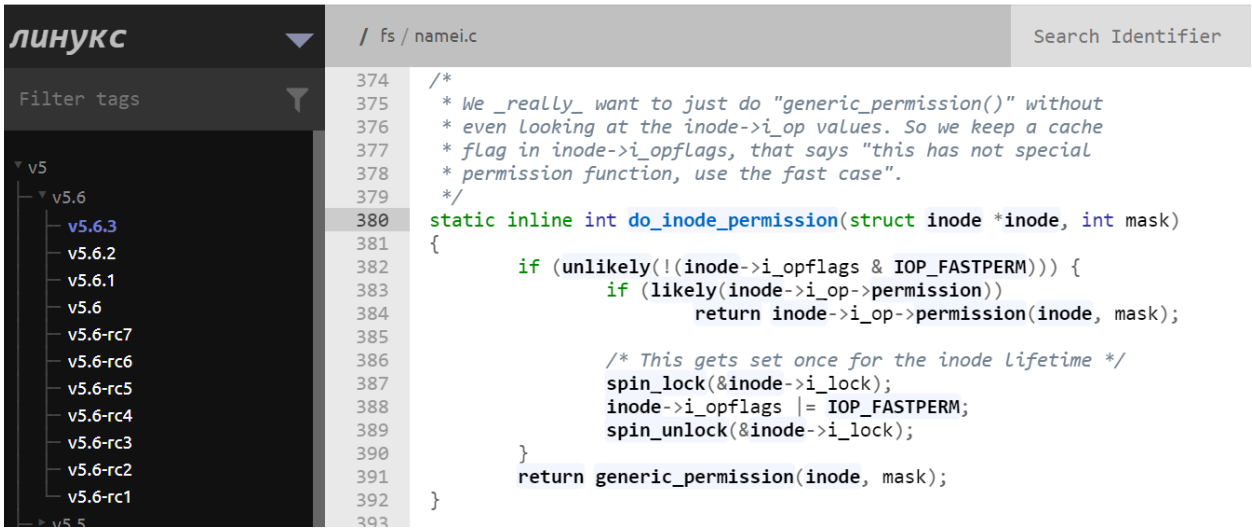
O_LARGEFILE (LFS) позволяет открывать файлы, размеры которых не могут быть представлены в off_t (но могут быть представлены в off64_t).

Другими словами, макрос **force_o_largefile**() проверяет разрядность системы, и в случае 64-битной разрядности добавляет флаг **O_LARGEFILE**, который позволяет открывать файлы, размер которых не может быть представлен типом off_t (long).

Кликнув по имени функции do_sys_open() осуществляется переход на эту функцию. Анализ функции показывает, что в теле функции вызываются две функции: build_open_how() и do_sys_openat2().

```
long do_sys_open(int dfd, const char __user *filename, int flags,
umode_t mode)
{
        struct open_how how = build_open_how(flags, mode);
        return do_sys_openat2(dfd, filename, &how);
}
```

В данном материале используется исходный код ядра Linux версии v5.6.3, как видно из приведенного скриншота.



В теле функции do_sys_open() вызывается функция build_open_how() и функция do_sys_openat2().

```
struct open_how {
        __u64 flags;
        __u64 mode;
        __u64 resolve;
};
/* List of all valid flags for the open/openat flags argument: */
#define VALID_OPEN_FLAGS \
        (O_RDONLY | O_WRONLY | O_RDWR | O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC | \
         O_APPEND | O_NDELAY | O_NONBLOCK | O_NDELAY | __O_SYNC | O_DSYNC | \
         FASYNC   | O_DIRECT | O_LARGEFILE | O_DIRECTORY | O_NOFOLLOW | \
         O_NOATIME | O_CLOEXEC | O_PATH | __O_TMPFILE)

#define S_IALLUGO (S_ISUID|S_ISGID|S_ISVTX|S_IRWXUGO)
```

```
inline struct open_how build_open_how(int flags, umode_t mode)
{
	struct open_how how = {
		.flags = flags & VALID_OPEN_FLAGS,
		.mode = mode & S_IALLUGO,
	};

	/* O_PATH beats everything else. */
	if (how.flags & O_PATH)
		how.flags &= O_PATH_FLAGS;
	/* Modes should only be set for create-like flags. */
	if (!WILL_CREATE(how.flags))
		how.mode = 0;
	return how;
}
```

Функция do_sys_openat2 вызывает функции: build_open_flags(), getname(filename), get_unused_fd_flags() и, если функция get_unused_fd_flags() возвращает fd > 0 , то вызывается функция do_filp_open().

```
static long do_sys_openat2(int dfd, const char __user *filename,
			   struct open_how *how)
{
	struct open_flags op;
	int fd = build_open_flags(how, &op);
	struct filename *tmp;

	if (fd)
		return fd;

	tmp = getname(filename);
	if (IS_ERR(tmp))
		return PTR_ERR(tmp);

	fd = get_unused_fd_flags(how->flags);
	if (fd >= 0) {
		struct file *f = do_filp_open(dfd, tmp, &op);
		if (IS_ERR(f)) {
			put_unused_fd(fd);
			fd = PTR_ERR(f);
		} else {
			fsnotify_open(f);
			fd_install(fd, f);
		}
	}
	putname(tmp);
	return fd;
}
```

Функция **do_sys_open**at2() выполняет все задачи системного вызова open. Сначала выполняется проверка правильности флагов и их преобразование во внутреннее представление функцией **build_open_flags**(). В случае неуспешного преобразования и проверки будет возвращена ошибка преобразования и проверки.

```
inline int build_open_flags(const struct open_how *how, struct
open_flags *op)
```

```
{
        int flags = how->flags;
        int lookup_flags = 0;
        int acc_mode = ACC_MODE(flags);

        /* Must never be set by userspace Никогда не должен быть
установлен пользователем*/
        flags &= ~(FMODE_NONOTIFY | O_CLOEXEC);

        /*
 * Older syscalls implicitly clear all of the invalid flags or argument
 * values before calling build_open_flags(), but openat2(2) checks all
 * of its arguments.
```
Старые системные вызовы неявно очищают все недопустимые флаги или значения аргументов перед вызовом build_open_flags (), но openat2 (2) проверяет все свои аргументы.
```
 */
        if (flags & ~VALID_OPEN_FLAGS)
                return -EINVAL;
        if (how->resolve & ~VALID_RESOLVE_FLAGS)
                return -EINVAL;

        /* Deal with the mode. */
        if (WILL_CREATE(flags)) {
                if (how->mode & ~S_IALLUGO)
                        return -EINVAL;
                op->mode = how->mode | S_IFREG;
        } else {
                if (how->mode != 0)
                        return -EINVAL;
                op->mode = 0;
        }

/*
 * In order to ensure programs get explicit errors when trying to use
 * O_TMPFILE on old kernels, O_TMPFILE is implemented such that it
 * looks like (O_DIRECTORY|O_RDWR & ~O_CREAT) to old kernels. But we have to require user
space to explicitly set it.
```
Чтобы гарантировать, что программы получают явные ошибки при попытке использовать O_TMPFILE на старых ядрах, O_TMPFILE реализован таким образом, что он выглядит (O_DIRECTORY | O_RDWR & ~ O_CREAT) для старых ядер. Но мы должны требовать, чтобы пользовательское пространство явно устанавливало его.
```
 */
        if (flags & __O_TMPFILE) {
                if ((flags & O_TMPFILE_MASK) != O_TMPFILE)
                        return -EINVAL;
                if (!(acc_mode & MAY_WRITE))
                        return -EINVAL;
        }
        if (flags & O_PATH) {
                /* O_PATH only permits certain other flags to be set. */
                if (flags & ~O_PATH_FLAGS)
                        return -EINVAL;
                acc_mode = 0;
        }

/*
```

```
 * O_SYNC is implemented as __O_SYNC|O_DSYNC.  As many places only
 *check for O_DSYNC if the need any syncing at all we enforce it's
 always set instead of having to deal with possibly weird behaviour
 for malicious applications setting only __O_SYNC.
```
O_SYNC реализован как __O_SYNC | O_DSYNC. Так как во многих местах проверка O_DSYNC выполняется только в том случае, если нам требуется принудительная синхронизация, она всегда установлена для того, чтобы не иметь дело с, возможно, странным поведением для вредоносных приложений, устанавливающих только __O_SYNC.
```
 */
        if (flags & __O_SYNC)
                flags |= O_DSYNC;

        op->open_flag = flags;

/* O_TRUNC implies we need access checks for write permissions
O_TRUNC подразумевает, что нам нужны проверки доступа для разрешений на запись
*/
        if (flags & O_TRUNC)
                acc_mode |= MAY_WRITE;

/* Allow the LSM permission hook to distinguish append access from general write access.
Разрешите ловушку разрешения LSM, чтобы отличить доступ добавления от общего доступа записи. */
        if (flags & O_APPEND)
                acc_mode |= MAY_APPEND;

        op->acc_mode = acc_mode;

        op->intent = flags & O_PATH ? 0 : LOOKUP_OPEN;

        if (flags & O_CREAT) {
                op->intent |= LOOKUP_CREATE;
                if (flags & O_EXCL)
                        op->intent |= LOOKUP_EXCL;
        }

        if (flags & O_DIRECTORY)
                lookup_flags |= LOOKUP_DIRECTORY;
        if (!(flags & O_NOFOLLOW))
                lookup_flags |= LOOKUP_FOLLOW;

        if (how->resolve & RESOLVE_NO_XDEV)
                lookup_flags |= LOOKUP_NO_XDEV;
        if (how->resolve & RESOLVE_NO_MAGICLINKS)
                lookup_flags |= LOOKUP_NO_MAGICLINKS;
        if (how->resolve & RESOLVE_NO_SYMLINKS)
                lookup_flags |= LOOKUP_NO_SYMLINKS;
        if (how->resolve & RESOLVE_BENEATH)
                lookup_flags |= LOOKUP_BENEATH;
        if (how->resolve & RESOLVE_IN_ROOT)
                lookup_flags |= LOOKUP_IN_ROOT;

        op->lookup_flags = lookup_flags;
        return 0;
}
```

Затем вызывается функция **getname** (const char __user *filename), которая выполняет копирование имени файла из пространства пользователя в пространство ядра.

```
struct audit_names;
struct filename {
        const char          *name;    /* pointer to actual string */
        const __user char   *uptr;    /* original userland pointer */
        int                 refcnt;
        struct audit_names  *aname;
        const char          iname[];
};
```

В структуре struct file name имеется поле с типом struct **audit_names** *aname;

```
/* When fs/namei.c:getname() is called, we store the pointer in name and bump
 * the refcnt in the associated filename struct.
```

Когда вызывается fs / namei.c: getname (), мы сохраняем указатель в имени и поднимаем **refcnt** в структуре ассоциированного имени файла.

```
 * Further, in fs/namei.c:path_lookup() we store the inode and device.
```

Далее, в fs / namei.c: path_lookup () мы храним индекс и устройство.

```
 */
struct audit_names {
        struct list_head     list;  /* audit_context->names_list */

        struct filename              *name;
        int                  name_len;     /* number of chars to log */
        bool                 hidden;       /* don't log this record */

        unsigned long        ino;
        dev_t                dev;
        umode_t                      mode;
        kuid_t               uid;
        kgid_t               gid;
        dev_t                rdev;
        u32                  osid;
        struct audit_cap_data        fcap;
        unsigned int         fcap_ver;
        unsigned char        type;         /* record type */
/*
 * This was an allocated audit_names and not from the array of names allocated in the task audit context.
* Thus this name should be freed on syscall exit.
```

Это были выделенные аудиторские имена, а не из массива имен, выделенных в контексте аудита задачи. Таким образом, это имя должно быть освобождено при выходе из системного вызова.

```
*/
        bool                 should_free;
};


struct filename *getname(const char __user * filename)
{
        return getname_flags(filename, 0, NULL);
}
```

Из функции getname() вызывается функция getname_flags(), которая копирует имя файла из пространства пользователя в пространство ядра. В теле функции getname_flags() происходит вызов функции **strncpy_from_user**(). В случае неуспешного копирования возвращается ошибка.

```
struct filename *getname_flags(const char __user *filename, int flags,
                               int *empty)
{
        struct filename *result;
```

```
        char *kname;
        int len;

        result = audit_reusename(filename);
```

Раскроем содержание данного вызова:

```
static inline struct filename *audit_reusename(const __user char *name)
{
    if (unlikely(!audit_dummy_context()))
            return __audit_reusename(name);
    return NULL;
}

/**
 * __audit_reusename - fill out filename with info from existing entry
 * @uptr: userland ptr to pathname
```

__audit_reusename - заполнить *filename* информацией из существующей записи @uptr: указатель пространства пользователя на pathname

```
 *
 * Search the audit_names list for the current audit context. If there is an
 * existing entry with a matching "uptr" then return the filename
 * associated with that audit_name. If not, return NULL.
```

Найдите в списке audit_names текущий контекст аудита. Если такая запись существует с соответствующим «uptr», тогда верните имя файла, связанное с этим audit_name. Если нет, вернуть NULL.

```
 */
struct filename *
__audit_reusename(const __user char *uptr)
{
            struct audit_context *context = audit_context();
            struct audit_names *n;

            list_for_each_entry(n, &context->names_list, list)
           {
                        if (!n->name)
                                    continue;
                        if (n->name->uptr == uptr) {
                                    n->name->refcnt++;
                                    return n->name;
                        }
            }
          return NULL;
}
```

```
        if (result)
                return result;

        result = __getname();
```

Строки, связанные с кэшированием данных и, в частности, со слаб кэшем

```
extern struct kmem_cache *names_cachep;

#define __getname()             kmem_cache_alloc(names_cachep, GFP_KERNEL)
#define __putname(name)                 kmem_cache_free(names_cachep, (void *)(name))
```

```
        if (unlikely(!result))
                return ERR_PTR(-ENOMEM);
```

```
/*
 * First, try to embed the struct filename inside the names_cache
* allocation
```
Во-первых, попробуйте встроить *struct filename* в распределение names_cache.
```
*/
        kname = (char *)result->iname;
        result->name = kname;

        len = strncpy_from_user(kname, filename, EMBEDDED_NAME_MAX);
        if (unlikely(len < 0)) {
                __putname(result);
                return ERR_PTR(len);
        }

/*
 * Uh-oh. We have a name that's approaching PATH_MAX. Allocate a separate struct filename so we can
dedicate the entire names_cache allocation for the pathname, and re-do the copy from userland.
```
Ой-ой. У нас есть имя, которое приближается к PATH_MAX. Выделите отдельное struct filename, чтобы мы могли выделить все имена names_cache для пути и заново скопировать из пользовательского пространства.
```
*/
        if (unlikely(len == EMBEDDED_NAME_MAX)) {
                const size_t size = offsetof(struct filename, iname[1]);
                kname = (char *)result;

/*
 * size is chosen that way we to guarantee that result->iname[0] is within the same object and that
* kname can't be equal to result->iname, no matter what.
```
размер выбирается таким образом, чтобы мы могли гарантировать, что result-> iname [0] находится внутри одного и того же объекта и что kname не может быть равен result-> iname, несмотря ни на что.
```
*/
                result = kzalloc(size, GFP_KERNEL);
                if (unlikely(!result)) {
                        __putname(kname);
                        return ERR_PTR(-ENOMEM);
                }
                result->name = kname;
                len = strncpy_from_user(kname, filename, PATH_MAX);
                if (unlikely(len < 0)) {
                        __putname(kname);
                        kfree(result);
                        return ERR_PTR(len);
                }
                if (unlikely(len == PATH_MAX)) {
                        __putname(kname);
                        kfree(result);
                        return ERR_PTR(-ENAMETOOLONG);
                }
        }

        result->refcnt = 1;
        /* The empty path is special. */
        if (unlikely(!len)) {
                if (empty)
                        *empty = 1;
                if (!(flags & LOOKUP_EMPTY)) {
                        putname(result);
```

```
                        return ERR_PTR(-ENOENT);
                }
        }

        result->uptr = filename;
        result->aname = NULL;
        audit_getname(result);
        return result;
}
```

Вызываемая функция get_unused_fd_flags() является оберткой функции **__alloc_fd()**, как видно из следующей последовательности описаний функций:

```
int __get_unused_fd_flags(unsigned flags, unsigned long nofile)
{
        return __alloc_fd(current->files, 0, nofile, flags);
}

int get_unused_fd_flags(unsigned flags)
{
        return __get_unused_fd_flags(flags, rlimit(RLIMIT_NOFILE));
}
EXPORT_SYMBOL(get_unused_fd_flags);
```

Функция **__alloc_fd()** находит для процесса свободный файловый дескриптор открытого файла и помечает его как занятый (это видно из следующего комментария. Эти действия выполняются в режиме монопольного доступа с использованием спин-блокировок. Если файловый дескриптор найден, то вызывается функция **__set_open_fd**(), которая помечает его как занятый.

```
/*
 * allocate a file descriptor, mark it busy.
 */
int __alloc_fd(struct files_struct *files,
               unsigned start, unsigned end, unsigned flags)
{
        unsigned int fd;
        int error;
        struct fdtable *fdt;

        spin_lock(&files->file_lock);/* захват спин-блокировки */
repeat:
        fdt = files_fdtable(files);
        fd = start;
        if (fd < files->next_fd)
                fd = files->next_fd;

        if (fd < fdt->max_fds)
                fd = find_next_fd(fdt, fd);

/*
 * N.B. For clone tasks sharing a files structure, this test will limit the total number of files that can be opened.
 */
        error = -EMFILE;
        if (fd >= end)
```

```
                goto out;
        error = expand_files(files, fd);
        if (error < 0)
                goto out;

/*
 * If we needed to expand the fs array we might have blocked - try again.
 */
        if (error)
                goto repeat;

        if (start <= files->next_fd)
                files->next_fd = fd + 1;

        __set_open_fd(fd, fdt);/*помечает дескриптор как занятый*/
        if (flags & O_CLOEXEC)
                __set_close_on_exec(fd, fdt);
        else
                __clear_close_on_exec(fd, fdt);
        error = fd;
#if 1
        /* Sanity check – санитарная проверка */
if (rcu_access_pointer(fdt->fd[fd]) != NULL) {
    printk(KERN_WARNING "alloc_fd: slot %d not NULL!\n", fd);
    rcu_assign_pointer(fdt->fd[fd], NULL);
}
#endif

out:
   spin_unlock(&files->file_lock);/*освобождение спин-блокировки*/
  return error;
}
```

Как видно из кода функции **do_sys_openat2()**, если выполняется условие fd >0, что означает – файловый дескриптор получен, вызывается функция do_filp_open(), которая возвращает указатель на struct **file**, т.е. создается дескриптор открытого файла в системной таблице открытых файлов.

Функция в качестве параметра получает дескриптор файла процесса, путь к файлу[2] (struct **filename** *pathname) и флаги.

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                const struct open_flags *op)
{
        struct nameidata nd;
        int flags = op->lookup_flags;
        struct file *filp;

        set_nameidata(&nd, dfd, pathname);
        filp = path_openat(&nd, op, flags | LOOKUP_RCU);
        if (unlikely(filp == ERR_PTR(-ECHILD)))
                filp = path_openat(&nd, op, flags);
        if (unlikely(filp == ERR_PTR(-ESTALE)))
                filp = path_openat(&nd, op, flags | LOOKUP_REVAL);
        restore_nameidata();
        return filp;
```

[2] Значение AT_FDCWD означает открытие файла в текущей директории.

```
}
```

В функции используется структура struct nameidata:

```
struct nameidata {
        struct path   path;/*путь к файлу*/
        struct qstr   last;
        struct path   root;/* корневая директория */
        struct inode  *inode; /* path.dentry.d_inode */
        unsigned int  flags;  /* флаги */
        unsigned      seq, m_seq, r_seq;
        int           last_type;
        unsigned      depth;
        int           total_link_count;
        struct saved {
                struct path link;
                struct delayed_call done;
                const char *name;
                unsigned seq;
        } *stack, internal[EMBEDDED_LEVELS];
        struct filename *name; /* указатель на структуру filename */
        struct nameidata *saved;
        struct inode  *link_inode; /* связанный inode */
        unsigned      root_seq;
        int           dfd;
} __randomize_layout;
```

Функция set_nameidata() инициализирует структуру nameidata.

```
static void set_nameidata(struct nameidata *p, int dfd, struct
filename *name)
{
        struct nameidata *old = current->nameidata;
        p->stack = p->internal;
        p->dfd = dfd;
        p->name = name;
        p->total_link_count = old ? old->total_link_count : 0;
        p->saved = old;
        current->nameidata = p;
}
```

В последней строке структура записывается в контекст текущего процесса. Затем вызывается функция path_openat(), которая возвращает указатель на struct file, т.е. на дескриптор открытого файла в системной таблице открытых файлов.

После инициализации структуры nameidata в функции **set_nameidata()** вызывается функция path_openat(). Функция path_openat выполняет поиск пути или, другими словами, пытается найти элемент каталога - dentry (directory entry) соответствующего пути, который ядро Linux использует для отслеживания иерархии файлов в каталогах, и инициализирует поля структуры struct file. Функция вызывается по следующим условиям:

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);

Как правила, ядро Linux открывает файл в режиме RCU. Это самый эффективный способ открыть файл. Если параметр 'rcu_walk' равен true, то вызывающий абонент выполняет обход по пути в режиме RCU-walk. RCU-walk - это алгоритм выполнения поиска по пути в Linux. Существенным отличием RCU-обхода является то, что он допускает возможность одновременного доступа. Четкое разграничение ролей. Самый простой способ управления параллелизмом - принудительно остановить любой другой поток, чтобы не допустить изменения структур данных, в которых заинтересован данный поток.

Если в режиме rcu-walk файловая система должна изменить dentry, то изменения сохранения в dentry, d_parent и d_inode не должны выполняться без взаимоисключения. Если возникает ситуация, которую rcu-walk не может обработать, возвращается - ECHILD, и будет выполнен вызов в режиме ref-walk.

Если попытка открыть файл в режиме RCU-walk не удастся, ядро перейдет в нормальный режим.

Третий вызов относительно редок, вероятно, будет использоваться только в файловой системе NFS. Одним из изменений, которые были недавно объединены для Linux 4.2, является существенное переписывание частей кода поиска пути на уровне виртуальной файловой системы Linux (VFS). Это переписывание в первую очередь влияет на обработку символических ссылок - хотя, как и многие такие переписывания, были найдены возможности для рационализации и улучшения других частей кода. ... Когда это происходит, поиск по всему пути прерывается и повторяется с установленным флагом LOOKUP_REVAL. Это заставляет повторную проверку быть более тщательной.

```
static struct file *path_openat(struct nameidata *nd,
                      const struct open_flags *op, unsigned flags)
{
  struct file *file;
  int error;

  file = alloc_empty_file(op->open_flag, current_cred());
   if (IS_ERR(file))
       return file;

   if (unlikely(file->f_flags & __O_TMPFILE)) {
       error = do_tmpfile(nd, flags, op, file);
   } else if (unlikely(file->f_flags & O_PATH)) {
           error = do_o_path(nd, flags, file);
           } else {
              const char *s = path_init(nd, flags);
              while (!(error = link_path_walk(s, nd)) &&
                 (error = do_last(nd, file, op)) > 0) {
              nd->flags &= ~(LOOKUP_OPEN|LOOKUP_CREATE|LOOKUP_EXCL);
                s = trailing_symlink(nd);
              }
              terminate_walk(nd);
       }
       if (likely(!error)) {
              if (likely(file->f_mode & FMODE_OPENED))
                     return file;
              WARN_ON(1);
              error = -EINVAL;
       }
       fput(file);
       if (error == -EOPENSTALE) {
              if (flags & LOOKUP_RCU)
```

```
                        error = -ECHILD;
                else
                        error = -ESTALE;
        }
        return ERR_PTR(error);
}
```

Функция **path_openat()** начинается с вызова функции **alloc_empty_file** (), которая выделяет новую файловую структуру с некоторыми дополнительными проверками, например, превышаем ли мы количество открытых файлов в системе или нет и т. д.

```
/* Find an unused file structure and return a pointer to it. Returns an error pointer if some error happend e.g. we
over file structures limit, run out of memory or operation is not permitted.
 *
 * Be very careful using this.  You are responsible for getting write access to any mount that you might assign to this
filp, if it is opened for write.  If this is not done, you will imbalance int the mount's writer count and a warning at
__fput() time.
```

Найдите неиспользуемую файловую структуру и верните указатель на нее. Возвращает указатель ошибки, если произошла какая-либо ошибка, например мы превышаем файловые структуры, не хватает памяти или операции не допускается.

*

* Будьте очень осторожны, используя это. Вы несете ответственность за получение доступа на запись к любому монтированию, которое вы можете назначить этому filp, если он открыт для записи. Если этого не сделать, вы нарушите баланс счетчика писателей монтирования и получите предупреждение во время __fput ().

```
 */
struct file *alloc_empty_file(int flags, const struct cred *cred)
{
        static long old_max;
        struct file *f;

        /*
         * Privileged users can go above max_files
         */
   if (get_nr_files() >= files_stat.max_files &&
                   capable(CAP_SYS_ADMIN))
    {
      /*
         * percpu_counters are inaccurate. Do an expensive check before
         * we go and fail.
```

*счетчики, связанные с определенным процессором (percpu_counters), неточны.
*нужно сделать «дорогую» проверку, прежде чем мы пойдем и
*потерпим неудачу.

```
       */
        if (percpu_counter_sum_positive(&nr_files) >=
                files_stat.max_files)
                goto over;
    }

        f = __alloc_file(flags, cred);
        if (!IS_ERR(f))
                percpu_counter_inc(&nr_files);

        return f;

over:
        /* Ran out of filps - report that */
        if (get_nr_files() > old_max) {
```

```
                pr_info("VFS: file-max limit %lu reached\n",
get_max_files());
                old_max = get_nr_files();
        }
        return ERR_PTR(-ENFILE);
}
```

После того, как была выделена новая структура для открытого файла struct file, вызываются функции **do_tmpfile** или **do_o_path** в случае если были установлены флаги **O_TMPFILE**| **O_CREATE** или **O_PATH** при вызове системного вызова **open():**

если (маловероятно (file->f_flags&**__O_TMPFILE**)

error = do_tmpfile (nd, flags, op, file)

иначе если (маловероятно (file->flags&**O_PATH**)

error = do_o_path (nd, flags, file)

иначе …

```
else {
        const char *s = path_init(nd, flags);
        while (!(error = link_path_walk(s, nd)) &&
                   (error = do_last(nd, file, op)) > 0)
         {
           nd->flags &= ~(LOOKUP_OPEN|LOOKUP_CREATE|LOOKUP_EXCL);
           s = trailing_symlink(nd);
         }
                terminate_walk(nd);
      }
```

Оба эти случая довольно специфичны, поэтому рассмотрим довольно обычный случай, когда нужно открыть уже существующий файл для чтения и записи.

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.

После выполнения функции **path_init()** выполняется цикл, в котором вызываются функции **link_path_walk()** и **do_last()**. Первая функция выполняет определение положения имен (элемента каталога) в дереве каталогов или, другими словами, эта функция запускает процесс прохода по заданному пути. Он обрабатывает pathname шаг за шагом, кроме последнего компонента пути к файлу. Эта обработка включает в себя проверку прав доступа и получение **файлового компонента**. При получении файлового компонента он передается в walk_component, который обновляет текущую запись каталога из кэша dentry (**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, полученного с помощью функции link_path_walk().

```
/*
 * Handle the last step of open()
 */
static int do_last(struct nameidata *nd,
                   struct file *file, const struct open_flags *op)
{
```

```
struct dentry *dir = nd->path.dentry;
kuid_t dir_uid = nd->inode->i_uid;
umode_t dir_mode = nd->inode->i_mode;
int open_flag = op->open_flag;
bool will_truncate = (open_flag & O_TRUNC) != 0;
bool got_write = false;
int acc_mode = op->acc_mode;
unsigned seq;
struct inode *inode;
struct path path;
int error;

nd->flags &= ~LOOKUP_PARENT;
nd->flags |= op->intent;

if (nd->last_type != LAST_NORM) {
        error = handle_dots(nd, nd->last_type);
        if (unlikely(error))
                return error;
        goto finish_open;
}

if (!(open_flag & O_CREAT)) {
        if (nd->last.name[nd->last.len])
                nd->flags |= LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
        /* we _can_ be in RCU mode here */
        error = lookup_fast(nd, &path, &inode, &seq);
        if (likely(error > 0))
                goto finish_lookup;

        if (error < 0)
                return error;

        BUG_ON(nd->inode != dir->d_inode);
        BUG_ON(nd->flags & LOOKUP_RCU);
} else {
        /* create side of things */
         * This will *only* deal with leaving RCU mode - LOOKUP_JUMPED
         * has been cleared when we got to the last component we are
         * about to look up
        * Это будет * только * иметь дело с выходом из режима RCU - LOOKUP_JUMPED
        был очищен, когда достигается последнего компонента, который ищется
        */
        error = complete_walk(nd); /* заканчивать проход */
        if (error)
                return error;

        audit_inode(nd->name, dir, AUDIT_INODE_PARENT); /* проверка inode */
        /* trailing slashes?  косые черты */
        if (unlikely(nd->last.name[nd->last.len]))
                return -EISDIR;
}

if (open_flag & (O_CREAT | O_TRUNC | O_WRONLY | O_RDWR)) {
        error = mnt_want_write(nd->path.mnt);
        if (!error)
                got_write = true;
        /*
         * do _not_ fail yet - we might not need that or fail with
         * a different error; let lookup_open() decide; we'll be
         * dropping this one anyway.
          пока не произойдет ошибка - это может быть не нужно или возникнет
          другая ошибка; пусть lookup_open() решит; мы все равно бросим это.
```

```
                */
        }
        if (open_flag & O_CREAT)
                inode_lock(dir->d_inode);
        else
                inode_lock_shared(dir->d_inode);
        error = lookup_open(nd, &path, file, op, got_write);
        if (open_flag & O_CREAT)
                inode_unlock(dir->d_inode);
        else
                inode_unlock_shared(dir->d_inode);

        if (error)
                goto out;

        if (file->f_mode & FMODE_OPENED) {
                if ((file->f_mode & FMODE_CREATED) ||
                   !S_ISREG(file_inode(file)->i_mode))
                        will_truncate = false;

                audit_inode(nd->name, file->f_path.dentry, 0);
                goto opened;
        }

        if (file->f_mode & FMODE_CREATED) {
                /* Don't check for write permission, don't truncate
                  Не проверять разрешение на запись, не обрезать */
                open_flag &= ~O_TRUNC;
                will_truncate = false;
                acc_mode = 0;
                path_to_nameidata(&path, nd);
                goto finish_open_created;
        }

        /*
         * If atomic_open() acquired write access it is dropped now due to
         * possible mount and symlink following (this might be optimized away if necessary...)
         Если atomic_open () получил доступ на запись, он теперь отбрасывается из-за возможного
         монтирования и последующей символической ссылки (это может быть оптимизировано при
         необходимости ...)
         */
        if (got_write) {
                mnt_drop_write(nd->path.mnt);
                got_write = false;
        }

        error = follow_managed(&path, nd);
        if (unlikely(error < 0))
                return error;

        /*
         * create/update audit record if it already exists.
         */
        audit_inode(nd->name, path.dentry, 0);

        if (unlikely((open_flag & (O_EXCL | O_CREAT)) == (O_EXCL | O_CREAT))) {
                path_to_nameidata(&path, nd);
                return -EEXIST;
        }

        seq = 0;    /* out of RCU mode, so the value doesn't matter */
        inode = d_backing_inode(path.dentry);
finish_lookup:
```

```
		error = step_into(nd, &path, 0, inode, seq);
		if (unlikely(error))
			return error;
finish_open:
		/* Why this, you ask?  _Now_ we might have grown LOOKUP_JUMPED... */
		error = complete_walk(nd);
		if (error)
			return error;
		audit_inode(nd->name, nd->path.dentry, 0);
		if (open_flag & O_CREAT) {
			error = -EISDIR;
			if (d_is_dir(nd->path.dentry))
				goto out;
			error = may_create_in_sticky(dir_mode, dir_uid,
						     d_backing_inode(nd->path.dentry));
			if (unlikely(error))
				goto out;
		}
		error = -ENOTDIR;
		if ((nd->flags & LOOKUP_DIRECTORY) && !d_can_lookup(nd->path.dentry))
			goto out;
		if (!d_is_reg(nd->path.dentry))
			will_truncate = false;

		if (will_truncate) {
			error = mnt_want_write(nd->path.mnt);
			if (error)
				goto out;
			got_write = true;
		}
finish_open_created:
		error = may_open(&nd->path, acc_mode, open_flag);
		if (error)
			goto out;
		BUG_ON(file->f_mode & FMODE_OPENED); /* once it's opened, it's opened */
		error = vfs_open(&nd->path, file);
		if (error)
			goto out;
opened:
		error = ima_file_check(file, op->acc_mode);
		if (!error && will_truncate)
			error = handle_truncate(file);
out:
		if (unlikely(error > 0)) {
			WARN_ON(1);
			error = -EINVAL;
		}
		if (got_write)
			mnt_drop_write(nd->path.mnt);
		return error;
}
```

При выполнении условия:

```
	if (open_flag & O_CREAT)
			inode_lock(dir->d_inode);
		else
			inode_lock_shared(dir->d_inode);
	    error = lookup_open(nd, &path, file, op, got_write);
		if (open_flag & O_CREAT)
			inode_unlock(dir->d_inode);
		else
```

```
                inode_unlock_shared(dir->d_inode);
```

вызывается функция lookup_open(). Функция **lookup_open()** создает inode открываемого файла, заполняя struct file в соответствии со следующим алгоритмом:

1) если не установлен флаг O_CREAT, завершить работу функции;

2) если inode открываемого файла существует, сбросить флаг O_CREAT и завершить работу функции;

3) создать inode открываемого файла с помощью функции **may_o_create**, проверяя при этом права доступа.

Комментарий мануала:

```
/*
 * Look up and maybe create and open the last component.
 *
 * Must be called with parent locked (exclusive in O_CREAT case).
 *
   * Returns 0 on success, that is, if  the file was successfully atomically created (if necessary) and
  opened,   or  the file was not completely opened at this time, though lookups and
 *  creations were performed.
 * These case are distinguished by presence of FMODE_OPENED on file->f_mode.
 * In the latter case dentry returned in @path might be negative if O_CREAT hadn't been specified.
 *
 * An error code is returned on failure.
* Посмотрите и, возможно, создайте и откройте последний компонент.
  *
  * Должен вызываться с заблокированным родителем (исключение в случае O_CREAT).
  *
  * Возвращает 0 в случае успеха, то есть, если файл был успешно атомарно создан (при
необходимости) и открыт, или файл не был полностью открыт в это время, хотя поиск и создание
были выполнены.
  * Эти случаи отличаются наличием FMODE_OPENED в файле-> f_mode.
  * В последнем случае dentry, возвращаемый в @path, может быть отрицательным, если O_CREAT не
был указан.
  *
  * Код ошибки возвращается при ошибке.
 */
```

Обратите внимание, если установлен флаг **O_CREATE**, то это может означать необходимость создания **struct inode.** При этом в ядре обеспечивается монопольный доступ, который обеспечивается **rw_semafore (семафор чтения-записи).** Монопольный доступ на запись реализуется вызовом inode_lock(), а на чтение – inode_lock_shared().

```
static int lookup_open(struct nameidata *nd, struct path *path,
                                        struct file *file,
                                        const struct open_flags *op,
                                        bool got_write)
{
        struct dentry *dir = nd->path.dentry;
        struct inode *dir_inode = dir->d_inode;
        int open_flag = op->open_flag;
        struct dentry *dentry;
        int error, create_error = 0;
        umode_t mode = op->mode;
        DECLARE_WAIT_QUEUE_HEAD_ONSTACK(wq);

        if (unlikely(IS_DEADDIR(dir_inode)))
                return -ENOENT;

        file->f_mode &= ~FMODE_CREATED;
```

```
	dentry = d_lookup(dir, &nd->last);
	for (;;) {
		if (!dentry) {
			dentry = d_alloc_parallel(dir, &nd->last, &wq);
			if (IS_ERR(dentry))
				return PTR_ERR(dentry);
		}
		if (d_in_lookup(dentry))
			break;

		error = d_revalidate(dentry, nd->flags);
		if (likely(error > 0))
			break;
		if (error)
			goto out_dput;
		d_invalidate(dentry);
		dput(dentry);
		dentry = NULL;
	}
	if (dentry->d_inode) {
		/* Cached positive dentry: will open in f_op->open */
		goto out_no_open;
	}

	/*
 * Checking write permission is tricky, bacuse we don't know if we are going to actually need it: O_CREAT opens should work as long as the file exists. But checking existence breaks atomicity. The trick is to check access and if not granted clear O_CREAT from the flags.
 *
 * Another problem is returing the "right" error value (e.g. for an O_EXCL open we want to return EEXIST not EROFS).
Проверить разрешение на запись сложно, потому что мы не знаем, действительно ли нам это нужно: открытия O_CREAT должны работать, пока файл существует. Но проверка существования нарушает атомарность. Хитрость в том, чтобы проверить доступ и, если не предоставлено, очистить O_CREAT от флагов.
 *
 * Другая проблема заключается в получении «правильного» значения ошибки (например, для открытия O_EXCL мы хотим вернуть EEXIST, а не EROFS).
 */
	if (open_flag & O_CREAT) {
		if (!IS_POSIXACL(dir->d_inode))
			mode &= ~current_umask();
		if (unlikely(!got_write)) {
			create_error = -EROFS;
			open_flag &= ~O_CREAT;
			if (open_flag & (O_EXCL | O_TRUNC))
				goto no_open;
			/* No side effects, safe to clear O_CREAT */
		} else {
			create_error = may_o_create(&nd->path, dentry, mode);
			if (create_error) {
				open_flag &= ~O_CREAT;
				if (open_flag & O_EXCL)
					goto no_open;
			}
		}
	} else if ((open_flag & (O_TRUNC|O_WRONLY|O_RDWR)) &&
		   unlikely(!got_write)) {
/*
 * No O_CREATE -> atomicity not a requirement -> fall back to lookup + open
Нет O_CREATE -> атомарность не обязательна -> вернуться к поиску + open
 */
		goto no_open;
	}

	if (dir_inode->i_op->atomic_open) {
		error = atomic_open(nd, dentry, path, file, op, open_flag,
				    mode);
		if (unlikely(error == -ENOENT) && create_error)
			error = create_error;
		return error;
	}
```

```
no_open:
        if (d_in_lookup(dentry)) {
                struct dentry *res = dir_inode->i_op->lookup(dir_inode, dentry,
                                                             nd->flags);
                d_lookup_done(dentry);
                if (unlikely(res)) {
                        if (IS_ERR(res)) {
                                error = PTR_ERR(res);
                                goto out_dput;
                        }
                        dput(dentry);
                        dentry = res;
                }
        }

        /* Negative dentry, just create the file */
        if (!dentry->d_inode && (open_flag & O_CREAT)) {
                file->f_mode |= FMODE_CREATED;
                audit_inode_child(dir_inode, dentry, AUDIT_TYPE_CHILD_CREATE);
                if (!dir_inode->i_op->create) {
                        error = -EACCES;
                        goto out_dput;
                }
                error = dir_inode->i_op->create(dir_inode, dentry, mode,
                                                open_flag & O_EXCL);
                if (error)
                        goto out_dput;
                fsnotify_create(dir_inode, dentry);
        }
        if (unlikely(create_error) && !dentry->d_inode) {
                error = create_error;
                goto out_dput;
        }
out_no_open:
        path->dentry = dentry;
        path->mnt = nd->path.mnt;
        return 0;

out_dput:
        dput(dentry);
        return error;
}

static int may_o_create(const struct path *dir, struct dentry *dentry, umode_t mode)
{
        struct user_namespace *s_user_ns;
        int error = security_path_mknod(dir, dentry, mode, 0);
        if (error)
                return error;

        s_user_ns = dir->dentry->d_sb->s_user_ns;
        if (!kuid_has_mapping(s_user_ns, current_fsuid()) ||
            !kgid_has_mapping(s_user_ns, current_fsgid()))
                return -EOVERFLOW;

        error = inode_permission(dir->dentry->d_inode, MAY_WRITE | MAY_EXEC);
        if (error)
                return error;

        return security_inode_create(dir->dentry->d_inode, dentry, mode);
}
```

При достижении последнего компонента данного пути к файлу, из функции **do_last()** вызывается **функция vfs_open()**. Открытие файла выполняется с помощью функции функции **vfs_open**, которая вызывается в функции **do_last**(). Функция **vfs_open**()

является оберткой над функцией **do_dentry_open**(), в которой вызывается **security_file_open**(), которая и создает файл.

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

```
/**
 * vfs_open - open the file at the given path
 * @path: path to open
 * @file: newly allocated file with f_flag initialized
 * @cred: credentials to use
vfs_open - открыть файл по указанному пути
 * @path: путь к открытию
 * @file: вновь выделенный файл с инициализированным f_flag
 * @cred: учетные данные для использования
 */
int vfs_open(const struct path *path, struct file *file)
{
        file->f_path = *path;
        return do_dentry_open(file, d_backing_inode(path->dentry), NULL);
}
```

Вернемся к функции **do_sys_open**. После ошибочного открытия файла выполняется освобождение файлового дескриптора (функция **put_unused_fd**) и возврат ошибки. В случае успешного открытия файла - уведомление файловой системы об открытии файла (функция **fsnotify_open**), запись дескриптора открытого файла в таблицу открытых файлов процесса (функция **fd_install**), вызвавшего системный вызов open и возврат дескриптора открытого файла.

Задание на лабораторную работу

Построить схему выполнения системного вызова open() в зависимости от значения основных флагов определяющих открытие файла на чтение, на запись, на выполнение и на создание нового файла. В схеме должны быть названия функций и кратко указаны выполняемые ими действия. По ГОСТу это делается с помощью выносных линий в фигурных скобках.

В схему нужно обязательно включить следующие действия, выполняемые соответствующими функциями ядра:

1) копирование названия файла из пространства пользователя в пространство ядра;
2) блокировка/разблокировка (spinlock) структуры files_struct и других действий в разных функциях;
3) алгоритм поиска свободного дескриптора открытого файла;
4) работу со структурой nameidata – инициализация ее полей;
5) алгоритм разбора пути (кратко);
6) инициализацию полей struct file;
7) «открытие» файла для чтения, записи или выполнения;
8) создание inode в случае отсутствия открываемого файла.

Отчет должен включать: титульный лист и схему алгоритма работы системного вызова open().

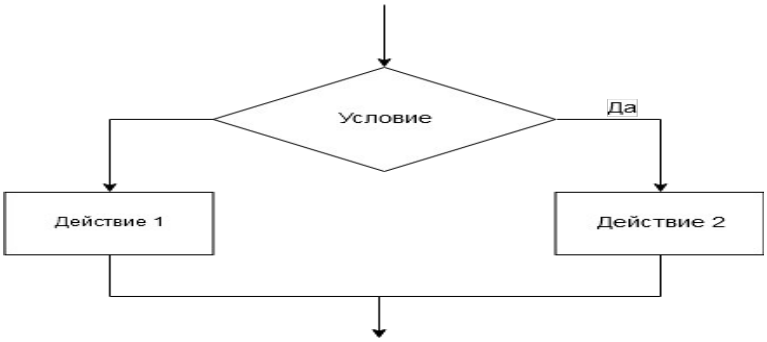
Список использованных источников

1. How does the open system call work

   https://www.bookstack.cn/read/linux-insides/SysCall-linux-syscall-5.md
2. man7.org > Linux > man-pages http://man7.org/linux/man-pages/man2/open.2.html
3. Программирование в Linux с нуля - Глава 5. НИЗКОУРОВНЕВЫЙ ВВОД-ВЫВОД https://www.linuxcenter.ru/lib/books/zlp/005.phtml?style=print

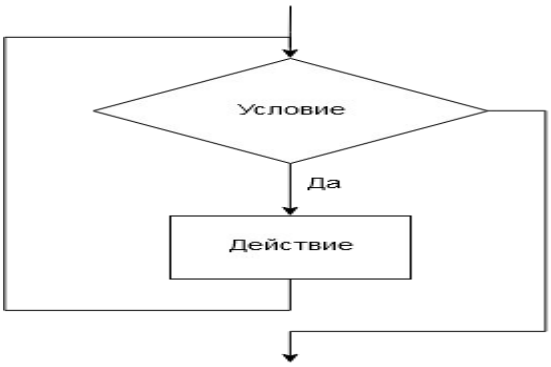
Памятка по оформлению схемы.

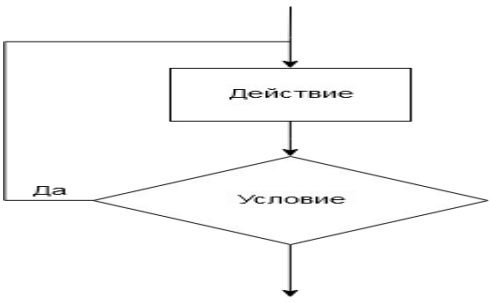
1. Оформление ветвление:



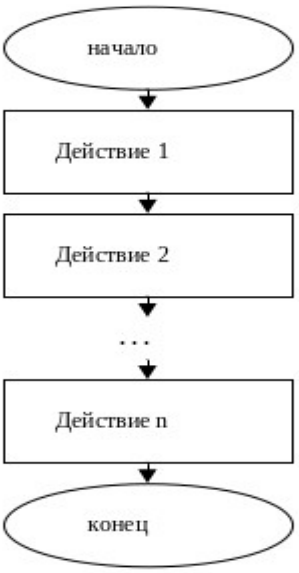
Все подписи должны быть или над стрелками, или справа от них. “НЕТ” не пишется.

2. Оформление повторения:





Все элементы алгоритма должны иметь примерно одинаковую ширину. Алгоритм начинается элементом «начало» и заканчивается элементом «конец». Алгоритм должен быть структурированным.



ГОСТ 19.701-90 (ИСО 5807-85) Единая система программной документации (ЕСПД). Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения http://docs.cntd.ru/document/9041994