

1. Билет №16

Открытые файлы: системный вызов `open()`: `int open(const char* pathname, int flags); int open(const char* pathname, int flags, mode_t mode)`; пояснить смысл параметров. Основные флаги. Флаг `CREATE`. Реализация системного вызова `open()` в системе – действия в ядре:

`SYSCALL_DEFINE3(open,...) -> ksys_open(filename, flags, mode) -> do_sys_open() -> do_sys_openat2() ...` найти наименьший файловый дескриптор... Действия, если флаг `O_CREATE` установлен?

1.1. Файл

Файл — важнейшее понятие в файловой подсистеме. Файл — информация, хранимая во вторичной памяти или во вспомогательном ЗУ с целью ее сохранения после завершения отдельного задания или преодоления ограничений, связанных в объеме основного ЗУ.

Файл — поименованная совокупность данных, хранимая во вторичной памяти (возможно даже целая). Файл — каждая индивидуально идентифицированная единица информации.

Существует 2 ипостаси файла:

1. файл, который лежит на диске;
2. открытый файл (с которым работает процесс).

Открытый файл – файл, который открывает процесс.

Для такого файла создается дескриптор файла в таблице открытых файлов процесса (`struct files_struct`).

Но этого мало.

Необходимо создать дескриптор открытого файла в сист. табл. открытых файлов (`struct file`)

1.2. Файловая подсистема

Файл != место на диске. В мире современной вычислительной техники файлы имеют настолько большие размеры, что не могут храниться в непрерывном физическом адресном пространстве, они хранятся вразброс (несвязанное распределение).

Файл может занимать разные блоки/сектора/дорожки на диске аналогично тому, как память поделена на страницы. В любой фрейм может быть загружена новая страница, как и файл.

Также, важно понимать адресацию. Соответственно, система должна обеспечить адресацию каждого такого участка.

Любая ОС без ФС не может быть полноценной. Задача ФС — обеспечивать сохранение данных и доступ к сохраненным данным (обеспечивать работу с файлами).

Чтобы обеспечить хранение файла и последующий доступ к нему, файл должен быть изолирован, то есть занимать некоторое адресное пространство, и это адресное пространство должно быть защищено. Доступ обеспечивается по тому, как файл идентифицируется в системе (доступ осуществляется по его имени).

ФС — порядок, определяющий способ организации хранения, именования и доступа к данным на вторичных носителях информации.

File management (управление файлами) — программные процессы, связанные с общим управлением файлами, то есть с размещением во вторичной памяти, контролем доступа к файлам, записью резервных копий, ведением справочников (directory).

Основные функции управления файлами обычно возлагаются на ОС, а дополнительные — на системы управления файлами.

Доступ к файлам: open, read, write, rename, delete, remove.

Разработка UNIX началась с ФС. Без ФС невозможно создание приложений, работающих в режиме пользователя (сложно разделить user mode и kernel mode).

Файловая подсистема взаимодействует практически со всеми модулями ОС, предоставляя пользователю возможность долговременного хранения данных, а также ОС возможность работать с объектами ядра.

1.2.1. struct file

Существует 2 типа файлов — файл, к-ый лежит на диске и открытый файл. Открытый файл — файл, который открывает процесс

Кратко

struct file описывает открытый файл.

Подробно

Если файл просто лежит на диске, то через дерево каталогов можно увидеть это.

Увидеть можно только подмонтированную ФС.

А есть открытые файлы — файлы, с которыми работают процессы.

Открыть файл может только процесс. Если файл открывается потоком, то он в итоге все равно открывается процессом (как ресурс). Ресурсами владеет процесс.

Таблицы открытых файлов

Помимо таблицы открытых файлов процесса (есть у каждого процесса), в системе есть одна таблица на все открытые файлы (на которую ссылаются таблицы процессов).

Причем в этой таблице на один и тот же файл (с одним и тем же inode) мб создано большое кол-во дескрипторов открытых файлов, т.к. один и тот же файл мб открыт много раз.

Каждое открытие файла с одним и тем же inode приведет к созданию дескриптора открытого файла.

При открытии файла его дескриптор добавляется:

1. в таблицу открытых файлов процесса (struct file_struct)
2. в системную таблицу открытых файлов

Каждый дескриптор struct file имеет поле f_pos. При работе с файлами это надо учитывать.

Один и тот же файл, открытый много раз без соотв. способов взаимоискл. будет атакован, что приведет к потере данных.

~~Гонки при разделении файлов — один и тот же файл мб открыт разными процессами.~~

Определение struct file

```
1  struct file {
2  struct path      f_path;
3  struct inode     *f_inode; /* cached value */
4  const struct file_operations *f_op;
5      ...
6  atomic_long_t     f_count; // колво — жестких ссылок
7  unsigned int      f_flags;
```

```

8   fmode_t      f_mode;
9   struct mutex   f_pos_lock;
10  loff_t        f_pos;
11  ...
12  struct address_space *f_mapping;
13  ...
14  };

```

Как осуществляется отображение файла на физ. страницы? - дескриптор открытого файла имеет указатель на inode (файл на диске).

Связь между **struct file** и **struct file operations**

Файл должен быть открыт. Соответственно для открытого файла должен быть создан дескриптор. В этом дескрипторе имеется указатель на **struct file_operations**. Это либо стандартные (установленные по умолчанию) операции на файлах для конкретной файловой системы, либо зарегистрированные разработчиком (собственные функции работы с файлами собственной файловой системы).

```

1   struct file_operations {
2   struct module *owner;
3   loff_t (*llseek) (struct file *, loff_t, int);
4   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5   ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6   ...
7   int (*open) (struct inode *, struct file *);
8   ...
9   int (*release) (struct inode *, struct file *);
10  ...
11 } __randomize_layout;

```

1.3. Системный вызов **open()**

Системный вызов **open()** открывает файл, определённый *pathname*.

Возвращаемое значение

open() возвращает файловый дескриптор — небольшое неотрицательное целое число, которое является ссылкой на запись в системной таблице открытых файлов и индексом записи в таблице дескрипторов открытых файлов процесса. Этот дескриптор используется далее в системных вызовах **read()**, **write()**, **lseek()**, **fcntl()** и т.д. для ссылки на открытый

файл. В случае успешного вызова будет возвращён наименьший файловый дескриптор, не связанный с открытым процессом файлом.

В случае ошибки возвращается -1 и устанавливается значение errno.

Параметры

pathname — имя файла в файловой системе. *flags* — режим открытия файла — один или несколько флагов открытия, объединенных оператором побитового ИЛИ.

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4
5 int open (const char *pathname, int flags);
6 int open (const char *pathname, int flags, mode_t mode);
```

2 варианта open():

1. Если ф-ция open предназначены для работы с существующим файлом, то это ф-ция вызывается с 2 параметрами.
2. Если пользователь желает создать файл и использует флаг O_CREATE или O_TMPFILE, то он должен указать 3-й пар-р — mode; Если эти флаги не указаны, то 3-й параметр игнорируется.

Так, можно открыть существующий файл, а можно открыть новый (создать) файл. Создать файл — создать inode.

1.4. Основные флаги. Флаг CREATE

O_CREAT

(если файл не существует, то он будет создан. Владелец (идентификатор пользователя) файла устанавливается в значение эффективного идентификатора пользователя процесса. Группа (идентификатор группы) устанавливается либо в значение эффективного идентификатора группы процесса, либо в значение идентификатора группы родительского каталога (зависит от типа файловой системы, параметров подключения (mount) и режима родительского каталога, см. например, параметры подключения bsdgroups и sysvgroups файловой системы ext2, как описано в руководстве mount(8)).);

O_EXCL

(Если он используется совместно с `O_CREAT`, то при наличии уже созданного файла вызов `open` завершится с ошибкой. В этом состоянии, при существующей символьной ссылке не обращается внимание, на что она указывает.);

`O_EXCL`

(Оно не работает в файловых системах NFS, а в программах, использующих этот флаг для блокировки, возникнет "race condition". Решение для атомарной блокировки файла: создать файл с уникальным именем в той же самой файловой системе (это имя может содержать, например, имя машины и идентификатор процесса), используя `link(2)`, чтобы создать ссылку на файл блокировки. Если `link()` возвращает значение 0, значит, блокировка была успешной. В противном случае используйте `stat(2)`, чтобы убедиться, что количество ссылок на уникальный файл возросло до двух. Это также означает, что блокировка была успешной);

`O_APPEND`

(Файл открывается в режиме добавления. Перед каждой операцией `write` файловый указатель будет устанавливаться в конце файла, как если бы использовался `lseek`); `O_APPEND` (может привести к повреждению файлов в системе NFS, если несколько процессов одновременно добавляют данные в один файл. Это происходит из-за того, что NFS не поддерживает добавление в файл данных, поэтому ядро на машине-клиенте должно эмулировать эту поддержку);

`O_RDONLY` - открыть файл только на чтение

`O_WRONLY` - открыть файл только на запись

`O_RDWR` - открыть файл для чтения и записи

`O_PATH` - получить лишь файловый дескриптор (сам файл не будет открыт). *Будет возвращен дескриптор `struct file` (он уже существует, мы его не создаем), при этом сам файл не открывается. Если флаг не установлен, то будет организован цикл по всем элементам пути и вызвана ф-ция `do_open`, которая открывает файл, т.е. создает дескриптор (инициализирует поля `struct file`).*

`O_TMPFILE` - создать неименованный временный обычный файл. Предполагает создание временного файла. Если он установлен, будет вызвана ф-ция `do_tmpfile`.

`O_APPEND` - установить смещение `f_pos` на конец файла.

`O_CREAT` - если файл не существует, то он будет создан, `id` владельца и группы файла устанавливаются действующим `id` пользователя и группы процесса.

Младшие 12 бит значения режима доступа к файлу устанавливаются равными значению аргумента `mode`, модифицированному следующим образом:

- Биты, соответствующие единичным битам маски режима создания файлов текущего процесса устанавливаются в 0

- Бит навязчивости устанавливается равным в 0

Если установлен флаг `O_CREAT` и указано несуществующее имя файла (система это контролирует), то д.б. создан `inode`.

`O_TRUNC` - если файл уже существует, он является обычным файлом и заданный режим позволяет записывать в этот файл, то его размер будет установлен в 0 (вся информация будет удалена). Режим доступа и владелец не меняются.

`O_EXCL` - если установлен `O_EXCL` и `O_CREAT` и указано имя существующего файла (файл уже существует), то `open` вернёт ошибку. Библиотечная функция `fopen` этого не делает.

`O_LARGEFILE` - позволяет открывать файлы, размер которых не может быть представлен типом `off_t` (`long`). Для установки должен быть указан макрос `_LARGEFILE64_SOURCE`.

`O_CLOEXEC` - устанавливает флаг *close-on-exec* для нового файлового дескриптора, указание этого флага позволяет программе избегать дополнительных операций `fcntl` `F_SETFD` для установки флага `FD_CLOEXEC`.

`O_EXEC` - открыть только для выполнения (результат не определен при открытии директории).

Режим (права доступа):

Если мы создаем новый файл, то мы должны указать права доступа к файлу.

Для режима предусмотрены константы (для пользователя/группы):

- `SR_IRWXU` / `SR_IRWXG` - права доступа на чтение, запись и исполнение

- `SR_IRUSR` / `SR_IRGRP` - права на чтение

- `SR_IWUSR` / `SR_IWGRP` - права на запись

- `SR_IXUSR` / `SR_IXGRP` - права на исполнение

1.5. Реализация системного вызова `open()` в системе — действия в ядре

`open()`, как и любой системный вызов переводит систему в режим ядра.

Сначала ищется свободный дескриптор в `struct files_struct` (в массиве дескрипторов открытых файлов процесса), потом при опр. усл-ях создается дескриптор открытого файла в системной таблице открытых файлов, затем при опр. усл-ях создается `inode`.

1.6. SYSCALL_DEFINE3(open,...)

В режиме ядра есть syscall table.

В системе есть 6 макросов – system call macro. У всех 1 параметр – имя сист. вызова.

С open() работает третий:

SYSCALL_DEFINE3(open, const char __user *filename, int flags, mode_t mode);

```
1  SYSCALL_DEFINE3(open, const char __user *filename, int flags, mode_t
    mode)
2  {
3  if (force_o_largefile())
4      flags |= O_LARGEFILE;
5  return do_sys_open(AT_FDCWD, filename, flags, mode)
6  }
```

filename – имя файла, которое передается из пространства пользователя в пр-во ядра. Это нельзя сделать напрямую. Впоследствии будет вызвана ф-ция str_copy_from_user() для передачи имени файла в ядро (это делается последовательно в результате ряда вызовов функций).

В макросе выполняется проверка того, какая у нас система: если 64-разр., то в ней есть большие файлы (largefile), и флаг O_LARGEFILE добавляется к флагам, к-ые были установлены.

Основная задача макроса – вызов ф-ции ядра do_sys_open()

1.7. do_sys_open()

```
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
7  if (fd)
8      return fd;
9
10 tmp = getname(filename);
11 if (IS_ERR(tmp))
```



```

12     return PTR_ERR(tmp);
13
14     fd = get_unused_fd_flags(flags); //обертка __alloc_fd()
15     if (fd >= 0) {
16         struct file *f = do_filp_open(dfd, tmp, &op);
17         if (IS_ERR(f)) {
18             put_unused_fd(fd);
19             fd = PTR_ERR(f);
20         } else {
21             fsnotify_open(f);
22             fd_install(fd, f);
23         }
24     }
25     putname(tmp);
26     return fd;
27 }

```

1.8. do_filp_open()

Основную работу по открытию файла и связанные с этим действия выполняет функция `do_filp_open()`.

struct filename и struct open_flags — эти структуры инициализированы в результате работы функций, которые были вызваны ранее

```

1     struct file *do_filp_open(int dfd, struct filename *pathname,
2     const struct open_flags *op)
3     {
4         struct nameidata nd; // внутренняя служебная структура
5         int flags = op->lookup_flags;
6         struct file *filp;
7
8         set_nameidata(&nd, dfd, pathname);
9         filp = path_openat(&nd, op, flags | LOOKUP_RCU);
10        if (unlikely(filp == ERR_PTR(-ECHILD)))
11            filp = path_openat(&nd, op, flags);
12        if (unlikely(filp == ERR_PTR(-ESTALE)))
13            filp = path_openat(&nd, op, flags | LOOKUP_REVAL);
14        restore_nameidata();

```

```
15 | return filp ;  
16 | }
```

1.9. build_open_flags()

Задачи ф-ции build_open_flags() – инициализация полей struct open_flags на основе флагов, указанных пользователем

В этой функции анализируются все флаги.

1.10. get_unused_fd_flags()

Можно предположить, что ф-ция get_unused_fd_flags должна найти неиспользуемый файловый дескриптор в таблице дескрипторов открытых файлов для того, чтобы выделить его, и open() мог его вернуть.

1.11. alloc_fd()

При этом ф-ция __alloc_fd() использует spin-lock'и, т.к. эти действия могут выполнять несколько процессов/потоков.

1.12. getname()

Ф-ция getname() вызывает get_name_flags(), которая копирует имя файла из пр-ва пользователя в пр-во ядра. При этом используется ф-ция str_copy_from_user().

Для любого процесса файловые дескрипторы 0, 1, 2 (stdin, stdout, stderr) занимаются автоматически, но для этих дескрипторов необходимо проделать все действия так же, как при вызове open() в приложении.

1.13. set_nameidata()

Ф-ция set_nameidata инициализирует поля struct nameidata.

1.14. path_openat()

Функция path_openat возвращает инициализированный дескриптор открытого файла (struct file)

1-й вызов

‘Быстрый проход’: игнорируются некоторые проверки. Это проход по всем флагам и выполнение соотв. анализа.

2-й вызов

‘Обычный’ (если быстрый проход вернул ошибку).

3-й вызов

Для файлов NFS (network filesystem).

В NFS не работает флаг O_APPEND, возникают гонки. O_APPEND позволяет дописывать данные в конец файла без потери данных в нем.

Гонки при разделении файлов — один и тот же файл мб открыт разными процессами.

```
1  static struct file *path_openat(struct nameidata *nd,
2      const struct open_flags *op, unsigned flags)
3  {
4      const char *s;
5      struct file *file;
6      int opened = 0;
7      int error;
8
9      file = get_empty_filp();
10     if (IS_ERR(file))
11         return file;
12
13     file->f_flags = op->open_flag;
14
15     if (unlikely(file->f_flags & __O_TMPFILE)) {
16         error = do_tmpfile(nd, flags, op, file, &opened);
17         goto out2;
18     }
19
20     if (unlikely(file->f_flags & O_PATH)) {
21         error = do_o_path(nd, flags, file);
22         if (!error)
23             opened |= FILE_OPENED;
```

```

24     goto out2;
25 }
26
27 s = path_init(nd, flags);
28 if (IS_ERR(s)) {
29     put_filp(file);
30     return ERR_CAST(s);
31 }
32 while (!(error = link_path_walk(s, nd)) &&
33     (error = do_last(nd, file, op, &opened)) > 0) {
34     nd->flags &= ~(LOOKUP_OPEN|LOOKUP_CREATE|LOOKUP_EXCL);
35     s = trailing_symlink(nd);
36     if (IS_ERR(s)) {
37         error = PTR_ERR(s);
38         break;
39     }
40 }
41 terminate_walk(nd);
42 out2:
43 if (!(opened & FILE_OPENED)) {
44     BUG_ON(!error);
45     put_filp(file);
46 }
47 if (unlikely(error)) {
48     if (error == -EOPENSTALE) {
49         if (flags & LOOKUP_RCU)
50             error = -ECHILD;
51         else
52             error = -ESTALE;
53     }
54     file = ERR_PTR(error);
55 }
56 return file;
57 }

```

Функции ядра специфицированы, но не стандартизованы (в отличие от сист. вызовов, которые стандартизованы POSIX). Поэтому функции и структуры ядра переписываются.

Для того, чтобы определить, существует ли файл, нужно пройти по цепочке *dentry*

(задействуется struct dentry).