

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... -> sys_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— жесткихссылок
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). В **write** стоит **buf** — означает, что **write** может записать разное количество байт.

```

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. Передача данных из пространства пользователя в пространство ядра и из ядра в пространство пользователя. Обоснование необходимости этих функций

Чтобы передать данные из адресного пространства пользователя в адресное пространство ядра и обратно используются функции `copy_from_user()` и `copy_to_user()`:

```
1 unsigned long __copy_to_user(void __user *to, const void *from, unsigned  
   long n);  
2 unsigned long __copy_from_user(void *to, const void __user *from,  
   unsigned long n);
```

Если некоторые данные не могут быть скопированы, эта функция добавит нулевые байты к скопированным данным до требуемого размера.

Обе функции возвращают количество байт, которые не могут быть скопированы. В случае выполнения будет возвращен 0.

Обоснование необходимости этих функций

Ядро работает с физическими адресами (адреса оперативной памяти), а у процессов адресное пространство виртуальное (это абстракция системы, создаваемая с помощью таблиц страниц).

Фреймы (физические страницы) выделяются по прерываниям.

Может оказаться, что буфер пространства пользователя, в который ядро пытается записать данные, выгружен.

И наоборот, когда приложение пытается передать данные в ядро, может произойти аналогичная ситуация.

Поэтому нужны специальные функции ядра, которые выполняют необходимые проверки.

Что можно передать из user в kernel?

Например, с помощью передачи из user mode выбрать режим работы загружаемого модуля ядра (какую информацию хотим получить из загружаемого модуля ядра в данный момент).

Такое "меню" надо писать в user mode и передавать соответствующие запросы модулям ядра.

1.4. Библиотечная функция `fopen()`

`fopen()` — это функция стандартной библиотеки `stdio.h` — библиотека буферизованного в/в.

```
1 FILE *fopen(const char *fname, const char *mode);
```

Функция `fopen()` открывает файл, имя которого указано аргументом `fname` и возвращает связанный с ним указатель. Тип операций, разрешенных над файлом, определяется аргументом `mode`.

`FILE` определена `define` над `IO_FILE` — описывает буферизированный ввод вывод, она описана в `stdio.h`. Ее связывает с дескриптором открытого файла: `fileno`

Если для `fopen()` указан `O_CREAT`, то если 2 раза вызвать так — данные затираются.

Возможно, стоит добавить, что функции библиотеки `stdio.h` могут работать с форматированными данными.

1.5. 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. Имя файла задается в `user mode`. *flags* — режим открытия файла — один или несколько флагов открытия, объединенных оператором побитового ИЛИ. *mode* — режим открытия файла.

```
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 функция `open` предназначена для работы с существующим файлом, то это функция вызывается с 2 параметрами.
2. Если пользователь желает создать файл и использует флаг `O_CREATE` или `O_TMPFILE`, то он должен указать 3-й параметр — `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`.

1.6. 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 подсоединения (`mount`) и режима родительского каталога, см. например, параметры подсоединения `bsdgroups` и `sysvgroups` файловой системы `ext2`, как описано в руководстве `mount(8)`.);

Реальный айди для процесса — который его запустил.

Эффективный айди для процесса — который установлен (может быть изменен, чтобы разрешить доступ не привилегированному пользователю).

`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.);

(Оно не работает в файловых системах `NFS`, а в программах, использующих этот флаг для создания файла блокировки (если хотим обеспечить создание процесса в единственном экземпляре), возникнет "race condition". Решение для атомарной блокировки файла: создать файл с уникальным именем в той же самой файловой системе.

`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 (этот флаг не работает в NFS, что приведет к гонкам, поэтому будет происходить затирание данных);

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_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_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 операцийfcntl F_SETFD для установки флага FD_CLOEXEC. Если close-on-exec установлен, то файловый дескриптор будет автоматически закрыт, когда любая функция из exec-семейства будет вызвана.

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):

- S_IRWXU / S_IRWXG - права доступа на чтение, запись и исполнение

- S_IRUSR / S_IRGRP - права на чтение

- S_IWUSR / S_IWGRP - права на запись

- S_IXUSR / S_IXGRP - права на исполнение

1.7. 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`.

1.8. `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()`

1.9. `do_sys_open()`

Создание и инициализация `struct`

`open_how` – проверка/установка флагов, режима, вызов `do_sys_openat2`.

```

1 long do_sys_open(int dfd , const char __user *filename , int flags , umode_t
    mode)
2 {
3     struct open_how how = build_open_how(flags , mode);
4     return do_sys_openat2(dfd , filename , &how);
5 }

```

1.10. do_sys_openat2()

Инициализация структуры open_flags, инициализация структуры filename, поиск свободного файлового дескриптора с пометкой его занятым. Открытие файла, инициализация полей структуры struct file. Если файл открыт, то уведомление ФС об открытии и запись дескриптора открытого файла в таблицу открытых файлов процесса.

```

1 static long do_sys_openat2(int dfd , const char __user *filename ,
2     struct open_how *how);

```

1.11. do_filp_open()

Основную работу по открытию файла и связанные с этим действия выполняет функция do_filp_open(). Функция осуществляет обход пути к файлу в 3-х возможных режимах:

1. с флагом LOOKUP_RCU — быстрый обход, допускается отсутствие некоторых проверок.
2. обычный обход, если быстрый обход вернул ошибку.
3. с флагом LOOKUP_REVAL — флаг (для работы с NFS) указывает, что необходимо выполнить повторную проверку (чтобы обеспечить принудительную повторную проверку записей, найденных в кеше). В NFS требуются дополнительные проверки, так как в ней не работает

O_APPEND, следовательно при разделении файлов возникают гонки, приводящие к потере данных.

struct filename и struct open_flags — эти структуры инициализированны 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);

```

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

1.13. 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. Вызывает alloc_fd().

```
1 int get_unused_fd_flags(unsigned flags);
```

1.14. alloc_fd()

При этом ф-ция __alloc_fd() использует spin-lock'и, т.к. эти действия могут выполнять несколько процессов/потоков. Ищет наименьший свободный файловый дескриптор с таблице открытых файлов процесса. Если не найден — расширяет таблицу. Помечает найденный файловый дескриптор занятым, устанавливает close-on-exec, возвращает найденный файловый дескриптор.

```
1 static int alloc_fd(unsigned start, unsigned end, unsigned flags);
```

1.15. getname()

Ф-ция getname() вызывает getname_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) занимают автоматически, но для этих дескрипторов необходимо проделать все действия так же, как при вызове open() в приложении.

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

1.16. set_nameidata()

Ф-ция set_nameidata инициализирует поля struct nameidata.

```
1  static inline void set_nameidata(struct nameidata *p, int dfd, struct  
    filename *name, const struct path *root);
```

1.17. restore_nameidata()

Ф-ция restore_nameidata восстанавливает структуру struct nameidata.

```
1  static void restore_nameidata(void);
```

1.18. path_openat()

Функция path_openat возвращает инициализированный дескриптор открытого файла (struct file)

```
1  static struct file *path_openat(struct nameidata *nd,  
2  const struct open_flags *op, unsigned flags);
```

1.19. open_last_lookups()

Функция open_last_lookups — разрешение пути при открытии файла (создание предшествующих директорий).

```
1  static const char *open_last_lookups(struct nameidata *nd,  
2  struct file *file, const struct open_flags *op);
```

1.20. lookup_open()

Функция lookup_open — создание директории, являющейся частью пути к открываемому файлу, если её нет в dentry cache. Создаются inode.

```
1  static struct dentry *lookup_open(struct nameidata *nd, struct file *  
    file, const struct open_flags *op, bool got_write);
```

1.21. do_open()

Функция `do_open` — проверяет флаги и открывает файл.

```
1  static int do_open(struct nameidata *nd,  
2  struct file *file, const struct open_flags *op);
```

1.22. may_open()

Функция `may_open` — проверяет, возможно ли открыть файл (права доступа, ...).

```
1  static int may_open(struct mnt_idmap *idmap, const struct path *path,  
2  int acc_mode, int flag);
```

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`).

Зогандки

Где видели `create_inode` из
`inode_operations`? — в `опене`, `lookup_open()`

Где видели семафоры `read/write`? — в `open_last_lookups()`:

```
1  if (open_flag & O_CREAT)  
2      inode_lock(dir->d_inode);  
3  else  
4      inode_lock_shared(dir->d_inode);  
5  dentry = lookup_open(nd, file, op, got_write);  
6  if (!IS_ERR(dentry) && (file->f_mode & FMODE_CREATED))  
7      fsnotify_create(dir->d_inode, dentry);  
8  if (open_flag & O_CREAT)  
9      inode_unlock(dir->d_inode);  
10 else  
11     inode_unlock_shared(dir->d_inode);
```

Реализация

```

1 static inline void inode_lock(struct inode *inode)
2 {
3     down_write(&inode->i_rwsem);
4 }
5 static inline void inode_unlock(struct inode *inode)
6 {
7     up_write(&inode->i_rwsem);
8 }
9 static inline void inode_lock_shared(struct inode *inode)
10 {
11     down_read(&inode->i_rwsem);
12 }
13 static inline void inode_unlock_shared(struct inode *inode)
14 {
15     up_read(&inode->i_rwsem);
16 }

```

LOOKUP_REVAL — флаг (для работы с NFS) указывает, что необходимо выполнить повторную проверку (чтобы обеспечить принудительную повторную проверку записей, найденных в кеше). В NFS требуются дополнительные проверки, так как в ней не работает O_APPEND, следовательно при разделении файлов возникают гонки, приводящие к потере данных.

Интересно, что сискал опена разбит на много функций!

Как проверяется существует ли файл? — по имени

Тонкий момент в alloc_fd? — если свободный ФД в таблице не найден, то создается новая таблица:

```

1 /*
2  * Expand files.
3  * This function will expand the file structures, if the requested size
4     exceeds
5  * the current capacity and there is room for expansion.
6  * Return <0 error code on error; 0 when nothing done; 1 when files were
7  * expanded and execution may have blocked.
8  * The files->file_lock should be held on entry, and will be held on exit.
9  */
10 static int expand_files(struct files_struct *files, unsigned int nr)

```

Давно когда-то файлы были маленькие, их было много, не было БД: 256 было достаточно. Сейчас есть большие файлы, соответственно количество меньше.

Что такое имя? — Каждый каталог и файл файловой системы имеет уникальное полное имя `full pathname` - имя, задающее полный путь от корня файловой системы через цепочку каталогов к соответствующему каталогу или файлу. Абсолютное имя содержит путь относительно корня `"/`. Короткое или относительное имя файла (`relative pathname`) - имя (возможно, составное), задающее путь к файлу от текущего рабочего каталога. Имя файла задается в `user mode`.