ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 12 — Управление открытыми файлами

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2023

2023.05.03

Оглавление

Открытие файла (повторение)

Чтобы начать работу с существующим файлом, его необходимо открыть.
С несуществующим — создать.
Операция открытия объявляет системе, что программа собирается работать с данным файлом.

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

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

Вызов **open()** используется, чтобы преобразовать путь к файлу в дескриптор файла — небольшое неотрицательно целое число, которое используется с вызовами **read()**, **write()** и т.п. при последующем вводе-выводе.

Дескриптор файла — это не индексный дескриптор, а объект данных ядра ОС.

Дескрипторы могут быть связаны не только с файлами, но и потоками произвольной природы.

Дескрипторы с номерами 0, 1, 2 обычно связаны со стандартными потоками ввода-вывода (**stdin**, **stdout**, **stderr**). На момент передачи управления пользовательскому коду программы обычно эти дескрипторы уже открыты. Тем не менее, их можно перенаправить.

Если системный вызов завершается успешно, возвращенный файловый описатель является наименьшим описателем, который еще не открыт процессом. В результате этого вызова появляется новый открытый файл, не используемый совместно никакими процессами (совместно используемые открытые файлы могут возникнуть, когда вызывается **fork(2)**).

Указатель устанавливается в начале файла.

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

Параметр **flags** — это битовые флаги, определяющие режим открытия файла.
Флаги собираются с помощью побитовой операции OR.

O_RDONLY — открывается в режиме «только для чтения»
O_WRONLY — открывается в режиме «только для записи»
O_RDWR — открывается в режиме «для чтения и записи»
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.

O_EXCL — если флаг 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 не работает в файловых системах NFS, а в программах, использующих этот флаг для блокировки, возникнет состояние гонки («race condition»).

O_NOCTTY — если **pathname** указывает на терминальное устройство, то оно не станет терминалом управления процесса, даже если процесс такового не имеет).

O_TRUNC — если файл уже существует, он является обычным файлом и режим позволяет записывать в этот файл (т.е. установлено **O_RDWR** или **O_WRONLY**), то его длина будет урезана до нуля.

Если файл является каналом FIFO или терминальным устройством, то этот флаг игнорируется. В остальных случаях действие флага **O_TRUNC** не определено.

O_APPEND — файл открывается в режиме добавления. Перед каждой операцией записи (**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_NONBLOCK или **O_NDELAY** — если возможно, то файл открывается в режиме **non-blocking**. Ни **open()**, ни другие последующие операции над возвращаемым дескриптором файла не заставляют вызывающий процесс ждать. Используется для работы с каналами FIFO.

O_SYNC — файл открывается в режиме синхронного ввода-вывода. Все вызовы **write()** для соответствующего описателя файла блокируют вызывающий процесс до тех пор, пока данные не будут физически записаны.

O_NOFOLLOW — если **pathname** — это символьная ссылка, то **open()** вернет код ошибки. Это расширение FreeBSD. Все прочие символьные ссылки в имени будут обработаны как обычно.

O_DIRECTORY — если **pathname** не является каталогом, то **open()** выдаст ошибку. Этот флаг используется только в Linux чтобы избежать проблем с «отказом от обслуживания» при вызове **opendir(2)** для канала FIFO или ленточного устройства.

O_CLOEXEC — устанавливает флаг **FD_CLOEXEC** (close-on-exec — закрыть файл при выполнении) для нового дескриптора файла.

Указание этого флага позволяет программе избегать дополнительных операций **fcntl(2)** **F_SETFD** для установки флага **FD_CLOEXEC**.

Использование этого флага необходимо в некоторых многопоточных программах. Использование отдельной операции **fcntl(2)** **F_SETFD** для установки флага **FD_CLOEXEC** недостаточно, чтобы избежать состояния гонки.

Когда один поток/нить открывает файловый дескриптор и пытается, используя **fcntl(2)**, установить ему флаг **FD_CLOEXEC** в то же время, когда другой поток выполняет **fork(2)** с последующим **execve(2)**, возникает состояние гонки. В зависимости от порядка выполнения, гонка может привести к непреднамеренной утечке файлового дескриптора, возвращаемого **open()**, в программу, выполняемую дочерним процессом, созданным **fork(2)**.

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

Символьные константы, используемые в mode

S_IRWXU 00700 владелец файла имеет права на чтение, запись и выполнение
S_IRUSR 00400 владелец имеет права на чтение файла
S_IWUSR 00200 владелец имеет права на запись в файл
S_IXUSR 00100 владелец имеет права на выполнение файла
S_IRWXG 00070 группа имеет права на чтение, запись и выполнение файла
S_IRGRP 00040 группа имеет права на чтение файла
S_IWGRP 00020 группа имеет права на запись в файл
S_IXGRP 00010 группа имеет права на выполнение файла
S_IRWXO 00007 все остальные имеют права на чтение, запись и выполнение
S_IROTH 00004 все остальные имеют права на чтение файла
S_IWOTH 00002 все остальные имеют права на запись в файл
S_IXOTH 00001 все остальные имеют права на выполнение файла

Бит чтения для всех типов файлов имеет одно и то же значение — он позволяет читать содержимое файла (получать листинг каталога командой **ls**).

Бит записи также имеет одно и то же значение — он позволяет писать в этот файл, включая и перезапись содержимого. Если у пользователя отсутствует право доступа на запись в каталоге, где находится данный файл, то пользователь не сможет его удалить. Также без этого же права пользователь не создаст новый файл в каталоге, хотя может сократить длину доступного на запись файла до нуля.

Бит выполнения. Если для некоторого файла установлен бит выполнения, то файл может выполняться как команда. В случае установки этого бита для каталога, этот каталог можно сделать текущим (перейти в него командой **cd**).

Эффект от установки остальных бит, согласно POSIX, не специфицируется.

В Linux используются еще три бита, упрощающие управление файлами и каталогами:
S_ISUID 0004000 — set-user-ID (SUID)
S_ISGID 0002000 — set-group-ID (SGID)
S_ISVTX 0001000 — sticky bit

Бит SUID (S_ISUID) — установленный бит **SUID** означает, что доступный пользователю на выполнение файл будет выполняться с правами (с эффективным идентификатором) владельца, а не пользователя, вызвавшего файл на исполнение, как это обычно происходит.

Бит SGID (S_ISGID) имеет несколько специальных применений.

1) Для каталога он указывает, что используется семантика BSD — файлы, создаваемые в каталоге, наследуют **ID** группы этого каталога, а не фактический **ID** группы создающего процесса. При этом для подкаталогов данного каталога также будет устанавливаться бит **SGID**.

2) Для исполняемого файла бит **SGID** заставляет изменить фактический ID группы процесса, который выполняет файл, согласно правилам, описанным в **execve(2)** — доступный пользователю на выполнение файл будет выполняться с правами (с эффективным идентификатором) группы-владельца, а не группы пользователя, вызвавшего файл на исполнение, как это обычно происходит.

3) Если файл не имеет бита выполнения группой **XGRP (S_IXGRP**), то бит **SGID** означает обязательную (mandatory) блокировку файла/записей, т.е. неизменность прав доступа на чтение и запись пока файл открыт определенной программой.

Клейкий (sticky) бит (S_ISVTX)

1) При установке обычным пользователем на файл этот бит сбрасывается. Значение этого бита при установке пользователем **root** зависит от версии ОС и иногда необходимо (ОС Solaris необходимо устанавливать клейкий бит для обычных файлов, используемых в качестве области подкачки).

2) Установка клейкого бита для каталога означает, что файл в этом каталоге может быть удален или переименован только пользователем-владельцем файла, пользователем-владельцем каталога, если файл доступен пользователю на запись и пользователем **root**.

Функция **fopen( )** стандартной библиотеки языка C (ISO/IEC 9899-2012) в общем случае предоставляет только два параметра при открытии/создании файла — имя файла и режим его открытия.

```
FILE *fopen(const char * restrict filename, // имя файла
            const char * restrict mode);    // режим открытия
```

Аргумент режима для файла при его создании отсутствует. Предполагается, что ОС установит права на файл в соответствии с политиками, принятыми по умолчанию.

Согласно стандарту аргумент **mode** указывает на строку. Если строка является одной из следующих, файл открывается в указанном режиме. В противном случае поведение не определено.

r открыть текстовый файл для чтения (**R**ead);
w обрезать до нулевой длины или создать текстовый файл для записи (**W**rite);
wx создать текстовый файл для записи в монопольном режиме (e**X**clusive);
a добавление; открыть или создать текстовый файл для записи в конце файла (**A**ppend);
rb открыть бинарный файл для чтения;
wb урезать до нулевой длины или создать двоичный файл для записи;
wbx создать двоичный файл для записи в монопольном режиме;
ab добавление; открыть или создать двоичный файл для записи в конце файла;
r+ открыть текстовый файл для обновления (чтение и запись);
w+ усечь до нулевой длины или создать текстовый файл для обновления;
w+x создать текстовый файл для обновления в монопольном режиме;
a+ открыть или создать текстовый файл для обновления и записи в конец файла;
r+b или **rb+** открыть двоичный файл для обновления (чтение и запись);
w+b или **wb+** усечь до нулевой длины или создать двоичный файл для обновления;
w+bx или **wb+x** создать двоичный файл для обновления в монопольном режиме;
a+b или **ab+** открыть или создать двоичный файл для обновления и записи в конец файла;

Каждой из вышеприведенных строк **mode** соответствует свой набор режимов дескриптора файла.

fcntl() — манипуляции с файловым дескриптором

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd,  // дескриптор файла, который возвратил вызов open()
          int cmd, // команда
          ...);    // аргументы команды, если это требуется
```

fcntl() позволяет выполнять различные команды над открытым файловым дескриптором **fd**.
Команда определяется содержимым аргумента **cmd**.
fcntl() может принимать необязательный третий аргумент **arg**.
Необходимость его указания зависит от значения, указанного в **cmd**.
Тип необходимого аргумента **arg** зависит от команды, в большинстве случаев требуется тип **int**, в некоторых случаях это будет указатель на структуру[1].
Системный вызов **fcntl(2)** позволяет выполнять следующие операции:
- создание дубликата файлового дескриптора;
- управление флагами файлового дескриптора;
- управление флагами состояния файла;
- совместная/рекомендательная (advisory) блокировка;
- обязательная (mandatory) блокировка;
- управление сигналами;
- аренда (лизинг, владение);
- уведомления об изменении файла и каталога;
- изменение ёмкости канала.

---

1) Далее тип **arg** указывается в скобках после каждого имени значения **cmd**, либо указывается **void**, если аргумент не требуется.

Некоторые из перечисленных операций поддерживаются только с определенной версии ядра.

Предпочтительный метод проверки того, поддерживает ли ядро хоста конкретную операцию, — вызвать **`fcntl()`** с желаемым значением **`cmd`** и затем проверить, завершился ли вызов с помощью **`EINVAL`**, указывая, что ядро не распознает это значение.

Управление флагами файлового дескриптора

С файловым дескриптором в настоящее время определён только один флаг **FD_CLOEXEC** — флаг **close-on-exec**. Он определяет что произойдет с дескриптором при вызове **exec().**

Если **FD_CLOEXEC** установлен, то файловый дескриптор будет закрыт, в противном случае (**FD_CLOEXEC** равен 0), файловый дескриптор при вызове **execve()** останется открытым.

С флагами файлового дескриптора работают следующие команды:

F_GETFD (void) — читает флаги файлового дескриптора, определенные в **<fcntl.h>**.

Флаги файловых дескрипторов связаны с одним файловым дескриптором и не влияют на другие файловые дескрипторы, которые ссылаются на тот же самый файл.

В частности, читает флаг **close-on-exec**.

F_SETFD (int) — устанавливает флаги файлового дескриптора согласно значению, указанному в аргументе **arg**.

Если флаг **FD_CLOEXEC** в третьем аргументе равен 0 (сброшен), дескриптор файла будет оставаться открытым при выполнении всех функций **exec**.

В противном случае дескриптор файла должен быть закрыт после успешного выполнения одной из функций **exec**.

Создание дубликата файлового дескриптора

F_DUPFD (int) — ищет наименьший доступный номер файлового дескриптора, который больше **reffd** и делает его копией дескриптора **fd**.

Фактически это форма вызова **dup2(2)** которая используется с явным указанием файлового дескриптора. Отличие от **dup2(2)** в том, что там дескриптор задаётся явно.

```
newfd = fcntl(fd,      // оригинальный fd
              F_DUPFD, // команда
              reffd);  // ссылочный fd
```

При успешном выполнении этой команды, возвращается новый файловый дескриптор, который будет ссылаться **на то же описание открытого файла, что и дескриптор исходного файла**, и будет совместно иметь с ним любые блокировки.

Флаг FD_CLOEXEC, связанный с новым дескриптором файла, будет сброшен, чтобы файл оставался открытым при вызовах функций класса exec.

F_DUPFD_CLOEXEC (int) — работает как и **F_DUPFD**, но на новом дескрипторе дополнительно устанавливается флаг **close-on-exec**.

Установка этого флага позволяет программам не делать дополнительный вызов **`fcntl()`** с командой **F_SETFD** для установки флага **FD_CLOEXEC**.

dup, dup2 — создать дубликат файлового дескриптора

```
#include <unistd.h>

int dup(int oldfd);  // оригинальный файловый дескриптор
int dup2(int oldfd,  // оригинальный файловый дескриптор
         int newfd); // новый файловый дескриптор
```

dup() — системный вызов **dup()** создает копию файлового дескриптора **oldfd**, используя для нового дескриптора неиспользуемый файловый дескриптор с наименьшим номером.

После успешного возврата старый и новый файловые дескрипторы могут использоваться как взаимозаменяемые. Они ссылаются на одно и то же описание открытого файла (**open(2)**) и совместно используют флаги смещения и состояния файла. Например, если смещение в файле изменяется с помощью **lseek(2)** для одного из файловых дескрипторов, смещение также изменится и для другого.

Два файловых дескриптора не имеют общих флагов файловых дескрипторов (флаг close-on-exec). **Флаг FD_CLOEXEC для копии дескриптора сбрасывается.**

Системный вызов **dup()** предоставляет альтернативный интерфейс к сервису, предоставляемому **fcntl()** с помощью команды **F_DUPFD**.

Вызов **dup(fd)** эквивалентен следующему:

```
fcntl(fd, F_DUPFD, 0);
```

dup2() — системный вызов **dup2()** выполняет то же, что и **dup()**, но вместо неиспользуемого файлового дескриптора с наименьшим номером будет взят номер файлового дескриптора, указанный в **newfd**.

Если файловый дескриптор **newfd** был ранее открыт, он автоматически закрывается перед повторным использованием.

Шаги закрытия и повторного использования файлового дескриптора **newfd** выполняются *атомарно*. Это важно, потому что попытка реализовать эквивалентную функциональность с помощью **close(2)** и **dup()** может привести к условиям возникновения гонки, в результате чего **newfd** между этими двумя вызовами может быть использован повторно.

Такое повторное использование может произойти из-за того, что основная программа может быть прервана обработчиком сигнала, который выделяет файловый дескриптор, или потому, что параллельный поток выделяет файловый дескриптор.

Дескриптор файла newfd ссылается на то же самое описание открытого файла, что и дескриптор файла oldfd, а также использует все его блокировки.

После успешного завершения, если **oldfd** не равно **newfd**, будет сброшен связанный с **newfd** флаг **FD_CLOEXEC**. Если **oldfd** равно **newfd**, связанный с **newfd** флаг **FD_CLOEXEC** не изменится.

Следует обратить внимание на следующие моменты:

- если **oldfd** не является допустимым файловым дескриптором, вызов завершается неудачно, и **newfd** не закрывается;

- если **oldfd** является допустимым дескриптором файла, а **newfd** имеет то же значение, что и **oldfd**, то **dup2()** ничего не делает и возвращает **newfd**.

В случае успеха эти системные вызовы возвращают новый дескриптор файла.

В случае ошибки возвращается **-1**, а значение **errno** устанавливается соответствующим образом.

Если **newfd** меньше 0 или больше или равно **OPEN_MAX**, **dup2()** вернет **-1** с **errno** == **EBADF**.

Ошибка, возвращаемая системным вызовом **dup2()**, отличается от ошибки, возвращаемой **fcntl(..., F_DUPFD, ...)**, если **newfd** находится вне допустимого диапазона.

Однако, в некоторых системах **dup2()** может возвращать **EINVAL**, как это происходит при использовании команды **F_DUPFD**.

Если **newfd** был открыт, то все ошибки, о которых было бы сообщено при вызове **close(2)**, теряются.

Если такое поведение неприемлемо, то в случае, когда программа является многопоточной и не выделяет файловые дескрипторы в обработчиках сигналов, правильным подходом будет не закрывать **newfd** перед вызовом **dup2()** из-за возможного возникновения состояния гонки, описанного выше. Вместо этого можно использовать следующий код:

```
// Получаем дубликат newfd, который впоследствии можно будет использовать для
// проверки ошибок close(); ошибка EBADF означает, что newfd не был открыт.
tmpfd = dup(newfd);
if (tmpfd == -1 && errno != EBADF) {
    // Обработка непредвиденной ошибки dup()
}

// Атомарное копирование oldfd в newfd
if (dup2(oldfd, newfd) == -1) {
    // Обработка ошибки dup2()
}

// Проверим наличие ошибок close() в файле, на который изначально ссылался
'newfd'
if (tmpfd != -1) {
    if (close(tmpfd) == -1) {
        // Обработка ошибки close()
    }
}
```

Управление флагами состояния файла

Каждый дескриптор открытого файла имеет несколько связанных с ним флагов создания и состояния, которые инициализируются вызовом **open(2)**.

Эти флаги совместно используются копиями файловых дескрипторов (созданными с помощью **dup(2)**, **fcntl(F_DUPFD)**, **fork(2)** и т.д.), которые указывают на одно и то же описание открытого файла.

Эти флаги состояния и их смысл описаны в **open(2)**.

Некоторые из этих флагов могут быть изменены вызовом **fcntl()**.

O_RDONLY — режим доступа по чтению
O_WRONLY — режим доступа по записи
O_RDWR — режим доступа по чтению и записи
O_CREAT — создается если не существует
O_EXCL — если файл существует и **O_CREAT**, **open( )** вернет ошибку
O_NOCTTY — если **fname** -> терминал, он не станет управляющим
O_TRUNC — если файл существует, будет урезан до 0

O_APPEND — файл открывается в режиме добавления
O_NONBLOCK — файл открывается в режиме non-blocking (FIFO)
O_SYNC — файл открывается в режиме синхронного ввода-вывода
O_NOFOLLOW — если **fname** символьная ссылка, **open( )** вернет ошибку
O_DIRECTORY — если **fname** не является каталогом, **open( )** вернет ошибку
O_LARGEFILE — позволяет открывать файлы, длина которых больше 31 бита
O_DIRECT — попытаться минимизировать влияние кэширования ввода-вывода
O_NOATIME — не обновлять время **st_atime** при вызове **read(2)** для файла

F_GETFL (void)

Получить права доступа к файлу и флаги состояния файла, определенные в **<fcntl.h>**.

Режимы доступа к файлу можно извлечь из возвращаемого значения с помощью маски **O_ACCMODE**, которая определена в **<fcntl.h>**.

Флаги состояния файла и режимы доступа к файлу связаны с *описанием* файла и не влияют на другие дескрипторы файлов, которые относятся к одному и тому же файлу с разными *описаниями* открытых файлов.

Возвращаемые флаги могут включать нестандартные флаги состояния файла, которые приложение не установило, при условии, что эти дополнительные флаги не изменяют поведение соответствующего приложения.

F_SETFL (int arg)

Установить флаги состояния файла, определенные в **<fcntl.h>**, согласно значению, указанному в третьем аргументе, взятом как тип **int**.

Биты, соответствующие режиму доступа к файлу и флагам создания файла, как они определены в **<fcntl.h>**, которые установлены в **arg**, должны игнорироваться. Если приложение изменяет какие-либо биты в **arg**, кроме упомянутых здесь, результат не определен. Если **fd** не поддерживает неблокирующие операции, то будет ли игнорироваться флаг **O_NONBLOCK**, не специфицируется.

Режимы доступа к файлу **O_RDONLY**, **O_WRONLY**, **O_RDWR** и флаги создания **O_CREAT**, **O_EXCL**, **O_NOCTTY**, **O_TRUNC**, указанные в **arg** игнорируются.

В Linux эта команда может изменять только флаги
O_APPEND
O_ASYNC
O_DIRECT
O_NOATIME
O_NONBLOCK

Консультативная (advisory) блокировка

Для установки, снятия и проверки существования блокировок записей (также известных как блокировки сегмента или области файла) используются команды **F_GETLK**, **F_SETLK** и **F_SETLKW**.

Третий аргумент, **lock**, является указателем на структуру, которая имеет, по крайней мере, следующие поля (в произвольном порядке):

```
struct flock {
    ...
    short l_type;   // Тип блокировки: F_RDLCK, F_WRLCK, F_UNLCK
    short l_whence; // Как интерпретировать l_start: SEEK_SET, SEEK_CUR, SEEK_END
    off_t l_start;  // Начальное смещение для блокировки (может быть < 0)
    off_t l_len;    // Количество блокируемых байт (если 0, то до конца файла)
    pid_t l_pid;    // PID процесса, установившего блокировку (только для F_GETLK)
    ...
};
```

Поля **l_whence**, **l_start** и **l_len** этой структуры задают диапазон байт, который мы хотим заблокировать.

Блокировки могут начинаться и выходить за пределы текущего конца файла, но не должны выходить за начало файла.

l_start — начальное смещение для блокировки, которое интерпретируется как:

- **SEEK_SET** — начало файла если значение;
- **SEEK_CUR** — текущая позиция в файле;
- **SEEK_END** — конец файла.

В случаях **SEEK_CUR**, и **SEEK_END** значение **l_start** может быть отрицательным при условии, что что смещение не находится перед началом файла.

l_len — количество байт, которые нужно заблокировать.

Если **l_len** > 0, то диапазон блокировки начинается с **l_start** и кончается **l_start + l_len - 1** включительно.

Если в **l_len** указан **0**, то блокируются все байты начиная с места, указанного **l_whence** и **l_start** и до конца файла, независимо от величины файла.

l_type — используется для указания типа устанавливаемой блокировки файла:

F_RDLCK — блокировка на чтение;
F_WRLCK — блокировка на запись;
F_UNLOCK — снятие блокировки.

Блокировку на чтение области файла (совместная блокировка) может удерживать любое количество процессов, но только один процесс может удерживать блокировку на запись (эксклюзивная блокировка).

Эксклюзивная блокировка исключает все другие блокировки, как общие так и эксклюзивные.

Один процесс может удерживать только один тип блокировки для области файла и если происходит новая блокировка уже заблокированной области (успешный возврат из запросов **F_SETLK** или **F_SETLKW**), то существующая блокировка будет преобразована в новый тип блокировки.

Если диапазон байтов, указанный новой блокировкой, не совпадает в точности с диапазоном существующей блокировки, то преобразования блокировки могут включать расщепление, сжатие или объединение с существующей блокировкой.

Запрос **F_SETLK** или **F_SETLKW** (соответственно) должен завершиться ошибкой или блокироваться, если другой процесс имеет существующие блокировки байтов в указанной области и тип любой из этих блокировок конфликтует с типом указанные в запросе.

Совместная блокировка (на чтение)

Если на сегменте файла установлена совместная блокировка, другие процессы на данном сегменте или его части могут тоже устанавливать совместные блокировки.

Совместная блокировка предотвращает установку исключительной (монопольной) блокировки любым другим процессом в любой части защищенной области.

Если файловый дескриптор не был открыт с доступом для чтения, запрос на совместную блокировку завершится ошибкой.

Исключительная блокировка (на запись)

Исключительная блокировка препятствует любому другому процессу устанавливать совместную блокировку или исключительную блокировку на любой части защищенной области.

Если файловый дескриптор не был открыт с доступом на запись, запрос на исключительную блокировку завершится ошибкой.

Команды блокировки

F_GETLK (struct flock *)

Получить информацию о возможности установить блокировку.

При вызове параметр **lock** должен описывать блокировку, которую мы хотели бы установить на файл.

Если блокировка, которая помешала бы созданию указанной в **lock** блокировки не обнаружена, **fcntl()** возвращает **F_UNLCK** в поле **l_type**, перезаписывая структуру **lock**.

Другие поля структуры останутся неизменёнными.

Если одна или более несовместимых блокировок мешают установке этой блокировки, то **fcntl()** возвращает подробности об одной из этих блокировок в полях **l_type**, **l_whence**, **l_start** и **l_len** структуры **lock** , а также присваивает **l_pid** значение PID того процесса, который удерживает блокировку.

Возвращенная информация может уже быть устаревшей к моменту ее проверки и использования

Поле **l_pid** структуры **flock** используется только с **F_GETLK** для возврата идентификатора процесса, удерживающего блокирующую блокировку.

После успешного запроса **F_GETLK**, когда препятствующая блокировка обнаружена, значения, возвращаемые в структуре **flock**, будут следующими:

l_type — тип обнаруженной блокирующей блокировки;
l_whence — **SEEK_SET**;
l_start — начало блокировки блокировки;
l_len — длина блокировки;
l_pid — ID процесса, удерживающего блокирующую блокировку.

F_SETLK (struct flock *)

Устанавливает или снимает блокировку в соответствии с типом операции, указанным в поле **l_type** третьего аргумента **lock** (**F_RDLCK**, **F_WRLCK**, **F_UNLCK**), на область байт, указанную полями **l_whence**, **l_start** и **l_len** там же (в структуре **lock**).

Если блокирующая блокировка удерживается другим процессом, то данный вызов вернёт **-1** и установит значение **errno** в **EACCES** или **EAGAIN**. Ошибка, возвращаемая в этом случае, может различаться в зависимости от реализации, поэтому POSIX требует, чтобы переносимое приложение проверяло обе ошибки.

F_SETLKW (struct flock *)

Команда эквивалентна **F_SETLK** за исключением того, что если совместная или исключительная блокировка конфликтует с другими блокировками, поток будет ждать, пока запрос не будет удовлетворен.

Если во время ожидания поступит сигнал, то данный вызов прерывается и после возврата из обработчика сигнала из вызова происходит немедленный возврат, при этом возвращается значение **-1** и **errno** устанавливается в **EINTR**.

В **<fcntl.h>** могут быть определены дополнительные значения **cmd**, определяемые реализацией. Их имена будут начинаться с **F_**.

Для того, чтобы установить блокировку на чтение, **fd** должен быть открыт на чтение.

Для того, чтобы установить блокировку на запись, **fd** должен быть открыт на запись.

Чтобы установить оба типа блокировки, **fd** должен быть открыт на запись и на чтение.

Все блокировки, связанные с файлом для данного процесса, автоматически снимаются, как только файловый дескриптор этого файла закрывается каким либо процессом, обладающим этим файловым дескриптором, или завершается процесс, обладающий этим файловым дескриптором.

Это плохо — процесс может потерять блокировки на файлах типа **/etc/passwd** или **/etc/mtab**, когда по какой-либо причине библиотечная функция решает их открыть, прочитать и закрыть.

Блокировки не наследуются потомком, созданным через **fork(2)**.

Блокировки сохраняются при вызове **execve(2)**, которая наследует все открытые файловые дескрипторы, на которых не было флага Close-On-Exec.

Взаимная блокировка (deadlock)

Если процесс, управляющий заблокированной областью, переводится в состояние ожидания, пытаясь заблокировать заблокированную область другого процесса, возникает опасность взаимоблокировки (deadlock, тупик). Если система обнаруживает, что уход процесса в состояние ожидания до тех пор, пока заблокированная область не будет разблокирована, вызовет взаимоблокировку, **fcntl()** завершится с ошибкой **EDEADLK**.

Запрос разблокировки (**F_UNLCK**), в котором **l_len** не равно нулю, а смещение последнего байта запрошенного сегмента является максимальным значением для объекта типа **off_t**, и при этом процесс имеет существующую блокировку, в которой **l_len** равно 0 и которая включает последний байт запрошенного сегмента, должен рассматриваться как запрос на разблокировку с начала запрошенного сегмента с **l_len**, равным 0. В противном случае запрос разблокировки (**F_UNLCK**) должен пытаться разблокировать только запрошенный сегмент.

Использование блокировок совместно с функциями из библиотеки **stdio(3)** нужно избегать из-за буферизации — вместо функций из **stdio(3)** следует использовать **read(2)** и **write(2)**.

Если файловый дескриптор **fd** относится к объекту совместно используемой памяти, поведение **fcntl()** будет таким же, как и для обычного файла, за исключением того, что действие следующих значений для аргумента **cmd** будет неопределено:

F_SETFL – установить флаги состояния файла
F_GETLK – проверить блокировку
F_SETLK – установить/снять блокировку
F_SETLKW – установить/снять блокировку

Пример блокировки и разблокировки файла

В следующем примере показано, как установить блокировку на байты файла от 100 до 109, а затем снять ее.

Чтобы процессу не приходилось ждать, когда несовместимая блокировка удерживается другим процессом, для выполнения запроса неблокирующей блокировки используется **F_SETLK** и вместо ожидания процесс может предпринять некоторые другие действия.

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    int          fd;
    struct flock fl;

    if ((fd = open("testfile", O_RDWR)) == -1) {
        /* Обработка ошибки */;
    }

    // Неблокирующий запрос, установки блокировки записи на байты 100-109
    fl.l_type   = F_WRLCK;  // запрос блокировки записи
    fl.l_start  = 100;      // с данного смещения
    fl.l_whence = SEEK_SET; // от начала файла
    fl.l_len    = 10;       // указанной длины
```

```
    if (fcntl(fd, F_SETLK, &fl) == -1) {
        if (errno == EACCES || errno == EAGAIN) { // обе ошибки, согласно POSIX
            // Получить блокировку в данный момент нет возможности
            printf("Уже заблокировано другим процессом\n");
        } else {
            // Обработка непредвиденной ошибки
        }
    } else { // Блокировка предоставлена

        // Выполняем ввод/вывод для байтов от 100 до 109
        // и разблокируем заблокированные байты

        fl.l_type   = F_UNLCK;
        fl.l_whence = SEEK_SET;
        fl.l_start  = 100;
        fl.l_len    = 10;
        if (fcntl(fd, F_SETLK, &fl) == -1) {
            // Обработка ошибки
        }
    }
    exit(EXIT_SUCCESS);
}
```

Установка флага Close-on-Exec

В примере показано, как установить флаг Close-on-Exec для файлового дескриптора **`fd`**.

```
#include <unistd.h>
#include <fcntl.h>
...
    int flags;

    flags = fcntl(fd, F_GETFD); // Получим флаги
    if (flags == -1) {
        // Обработка ошибки
    }
    flags |= FD_CLOEXEC;        // добавим FD_CLOEXEC
    if (fcntl(fd, F_SETFD, flags) == -1) {
        // Обработка ошибки
    }
```

Замечания по блокировкам

POSIX.1-2017 дает возможность одновременного доступа для чтения и записи к данным файла с помощью функции **fcntl()**. Однако без элементов управления параллелизмом эту функцию нельзя полностью использовать, чтобы не потерять случайно данные.

Потеря данных происходит несколькими способами.

Один из них возникает, когда несколько процессов пытаются обновить одну и ту же запись без упорядоченного контроля — несколько обновлений могут происходить параллельно, и последний записавший «выигрывает».

Другой случай — это двоичное дерево или какая-либо другая внутренняя база данных на основе списков, которая подвергается реорганизации. Без исключительного использования процессом обновления сегмента дерева другие процессы чтения могут потеряться в базе данных при разделении, сжатии, вставке или удалении индексных блоков. Хотя **fcntl()** полезна для многих приложений, она не предназначена для слишком общего применения и плохо справляется с примером двоичного дерева.

Возможности, предоставляемые **fcntl()**, требуются только для обычных файлов и они не подходят для многих устройств, таких как терминалы и сетевые соединения.

Поскольку **fcntl()** работает с «любым файловым дескриптором, связанным с этим файлом, каким бы способом он не был получен», дескриптор файла может быть также унаследован через операцию **fork()** или **exec** и, таким образом, может повлиять на файл, который был открыт другим процессом.

Использование дескриптора открытого файла чтобы определить что блокировать, требует дополнительных вызовов и создает проблемы если несколько процессов совместно используют дескриптор открытого файла.

Плюс существует слишком много реализаций, трактующих представленный механизм кучей способов.

Другим следствием этой модели является то, что закрытие любого дескриптора файла для данного файла (независимо от того, является ли он тем дескриптором, который создал блокировку) приводит к снятию блокировок этого файла для данного процесса.

Точно так же любое закрытие для любой пары файл/процесс снимает блокировки, принадлежащие этому файлу для этого процесса. Но следует обратить внимание, что хотя дескриптор открытого файла может быть передан через **fork( )**, блокировки через **fork( )** не наследуются. Однако блокировки могут быть унаследованы через одну из функций **exec**.

Идентификация машины в сетевой среде выходит за рамки данного тома POSIX.1-2017. Таким образом, член **l_sysid**, такой как в System V, не включается в структуру блокировки.

Изменение типов блокировки может привести к расщеплению ранее заблокированной области на более мелкие области.

Обязательная (mandatory) блокировка

Блокировки могут быть или консультативные (рекомендательные), или обязательные. По умолчанию используются консультативные. Они не обязательны к выполнению и полезны только в кооперативных процессах.

Обязательные блокировки влияют на все процессы.

Если процесс пытается получить несовместимый доступ (например, **read(2)** и **write(2)**) к области файла, на которую установлена несовместимая обязательная блокировка, то результат зависит от состояния флага **O_NONBLOCK** в описании (структура данных, связанная с файлом) этого открытого файла.

Если флаг **O_NONBLOCK** не установлен, то системный вызов блокируется до удаления блокировки или преобразуется в режим, который совместим с доступом.

Если флаг **O_NONBLOCK** установлен, то системный вызов завершается с ошибкой **EAGAIN**.

Чтобы использовать обязательные блокировки, в файловой системе, содержащей файл, и на самом файле должно быть включено обязательное блокирование.

Обязательное блокирование в файловой системе включается с помощью параметра «**-o mand**» команды **mount(8)** или с помощью флага **MS_MANDLOCK** в **mount(2)**.

Обязательное блокирование на файле включается посредством отключения права исполнения группе и установкой бита set-group-ID.

Чтобы рекомендательная/консультативная блокировка записи в файл была эффективной, все процессы, имеющие доступ к файлу, должны взаимодействовать и использовать консультативный механизм перед выполнением операций ввода/вывода в файле.

Блокировка записи в принудительном режиме важна, когда нельзя быть уверенным, что все процессы взаимодействуют друг с другом. Например, если один пользователь использует редактор для обновления файла в то же время, когда второй пользователь выполняет другой процесс, обновляющий тот же файл, и если только один из двух процессов использует рекомендательную блокировку, эти процессы не взаимодействуют. В данном случае блокировка записи в принудительном режиме защитит от случайных столкновений.

Используя блокировку файлов и записей в принудительном режиме, процесс может заблокировать файл один раз и разблокировать его после завершения всех операций ввода-вывода.

Блокировка записи в принудительном режиме обеспечивает основу, которую можно расширить, например, использовать дополнительно совместно используемые блокировки. То есть механизм может быть расширен, чтобы позволить процессу заблокировать файл таким образом, чтобы другие процессы могли его читать, но ни один из них не мог его записать.

Обязательные блокировки, в том числе и в Linux, ненадежны по нескольким причинам:

1. Обязательная установка блокировки была сделана мультиплексированием бита set-group-ID в большинстве реализаций. В лучшем случае это сбивало с толку.

2. Связь с усечением файлов, поддерживаемая в 4.2 BSD, не была четко определена.

3. Любой общедоступный файл может быть заблокирован кем угодно. Многие исторические реализации хранят базу паролей в общедоступном файле. Таким образом, злоумышленник может запретить вход в систему. Другой вариант — оставить открытой междугороднюю телефонную линию.

4. Некоторые реализации с предоставлением страниц памяти по запросу (demand-paged system) предлагают доступ к файлам с отображением их в памяти, и для файлов такого типа принудительное блокирование не может быть выполнено.

Поскольку ожидание в области блокировки прерывается любым сигналом, для обеспечения возможности тайм-аута в приложениях, может использоваться системный вызов **`alarm()`**. Это полезно при обнаружении тупиковых ситуаций.

Поскольку реализация полного обнаружения взаимоблокировок не всегда возможна, ошибка **`EDEADLK`** была сделана необязательной.

Потерянные блокировки

Если консультативная блокировка получена в сетевой файловой системе, такой как NFS, возможно, что блокировка будет потеряна. Это может произойти из-за административных действий на сервере или из-за разрыва сетевого соединения (т.е. потери сетевого подключения к серверу), который длится достаточно долго, чтобы сервер мог предположить, что клиент больше не функционирует.

Когда файловая система определяет, что блокировка была потеряна, будущие запросы **`read(2)`** или **`write(2)`** могут завершиться ошибкой **`EIO`**. Эта ошибка будет сохраняться до тех пор, пока блокировка не будет снята или дескриптор файла не будет закрыт. Начиная с Linux 3.12, это происходит, по крайней мере, для NFSv4 (включая все минорные версии).

Некоторые версии UNIX в этом случае отправляют сигнал (**`SIGLOST`**).

Linux не определяет этот сигнал и не предоставляет никаких асинхронных уведомлений о потерянных блокировках.

Управление сигналами

Функция **sigaction()** указывает, как должен обрабатываться сигнал процессом.

```
#include <signal.h>
int sigaction(int sig,
              const struct sigaction *restrict act,
              struct sigaction *restrict oldact)
```

Структура **sigaction** определяется примерно так:

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

Если в **sa_flags** установлен бит **SA_SIGINFO**, это значит, что обработчик сигнала принимает три аргумента, а не один. В этом случае вместо **sa_handler** следует установить **sa_sigaction**.

```
void func(int signo,       // продвинутый обработчик sa_sigaction
          siginfo_t *info, //
          void *context);  //
```

Имеет два дополнительных аргумента, которые передаются функции-обработчику сигнала.

Второй аргумент должен указывать на объект типа **siginfo_t**, объясняющий причину, по которой был сгенерирован сигнал.

Аргумент siginfo_t для SA_SIGINFO обработчика

```
siginfo_t {
    int      si_signo;     // Номер сигнала
    int      si_errno;     // Значение errno
    int      si_code;      // Код сигнала
    int      si_trapno;    // Номер ловушки, которую вызвал аппаратный сигнал
    pid_t    si_pid;       // ID процесса, пославшего сигнал
    uid_t    si_uid;       // ID реального пользователя процесса, пославшего сигнал
    int      si_status;    // Выходное значение или номер сигнала
    clock_t  si_utime;     // Использованное пользовательское время
    clock_t  si_stime;     // Использованное системное время
    sigval_t si_value;     // Значение сигнала
    int      si_int;       // sigqueue(3)
    void    *si_ptr;       // sigqueue(3)
    int      si_overrun;   // Счётчик переполнения таймера
    int      si_timerid;   // ID таймера; таймеры POSIX.1b
    void    *si_addr;      // Адрес памяти, приводящий к ошибке
    long     si_band;      // Внутреннее событие
    int      si_fd;        // Файловый дескриптор
    short    si_addr_lsb;  // Наименее значимый бит адреса
    void    *si_lower;     // Нижняя граница при нарушении адреса
    void    *si_upper;     // Верхняя граница при нарушении адреса
    int      si_pkey;      // Ключа защиты в PTE, который привёл к ошибке
    void    *si_call_addr; // Адрес инструкции системного вызова
    int      si_syscall;   // Количество попыток системного вызова
    unsigned int si_arch;  // Архитектура пытавшегося системного вызова
}
```

Для управления сигналами доступности ввода/вывода используются команды

F_GETOWN
F_SETOWN
F_GETOWN_EX
F_SETOWN_EX

F_GETSIG (не POSIX)
F_SETSIG (не POSIX)

F_SETSIG (int)

Установить сигнал, который будет послан, когда станет возможен ввод или вывод, равным значению, указанному в **arg**. Нулевое значение означает отправку сигнала **SIGIO** по умолчанию. Любое другое значение (включая **SIGIO**) является сигналом для отправки вместо **SIGIO**, и в этом случае для обработчика сигнала доступна дополнительная информация, если он был установлен с **SA_SIGINFO**.

В случае использования **F_SETSIG** с ненулевым значением и установкой **SA_SIGINFO** для обработчика сигнала (**sigaction(2)**), обработчику передаётся дополнительная информация о событиях ввода/вывода в структуре **siginfo_t** — если поле **si_code** показывает, что сигналом является **SI_SIGIO**, то поле **si_fd** содержит файловый дескриптор, ассоциированный с событием вывода/вывода.

F_GETSIG (void)

Возвращает (как результат функции **fcntl()**) сигнал, отправляемый тогда, когда становится возможным ввод или вывод. Нулевое значение означает отправку **SIGIO**.

Любое другое значение (включая **SIGIO**) является другим сигналом. В этом случае обработчику сигнала, если он был установлен с **SA_SIGINFO**, будет доступна дополнительная информация.

arg игнорируется.

В противном случае, не существует никакого механизма, чтобы сообщить с каким файловым дескриптором связан полученный сигнал и, чтобы определить, какой файловый дескриптор доступен для ввода/вывода, необходимо использовать вызовы **select(2)**, **poll(2)**, **read(2)** с установленным флагом **O_NONBLOCK** и т.д.

Следует обратить внимание, что дескриптор файла, предоставленный в **si_fd**, является одним из тех, что были указаны во время операций **F_SETSIG**. Это может привести к необычному крайнему случаю. Если файловый дескриптор дублируется (**dup(2)** или аналогичное), а исходный файловый дескриптор закрылся, то события ввода-вывода будут продолжать генерироваться и поле **si_fd** будет содержать номер закрытого к этому времени файлового дескриптора.

При выборе сигнала реального времени (value >= **SIGRTMIN**), в очередь будут добавляться несколько событий ввода вывода, с одинаковыми номерами сигналов. Размер очереди зависит от доступной памяти. Если для обработчика сигнала будет установлено **SA_SIGINFO**, будет доступна, как описано выше, дополнительная информация.

Linux налагает ограничение на количество сигналов в реальном времени, которые могут быть для процесса поставлены в очередь, и если этот предел достигнут, то ядро возвращается к доставке **SIGIO**, и это сигнал доставляется всему процессу, а не конкретному потоку.

F_GETOWN (void)

Вернуть (в качестве результата функции **fcntl()**) идентификатор процесса или группы процессов, которые в настоящее время получают сигналы **SIGIO** и **SIGURG** для событий на файловом дескрипторе **fd**.

Идентификаторы процесса возвращаются как положительные значения, идентификаторы групп процессов возвращаются как отрицательные значения (но см. ОШИБКИ ниже).

POSIX: Если **fd** относится к сокету и когда будут доступны внеполосные данные вызов вернет идентификатор процесса или идентификатор группы процессов, указанный для получения сигналов **SIGURG**. Нулевое значение указывает, что сигналы **SIGURG** не отправляются. Если **fd** не относится к сокету, результаты не определены.

F_SETOWN (int)

Установить идентификатор процесса или идентификатор группы процессов, которые будут принимать сигналы **SIGIO** и **SIGURG** для событий на файловом дескрипторе **fd**.

Целевой процесс или идентификатор группы процессов указывается в **arg**. Идентификатор процесса задаётся положительным числом, идентификатор группы задаётся отрицательным числом. Обычно, вызывающий процесс указывает самого себя в качестве владельца (принимающего), то есть в **arg** указывается результат **getpid(2)**.

Помимо установки владельца файлового дескриптора, необходимо также разрешить генерацию сигналов на файловом дескрипторе. Это делается с помощью команды **F_SETFL** системного вызова **fcntl()**, которая может установить в файловом дескрипторе флаг состояния файла **O_ASYNC**. После этого сигнал **SIGIO** отправляется всякий раз, когда становится возможным ввод или вывод в файловом дескрипторе. Для получения сигнала, отличного от **SIGIO**, может использоваться команда **F_SETSIG** системного вызова **fcntl()**.

Отправка сигнала процессу (группе)-владельцу, указанному в **F_SETOWN**, подлежит тем же проверкам разрешений, что и для **kill(2)**, где отправляющий процесс — это тот, который использует **F_SETOWN**. Если эта проверка разрешений терпит неудачу, сигнал игнорируется.

Примечание.

Операция **F_SETOWN** записывает учетные данные вызывающего абонента во время вызова **fcntl()**, и именно эти сохраненные учетные данные используются для проверок разрешений.

Если файловый дескриптор **fd** ссылается на сокет, **F_SETOWN** также выбирает получателя сигналов **SIGURG**, которые доставляются, когда на этот сокет поступают внеполосные данные. (**SIGURG** отправляется в любой ситуации, когда **select (2)** сообщит, что сокет имеет «исключительное состояние».)

F_SETOWN_EX (struct f_owner_ex *)

Эта команда выполняет задачу, аналогичную **F_SETOWN**. Она позволяет вызывающему направить сигналы доступности ввода-вывода конкретной нити, процессу или группе процессов. Вызывающий указывает принимающих сигнал в **arg**, который является указателем на структуру **f_owner_ex**:

```
struct f_owner_ex {
    int    type; // F_OWNER_TID, либо F_OWNER_PID, либо F_OWNER_PGRP
    pid_t  pid;  // ID нити, ID процесса или ID группы процессов
};
```

Поле типа имеет одно из следующих значений, которые определяют, как интерпретируется **pid**:

F_OWNER_TID — отправить сигнал потоку, идентификатор потока которого (значение, возвращаемое вызовом **clone(2)** или **gettid(2)**) указан в **pid**.

F_OWNER_PID — отправить сигнал процессу, идентификатор которого указан в **pid**.

F_OWNER_PGRP — отправить сигнал группе процессов, идентификатор которой указан в **pid**.

Следует заметить, что, в отличие от **F_SETOWN**, идентификатор группы процессов указывается здесь как положительное значение.

F_GETOWN_EX (struct f_owner_ex *)

получить настройки владения текущим файловым дескриптором, установленные предыдущей командой **F_SETOWN_EX**. Информация возвращается в структуре **f_owner_ex**, указанной в **arg**

Поле **type** должно быть равно либо **F_OWNER_TID**, либо **F_OWNER_PID**, либо **F_OWNER_PGRP**.

Значение поля **pid** — положительное целое, представляющее ID нити, ID процесса или ID группы процессов, соответственно.

Аренда файла

И рекомендательное, и обязательное блокирование предназначены для предотвращения доступа процесса к файлу или его части, которая используется другим процессом. Когда установлена блокировка, процесс, которому необходим доступ к файлу, должен подождать завершения процесса, владеющего блокировкой.

Данный паттерн подходит для большинства применений, но иногда программа желает использовать файл в режиме эксклюзивного доступа, но до тех пор, пока он не понадобится другой программе. Для этого Linux предлагает механизм аренды файлов (в других системах это называется периодическими блокировками (oplocks)).

Аренда файла предоставляет механизм, посредством которого процесс, который удерживает аренду («арендатор»), уведомляется сигналом, если процесс («нарушитель аренды») пытается выполнить вызов **open(2)** или **truncate(2)** для этого файла/дескриптора.

Для установки новой и получения текущей аренды открытого дескриптора файла используются, соответственно, команды **`fcntl()`** **`F_SETLEASE`** и **`F_GETLEASE`**.

Существуют два типа аренды:

- аренда чтения;
- аренда записи.

Аренда чтения вызывает передачу сигнала при открытии файла для записи, открытии с указанием **`O_TRUNC`** или вызове **`truncate()`**.

Аренда записи посылает сигнал при открытии файла для чтения или записи.

Аренды файлов работают только для модификаций, вносимых в файл той же системой, которая владеет арендой. Если файл локальный (не файл, доступ к которому возможен через сеть), любой подходящий доступ к файлу инициирует сигнал. Если доступ к файлу возможен через сеть, передачу сигнала вызывают только процессы на одной машине с процессом-арендатором; доступ с любой другой машины удается в случае отсутствия аренды.

Аренды можно размещать только на обычных файлах (для каналов и каталогов это невозможно), кроме того, аренды записи предоставляются только владельцу файла.

F_SETLEASE

В зависимости от значения последнего параметра, передаваемого в **fcntl()** аренда создается или удаляется:

F_RDLCK — установить аренду чтения. Это приведёт к генерации уведомления вызывающего процесса, когда файл открывается для записи или усечения. Аренда чтения может быть выделена только на файловый дескриптор, открытый только на чтение.

F_WRLCK — установить аренду записи. Это приведёт к генерации уведомления вызывающего процесса, когда файл открывается для чтения или записи или выполняется его усечение. Аренда записи может быть установлена на файл, только если этот файл не имеет других открытых файловых дескрипторов.

F_UNLCK — удалить аренду с указанного файла.

Если запрашивается новая аренда, она заменяет любую существующую аренду.

В случае ошибки возвращается отрицательное число.

Ноль или положительное число свидетельствуют об успехе операции.

Аренды ассоциируются с открытым файловым описанием (**open(2)**). Это значит, что дублированные файловые дескрипторы (созданные, например, пр вызове **fork(2)** или **dup(2)**) указывают на одну и ту же аренду, и эта аренда может изменяться или удаляться через любой из этих дескрипторов. Более того, аренда удаляется или через явную команду **F_UNLCK** на любом из этих дублированных дескрипторов, или когда все эти дескрипторы будут закрыты.

Аренды могут быть выданы только на обычные файлы. Непривилегированный процесс может получить аренду только на файл, чей UID (владельца) совпадает с UID на файловой системе процесса. Процесс с мандатом **CAP_LEASE** может получить аренду на любые файлы.

F_GETLEASE

Указывает, какой тип аренды связан с файловым дескриптором **fd**.

Возвращается одно из значений **F_RDLCK**, **F_WRLCK** или **F_UNLCK**, соответственно означающих аренду на чтение, запись или что аренды нет. Аргумент **arg** игнорируется.

Что происходит?

Когда процесс («нарушитель аренды») выполняет вызов **open(2)** или **truncate(2)**, который конфликтует с арендой, установленной через **F_SETLEASE**, то системный вызов блокируется ядром и ядро уведомляет арендатора сигналом (по умолчанию **SIGIO**).

Владелец аренды должен отреагировать на получение этого сигнала, выполнив любую очистку, которая требуется для подготовки файла к доступу другим процессом (например, очистка кэшированных буферов), а затем либо удалить, либо понизить его аренду.

Аренда удаляется по команде **F_SETLEASE** с аргументом **arg**, установленным в **F_UNLCK**.

Если владелец аренды в настоящее время имеет аренду на запись в файл, а прерыватель аренды открывает файл для чтения, то держателю аренды достаточно понизить аренду до аренды для чтения. Это делается путем выполнения команды **F_SETLEASE** с указанием **arg** как **F_RDLCK**.

Если арендатор не освободит аренду или не снизит условия в течении определённого количества секунд, указанного в файле **/proc/sys/fs/lease-break-time**, то ядро принудительно удалит аренду или снизит условия аренды для арендатора.

Как только был инициировано нарушение аренды, **F_GETLEASE** будет возвращать целевой тип аренды (либо **F_RDLCK**, либо **F_UNLCK**, в зависимости от того, что будет совместимо с нарушителем аренды) до тех пор, пока владелец аренды добровольно не снизит или не удалит аренду, либо ядро не сделает это принудительно после того, как таймер нарушения аренды истечет.

После того, как аренда была добровольно или принудительно удалена либо понижена, и при условии, что нарушитель аренды не разблокировал свой системный вызов, ядро разрешает выполнение системного вызова нарушителя аренды.

Если блокировка **read(2)** или **truncate(2)** нарушителя аренды прерывается обработчиком сигнала, то системный вызов завершается ошибкой с ошибкой **EINTR**, но все другие шаги по-прежнему выполняются, как описано выше.

Если нарушитель аренды завершился по сигналу в то время, когда он был заблокирован в состоянии **open(2)** или **truncate(2)**, другие шаги по-прежнему выполняются, как описано выше.

Если нарушитель аренды указывает флаг **O_NONBLOCK** при вызове **open(2)**, то вызов немедленно завершается ошибкой **EWOULDBLOCK**, но другие шаги по-прежнему будут выполняться, как описано выше.

Когда в арендованном файле происходит одно из контролируемых событий, ядро передает сигнал процессу, удерживающему аренду.

По умолчанию передается **SIGIO**, но процесс может выбрать, какой сигнал передавать этому файлу, с помощью вызова **fcntl()**, в котором второй параметр установлен в **F_SETSIG**, а последний — в сигнал, который должен использоваться вместо **SIGIO**.

Использование **F_SETSIG** дает один значительный эффект. По умолчанию при доставке **SIGIO** обработчику не передается **siginfo_t**. Если же используется **F_SETSIG**, даже когда сигналом, передаваемым в ядро, является **SIGIO**, а при регистрации обработчика сигнала был установлен **SA_SIGINFO**, файловый дескриптор, аренда которого инициировала событие, передается в обработчик сигналов в члене **si_fd** одновременно с элементом **siginfo_t**. Это позволяет применять один сигнал к аренде набора файлов, в то время как **si_fd** сообщает сигналу, какому файлу необходимо уделить внимание.

Единственные два системных вызова, которые могут инициировать передачу сигнала для арендуемого файла — это **open()** и **truncate()**. Когда они вызываются процессом для арендуемого файла, они блокируются, и процессу-владельцу передается сигнал.

open() или **truncate()** завершаются после удаления аренды с файла (или его закрытия процессом-владельцем, что вызывает удаление аренды). Если процесс, удерживающий аренду, не отменяет снятие в течение времени, указанного в файле **/proc/sys/fs/lease-break-time**, ядро прерывает аренду и позволяет завершиться запускающему системному вызову.

Ниже приведен пример применения аренды для уведомления о намерении другого процесса получить доступ к файлу. Список файлов берется командной строки, и на каждый файл устанавливается аренда по записи. Когда другой процесс намеревается получить доступ к файлу (даже для чтения, поскольку использовалась блокировка записи), программа освобождает блокировку файла, позволяя другому процессу продолжать работу. Она также выводит сообщение об освобождении файла.

```
#define GNU_SOURCE

#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

const char **fileNames;
int   numFiles;

void handler3(int sig, siginfo_t *siginfo, void *context) {
// Когда аренда истечет, вывод сообщения и закрытие файла. Предполагается, что
// первый открываемый файл получит файловый дескриптор 3, следующий - 4 и т.д.
    write(1, "освобождение ", 13);
    write(1,
          fileNames[siginfo->si_fd – 3], strlen(fileNames[siginfo->si_fd - 3]));
    write(1, " ", 1);
    fcntl(siginfo->si_fd, F_SETLEASE, F_UNLCK); // снять аренду
    close(siginfo->si_fd);
    numFiles--;
}
```

```
int main(int argc, const char **argv) {

    int fd;
    const char **file;
    struct sigaction act;

    if (argc < 2) {
        fprintf(stderr, "использование: %s <filename>+ ", argv[0]);
        return 1;
    }

   // Регистрация обработчика сигналов. Указание SA_SIGINFO предоставляет
   // обработчику возможность узнать, какой fd имеет истекшую аренду.
    act.sa_sigaction = handler3;
    act.sa_flags     = SA_SIGINFO;
    sigemptyset(&act.sa_mask);
    sigaction(SIGRTMIN, &act, NULL); // пользовательский RT-сигнал SIGRTMIN

    // Сохраняем список имен файлов в глобальной переменной, чтобы обработчик
    // сигналов мог иметь к нему доступ.
    fileNames = argv + 1;
    numFiles  = argc - 1;
```

```
    // Открываем файлы, устанавливаем используемые сигнал и создаем аренду
    for (file = fileNames; *file; file++) {
        if ((fd = open(* file, O_RDONLY)) < 0) {
            perror("open");
            return 1;
        }

        // Для правильного заполнения структуры siginfo
        // необходимо использовать F_SETSIG
        if (fcntl(fd, F_SETSIG, SIGRTMIN) < 0) {
            perror("F_SETSIG");
            return 1;
        }
        // регистрируем аренду
        if (fcntl(fd, F_SETLEASE, F_WRLCK) < 0) {
            perror("F_SETLEASE");
            return 1;
        }
    }

    // Пока есть открытые файлы, ожидаем поступления сигналов.
    while (numFiles) {
        pause();
    }

    return 0;
}
```

Уведомления об изменении файла и каталога

F_NOTIFY (int)

Уведомлять при изменении каталога, на который указывает **fd** или когда изменились файлы, которые в нём содержатся.

События, о наступлении которых делается уведомление, задаются в аргументе **arg**, который является битовой маской, получаемой сложением (OR) одного или более следующих бит:

DN_ACCESS — был произведён доступ к файлу (read(2), pread(2), readv(2), ...);
DN_MODIFY — файл был изменён (write(2), pwrite(2), writev(2), truncate(2), ftruncate(2), ...);
DN_CREATE — файл был создан (open(2), creat(2), mknod(2), mkdir(2), link(2), symlink(2), rename(2));
DN_DELETE — файл был удалён (unlink(2), rename to another directory, rmdir(2));
DN_RENAME — файл был переименован внутри каталога (rename(2));
DN_ATTRIB — у файла были изменены атрибуты (chown(2), chmod(2), utime[s](2), ...).

Уведомления об изменении состояния каталога обычно однократные и приложение должно перерегистрировать установку уведомлений, чтобы и дальше их получать. Однако, если в аргумент **arg**, добавить **DN_MULTISHOT**, то уведомления будут приходить до тех пор, пока не будут явно отменены.

Серии запросов **F_NOTIFY** добавляются к событиям в **arg**, которые уже установлены.

Чтобы выключить уведомления о всех событиях, следует выполнить вызов **F_NOTIFY**, указав 0 в **arg**.

Уведомления выполняются через доставку сигнала. По умолчанию это **SIGIO**, но его можно изменить, используя команду **F_SETSIG** системного вызова **fcntl()**. В последнем случае обработчик сигнала получает структуру **siginfo_t** в качестве своего второго аргумента, если он был установлен с использованием **SA_SIGINFO** и поле **si_fd** этой структуры будет содержать дескриптор файла, который сгенерировал уведомление (полезно при установке уведомления в нескольких каталогах).

Следует обратить внимание, что **SIGIO** является одним из стандартных сигналов, которые не ставятся в очередь. Поэтому, особенно в случае использования **DN_MULTISHOT** для уведомления, может быть полезным использовать сигналы реального времени. Это позволяет поставить несколько уведомлений в очередь.

ПРИМЕЧАНИЕ

Новые приложения могут использовать интерфейс **inotify(7)** (доступный начиная с ядра 2.6.13), который обеспечивает гораздо лучший интерфейс для получения уведомлений о событиях файловой системы. Однако этот интерфейс есть только у Linux.

Изменение емкости канала

F_SETPIPE_SZ (int)

Изменяет ёмкость канала, на который указывает **fd**. Она становится равной не менее **arg** байт.

Непривилегированный процесс может подстроить емкость канала до любого значения начиная с размера системной страницы до предела, заданного в **/proc/sys/fs/pipe-max-size**.

Попытки установить пропускную способность канала ниже размера страницы автоматически округляются до размера страницы.

Задание непривилегированным процессом ёмкости канала выше предела из **/proc/sys/fs/pipe-max-size** приведёт к ошибке **EPERM**.

Привилегированный процесс (**CAP_SYS_RESOURCE**) может переопределить ограничение.

При выделении буфера для канала ядро может использовать емкость больше, чем **arg**, если это удобно для реализации. (В текущей реализации выделение — это следующая большая величина, кратная размеру страницы и кратная запрошенному размеру.)

Попытка установить пропускную способность канала меньше, чем объем буферного пространства, используемого в настоящее время для хранения данных, приводит к ошибке **EBUSY**.

Следует обратить внимание, что из-за способа использования страниц буфера конвейера при записи данных в конвейер, количество байтов, которые могут быть записаны, может быть меньше номинального размера, в зависимости от размера записи.

В качестве результата возвращается установленная фактическая емкость в байтах.

F_GETPIPE_SZ (void)

Возвращает ёмкость канала, указываемого **fd**.