Тема 8. Управление файлами в ОС GNU/Linux

Главные вопросы, которые мы обсудим, представлены на СЛАЙДЕ 1. По ходу лекции даются примеры системных вызовов ввода-вывода.

8.1 Файлы и файловые системы

Подавляющее большинство современных ОС предлагает пользователям абстракцию, называемую **файлами**.

Файл – это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Файлы хранятся в памяти, не зависящей от энергопитания. Одним из исключений является так называемый виртуальный диск, когда в оперативной памяти создается структура, имитирующая ФС. Можно выделить следующие основные цели использования файлов (СЛАЙД 2):

- **Долговременное и надежное хранение информации**.
- **Совместное использование информации**.

Эти цели достигаются с использование файловых систем (ФС). **Файловая система** – это часть подсистемы ввода-вывода ОС, включающая (СЛАЙД 3):

- совокупность всех файлов на диске;
- наборы структур данных, используемых для управления файлами, такие, например как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске;
- комплекс программных средств, реализующих различные операции над файлами, такие как создание, уничтожение, чтение, запись, именование и поиск файлов.

ФС позволяет программам обходиться набором достаточно простых операций для выполнения действий над некоторым абстрактным объектом, которым является файл. При этом программистам не нужно иметь дело с низкоуровневыми проблемами передачи данных с долговременного ЗУ. Все эти функции ФС берет на себя: распределяет дисковую память, поддерживает именование файлов, преобразует имена файлов в соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление файлов.

Задачи, решаемые ФС, конечно, зависят от способа организации вычислительного процесса в целом. Основные функции ФС нацелены на решение следующих задач (СЛАЙД 4):

- именование файлов;
- программный интерфейс для прикладных программ;
- отображение логической модели ФС на физическую организацию хранилища данных;
- устойчивость ФС к сбоям электропитания, ошибкам программных и аппаратных средств;
- организацию совместного доступа к файлу нескольких процессов;
- защиту файлов одного пользователя от несанкционированного доступа другого и т. д.

ФС поддерживают несколько функционально различных типов файлов, в число которых, как правило, входят обычные файлы, каталоги, специальные файлы, именованные «конвейеры», «почтовые ящики», символические ссылки и связи, сокеты. Иногда сюда относят и отображаемые в память файлы. СЛАЙД 5.

Обычные файлы, или просто **файлы**, содержат информацию произвольного характера, которую заносит в них пользователь или которая образуется в результате

работы системных и пользовательских программ. Большинство современных ОС никак не ограничивает и не контролирует содержимое и структуру обычного файла, но все ОС должны распознавать хотя бы один тип файлов – их собственные исполняемые файлы.

Каталоги – это особый тип файлов, которые содержат системную справочную информацию о наборе файлов, сгруппированных пользователями по какому-либо неформальному признаку. Во многих ОС в каталог могут входить файлы любых типов, в том числе другие каталоги, за счет чего образуется иерархическая структура, удобная для поиска. Каталоги устанавливают соответствие между именами файлов и их характеристиками, используемыми ФС для управления файлами. В число таких характеристик входит, в частности, информация о типе файла и расположении его на диске, правах доступа к файлу и датах его создания и модификации. Во всех остальных отношениях каталоги рассматриваются ФС как обычные файлы.

Специальные файлы – это фиктивные файлы, ассоциированные с устройствами ввода-вывода, используемые для унификации механизма доступа к файлам и устройствам.

Символьные ссылки – файлы в ФС, для которых не формируются никакие данные, кроме одной текстовой строки с указателем. Эта строка трактуется как путь к файлу (это символьная ссылка) или каталогу (это символьная связь), который должен быть открыт при попытке обратиться к данной ссылке (файлу).

Конвейеры и сокеты являются средствами межпроцессорного взаимодействия и рассматриваются в одном из следующих разделов лекционного курса.

Все типы файлов имеют символьные имена. В иерархически организованных ФС обычно используются три типа имен файлов: простые, составные и относительные. СЛАЙД 6.

Простое, (**короткое**), символьное имя идентифицирует файл в пределах одного каталога. Простые имена присваивают файлам пользователи и программисты, при этом они должны учитывать ограничения ОС как на набор символов, так и на длину имени. Современные ФС, как правило, поддерживают длинные простые символьные имена файлов. Во многих ФС GNU/Linux имя файла может содержать до 256 байт.

В иерархических ФС разным файлам разрешено иметь одинаковые простые символьные имена при условии, что они принадлежат разным каталогам. То есть здесь работает схема «**много файлов – одно простое имя**». Для однозначной идентификации файлов в таких системах используется так называемое полное имя.

Полное имя представляет собой цепочку простых имен всех каталогов, через которые проходит путь от корня до данного файла. Таким образом, полное имя является составным, в нем простые имена отделены друг от друга принятым в ОС разделителем. Часто в качестве разделителя используется прямой и обратный слеш, при этом не принято указывать имя корневого каталога.

В древовидной ФС между файлом и его полным именем имеется взаимно однозначное соответствие «**один файл – одно полное имя**». В ФС с сетевой структурой файл может входить в несколько каталогов, т. е. иметь несколько полных имен; здесь справедливо соответствие «**один файл – много полных имен**». В обоих случаях файл однозначно идентифицируется полным именем.

Файл может быть идентифицирован также **относительным именем**. Относительное имя файла определяется через понятие «текущий каталог». Для каждого пользователя в каждый момент времени один из каталогов является текущим, этот каталог выбирается самим пользователем по команде ОС. ФС фиксирует имя текущего каталога, для использования его как дополнения к относительным именам при образовании полного имени файла. При использовании относительных имен пользователь идентифицирует файл цепочкой имен каталогов, через которые проходит маршрут от текущего каталога до данного файла.

В некоторых ОС разрешено присваивать одному и тому же файлу несколько простых имен, которые можно интерпретировать как псевдонимы. В этом случае также

как в системе с сетевой структурой, устанавливается соответствие «один файл – много полных имен», так как каждому простому имени файла соответствует по крайней мере одно полное имя. СЛАЙД 7.

Хотя полное имя однозначно определяет файл, ОС проще работать с файлом, если между файлами и именами имеется взаимно однозначное соответствие. С этой целью она присваивает файлу **уникальное имя**, так что справедливо соотношение «**один файл – одно уникальное имя**». Уникальное имя существует наряду с одним или несколькими символьными именами, присваиваемыми файлу пользователями или программами. Уникальное имя представляет собой числовой идентификатор и предназначено только для ОС. Примером такого уникального имени файла является **номер индексного дескриптора** во многих ФС, поддерживаемых UNIX-подобными ОС, в том числе GNU/Linux.

8.2 Файловые операции

Для выполнения некоторых файловых операций есть несколько системных вызовов, выглядящих как обычные функции языка программирования высокого уровня. При выполнении этих операций участвуют так называемые **файловые дескрипторы**. Чтобы использовать эти функции в программах, необходимо подключить заголовочные файлы *fcntl.h*, *sys/types.h*, *sys/stat.h*, *unistd.h*.

Дескриптор представляет собой целое число, обозначающее конкретный экземпляр файла, открытого в одном процессе. Файл можно открыть для чтения, записи, одновременно и того, и другого. Файловому дескриптору не обязательно соответствует обычный файл, это также может быть другой системный компонент, способный передавать и принимать данные (спецфайл, сокет, один из концов конвейера).

Чтобы открыть файл и получить дескриптор для работы с ним, необходимо вызвать функцию *open()*. СЛАЙД 8.

```
#include <fcntl.h>
int open(char* file   // имя файла
         , int oflag  // способ открытия файла
         , ...        // необязательный третий аргумент
);
```

В качестве аргументов принимается строка с именем файла и флаги, определяющие способ открытия. С помощью данной функции можно создавать новый файл, для этого предназначен третий аргумент функции – права доступа к файлу. Возможные значения второго аргумента (СЛАЙД 9):

- *O_RDONLY* – файл открывается только для чтения;
- *O_WRONLY* – файл доступен только для записи;
- *O_RDWR* – файл открывается и для чтения, и для записи;
- *O_TRUNC* – приводит к очистке существующего файла;
- *O_APPEND* – приводит к открытию файла в режиме добавления;
- *O_CREAT* – означает создание нового файла. Если файл уже существует, он будет открыт;
- *O_EXCL* – при использовании совместно с *O_CREAT* откажется открывать существующий файл.

Когда задан флаг *O_CREAT*, должен присутствовать третий аргумент, определяющий права доступа к создаваемому файлу, как в примере кода на СЛАЙДЕ 10. После успешного завершения работы этой программы мы получим файл, длина которого равно нулю.

Можно воспользоваться функцией создания файла, которая объявляется следующим образом (СЛАЙД 11).

```
#include <unistd.h>
int creat(char* file       // символьное имя файла
```

```
          , mode_t mode    // режим создания файла
);
```

По окончании работы с файлом его необходимо закрыть (СЛАЙД 11) вызовом функции *close()*. В некоторых случаях, например, в программе с предыдущего листинга, функцию закрытия можно не вызывать, поскольку Linux автоматически закрывает все открытые файлы при завершении программы. После того, как файл закрыт, очевидно, обращаться к нему нельзя.

```
#include <unistd.h>
int close(int fd     // дескриптор открытого файла
);
```

Закрытие файла вызывает разную реакцию ОС, в зависимости от типа файла. Например, закрытие сокета может сопровождаться разрывом сетевого соединения между двумя сетевыми узлами, взаимодействующими через сокет.

Linux ограничивает количество файлов, которые могут быть открыты процессом в определенный момент времени. Дескрипторы занимают ресурсы ядра ОС, поэтому желательно файлы закрывать вовремя, чтобы дескрипторы удалялись из системных таблиц. Обычно процессам назначается ограничения в 1024 дескриптора.

Запись данных в файл

Для этого (СЛАЙД 12) предназначена функция *write()*.

```
#include <unistd.h>
ssize_t write(int fd        // дескриптор файла
              , void* Buf   // адрес буфера, в котором хранятся данные для записи
              , size_t BytesToWrite // сколько байтов записать
);
```

Очевидно, что файл должен быть открыт к моменту выполнения данной операции. Природа данных, которые записываются посредством функции *write()*, ей неинтересна. Она работает с байтовыми последовательностями.

В программе, чей листинг приведен на СЛАЙДЕ 13, в заданный файл записывается значение текущего времени. Если файл не существует, то он создается. Для получения и форматирования значений времени используются функции *time()*, *localtime()*, *asctime()*.

При первом вызове этой программы файл был создан, а при втором – дополнен. Функция возвращает число записанных байтов, или -1, в случае ошибки. Для некоторых типов файлов число фактически записанных байтов может оказаться меньше требуемого. Программа должна выявлять подобные случаи и вызывать функцию *write()* повторно, чтобы передать оставшуюся часть данных. Такая техника продемонстрирована в листинге кода, приведенного на СЛАЙДЕ 14.

Чтение данных из файла

Чтение осуществляется (СЛАЙД 15) функцией *read()*.

```
#include <unistd.h>
ssize_t read(int fd               // дескриптор файла
             , void* Buf          // адрес буфера, в который читаются данные
             , size_t BytesToRead // сколько байтов прочитать
);
```

Очевидно, что файл должен быть открыт к моменту выполнения данной операции. Природа данных, которые считываются из файла посредством функции *read()*, ей неинтересна. Она работает с байтовыми последовательностями.

Иногда считывается меньше байтов, чем требовалось, например, в файле содержится недостаточно байтов.

В программе, чей листинг приведен на СЛАЙДЕ 16, демонстрируется применение

этой функции. Программа отображает шестнадцатеричный дамп файла, заданного в командной строке. В каждой строке показано смещение от начала файла, а затем – следующие 16 байтов.

Перемещение по файлу

В файловом дескрипторе запоминается текущая позиция в файле. При выполнении операций ввода-вывода указатель будет перемещен на прочитанное количество байтов. Для выполнения еще одной полезной файловой операции, а именно перемещения внутри файла принято использовать функцию *lseek()*. См. СЛАЙД 17.

```
#include <unistd.h>
off_t lseek(int fd                  // Дескриптор открытого файла
            , off_t bytesToMove     // Количество байтов, на которые надо перейти
            , int whence            // С какой позиции перемещаться
);
```

Функция возвращает новую позицию файлового указателя, или -1, в случае ошибки. Если аргумент *whence* равен *SEEK_SET*, то второй аргумент интерпретируется как смещение от начала файла. Если *whence* равен *SEEK_CUR*, то смещение производится от текущей позиции в файле. Если *whence* равен *SEEK_END*, то аргумент *bytesToMove* интерпретируется как смещение от конца файла. Функция неприменима к некоторым типам файлов, например, сокетам.

Можно задавать нулевое значение смещения, чтобы узнать текущую позицию в файле. Например:

```
off_t pos = lseek(fd, 0, SEEK_CUR);
```

Можно перемещать указатель за пределы файла. Если текущая позиция находится за концом файла и выполняется запись в него, то ОС автоматически увеличивает файл, чтобы вместить в него новые данные. Промежуток между «старым» концом файла и указателем текущей позиции не записывается на диск, вместо этого ОС только помечает его длину. Если потом попытаться прочесть файл, то окажется, что данный промежуток заполнен нулевыми байтами.

Как следствие этой особенности, возможно создание файлов огромных размеров, которые при этом не занимают места на диске. Это демонстрируется программой, листинг которой приведен на СЛАЙДЕ 18. В качестве аргументов командной строки программа принимает имя файла и размер в мегабайтах. Она создает файл, перемещается с помощью нашей функции на требуемое расстояние и записывает нулевой байт, а затем закрывает файл.

Далее пробуем создать файл размером 1 Гб, и обращаем внимание на объем свободного места на диске до и после выполнения программы.

Видно, что файл практически не занимает место на диске, несмотря на свой огромный размер, но если попытаться открыть его и прочитать данные, то окажется, что в нем ровно 1 Гб нулей. Можно проверить это с помощью нашей предыдущей программы.

Функция *lseek()* не сможет перейти на указатель перед началом файла.

Функции stat() и fstat()

Функция *read()*, конечно, весьма полезна, но она позволяет прочитать только содержимое файла. Остальную полезную информацию о файле (СЛАЙД 19) можно получить с помощью функций *stat()* и *fstat()*.

```
#include <sys/stat.h>
int stat(char* filename            // символьное имя файла
         , struct stat* stat_info // указатель на дескриптор информации о файле
);
```

```
int fstat(int fd // дескриптор файла
          , struct stat* stat_info // указатель на дескриптор информации о файле
);
```

В случае успеха эти функции возвращают 0 и заполняют структуру с данными о файле, в противном случае возвращается -1. Полезными полями структуры *stat* являются (СЛАЙД 20):

- *st_mode* – код доступа к файлу.
- *st_uid* и *st_gid* – идентификаторы пользователя и группы.
- *st_size* – размер файла в байтах.
- *st_atime* – время последнего обращения к файлу.
- *st_mtime* – время последней модификации файла.

Следующие макросы, проверяя поле *st_mode*, служат для определения того, к какому типу принадлежит файл (СЛАЙД 21):

- *S_ISBLK* – блочное устройство.
- *S_ISCHR* – символьное устройство.
- *S_ISDIR* – каталог.
- *S_ISFIFO* – именованный конвейер.
- *S_ISLNK* – символическая ссылка.
- *S_ISREG* – обычный файл.
- *S_ISSOCK* – сокет.

Все они получают в качестве аргумента код доступа и возвращают ненулевое значение, если их догадка о типе файла подтвердилась.

В поле *st_dev* структура *stat* содержится младший и старший номера устройства, на котором расположен файл. В поле *st_info* содержится номер индексного дескриптора файла, определяющий местоположение файла в ФС.

Если вызвать функции для символической ссылки, то они проследят, куда указывает ссылка, и вернут информацию о самом файле, а не о ссылке. В связи с этим макрос *S_ISLNK* всегда возвращает 0. Однако есть еще одна функция – *lstat()*, которая не пытается отслеживать символические ссылки, а во всем остальном – аналогичная функции *stat()*. Возможна ситуация, когда символьная ссылка оказалась поврежденной. Тогда функции *stat()* и *fstat()* возвращают ошибку, а *lstat()* выполняется успешно.

Если файл уже открыт для чтения/записи, то лучше пользоваться функцией *fstat()*, т.к. она в качестве первого аргумента принимает файловый дескриптор.

В листинге кода, приведенного на СЛАЙДЕ 22, показана функция, создающая буфер достаточного размера и загружающая в него содержимое указанного файла. Размер файла определяется функцией *fstat()*. Она же проверяет, является ли заданный файл обычным.

Векторные чтение и запись

Выше мы показали, что аргументами функции *write()* являются указатель на буфер и его длина. Эта функция записывает в файл непрерывный блок данных из памяти. Однако во многих случаях программам необходимо записывать группы блоков с разными адресами. Понятно, что при использовании *write()* мы тоже можем сделать это, но тогда придется объединять блоки в памяти (а это может быть неэффективно) либо многократно вызывать функцию. Последнее может оказаться тоже не всегда приемлемым. .

Функция *writev()* записывает в файл несколько несвязанных буферов одновременно. Это называется **векторной записью**. Сложность ее применения заключается в создании структуры, задающей начало и конец каждого буфера и представляющей собой массив элементов типа *struct iovec*. Каждый из них описывает одну область памяти. В поле *iov_base* задается адрес начала области, а в поле *iov_len* – ее длина. Если число буферов известно заранее, то достаточно объявить массив структур *iovec*. В противном случае – выделить память для массива динамически. СЛАЙД 23.

Функции векторной записи передается дескриптор записываемого файла, массив структур *iovec* и размер массива. Функция возвращает общее число записанных байтов.

Программа, листинг которой приведен на СЛАЙДАХ 24-25, записывает аргументы командной строки во внешний файл с помощью единственного вызова функции *writev()*. Первый аргумент – имя файла, в который построчно сохраняются все остальные аргументы. Количество аргументов заранее неизвестно, поэтому массив создается функцией *malloc()*.

Также имеется функция **векторного чтения** *readv()*, которая загружает содержимое внешнего файла в несколько несвязанных областей памяти. В ней массив структур *iovec* определяет начало и размер области.

Работа с каталогами

Функция *getcwd()* возвращает имя текущего каталога, *chdir()* делает текущим заданный каталог, *mkdir()* – создает новый каталог, *rmdir()* – удаляет каталог (СЛАЙД 26).

```
#include <unistd.h>
char* getcwd(char* buf    // буфер для заполнения полным именем текущего каталога
             , size_t size // размер буфера
);

int chdir(char* path // символьное имя каталога
);

int rmdir(char* path // символьное имя каталога
);

#include <sys/stat.h>
int mkdir(char* path // символьное имя создаваемого каталога
    , mode_t mode    // режим создания каталога
);
```

В Linux имеются функции для чтения содержимого каталога. В программах необходимо придерживаться следующей последовательности действий (СЛАЙД 27).

1. Вызвать функцию *opendir()* с именем требуемого каталога. Она возвращает указатель на структуру типа *DIR**, который может использоваться для доступа к содержимому каталога, или NULL.

```
#include <dirent.h>
#include <sys/types.h>
DIR* opendir(char* path   // символьное имя каталога
);
```

2. Далее нужно последовательно вызывать функцию *readdir()*, передавая ей дескриптор, который был возвращен функцией *opendir()*. Функция *readdir()* будет всякий раз возвращать указатель на структуру типа *dirent**, содержащую информацию о следующем элементе каталога. Когда будет достигнут конец каталога, функция возвратит NULL. В структуре *dirent* есть поле *d_name* (*char d_name[256];*), где содержится имя элемента каталога.

```
#include <dirent.h>
#include <sys/types.h>
struct dirent* readdir (DIR* dir_p // указатель на структуру каталога
);
```

3. Для завершения работы с каталогом необходимо вызвать функцию *closedir()*, передавая ей дескриптор, возвращенный функцией *opendir()*.

```
#include <dirent.h>
#include <sys/types.h>
int closedir (DIR* dir_p // указатель на структуру каталога
);
```

В листинге, приведенном на СЛАЙДАХ 28-29, показана программа, отображающая список содержимого каталога, имя которого задано в командной строке. Если этого не сделать, то отображается содержимое текущего каталога. Для каждого элемента каталога отображается его тип и полное имя, Функция *GetFileType()* определяет тип объекта ФС с помощью функции *lstat()*.

Другие файловые функции

За удаление файлов отвечает несколько функций, одна из них – *remove()*. За переименование файлов ответственность несет функция *rename()*. См. СЛАЙД 30.

```
#include <unistd.h>
int rename(char* old_path   // старое имя файла
           , char* new_path // новое имя файла
);
int remove(char* path // имя файла для удаления
);
```

Еще одна функция позволяет удалять обычные файлы, именованные конвейеры, специальные файлы – *unlink()*.

```
#include <unistd.h>
int unlink(char* path   // имя файла
);
```

На самом деле эта функция всего лишь удаляет элемент из каталога, в котором располагается указанный файл. Если это имя было последней ссылкой на файл, и больше нет процессов, которые держат этот файл открытым, данный файл удаляется, и место, которое он занимает, освобождается для дальнейшего использования.

Если имя было последней ссылкой на файл, но какие-либо процессы все еще держат этот файл открытым, то файл будет оставлен пока последний дескриптор, указывающий на него, не будет закрыт.

Если имя указывает на символьную ссылку, ссылка будет удалена.

Если имя указывает на сокет, именованный конвейер или устройство, то имя будет удалено, но процессы, которые открыли любой из этих объектов, могут продолжать его использовать.

Чтение символьных ссылок

В программе, листинг которой приведен на СЛАЙДЕ 31, показано использование функции *readlink()*. Она определяет адресат символьной ссылки, принимает три аргумента: имя, буфер для записи адресата и его длину. Полное имя, помещаемое в буфер, не завершается нулевым символом. Однако в третьем аргументе возвращается длина буфера, поэтому добавить этот символ несложно.

Если первый аргумент не ссылка, то функция возвращает -1, а в переменную *errno* записывается константа *EINVAL*.

Проверка прав доступа к файлу

Функция *access()* определяет, имеет ли вызвавший ее процесс право доступа к указанному файлу. Функция способна проверить любую комбинация привилегий чтения, записи, выполнения и существования файла. СЛАЙД 32.

Она принимает два аргумента: имя проверяемого файла и битовое объединение флагов *R_OK*, *W_OK* и *X_OK*. Они соответствуют правам чтения, записи и выполнения. При наличии у процесса привилегий возвращается 0. Если файл существует, а нужные привилегии отсутствуют, то возвращается -1, а в переменной *errno* оказывается значение *EACCESS* либо *EROFS*.

В качестве второго аргумента можно передавать *F_OK*. В этом случае проверяется

только существование файла. Если он есть, то возвращается 0. Если его нет, возвращается -1, а в *errno* записывается *ENOENT*. Если один из каталогов на пути к файлу недоступен, то в *errno* помещается код *EACCESS*.

Программа, код которой показан на СЛАЙДЕ 33, с помощью функции *access()* проверяется существование файла (его имя указывается в командной строке), и определяется, разрешен ли доступ на чтение и/или запись.

Блокирование файлов

Функция *fcntl()* – это точка доступа к нескольким особенным файловым операциям. Первый аргумент – дескриптор файла, второй – код операции. Для некоторых операций используется третий, дополнительный аргумент, но мы покажем только операцию блокирования файла, которой этот аргумент не нужен.

Функция *fcntl()* позволяет программе поставить на файл блокировку чтения или записи, аналогично мьютексам, рассматривавшемся в одном из предыдущих разделов лекционного курса. Разумеется, блокировка чтения ставится на файл, доступный для чтения, а блокировка записи – на файл, доступный для записи. Несколько процессов могут удерживать блокировку чтения одного и того же файла, однако только один процесс сможет поставить блокировку записи. Файл одновременно не может быть блокирован для и чтения, и записи. Следует отметить, что наличие блокировки не мешает процессам открывать файл и осуществлять чтение или запись его данных, если, конечно, они сами не попробуют вызвать функцию *fcntl()*.

Перед блокированием файла необходимо создать и проинициализировать структуру типа *flock*. В поле *l_type* следует указать константу *F_RDLCK* для блокировки чтения и *F_WRLCK* – для блокировки записи. Затем вызывается функция *fcntl()*, которой передается дескриптор файла, код операции *F_SETLCLW* и указатель на структуру типа *flock*. Если такая блокировка была сделана другим процессом, то функция *fcntl()* перейдет в режим ожидания, пока «чужая» блокировка не будет снята. СЛАЙДЫ 34 и 35.

В листинге программы, приведенном на СЛАЙДЕ 36, показаны основные манипуляции для блокирования файла. Сначала программа открывает файл, затем ставит на него блокировка. Программа ожидает нажатия клавиши Enter, после чего разблокирует и закрывает файл.

Чтобы эта функция не переходила в режим ожидания, когда блокировку поставить невозможно, следует задать в качестве кода операции константу *F_SETLCK*. Если файл уже заблокирован, то функция немедленно вернет -1.

В Linux есть системный вызов *flock()*, который тоже реализует блокировку файла, однако она не работает с сетевыми ФС, например, NFS. Наша программа, использующая функцию *fcntl()*, от этого не зависит.

Литература и дополнительные источники к Теме 8

1. Робачевский, А. Операционная система Unix, 2 изд./ А.Робачевский, С.Немнюгин, О.Стесик. – СПб.: БХВ-Петербург, 2010. – 656 с.
2. Инструменты Linux для Windows-программистов - http://rus-linux.net/nlib.php?name=/MyLDP/BOOKS/Linux-tools/index.html
3. Лав, Р. Linux. Системное программирование/ Р.Лав. – СПб.: Питер, 2008. – 416 с.
4. Файловая система NTFS - http://www.ixbt.com/storage/ntfs.html
5. Volume and File Structure of Disk Cartridges for Information Interchange - http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-107.pdf
6. Second Extended File System - http://www.nongnu.org/ext2-doc/
7. НИЗКОУРОВНЕВЫЙ ВВОД-ВЫВОД - http://www.opennet.ru/docs/RUS/zlp/005.html
8. Network File System Version 4 (nfsv4) - http://datatracker.ietf.org/wg/nfsv4/charter/