Linux API – работаем с файловой системой

Андрей Боровский

Управление хранением данных на диске – одна из самых важных задач любой ОС, настолько важных, что система DOS так и называлась – дисковая операционная система. Вероятно, читатель этой статьи, желающий стать Linux-программистом, уже знает, как устроена файловая система Linux с точки зрения пользователя. Мы рассмотрим эту систему с точки зрения программиста. Один из основополагающих принципов Unix/Linux – everything is a file – в вольном переводе означает: «файлы - наше все». Linux отображает в виде файлов не только сами файлы, но и различные типы установленных в системе устройств, а также некоторые структуры данных, создаваемые в ходе работы системы. Мы не будем останавливаться на классических приемах работы с файлами как с хранилищами данных на диске, а рассмотрим несколько специальных возможностей, предоставляемых файловой системой Unix/Linux.

Стандартная библиотека Linux glibc предоставляет нам полный набор функций для работы с файлами, а точнее даже два полных набора – системные вызовы Linux и реализованные на их основе функции стандартной библиотеки C (прежде, чем приступать к изучению примеров из этой статьи, рекомендуется ознакомиться с документацией к этой библиотеке). Может показаться, что системные вызовы более эффективны, чем библиотечные функции, но это не так. Как правило, чем меньше системных вызовов, тем быстрее будет работать ваша программа. Дело в том, что для выполнения своей работы системные вызовы переключают систему в режим ядра, а затем возвращаются в пользовательский режим. В прежних версиях Linux для платформы IA32 переключение осуществлялось с помощью прерывания int 80h, в новых, где указана архитектура i686, - с помощью специальной процессорной инструкции, появившейся в Pentium II [1]. Переключение между режимами занимает сравнительно много процессорного времени, поэтому библиотечные функции минимизируют количество системных вызовов, заставляя каждый такой вызов выполнять как можно больше полезной работы. И системные вызовы, и функции стандартной библиотеки C экспортируются библиотекой glibc. Функции для работы с файлами open(), close(), read(), write() и им подобные, использующие дескрипторы файлов, представляют собой обертки для соответствующих системных вызовов. Функции fopen(), fclose(), fread(), fwrite(), fseek(), и другие, работающие со структурой FILE, - являются частью стандартной библиотеки. Особую роль среди системных вызовов играет вызов ioctl(2), являющийся, фактически, универсальным средством управления устройствами, представленными в виде файлов (за универсальность вызов ioctl() называют швейцарским армейским ножом). Первым параметром функции ioctl() должен быть дескриптор открытого файла. Второй параметр – запрос или команда. Помимо этого, при вызове ioctl() могут передаваться дополнительные параметры, число и типы которых зависят от значения второго параметра функции. Функция ioctl() возвращает результаты вызова в переменных, переданных по ссылке или через стек (в виде возвращаемого значения). Следует отметить, что в последнее время среди разработчиков Linux наметилась тенденция на отказ от использования ioctl().

Управление файлами устройств

В качестве примера использования системных вызовов для работы с файлами устройств мы напишем программу, копирующую аудиоданные с audio-CD в wav-файлы (так называемый «риппер»). Файлы устройств, как известно, расположены в директории /dev/. В этой директории мы найдем и устройство /dev/cdrom, представляющее первый из установленных в вашей системе CD-приводов.

Прежде, чем мы приступим к написанию программы-риппера, имеет смысл обратить внимание на врезку, посвященную устройству Audio CD.

Audio CD изнутри

Запись на любом компакт-диске состоит из нескольких треков. Треки нумеруются начиная с нуля (трек 0 содержит оглавление диска). Номер трека не может превышать значение 99. На аудио CD каждый музыкальный фрагмент как правило записывается на отдельном треке. На одном и том же диске могут быть записаны как аудио-треки, так и треки данных. Аудиоданные на CD записываются в 16-битном представлении с чередующимися сэмплами для правого и левого канала, с частотой дискретизации 44.1 КГц (если вы не знаете, что такое сэмплы и частота дискретизации, не волнуйтесь, для нашего примера это не принципиально).

Запись на диске разбивается на фреймы. Каждый фрейм содержит 2352 байта. Нетрудно подсчитать, что для обеспечения указанных выше характеристик цифровой записи чтение данных должно выполняться со скоростью 75 фреймов в секунду (что и соответствует однократной скорости чтения CD-ROM). С фреймами связан и один из форматов адресации на аудио CD. Адресация осуществляется в единицах MSF - минуты, секунды, фреймы - где фрейм можно рассматривать как 1/75 секунды. Другой формат адресации, связанный с логическими блоками (LBA), используется в основном при работе с не-аудиодисками.

Работа с CD-ROM с помощью устройства /dev/cdrom обычно выполняется по следующему сценарию: открытие файла устройства, настройка параметров с помощью ioctl(2), чтение (запись) данных, закрытие устройства. Полный текст программы вы найдете на прилагаемом к журналу диске, здесь мы рассмотрим только самые интересные части, имеющие отношение к управлению устройствами-файлами. Текст программы начинается с директив включения заголовочных файлов. Файлы unistd.h и sys/fcntl.h содержат функции для работы с системными вызовами. Заголовочный файл linux/cdrom.h содержит различные константы и макросы, используемые при работе с CD-ROM, но, увы, не содержит макросов, с помощью которых можно было бы преобразовать MSF во фреймы и обратно. Мы сами определяем соответствующие функции. Мы открываем файл устройства с помощью системного вызова open(2):

```
int cdd = open("/dev/cdrom", O_RDONLY);
```

Флаг, переданный функции open, указывает, что файл открыт только для чтения. Дальнейший доступ к устройству будет выполняться с помощью полученного дескриптора cdd. В Linux 2.4.22 каждый процесс может открыть не более 1048576 дескрипторов одновременно [2]. Нашим программам этого будет вполне достаточно. Мы предполагаем, что устройство /dev/cdrom установлено в системе и работает правильно, однако, в общем случае неплохо проверить значение дескриптора, возвращенное open, на предмет ошибки (в этом случае функция возвращает -1, переменная errno содержит дополнительный код ошибки). Вызовы ioctl, связанные с воспроизведением Audio CD, приведены в таблице 1.

Вызов	Описание	Дополнительный параметр
CDROM_DRIVE_STATUS	Получение данных о состоянии устройства	константа CDSL_XXX
CDROM_DISC_STATUS	Получение	константа

	данных о диске	CDSL_XXX
CDROMREADTOCHDR	Чтение заголовка оглавления диска	структура cdrom_tochdr
CDROMREADTOCENTRY	Чтение элемента оглавления диска	структура cdrom_tocentry
CDROMSUBCHNL	Чтение данных о параметрах воспроизведения	структура cdrom_subchnl
CDROMPLAYTRKIND, CDROMPLAYMSF	Воспроизведение аудиозаписи	Структуры cdrom_ti и cdrom_msf
CDROMSTOP	Остановка воспроизведения	значение 0
CDROMPAUSE, CDROMRESUME	Приостановка, возобновление воспроизведения	значение 0
CDROMEJECT	Открытие лотка устройства	значение 0
CDROMCLOSETRAY	Закрытие лотка устройства	значение 0

Таблица 1. Вызовы ioctl, связанные с воспроизведением Audio CD

Результат запросов CDROM_DRIVE_STATUS и CDROM_DISC_STATUS возвращается не в параметре-ссылке, а как результат функции ioctl(). В качестве третьего аргумента в этих запросах выступает одна из констант CDSL_XXX, определенных в файле cdrom.h. Эти константы предназначены для работы с устройствами автоматической смены компакт-дисков (CD changers). В случае "однодискового" устройства следует использовать CDSL_CURRENT. Результатом вызова CDROM_DRIVE_STATUS могут быть значения CDS_NO_DISC (нет диска в устройстве), CDS_DRIVE_NOT_READY (устройство не готово), CDS_DISC_OK (диск обнаружен), а также некоторые другие константы из файла cdrom.h. Среди значений, возвращаемых вызовом CDROM_DISC_STATUS, следует отметить CDS_NO_DISC (см. выше) CDS_AUDIO (диск опознан как аудио) и CDS_MIXED (диск опознан как "смешанный"). Остальные значения соответствуют не-аудиодискам. Нижеследующий фрагмент программы проверяет, готов ли CD-дисковод к передаче данных:

```
disc_stat = ioctl(cdd, CDROM_DRIVE_STATUS, CDSL_CURRENT);
if ((disc_stat != CDS_DISC_OK) && (disc_stat != CDS_NO_INFO))
{
  close(cdd);
  printf("Устройство не готово\n");
  return 1;
}
```

Вызовы CDROMREADTOCHDR и CDROMREADTOCENTRY предназначены для работы с оглавлением диска. Вызов CDROMREADTOCHDR позволяет получить данные о номере первого и последнего информационных треков на диске, а вызов CDROMREADTOCENTRY - данные об отдельном треке: адрес начала трека (в формате MSF или LBA), тип трека (аудио или данные) и т.п. Вызов CDROMSUBCHNL позволяет получить информацию о текущем состоянии устройства - находится ли диск в режиме воспроизведения, и в какой позиции выполняется чтение данных. Строка программы

```
ioctl(cdd, CDROMREADTOCHDR, &toc);
```

заполняет переменную toc типа `cdrom_tochdr` данными заголовка оглавления диска. Структура `cdrom_tochdr` позволяет нам узнать количество треков на диске.

Вызов

```
ioctl(cdd, CDROMREADTOCENTRY, &entry);
```

позволяет получить информацию о заданном треке. Дополнительный параметр вызова имеет тип «указатель на структуру `cdrom_tocentry`». Перед вызовом ioctl() мы заполняем поля format (формат длительности трека) и track (номер трека) этой структуры. В этой же структуре системный вызов возвращает информацию о выбранном треке, в том числе тип трека (аудио или данные) и длительность трека. В файле cdrom.h определена константа CDROM_LEADOUT, указывающая на условный трек, расположенный после последнего трека.

Чтение данных трека выполняется с помощью вызова

```
ioctl(cdd, CDROMREADAUDIO, &rdaudio);
```

где rdaudio – структура `cdrom_read_audio`.

Наша программа считывает данные CD и записывает их в файл формата wav. Строка вызова программы должна выглядеть так (исполнимый файл названии cdripper)

```
cdripper <трек> <файл>
```

где трек – номер трека (первый трек, содержащий пользовательские данные имеет номер 1), файл – имя файла в котором будут сохранены аудиоданные в формате wav.

Принцип, согласно которому любой объект системы должен быть представлен в виде файла, приводит к тому, что даже дескрипторы файлов представлены в Linux в виде файлов. В директории /dev/fd можно увидеть файлы-ссылки и именами 0, 1, 2 и так далее. Эти файлы представляют дескрипторы файлов, открытых процессом, который читает директорию /dev/fd. Именно так, каждый процесс видит в этой директории только свои дескрипторы. Как некий артефакт из фантастического мира, директория /dev/fd выглядит по-разному в зависимости от того, кто на нее «смотрит» (этим свойством обладают также многие директории и файлы из виртуальной файловой системы /proc, которую мы рассмотрим ниже). Открытие файла ссылки из /dev/fd эквивалентно созданию дубликата дескриптора, который представляет файл. Например, вызов

```
fd = open("/dev/fd/1", 0);
```

присваивает fd дубликат дескриптора, представленного файлом /dev/fd/1 (файлы /dev/fd/0, /dev/fd/1 и /dev/fd/2 по умолчанию соответствуют стандартным потокам ввода, вывода и ошибок).

Файловая система /proc

Помимо файловой системы /dev в Linux есть еще один источник необычных файлов – файловая система /proc. С помощью этой файловой системы можно получить множество ценных сведений о состоянии различных устройств и системных объектов (модулей ядра, например) а также о выполняющихся процессах (собственно, отсюда и происходит ее название).

Сведения об устройствах понадобятся, вероятнее всего, только всяким настраивающим/диагностическим утилитам. Мы же рассмотрим некоторые элементы системы /proc, которые могут пригодиться в программах самого разного назначения. Данные о каждом процессе хранятся в специальной поддиректории директории /proc, с именем, соответствующим численному значению идентификатора процесса. В директории процесса находятся несколько файлов и поддиректорий, из которых можно почерпнуть данные о нем (см. таблицу 2)

Элемент	Тип	Содержание
cmdline	файл	Командная строка, использовавшаяся при запуске процесса.
cwd	символическая ссылка	Указывает на директорию процесса
environ	файл	Список переменных окружения для данного процесса
exe	символическая ссылка	Указывает на файл, хранящий образ процесса
fd	директория	Ссылки на файлы, используемые процессом
root	гибкая ссылка	Указывает на корень файловой системы процесса
stat	файл	Различные сведения о процессе

Таблица 2. Файлы и дочерние каталоги /proc/<PID>, позволяющие получить различную информацию о процессе.

Если вы не root, то доступ ко многим поддиректориям процессов будет вам запрещен, но к своей собственной поддиректории процесс может получить доступ всегда. Как найти свою поддиректорию? С помощью getpid(2) процесс может узнать свой идентификатор и сконструировать путь к поддиректории, но есть и более простой способ. Помимо поддиректорий с именами, соответствующими идентификаторам процессов, каждый процесс «видит» в директории /proc поддиректорию-ссылку self, которая указывает на каталог с его данными. Использование данных из директории процесса мы рассмотрим на примере небольшой программы printenv, которая распечатывает в стандартный поток вывода полный список своих переменных окружения (то же самое делает команда Linux set) .

```
include <stdio.h>

#define BUF_SIZE 0x100

int main(int argc, char * argv[])
{
  char buf[BUF_SIZE];
  int len, i;
  FILE * f;
  f = fopen("/proc/self/environ", "r");
  while((len = fread(buf, 1, BUF_SIZE-1, f)) > 0)
  {
    for (i = 0; i < len; i++) if (buf[i]==0) buf[i] = 10;
    buf[len] = 0;
    printf("%s", buf);
  }
  fclose(f);
  return 0;
}
```

Строки в файле environ разделены не символами перевода строки (имеющим код 10 или 0x0A), а нулями.

Два способа прочесть содержимое директории

Задача перечисления всех элементов директории возникает довольно часто. Стандартная библиотека Linux предоставляет два способа перечисления содержимого директорий: первый способ – с помощью функции scandir() и функций обратного вызова, второй – с использованием набора функций opendir(), readdir(), closedir(). Рассмотрим два варианта программы printdir, распечатывающей содержимое директории, переданной ей в качестве аргумента.

```
#include <stdio.h>
#include <dirent.h>


int sel(struct dirent * d)
{
  return 1; // всегда подтверждаем
}


int main(int argc, char ** argv) {
  int i, n;
  struct dirent ** entry;
  if (argc != 2)
  {
    printf("Использование: %s <директория>\n", argv[0]);
    return 0;
  }
  n = scandir(argv[1], &entry, sel, alphasort);
  if (n < 0)
  {
     printf("Ошибка чтения директории\n");
     return 1;
  }
  for (i = 0; i < n; i++)
    printf("%s inode=%i\n", entry[i]->d_name, entry[i]->d_ino);
  return 0;
}
```

Функция scandir() создает список элементов указанной директории. Ей необходимо передать указатель на функцию обратного вызова, которая, получая данные об очередном элементе, принимает решение, включать этот элемент в результирующий список. В нашем примере это функция sel(). Если при очередном вызове функция sel() вернет значение 0, соответствующий элемент директории не будет включен в конечный список. Последний параметр scandir - функция сортировки элементов директории. Мы используем функцию alphasort(), сортирующую элементы в лексикографическом порядке.

Данные об элементах директории передаются в структурах dirent. Можно было бы ожидать, что структуры типа dirent содержат много полезной информации об элементах директории, но это не так. Кроме имени файла dirent содержит номер inode для этого элемента (простым программам обычно не зачем знать номера inode, но, чтобы наш пример чем-то отличался от стандартного, мы включаем эту информацию). У структуры dirent есть еще поле d_type типа char *, но оно, как правило, содержит null.

Функция scandir() позволяет нам получить полный отсортированный список элементов директории за один вызов. У нас есть возможность использовать низкоуровневые средства, которые могут оказаться быстрее в том случае, если сортировка файлов нам не нужна. Рассмотрим второй вариант программы:

```
#include <stdio.h>
#include <dirent.h>

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

```
{
  DIR * d;
  struct dirent * entry;
  if (argc != 2)
  {
    printf("Использование: %s <директория>\n", argv[0]);
    return 0;
  }
  d = opendir(argv[1]);
  if (d == NULL)
  {
     printf("Ошибка чтения директории\n");
     return 1;
  }
  while (entry = readdir(d))
    printf("%s inode=%i\n", entry->d_name, entry->d_ino);
  closedir(d);
  return 0;
}
```

Этот вариант программы использует функции opendir(), readdir() и closedir(), которые работают с директорией как с файлом. Функция readdir() возвращает значение TRUE до тех пор, пока не будут прочитаны все элементы директории.

Разреженные файлы

Unix-системы позволяют создавать файлы, логический размер которых превышает физический. Такие файлы могут быть удобны, когда необходимо отобразить какую либо незаполненную структуру данных (например, матрицу с большим количеством нулей). Наглядным примером разреженных файлов являются знакомые всем нам файлы core.dump. Рассмотрим текст программы makehole:

```
#include <stdio.h>
#include <string.h>
#define BIG_SIZE 0x1000000
int main(int argc, char * argv[])
{
  FILE * f;
  f = fopen(argv[1], "w");
  if (f == NULL)
  {
     printf("Невозможно создать файл: %s", argv[1]);
     return 1;
  }
  fwrite(argv[1], 1, strlen(argv[1]), f);
  fseek(f, BIG_SIZE, SEEK_CUR);
  fwrite(argv[1], 1, strlen(argv[1]), f);
  fclose(f);
}
```

Если скомпилировать эту программу под именем makehole и запустить

```
makehole bighole.txt
```

то на диске будет создан файл bighole.txt. Команда ls –al сообщит нам, что размер файла составляет чуть больше 16 мегабайт (см. значение константы `BIG_SIZE` в программе). Однако, с помощью команды

```
du bighole.txt
```

мы узнаем, что на диске этот файл занимает 24 байта. Причиной появления пропусков в открытом для записи файле стало смещение с помощью функции fseek() в область после конца файла. Выход за пределы файла с помощью fseek() – стандартный метод получения разреженных файлов. В момент вызова fseek() в нашей программе позиция записи находится в конце файла. Флаг SEEK_CUR указывает, что смещение отсчитывается от текущей позиции. Таким образом, в файле образуется пропуск, величина которого в байтах соответствует значению BIG_SIZE. При чтении пустых блоков в разреженном файле функция чтения данных будет возвращать блоки, заполненные нулями.

Блокировка областей файла

Блокировка областей файла позволяет нескольким программам совместно работать с содержимым одного и того же файла, не мешая друг другу, или, точнее, мешая друг другу испортить данные. Мы рассмотрим интерфейс блокировки областей, основанный на использовании функции fcntl(2). Функция fcntl() тоже представляет собой нечто вроде швейцарского армейского ножа. С помощью этой функции можно манипулировать дескрипторами файлов и устанавливать рекомендательные (advisory) блокировки. Рекомендательными эти блокировки называются потому, что следование им является для программ, работающих с файлом, делом доброй воли. Если программа сама не использует блокировок, блокировки, установленные другими программами, не будут иметь для нее никакого эффекта. Существует возможность придать рекомендательным блокировкам fcntl() обязательный характер, но для этого соответствующая файловая система должна быть смонтирована со специальным ключом. Для изучения работы блокировок напишем программу testlocks (файл testlocks.c). При работе с блокировками во втором параметре функции fcntl() передается одна из команд управления блокировками, третий же параметр должен содержать адрес структуры flock, в которую записывается информация о блокировке (см. таблицу 3).

Поле	Значение
l_type	Тип блокировки: записи – F_RDLCK, чтения – F_WRLCK, сброс – F_UNLCK.
l_whence	Точка отсчета смещения
l_start	Начальный байт области
l_len	Длина области
l_pid	Идентификатор процесса, установившего блокировку (для GETLCK)

Таблица 3. Описание полей структуры f_lock

Для установки блокировки мы заполняем поля структуры flock необходимыми значениями и вызываем fcntl() с командой F_SETLK (установить блокировку):

```
fi.l_type = F_WRLCK;
fi.l_whence = SEEK_SET;
fi.l_start = 0;
fi.l_len = 64;
off = 0;
while (fcntl(fd, F_SETLK, &fi) == -1)
{
   fcntl(fd, F_GETLK, &fi);
   ...
   printf("байты %i - %i заблокированы процессом %i\n", off, off+64,
      fi.l_pid);
}
```

Если заданная область уже заблокирована, fcntl возвращает -1. С помощью команды F_GETLK можно узнать, идентификатор процесса, заблокировавшего

данную область. Для того, чтобы снять блокировку, мы вызываем fctnl() с командой F_SETLK (странно, не правда ли?) и параметром l_type структуры flock, равным F_UNLCK:

```
fi.l_type = F_UNLCK;
if (fcntl(fd, F_SETLK, &fi) == -1)
  printf("Ошибка разблокирования\n");
```

Скомпилируйте программу testlocks и запустите на выполнение сразу несколько экземпляров. Первый экземпляр testlocks создаст файл testlocks.txt. Каждый процесс заблокирует 64 байта в этом файле и сделает запись в заблокированную область. Второй, третий и все последующие экземпляры процессов сообщат, какие области файла уже заблокированы другими процессами. Завершить программу testlocks можно, нажав любую символьную клавишу и, затем, ввод.

Файлы Linux, – это не только удобные хранилища данных. С их помощью можно решать множество задач, начиная с управления устройствами и заканчивая разграничением доступа к ресурсам. Однако, работа с файлами – далеко не единственное, что может Linux.

Литература:

1. D. P. Bovet, M. Cesati, **Understanding the Linux Kernel, 3rd Edition,** O'Reilly, 2005
2. W. R. Stevens, S. A. Rago, **Advanced Programming in the UNIX® Environment: Second Edition,** Addison Wesley Professional, 2005