6. Работа с файловой системой (продолжение)

6.1. 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 1)

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 1. *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.

```
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.

6.2. 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 как-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.

6.3. Разреженные файлы

Unix-системы позволяют создавать файлы, логический размер которых превышает физический. Такие файлы могут быть удобны, когда необходимо следовать 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.

6.4. 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 2).

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 2. *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