Создание виртуальных файловых систем в Linux

Пешеходов А. П. aka fresco, Четверг, 25 Май 2006, 00:53

Создание виртуальных файловых систем в Linux

Документ составлен на основе статьи Jonathan Corbet "Creating virtual filesystems with libfs" (http://lwn.net/Articles/57369/).

Линус (Linus) и многочисленные разработчики ядра Linux негативно относятся к использованию системного вызова ioctl(), не без оснований считая его, по сути, неконтролируемы способом добавления совершенно нестандартных интерфейсов в ядро. Создание новых файлов в каталоге /proc так же не выглядит хорошим решением, т.к. там уже достаточно беспорядка. Разработчики, "населяю" свой код реализациями ioctl() или файлами в /proc, часто бывают обескуражены возможностью создания вместо этого обыкновенной виртуальной файловой системы. Файловые системы делают интерфейс явным и видимы в пространстве пользователя, они так же позволяют существенно упростить написание различных административны скриптов.

Ядра серии 2.6 (начиная с 2.5.7) содержат набор подпрограмм, называемый libfs, специально спроектированны для упрощения задачи написания виртуальный файловых систем. Libfs берет на себя выполнение многих стандартных для виртуальных файловых систем задач, позволяя неквалифицированым разработчикам концентрироватья только на реализации характерной для их задач функциональност.

В этой статье мы рассмотрим реализацию простой виртуальной файловой системы (lwnfs), заполненной файлами-счетчиками. Каждое чтение такого файла возвращает текущее значение счетчика и инкрементирует его:

cat /lwnfs/counter
0
cat /lwnfs/counter
1
Также возможна запись в файл числового значения, которое будет присвоено счетчику:

echo 1000 > /lwnfs/counter
cat /lwnfs/counter
1000

Общая архитектура файловой системы

Т.к. наша файловая система будет виртуальной, никаких операций работы с диском мы не предусматриваем а все данные будут располагаться в различных кэшах ядра. Для связывания

файловой системы с VFS необходимо определить некий минимальный набор функций и структур данных. Рассмотрим эти структуры.

struct file_system_type является глобальным "" файловой системы и содержит имя ФС, а также функции получения и уничтожения суперблока (lfs_type).
struct super_operations содержит набор функций работы с глобальными данными файловой системы. Здесь мы используем "", предоставляемые libfs. (lfs_s_ops).
struct file_operations определяет набор функций для работы с файлами. Для файлов счетчиков мы реализуем только три из них – open, read и write (составляют lfs_file_ops) – для каталогов же используем "" libfs.

Алгоритм работы нашей ФС таков:

Точкой входа в модуль является функция lfs_init, выполняющая регистрацию ФС вызовом register_filesystem(&lfs_type).
При монтировании ФС вызывается функция lfs_type->get_sb(), указывающая на lfs_get_super. Эта функция вызывает libfs-helper get_sb_single(), передавая ей в качестве параметра указатель на lfs_fill_super. get_sb_single() размещает структуру struct super_block, и инициализирует некоторыми разумными значениями все ее поля, вызывая в завершение lfs_fill_super().
lfs_fill_super() устанавливает некоторые значащие поля суперблока, а также создает и инициализируетс логическую структуру файловой системы – файл "counter" в корневом каталоге, и файл "subcounter" в каталоге "subdir", размещая и вставляя в соответствующие кэши VFS все необходимые структуры.

Теперь рассмотрим реализацию описанных выше механизмов более детально.

Инициализация и установка суперблока

Модуль ядра, реализующий ФС, должен, в момент загрузки, зарегистрироват файловую систему в VFS. Код инициализации lwnfs прост:

static int __init lfs_init(void) {
return register_filesystem(&lfs_type);
}

Аргумент lfs_type инициализируетс следующим образом:

static struct file_system_type lfs_type = {
.owner   = THIS_MODULE,
.name   = "lwnfs",
.get_sb   = lfs_get_super,
.kill_sb   = kill_litter_super,
};

Это основная структура данных, описывающая файловую систему в ядре (определена в linux/fs.h). Поле owner используется для управления счетчиком ссылок на модуль (необходим

для предотвращения выгрузки используемого модуля). Поле name — строка, которая будет передана вызову mount() в качестве типа ФС. Далее следуют 2 функции управления суперблоком. kill_little_super() — это generic-функция, предоставляемая VFS, она просто освобождает все внутренние структуры при размонтированииФС; т.о. авторы простых виртуальных файловых систем не должны заботится об этом аспекте (конечно, необходимо дерегистрироват ФС в момент выгрузки модуля, это будет реализовано в exit-функции модуля lwnfs).

Окончательная инициализация суперблока — задача разработчика, она уже значительно проще, нежели кодирование всех требуемых операций вручную. С этим справляется функция lfs_get_super():

static struct super_block *lfs_get_super(struct file_system_type *fst,
int flags, const char *devname, voi *data) {
return get_sb_single(fst,flags,data,lfs_fill_super);
}

Аналогично, get_sb_single() — есть generic-код, выполняющий большую часть задачи создания суперблока (выделение памяти, инициализация полей, и т.д.). Однако по завершении он вызывает lfs_fill_super(), которая производит установку специфичных для нашей ФС полей.

static int lfs_fill_super (struct super_block *sb,
void *data, int silent) {
struct inode *root;
struct dentry *root_dentry;

/* Устанавливаем поля суперблока */
sb->s_blocksize = PAGE_CACHE_SIZE;
sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
sb->s_magic = LFS_MAGIC;
sb->s_op = &lfs_s_ops;

/* Создание inode для корневого каталога */
root = lfs_make_inode (sb, S_IFDIR | 0755);
if (! root)
goto out;
root->i_op = &simple_dir_inode_operations;
root->i_fop = &simple_dir_operations;

/* Создание dentry для корневого каталога */
root_dentry = d_alloc_root(root);
if (! root_dentry)
goto out_iput;
sb->s_root = root_dentry;

/* Создание логической структуры файлов и папок */
lfs_create_files (sb, root_dentry);
return 0;

out_iput:
/* Если выделение dentry провалилось, уничтожаем
* inode и выходим */
iput(root);
out:
return -ENOMEM;
}

Функция принимает 3 аргумента; первый – указатель на конструируемый суперблок, последние 2 могут быть проигнорированы Инициализация суперблока сводится к установке размера блока, magic-идентификатора и superblock-операций, описываемых структурой super_operations. Для простой виртуальной ФС нет необходимости реализовывать все операции, определенные в этой структуре — libfs предоставит необходимые "". Достаточно установить ее так:

static struct super_operations lfs_s_ops {
.statfs = simple_statfs
.drop_inode = generic_delete_inode;
}

Проинициализироав суперблок, lfs_fill_super берется за построение корневого каталога нашей ФС. Первым делом для него создается inode – вызовом lfs_make_inode(), реализация которого будет рассмотрена ниже. Он нуждается в указателе на суперблок и аргументе mode, который задает разрешения на создаваемый файл в формате вызова stat(), маска S_IFDIR говорит функции, что мы создаем каталог, файловые и inode-операции, которые мы назначаем новому inode, взяты из libfs.

Далее для корневого каталога создается структура dentry, через которую он помещается в directory-кэш. Заметим, что суперблок имеет специальное поле, хранящее указатель на dentry корневого каталога, которое также устанавливается lfs_fill_super().
[newpage]
Создание файлов

Теперь суперблок имеет полностью работоспособныйкорневой каталог. Все реальные операции с каталогом будут обрабатываться libfs и уровнем VFS. Однако libfs не в состоянии создать что-либо интересное в корневом каталоге, и этим придется заняться нам. Перед возвратом управления функция lfs_fill_super() делает вызов lfs_create_files(), который создает и размещает логическую структуру нашей ФС. Заметим, что счетчики в нашем модуле реализованы в виде глобальных переменных типа atomic_t.

static atomic_t counter, subcounter;

static void lfs_create_files (struct super_block *sb, struct dentry *root) {
struct dentry *subdir;

/* Создаем файл "counter" в корневом каталоге */
atomic_set(&counter, 0);
lfs_create_file(sb, root, "counter", &counter);

/* Создаем каталог "subdir" */
atomic_set(&subcounter, 0);
subdir = lfs_create_dir(sb, root, "subdir");

/* Создаем файл "subcounter" в "subdir" */
if (subdir)
lfs_create_file(sb, subdir, "subcounter", &subcounter);
}

Понятно, что lfs_create_files() выполняет только инициализацию счетчиков. Реальную работу по созданию файлов и каталогов выполняют другие функции.

static struct dentry *lfs_create_file (struct super_block *sb,
struct dentry *dir, const char *name,
atomic_t *counter) {

struct dentry *dentry;
struct inode *inode;
struct qstr qname;

/* Инициализируем qstr, считаем хэш */
qname.name = name;
qname.len = strlen (name);
qname.hash = full_name_hash(name, qname.len);

/* Создаем dentry для файла */
dentry = d_alloc(dir, &qname);
if (! dentry)
goto out;

/* Создаем inode для файла */
inode = lfs_make_inode(sb, S_IFREG | 0644);
if (! inode)
goto out_dput;
inode->i_fop = &lfs_file_ops;
inode->u.generic_ip = counter;

d_add(dentry, inode);
return dentry;

out_dput:
dput(dentry);
out:
return 0;
}

В качестве аргументов эта функция принимает указатель на суперблок, dentry родительского каталога и имя создаваемого файла. Первым делом создается dentry для нового файла – вызовом d_alloc(), который принимает указатель на родительский dentry и структуру типа struct qstr, служащую для удобно представления имени файла. Эта структура, помимо самого имени и его длины, содержит также хэш, вычисляемый вызовом ful_name_hash(), по которому указанный dentry может быстро найден в кэше.

Для инициализации inode мы снова пользуемся функцией lfs_make_inode(), однако теперь мы создаем регулярный файл, о чем говорит маска S_IFREG. В inode мы используем 2 поля:

поле i_fop устанавливается указателем на структуру с файловыми операциями, реализующими чтение и запись счетчиков (lfs_file_ops).
поле u.generic_ip мы используем для хранения счетчика типа atomic_t.

Другими словами, i_fop определяет поведение данного конкретного файла, а u.generic_ip хранит специфичные для файла данные. Практически все виртуальные файловые системы используют эти 2 поля для установки требуемого поведения файла.

Последний этап создания файла – добавление его в dentry-кэш вызовом d_add(). Это позволяет VFS отыскивать файл без обращения к directory-операциям, онам – обойтись без реализации directory-операций. Вся наша файловая система находится внутри кэша ядра, модуль может не запоминать ее структуру и в состоянии обойтись без реализации операций просмотра (lookup) ФС. Это делает жизнь проще.

Каталог "subdir" создается функцией lfs_create_dir():

static struct dentry *lfs_create_dir (struct super_block *sb,
struct dentry *parent, const char *name)
{
struct dentry *dentry;
struct inode *inode;
struct qstr qname;

qname.name = name;
qname.len = strlen (name);
qname.hash = full_name_hash(name, qname.len);

dentry = d_alloc(parent, &qname);
if (! dentry)
goto out;

inode = lfs_make_inode(sb, S_IFDIR | 0644);
if (! inode)
goto out_dput;
inode->i_op = &simple_dir_inode_operations;
inode->i_fop = &simple_dir_operations;

d_add(dentry, inode);
return dentry;

out_dput:
dput(dentry);
out:
return 0;
}

Т.к. от реализации directory-операций мы отказались, поле i_fop устанавливается указателем на generic-функцию simple_dir_operations().

Создание inode

Теперь посмотрим, как работает lfs_make_inode():

static struct inode *lfs_make_inode(struct super_block *sb, int mode) {
struct inode *ret = new_inode(sb);

if (ret) {
ret->i_mode = mode;
ret->i_uid = ret->i_gid = 0;
ret->i_blksize = PAGE_CACHE_SIZE;
ret->i_blocks = 0;
ret->i_atime = ret->i_mtime = ret->i_ctime = CURRENT_TIME;
}
return ret;
}

Она просто размещает новую структуру inode (вызовом new_inode()) и заполняет ее некоторыми осмысленными значениями. Аргумент mode определяет не только права доступа к файлу, но и его тип – регулярный файл или каталог.
[newpage]
Реализация файловых операций

До этого момента мы почти не касались работы собственно счетчиков, занимаясь реализацией внутренних VFS-механизмов, необходимых для работы любой файловой системы. Пришло время посмотреть, как будет выполняться реальная работа.

Операции над счетчиками находятся в структуре file_operations, которую мы ассоциируем с inodes файлов-счетчиков:

static struct file_operations lfs_file_ops = {
.open =lfs_open,
.read =lfs_read_file,
.write =lfs_write_file,
};

Впомним, что указатель на эту структуру помещается в inode каждого файла-счетчика функцией lfs_create_file().

Простейшей операцией является open():

static int lfs_open(struct inode *inode, struct file *filp) {
filp->private_data = inode->u.generic_ip;
return 0;
}

Все, что она делает – помещает указатель на atomic_t прямо в структуру file, что несколько упрощает доступ к нему.

Интересная работа выполняется функцией read(), которая должна инктрементроват счетчик, а затем возвращать его значение в пространство пользователя. Она начинается с чтения и инкрементированя счетчика:

static ssize_t lfs_read_file(struct file *filp, char *buf,
size_t count, loff_t *offset) {

atomic_t *counter = (atomic_t *) filp->private_data;
int v, len;
char tmp[TMPSIZE];

v = atomic_read(counter);
if (*offset > 0)
v -= 1;
else
atomic_inc(counter);
len = snprintf(tmp, TMPSIZE, "%dn", v);
if (*offset > len)
return 0;
if (count > len - *offset)
count = len - *offset;

if (copy_to_user(buf, tmp + *offset, count))
return -EFAULT;
*offset += count;

return count;
}

Заметим ,что здесь возможны "" – 2 процесса могут прочитать счетчик до его инкрементированя, в результате чего одно и тоже значение счетчика будет возвращено дважды. Серьезный модуль, вероятно, попытался бы упорядочить доступ к счетчику путем применения блокировок (spinlocks), однако наша ФС предназначена только для демонстрационны целей и подобными сложностями не обременена.

Но в любом случае, мы имеем некоторое значение счетчика и должны вернуть его в пространство пользователя. Далее следует перекодированиеего в символьную форму и перемещение в пользовательски буфер, а также корректированиесмещения (seek) в файле.

Теперь рассмотрим функцию write, которая позволяет пользователям устанавливать значение счетчика:

static ssize_t lfs_write_file(struct file *filp, const char *buf,
size_t count, loff_t *offset) {

atomic_t *counter = (atomic_t *) filp->private_data;
char tmp[TMPSIZE];

if (*offset != 0)
return -EINVAL;

if (count >= TMPSIZE)
return -EINVAL;
memset(tmp, 0, TMPSIZE);
if (copy_from_user(tmp, buf, count))
return -EFAULT;

atomic_set(counter, simple_strtol(tmp, NULL, 10));
return count;
}

Эта функция копирует из пользовательскоо буфера записываемые данные в символьном виде, конвертирует их к численной форме и устанавливает счтчик.

Завершение работы модуля

Отмонтирование ФС выполняется generic-функцией kill_little_super. При выгрузке модуля вызывается lfs_exit(), выполняющая дерегистрацию файловой системы:

static void __exit lfs_exit() {
unregister_filesystem(&lfs_type);
}

Заключение

Кода libfs полностью достаточно для реализации большинства driver-specific виртуальных файловых систем. Дополнительные примеры могут быть найдены в исходниках ядра Linux-2.6:

drivers/hotplug/pci_hotplug_core.c
drivers/usb/core/inode.c
drivers/oprofile/oprofilefs.c
fs/ramfs/inode.c

Аналогичные по тематике переводы и статьи можно найти на www.fresco.h16.ru.