《操作系统》
实验指导书

南昌大学软件学院

实验 7： Linux 文件系统

7.1 实验概述

实验目的：通过使用 Linux 的文件和目录的 API，增加对存储设备及文件系统的了解，并能创建文件系统的实用工具。

实验要求：通过验证实验内容掌握 API，解决实验设计问题，提交报告。

实验语言：c

实验环境：linux、gcc

7.2 实验背景

永久存储设备永久地（或至少长时间地）存储信息，如传统硬盘驱动器（hard disk drive）或更现代的固态存储设备（solid-state storage device）。持久存储设备与内存不同。内存在断电时，其内容会丢失，而持久存储设备会保持这些数据不变。

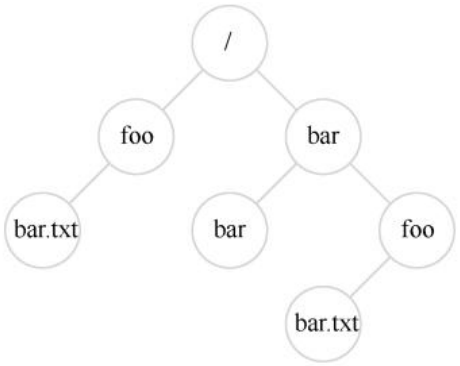
操作系统应该如何管理持久存储设备？都需要哪些 API？实现有哪些重要方面？本实验以 Linux 操作系统为对象，掌握使用其关于文件和目录的 API。

存储虚拟化形成了两个关键的抽象。第一个是文件（file）。文件就是一个线性字节数组，每个字节都可以读取或写入。每个文件都有某种低级名称（low-level name），通常是某种数字。用户通常不知道这个名字。由于历史原因，Linux 文件的低级名称通常称为 inode 号（inode number）。每个文件都有一个与其关联的 inode 号。一般来说，操作系统不太会去了解用户文件的结构和内容。

第二个抽象是目录（directory）。一个目录，像一个文件一样，也有一个低级名字（即 inode 号），但是它的内容非常具体：它包含一个（用户可读名字，低级名字）对的列表。例如，假设存在一个低级别名称为“10”的文件，它的用户可读的名称为“foo”。“foo”所在的目录因此会有条目（“foo”，“10”），将用户可读名称映射到低级名称。目录中的每个条目都指向文件或其他目录。通过将目录放入其他目录中，用户可以构建任意的目录树（directory tree，或目录层次结构，directory hierarchy），在该目录树下存储所有文件和目录。

目录层次结构从根目录（root directory）开始（在基于 UNIX 的系统中，根目录就记为“/”），并使用某种分隔符（separator）来命名后续子目录（sub-directories），直到命名所需的文件或目录。例如，如果用户在根目录

中创建了一个目录 foo，然后在目录 foo 中创建了一个文件 bar.txt，我们就可以通过它的绝对路径名（absolute pathname）来引用该文件，在这个例子中，它将是/foo/bar.txt。更复杂的目录树，请参见下图，有效目录是/、/foo、/bar、/bar/bar、/bar/foo，有效的文件是/foo/bar.txt 和/bar/foo/bar.txt。目录和文件可以具有相同的名称，只要它们位于文件系统树的不同位置（例如，图中有两个名为 bar.txt 的文件：/foo/bar.txt 和/bar/foo/bar.txt）。



这个例子中的文件名包含两部分：bar 和 txt，以句点分隔。第一部分是任意名称，而文件名的第二部分通常用于指示文件的类型（type），例如，它是 C 代码（例如，.c）还是图像（例如，.jpg），或音乐文件（例如，.mp3）。然而，这通常只是一个惯例（convention）：**一般不会强制名为 main.c 的文件中包含的数据确实是 C 源代码**。

7.3 实验内容

7.3.1 创建文件

通过系统调用 open(),并传入 O_CREAT 标志，程序可以创建一个新文件。下面是示例代码，用于在当前工作目录中创建名为“foo”的文件。

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
```

在 Linux 中，可以通过 shell 命令查看 open()的函数原型及各种参数规范：

```
man open
```

在本例中，程序创建文件（O_CREAT），只能写入该文件，因为以（O_WRONLY）这种方式打开，并且如果该文件已经存在，则首先将其截断为零字节大小，即删除所有现有内容（O_TRUNC）。

open()的返回值很重要：文件描述符（file descriptor）。文件描述符只是一个整数，是每个进程私有的，在 UNIX 系统中用于访问文件。因此，一旦文件被打开，你就可以使用文件描述符来读取或写入文件（假定 open 时设置了相应的权限）。这样，获重一个文件描述符就相当于获得了一种权限（capability），即一个不透明的句柄，让你执行某些操作。另一种看待文件描述符的方法，是将

它作为指向文件类型对象的指针。一旦你有这样的对象，就可以调用其他“方法”来访问文件，如 read()和 write()。

7.3.2 读写文件

读取一个现有的文件。如果在命令行键入，可以用 cat 程序，将文件的内容显示到屏幕上。

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

在这段代码中，将程序 echo 的输出重定向到文件 foo，然后文件中就包含单词“hello”；然后用 cat 来查看文件的内容。但是，cat 程序如何访问文件 foo？在 Linux 上，使用 strace 工具，来跟踪程序所做的系统调用。为了可读性删除了其输出的一些调用。

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6) = 6
hello
read(3, "", 4096) = 0
close(3) = 0
...
prompt>
```

首先，调用 open()以只读的方式打开 foo 文件，返回一个文件描述符，其值为 3。每个正在运行的进程已经打开了 3 个文件：标准输入（进程可以读取以接收输入），标准输出（进程可以写入以便将信息显示到屏幕），以及标准错误（进程可以写入错误消息）。这些分别由文件描述符 0、1 和 2 表示。

打开成功后，cat 使用 read()重复读取文件中的一些字节。read()的第一个参数是文件描述符，从而告诉文件系统读取哪个文件。第二个参数指向一个用于放置 read()结果的缓冲区。在上面的系统调用跟踪中，strace 显示了这时的读取结果（“hello”）。第三个参数是缓冲区的大小，在这个例子中是 4KB。对 read()的调用也成功返回，这里返回它读取的字节数（6，其中包括“hello”中的 5 个字母和一个行尾标记）。

之后针对文件描述符 1（标准输出）调用 write()，用于将单词“Hello”写到屏幕上。

程序调用 close()，传入相应的文件描述符，表明它已用完文件“foo”。该文件因此被关闭，对它的读取完成了。

7.3.3 随机定位文件读写位置

有时能够读取或写入文件中的特定偏移量是有用的。例如，如果你在文本文件上构建了索引并利用它来查找特定单词，最终可能会从文件中的某些随机（random）偏移量中读取数据。为此，将使用 lseek()系统调用。下面是函数原型：

```
off_t lseek(int fileId, off_t offset, int whence);
```

通过“man lseek”可以看到更详细的描述。

NAME
lseek - reposition read/write file offset

SYNOPSIS
```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

DESCRIPTION
The **lseek**() function repositions the offset of the open file associated with the file descriptor fd to the argument offset according to the directive whence as follows:

SEEK_SET
The offset is set to offset bytes.

SEEK_CUR
The offset is set to its current location plus offset bytes.

SEEK_END
The offset is set to the size of the file plus offset bytes.

The **lseek**() function allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a "hole") return null bytes ('\0') until data is actually written into the gap.

7.3.4 立即写入

大多数情况下，当程序调用 write()时，它只是告诉文件系统：请在将来的某个时刻，将此数据写入持久存储。出于性能的原因，文件系统会将这些写入在内存中缓冲（buffer）一段时间（例如 5s 或 30s）。在稍后的时间点，写入将实际发送到存储设备。但是，有些应用程序，如数据库管理系统（DBMS），要求能够经常强制立即写入磁盘。

UNIX 中，提供给应用程序的接口被称为 fsync(int fd)。当进程针对特定文件描述符调用 fsync()时，文件系统通过强制将所有脏（dirty）数据（即尚未写入的）写入磁盘。

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
```

```
rc = fsync(fd);
assert(rc == 0);
```

7.3.5 获取文件信息

除了文件访问之外，我们还希望文件系统能够保存关于它正在存储的每个文件的大量信息。通常将这些数据称为文件元数据（metadata）。要查看特定文件的元数据，可以使用 stat()或 fstat()系统调用。这些调用将一个路径名（或文件描述符）添加到一个文件中，并填充一个 stat 结构，如下所示：

```
struct stat {
  dev_t st_dev; /* ID of device containing file */
  ino_t st_ino; /* inode number */
  mode_t st_mode; /* protection */
  nlink_t st_nlink; /* number of hard links */
  uid_t st_uid; /* user ID of owner */
  gid_t st_gid; /* group ID of owner */
  dev_t st_rdev; /* device ID (if special file) */
  off_t st_size; /* total size, in bytes */
  blksize_t st_blksize; /* blocksize for filesystem I/O */
  blkcnt_t st_blocks; /* number of blocks allocated */
  time_t st_atime; /* time of last access */
  time_t st_mtime; /* time of last modification */
  time_t st_ctime; /* time of last status change */
};
```

可以看到有关于每个文件的大量信息，包括其大小（以字节为单位），其低级名称（即 inode 号），一些所有权信息以及有关何时文件被访问或修改的一些信息，等等。要查看此信息，可以使用命令行工具 stat：

```
prompt> echo hello > file
prompt> stat file
 File: 'file'
 Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ remzi) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

7.3.6 创建目录

要创建目录，可以用系统调用 mkdir()。同名的 mkdir 程序可以用来创建这样一个目录。

```
prompt> strace mkdir foo
...
mkdir("foo", 0777) = 0
```

```
...
prompt>
```

这样的目录创建时，它被认为是“空的”，尽管它实际上包含最少的内容。具体来说，空目录有两个条目：一个引用自身的条目，一个引用其父目录的条目。前者称为“.”（点）目录，后者称为“..”（点-点）目录。你可以通过向程序 ls 传递一个标志（-a）来查看这些目录：

```
prompt> ls -a
./ ../
prompt> ls -al
total 8
drwxr-x--- 2 remzi remzi 6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

7.3.7 读取目录

打印目录内容的示例程序。

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%d %s\n", (int)d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

该程序使用了 opendir()、readdir()和 closedir()这 3 个调用来完成工作，使用循环一次读取一个目录条目，并打印目录中每个文件的名称和 inode 编号。在 struct dirent 数据结构中，展示了每个目录条目中可用的信息：

```
struct dirent {
  char d_name[256]; /* filename */
  ino_t d_ino; /* inode number */
  off_t d_off; /* offset to the next dirent */
  unsigned short d_reclen; /* length of this record */
  unsigned char d_type; /* type of file */
};
```

7.3.8 硬链接

在 Linux 中，多个文件名指向同一索引节点是存在的。比如：A 是 B 的硬链接（A 和 B 都是文件名），则 A 的元数据项中的 inode 号与 B 的 inode 号相同，即一个 inode 对应两个不同的文件名，两个文件名指向同一个文件，A 和 B 对文件系统来说是完全平等的。删除其中的一个不会影响另外一个的访问。这种方式

可以让一个文件拥有多个有效路径名，这样用户就可以建立硬链接到重要文件，以防止“误删”。

命令行程序 ln 用于执行此操作：

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
prompt> ls -i file file2
67158084 file
67158084 file2
```

注意，此操作在 vmware 中与 windows 共享的文件上不被支持。

因此，在删除文件时，并不能简单地移除文件，需要考虑链接这个因素。通过观察删除程序 rm 的执行，发现其秘密。

```
prompt> strace rm foo
...
unlink("foo") = 0
...
```

文件有一个基于 inode 号中的引用计数（reference count），该引用计数（有时称为链接计数，link count）允许文件系统跟踪有多少不同的文件名已链接到这个 inode。调用 unlink()时，会删除人类可读的名称（正在删除的文件）与给定 inode 号之间的“链接”，并减少引用计数。只有当引用计数达到零时，文件系统才会释放 inode 和相关数据块，从而真正“删除”该文件。在 stat 程序执行的结果中，可以观察到这个引用计数。

7.3.9 符号链接

符号链接（symbolic link），有时称为软链接（soft link），类似于 Windows 的快捷方式。

在符号链接中，文件实际上是一个文本文件，其中包含的有另一文件的位置信息。比如：A 是 B 的软链接（A 和 B 都是文件名），A 的元数据项中的 inode 号与 B 的 inode 号不相同，A 和 B 指向的是两个不同的 inode，继而指向两块不同的数据块。但是 A 的数据块中存放的只是 B 的路径名（可以根据这个找到 B 的位置）。A 和 B 之间是“主从”关系，如果 B 被删除了，A 仍然存在（因为两个是不同的文件），但指向的是一个无效的链接（悬空引用，dangling reference）。

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

```
prompt> stat file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...
prompt> ls -al
drwxr-x--- 2 remzi remzi 29 May 3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ../
-rw-r----- 1 remzi remzi 6 May 3 19:10 file
lrwxrwxrwx 1 remzi remzi 4 May 3 19:10 file2 -> file
```

仔细观察 ls 输出的长格式的第一个字符，可以看到常规文件最左列中的第一个字符是“-”，目录是“d”，软链接是“l”。还可以看到符号链接的大小（本例中为 4 个字节），以及链接指向的内容（名为 file 的文件）。

7.3.10 创建并挂载文件系统

在指定的分区上创建一个文件系统，Linux 提供了一个名为 mkfs 的工具。它的执行思路：为该工具提供一个设备（例如磁盘分区，例如/dev/sda1），一种文件系统类型（例如 ext3），它就在该磁盘分区上写入一个空文件系统，从根目录开始。

一旦创建了这样的文件系统，就需要在统一的文件系统树中进行访问。这个任务是通过 mount 程序实现的(它使底层系统调用 mount()完成实际工作)。mount 的作用很简单：以现有目录作为目标挂载点（mount point），本质上是将新的文件系统粘贴到目录树的这个点上。

想象一下，我们有一个未挂载的 ext3 文件系统，存储在设备分区/dev/sda1 中，它的内容包括：一个根目录，其中包含两个子目录 a 和 b，每个子目录依次包含一个名为 foo 的文件。假设希望在挂载点/home/users 上挂载此文件系统。我们会输入以下命令：

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

如果成功，mount 就让这个新的文件系统可用了。但是，请注意现在如何访问新的文件系统。要查看那个根目录的内容，我们将这样使用 ls：

```
prompt> ls /home/users/
a b
```

路径名/home/users/现在指的是新挂载目录的根。同样，我们可以使用路径名 /home/users/a 和 /home/users/b 访问文件 a 和 b。最后，可以通过 /home/users/a/foo 和/home/users/b/foo 访问名为 foo 的两个文件。因此 mount 的美妙之处在于：它将所有文件系统统一到一棵树中，而不是拥有多个独立的文件系统，这让命名统一而且方便。

要查看系统上挂载的内容，以及在哪些位置挂载，只要运行 mount 程序。

```
/dev/sda1 on / type ext3 (rw)
```

```
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

这里展示了许多不同的文件系统，包括 ext3（标准的基于磁盘的文件系统）、proc 文件系统（用于访问当前进程信息的文件系统）、tmpfs（仅用于临时文件的文件系统）和 AFS（分布式文件系统）。

7.4 实验设计

1、在 7.3.5 中我们看到了 Linux 提供的 stat 程序，请编写您自己版本的命令行程序 mystat，实现对给定文件或目录进行信息读取分析，并打印出文件大小、分配的块数、引用（链接）计数等。【接口：stat()】

2、编写一个命令行程序 myls（如同 Linux 提供的 ls 程序），列出指定目录内容。如果没有传参数，则程序仅输出指定目录中的文件名。当传入 -l 参数时，程序需要打印出文件的所有者，所属组权限以及 stat() 函数获得的一些其他信息。另外还要支持传入要读取的目录作为参数，比如 myls -l directory。如果没有传入目录参数，则用当前目录作为默认参数。【接口：stat()、opendir()、readdir()和 getcwd()】