LAB 4: FILE SYSTEMS

Each file is referenced by an inode, which is addressed by a filesystem-unique numerical value known as an inode number. An inode is both a physical object located on the disk of a Unix-style filesystem and a conceptual entity represented by a data structure in the Linux kernel. The inode stores the metadata associated with a file, such as a file's access permissions, last access timestamp, owner, group, and size, as well as the location of the file's data.

The file name is not available in an inode is the file's name and it is stored in the directory entry.

Obtain the inode number for a file using the -i flag to the ls command:
$ ls -i

To obtain information of files in the current directory in detail, use -il:
$ ls -il

To display the filesystem inode space information, the df command could be used:
$ df -i

Stat functions

Linux provides a family of functions for obtaining the metadata of a file:

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

int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *path, struct stat *buf);
```

Each of these functions returns information about a file.
stat() returns information about the file denoted by the path, while fstat() returns information about the file represented by the file descriptor fd.

lstat() is identical to stat(), except that in the case of a symbolic link, lstat() returns information about the link itself and not the target file.

Each of these functions stores information in a stat structure, which is provided by the user. The stat structure is defined in <bits/stat.h>, which is included from <sys/stat.h>:

```
struct stat {
dev_t st_dev; /* ID of device containing file */
ino_t st_ino; /* inode number */
mode_t st_mode; /* permissions */
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; /* last access time */
time_t st_mtime; /* last modification time */
time_t st_ctime; /* last status change time */
};
```

A file is mapped to its inode by its parent directory.

The inode number of / (topmost) directory is fixed, and is always 2.

$ stat /
File: '/'
Size: 4096 Blocks: 8 IO Block: 4096 directory
Device: 806h/2054d Inode: 2 Links: 27
Access: (0755/drwxr-xr-x) Uid: (0/ root) Gid: (0/ root)
Access: 2019-12-07 01:40:01.565097799 +0100
Modify: 2019-12-07 01:27:33.651924301 +0100
Change: 2019-12-07 01:27:33.651924301 +0100
Birth:

The fields are as follows:
• The st_dev field describes the device node on which the file resides. If the file is not backed by a device—for example, if it resides on an NFS volume—this value is 0.

• The st_ino field provides the file's inode number.

• The st_mode field provides the file's mode bytes, which describe the file type (such as a regular file or a directory) and the access permissions.

• The st_nlink field provides the number of hard links pointing at the file. Every file on a filesystem has at least one hard link. The st_uid field provides the user ID of the user who owns the file.

• The st_gid field provides the group ID of the group who owns the file.

• If the file is a device node, the st_rdev field describes the device that this file represents.

• The st_size field provides the size of the file, in bytes.

• The st_blksize field describes the preferred block size for efficient file I/O. This value is the optimal block size for user-buffered I/O.

• The st_blocks field provides the number of filesystem blocks allocated to the file. This value multiplied by the block size will be smaller than the value provided by st_size if the file has holes.

• The st_atime field contains the last file access time. This is the most recent time at which the file was accessed.

• The st_mtime field contains the last file modification time—that is, the last time the file was written to.

• The st_ctime field contains the last file change time. The field contains the last time that the file's metadata (for example, its owner or permissions) was changed.

Return Values

On success, all three calls return 0 and store the file's metadata in the provided stat structure. On error, they return −1 and set errno to one of the following:

EACCES
The invoking process lacks search permission for one of the directory components of path (stat() and lstat() only).

EBADF
fd is invalid (fstat() only).

EFAULT
path or buf is an invalid pointer.

ELOOP
path contains too many symbolic links (stat() and lstat() only).

ENAMETOOLONG
path is too long (stat() and lstat() only).

ENOENT
A component in path does not exist (stat() and lstat() only).

ENOMEM
There is insufficient memory available to complete the request.

ENOTDIR
A component in path is not a directory (stat() and lstat() only).

Program that uses stat() to retrieve the size of a file provided on the command line:

```
/* Filename: stat.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
struct stat sb;
int ret;
if (argc < 2) {
fprintf (stderr, "usage: %s <file>\n", argv[0]);
return 1;
}
ret = stat (argv[1], &sb);
if (ret) {
perror ("stat");
return 1;
}
printf ("%s is %ld bytes\n", argv[1], sb.st_size);
return 0;
}
```

Here is the result of running the program on its own source file:

```
$ ./stat stat.c
stat.c is 392 bytes
```

Program to find the file type (such as symbolic link or block device node) of the file given by the first argument to the program:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
struct stat sb;
```

```
int ret;
if (argc < 2) {
fprintf (stderr, "usage: %s <file>\n", argv[0]);
return 1;
}
ret = stat (argv[1], &sb);
if (ret) {
perror ("stat");
return 1;
}
printf ("File type: ");
switch (sb.st_mode & S_IFMT) {
case S_IFBLK:
printf("block device node\n");
break;
case S_IFCHR:
printf("character device node\n");
break;
case S_IFDIR:
printf("directory\n");
break;
case S_IFIFO:
printf("FIFO\n");
break;
case S_IFLNK:
printf("symbolic link\n");
break;
case S_IFREG:
printf("regular file\n");
break;
case S_IFSOCK:
printf("socket\n");
break;
default:
printf("unknown\n");
break;
}
return 0;
}
```

Links

Each name-to-inode mapping in a directory is called a link. A link is essentially just a name in a list (a directory) that points at an inode. A single inode (and thus a single file) could be referenced from, say, both /etc/customs and /var/run/ledger. However, because links map to inodes and inode numbers are specific to a particular filesystem, /etc/customs and /var/run/ledger must both reside on the same filesystem. Within a single filesystem, there can be a large number of links to any given file. The only limit is in the size of the integer data type used to hold the number of links. Among various links, no one link is the “original” or the “primary” link. All of the links get the same status, pointing at the same file and called as hard links. Files can have 0, 1, or many links. Most files have a link count of 1 i.e. they are pointed at by a single directory entry, but some files have 2 or even more links. Files with a link count of 0 have no corresponding directory entries on the filesystem. When a file’s link count reaches 0, the file is marked as free, and its disk blocks are made available for reuse.5 Such a file, however, remains on the filesystem if a process has the file open. Once no process has
the file open, the file is removed.

The link() system call standardized by POSIX, creates a new link for an existing file:

```
#include <unistd.h>
int link (const char *oldpath, const char *newpath);
```

A successful call to link() creates a new link under the path new path for the existing file oldpath and then returns 0. Upon completion, both the oldpath and newpath refer to the same file. There is no way to tell which was the “original” link.

On failure, the call returns −1 and sets errno to one of the following:

EACCES
The invoking process lacks search permission for a component in oldpath, or the invoking process does not have write permission for the directory containing newpath.

EEXIST
newpath already exists—link() will not overwrite an existing directory entry.

EFAULT
oldpath or newpath is an invalid pointer.

EIO
An internal I/O error occurred (this is bad!).

ELOOP
Too many symbolic links were encountered in resolving oldpath or newpath.

EMLINK
The inode pointed at by oldpath already has the maximum number of links pointing at it.

ENAMETOOLONG
oldpath or newpath is too long.

ENOENT
A component in oldpath or newpath does not exist.

ENOMEM
There is insufficient memory available to complete the request.

ENOSPC
The device containing newpath has no room for the new directory entry.

ENOTDIR
A component in oldpath or newpath is not a directory.

EPERM
The filesystem containing newpath does not allow the creation of new hard links, or oldpath is a directory.

EROFS
newpath resides on a read-only filesystem.

EXDEV
newpath and oldpath are not on the same mounted filesystem. (Linux allows a single filesystem to be mounted in multiple places, but even in this case, hard links cannot be created across the mount points.)

This example creates a new directory entry, pirate, that maps to the same inode (and thus the same file) as the existing file privateer, both of which are in /home/nayan:

```
int ret;
/* create a new directory entry, '/home/nayan/somecontents', that points at the same inode as '/home/nayan/contents' */

ret = link ("/home/ nayan/somecontents", /home/nayan/contents");
if (ret)
perror ("link");
```

Symbolic Links

Symbolic links, also known as symlinks or soft links, are similar to hard links in that both point at files in the filesystem. The symbolic link differs, however, in that it is not merely an additional directory entry, but a special type of file altogether. This special file contains the pathname for a different file, called the symbolic link's target. At runtime, on the fly, the kernel substitutes this pathname for the symbolic link's pathname. Thus, whereas one hard link is indistinguishable from another hard link to the same file, it is easy to tell the difference between a symbolic link and its target file.

A symbolic link may be relative or absolute. It may also contain the special dot directory discussed earlier, referring to the directory in which it is located, or the dot-dot directory, referring to the parent of this directory. Soft links, unlike hard links, can span filesystems. Symbolic links can point at files that exist or at nonexistent files. The latter type of link is called a dangling symlink. Sometimes, dangling symlinks are unwanted—such as when the target of the link was deleted, but not the symlink—but at other times, they are intentional. Symbolic links can even point at other symbolic links.

This can create loops. System calls that deal with symbolic links check for loops by maintaining a maximum traversal depth. If that depth is surpassed, they return ELOOP.

The system call for creating a symbolic link is very similar as seen above:

```
#include <unistd.h>
int symlink (const char *oldpath, const char *newpath);
```

A successful call to symlink() creates the symbolic link newpath pointing at the targetoldpath, and then returns 0. On error, symlink() returns −1 and sets errno to one of the following:

EACCES
The invoking process lacks search permission for a component in oldpath, or the invoking process does not have write permission for the directory containing new path.

EEXIST
newpath already exists—symink() will not overwrite an existing directory entry.

EFAULT
oldpath or newpath is an invalid pointer.

EIO
An internal I/O error occurred

ELOOP
Too many symbolic links were encountered in resolving oldpath or newpath.

EMLINK
The inode pointed at by oldpath already has the maximum number of links pointing at it.

ENAMETOOLONG
oldpath or newpath is too long.

ENOENT
A component in oldpath or newpath does not exist.

ENOMEM
There is insufficient memory available to complete the request.

ENOSPC
The device containing newpath has no room for the new directory entry.

ENOTDIR
A component in oldpath or newpath is not a directory.

EPERM
The filesystem containing newpath does not allow the creation of new symbolic links.

EROFS
newpath resides on a read-only filesystem.

This snippet is the same as our previous example, but it creates /home/nayan/somecontents as a symbolic link to /home/nayan/contents:

```
int ret;
/* create a symbolic link, '/home/nayan/somecontents ', that points at
'/home/nayan/contents' */

ret = symlink ("/home/nayan/somecontents", "/home/nayan/contents");
if (ret)
perror ("symlink");
```

Unlinking

The converse to linking is unlinking, the removal of pathnames from the filesystem. A single system call, unlink(), handles this task:

```
#include <unistd.h>
int unlink (const char *pathname);
```

A successful call to unlink() deletes pathname from the filesystem and returns 0. If that name was the last reference to the file, the file is deleted from the filesystem. If, however, a process has the file open, the kernel will not delete the file from the filesystem until that process closes the file. Once no process has the file open, it is deleted.

If pathname refers to a symbolic link, the link, not the target, is destroyed. If pathname refers to another type of special file, such as a device, FIFO, or socket, the special file is removed from the filesystem, but processes that have the file open may continue to utilize it.

On error, unlink() returns −1 and sets errno to one of the following error codes:

EACCES
EFAULT
EIO
EISDIR
ELOOP
ENAMETOOLONG
ENOENT
ENOMEM
ENOTDIR
EPERM
EROFS

unlink() does not remove directories. For that, applications should use rmdir.

To ease the want on destruction of any type of file, the remove() function is provided:

```
#include <stdio.h>
int remove (const char *path);
```

A successful call to remove() deletes path from the filesystem and returns 0. If path is a file, remove() invokes unlink(); if path is a directory, remove() calls rmdir().

On error, remove() returns −1 and sets errno to any of the valid error codes set by unlink() and rmdir(), as applicable.

Lab Exercises:

1. Write a program to find the inode number of an existing file in a directory. Take the input as a filename and print the inode number of the file.
2. Write a program to print out the complete stat structure of a file.
3. Write a program to create a new hard link to an existing file and unlink the same. Accept the old path as input and print the newpath.
4. Write a program to create a new soft link to an existing file and unlink the same. Accept the old path as input and print the newpath.

Additional Exercises:

1. Write a program to find the inode number of all files in a directory. Take the input as a directory name and print the inode numbers of all the files in it.
2. Write a program to print the full stat structure of a directory.