东南大學 SOUTHEAST UNIVERSITY

OPERATING SYSTEM CONCEPTS

Chapter 11. File-System Interface

A/Prof. Kai Dong

Objectives

- To explain the function of file systems.
- To describe the interfaces to file systems.
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- To explore file-system protection.

Contents

Contents

Files and Directories

- Virtualizing the persistent storage
- Two key abstractions
  - A **file** is simply a linear array of bytes, each of which you can read or write
    - Each file has some kind of low-level name, which is often referred to as its **inode number**
  - A **directory**, contains a list of entries, each of which is a (user-readable name, low-level name) pair, referring to either a file or other directory
    - Also has a low-level name (i.e., an inode number)

Files and Directories

- By placing directories within other directories, users are able to build an arbitrary directory tree (or directory hierarchy), under which all files and directories are stored.
- The directory hierarchy starts at a root directory, and uses some kind of separator to name subsequent sub-directories until the desired file or directory is named.
- A file can thus be referred to by its absolute pathname.
- One great thing provided by the file system: a convenient way to name all the files we are interested in.
- The file name often has two parts: x.y, separated by a period. The first part is an arbitrary name, whereas the second part of the file name is usually used to indicate the type of the file.
  - However, this is usually just a convention: there is usually no enforcement that the data contained in a file named main.c is indeed C source code.

Contents

File Interface

Creating Files

- The *open*() system call

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

- Creates a file called “foo” in the current working directory.
- The routine *open*() takes a number of different flags.
  - The program creates the file (*O_CREAT*), can only write to that file while opened in this manner (*O_WRONLY*), and, if the file already exists, first truncate it to a size of zero bytes thus removing any existing content (*O_TRUNC*).

File Interface

Creating Files (contd.)

- A file descriptor is what *open*() returns.
- Is just an integer, private per process, and is used in UNIX systems to access files, i.e., to call *read*() and *write*().
- We will see how to use a file descriptor.

File Interface

Reading Files

- Reading an existing file

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

- Let's trace the system calls

```
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>
```

File Interface

Reading Files (contd.)

```
open ("foo", O_RDONLY | O_LARGEFILE)     = 3
```

- Each running process already has three files open, standard input, standard output, and standard error.

```
read(3, "hello\n", 4096)                 = 6
```

- The first argument to *read*() is the file descriptor, thus telling the file system which file to read;
- The second argument points to a buffer where the result of the *read*() will be placed (the system-call trace above shows the results of the read in this spot).
- The third argument is the size of the buffer, which in this case is 4 *KB*.
- The call to *read*() returns the number of bytes it read (take into account the end-of-line marker).

File Interface

Reading Files (contd.)

```
write(1, "hello\n", 6)                    = 6
```

- The file descriptor 1 is the standard output, and thus is used to write the word “hello” to the screen as the program cat is meant to do.

```
read(3, "", 4096)                         = 0
```

- The cat program then tries to read more from the file, but since there are no bytes left in the file, the *read*() returns 0 and the program knows that this means it has read the entire file.

```
close(3)                                  = 0
```

- Close the file.

File Interface

Writing Files

- Writing a file has similar steps
  - First, a file is opened for writing;
  - Then the *write*() system call is called;
  - Perhaps repeatedly for larger files;
  - And then *close*().

File Interface

Reading / Writing NOT Sequentially

- Thus far, we've discussed how to read and write files, but all access has been sequential.
- How to read or write to a specific offset within a file.
- The *lseek*() system call

```c
off_t lseek(int fildes, off_t offset, int whence);
```

- The first argument is a file descriptor.
- The second argument is *offset*, which positions the file offset to a particular location within the file.
- The third argument, called *whence* for historical reasons, determines exactly how the seek is performed.

File Interface

Reading / Writing NOT Sequentially (contd.)

- Details about *whence*

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current location plus
    offset bytes.
If whence is SEEK_END, the offset is set to the size of the file plus
    offset bytes.
```

- Thus, we know that an open file has a current offset, which is updated in one of two ways.
  - The first is when a read or write of N bytes takes place, N is added to the current offset; thus each read or write **implicitly** updates the offset.
  - The second is **explicitly** with *lseek*, which changes the offset as specified above.

File Interface

Writing Immediately

- Most times when a program calls *write*(), it is just telling the file system: please write this data to persistent storage, at some point in the future.
- The file system, for performance reasons, will **buffer** such writes in memory for some time, then the write(s) will actually be issued to the storage device.
- How to write immediately?

File Interface

Writing Immediately (contd.)

- The *fsync*() system call.
- When a process calls *fsync*(int *fd*), the file system responds by forcing all dirty (i.e., not yet written) data to disk, for the file referred to by the specified file descriptor.

```
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);
```

- If the file is newly created, we also need to *fsync*() the directory that contains the file.
- What if something bad (power off) happens when performing *fsync*()?

File Interface

Renaming Files

- How to give a file a different name.

```
prompt> mv foo bar
```

- The *rename*() system call

```
rename(char *old, char *new)
```

  - the original name of the file (*old*)
  - the new name (*new*).
- It is implemented as an **atomic** call with respect to system crashes.
- It is critical for supporting certain kinds of applications that require an atomic update to file state.

File Interface

Renaming Files (contd.)

- Atomic update to file state

```
int fd = open("foo.txt.tmp", O_WRONLY | O_CREAT | O_TRUNC);
write(fd, buffer, size);
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

File Interface

Getting Information about Files

- File system keeps a fair amount of information about each file it is storing.
- Such data is called **metadata**.

```
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
}
```

File Interface

Getting Information about Files (contd.)

- The *stat*() or *fstat*() system calls

```
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/ kai) Gid: (30686/ kai)
Access: (2019-07-13) 15:50:20.157594748 -0500
Modify: (2019-07-13) 15:50:20.157594748 -0500
Change: (2019-07-13) 15:50:20.157594748 -0500
```

- Such information is kept in a structure called an **inode**, we will dive into it in future.

File Interface

Permission Bits

- UNIX permission bits

```
prompt> echo hello > file
prompt> ls -l foo.txt
-rw-r--r-- 1 kai kaigroup 0 Jul 13 16:29 foo.txt
```

  - Nine bits determine, for each regular file, directory, and other entities, exactly who (**owner**, **group**, **other**) can access it and how (**read**, **write**, **execute**).

- Change the file mode (*chmod*() command)

```
prompt> chmod 777 test.sh
```

File Interface

Removing Files

- Delete files: *rm*() in UNIX.
- Let's trace the system calls in program *rm*.

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

- The *unlink*() system call
  - Takes the name of the file to be removed.
  - Why not something like *remove*() or *delete*()?
    - To understand the answer to this puzzle, we must first understand more than just files, but also directories.

Contents

Directory Interface

Making Directories

- The *mkdir*() system call

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

- A newly created directory is considered “empty”, (i.e., only has “.” and “..” entries).

```
prompt> ls -a
./      ../
prompt> ls -al
total 8
drwxr-x--- 2 kai kai 6 Jul 13 16:17 ./
drwxr-x--- 26 kai kai 4096 Jul 13 16:17 ../
prompt>
```

Directory Interface

Reading Directories

- The *ls*() system call

```c
/* a ls() like program */
struct dirent {
        char            d_name[256];    // filename
        ino_t           d_no;           // 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
};

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

Directory Interface

Deleting Directories

- The program *rmdir*
- The *rmdir*() system call
- *rmdir*() has the requirement that the directory be empty (i.e., only has "." and ".." entries) before it is deleted.

Contents

Links

Hard Links

- The *link*() system call takes two arguments, an old pathname and a new one; when you "link" a new file name to an old one, you essentially create another way to refer to the same file.

```
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
prompt>
```

- The way link works is that it simply creates another name in the directory you are creating the link to, and refers it to the same inode number (i.e., low-level name) of the original file. The file is **NOT** copied in any way.

Links

Hard Links (contd.)

- Recall that removing a file is performed via *unlink*().
- On creating a file
  - First, you are making a structure (the inode) that will track virtually all relevant information about the file, including its size, where its blocks are on disk, and so forth.
  - Second, you are linking a human-readable name to that file, and putting that link into a directory.
- When unlinking a file
  - The file system checks a reference count (sometimes called the link count) within the inode number.
  - Only when the reference count reaches zero, does the file system also free the inode and related data blocks, and thus truly “delete” the file.

Links

Symbolic Links

- Hard links are somewhat limited.
  - You can't create one to a directory (for fear that you will create a cycle in the directory tree);
  - You can't hard link to files in other disk partitions (because inode numbers are only unique within a particular file system, not across file systems);
  - Etc.
- There is one other type of link that is really useful, and it is called a symbolic link or sometimes a soft link.

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

Links

Symbolic Links

- A symbolic link is actually a file itself, of a different type.

```
prompt> stat file
··· regular file ···
prompt> stat file2
··· symbolic link ···
```

- Running *ls*

```
prompt> ls -al
drwxr-x--- 2 kai kai 6 Jul 13 19:10 ./
drwxr-x--- 26 kai kai 4096 Jul 13 16:17 ../
-rw-r----- 1 kai kai 6 Jul 1319:10 file
lrwxrwxrwx 1 kai kai 4 Jul 1319:10 file2 -> file
```

  - The first character in the left-most column is a “-” for regular files, a “d” for directories, and an “l” for soft links.
  - The size of the symbolic link is 4 bytes in this case.

Links

Symbolic Links

- A dangling reference:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

- Removing the original file causes the link to point to a pathname that no longer exists.

Contents

File System Interface

Making and Mounting a File System

- How to assemble a full directory tree from many underlying file systems?
  - Make file systems (*mkfs*() command)
    - Give the tool, as input, a device (such as a disk partition, e.g., /dev/sda1) and a file system type (e.g., ext3), and it simply writes an empty file system, starting with a root directory, onto that disk partition.
  - Mount them to make their contents accessible (*mount*() command)
    - Takes an existing directory as a target mount point and essentially paste a new file system onto the directory tree at that point.

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