39. Files and Directories

Operating Systems in Three Easy Pieces

Keith Perkins

Original slides by
Youjip Won
Hanyang University

Embedded Software Systems Lab

Persistent Storage

- Keep all data **intact** even if there is a power loss.
  - Hard disk drive
  - Solid-state storage device

- Two key abstractions in the virtualization of storage
  - File
  - Directory

File

- A linear array of bytes

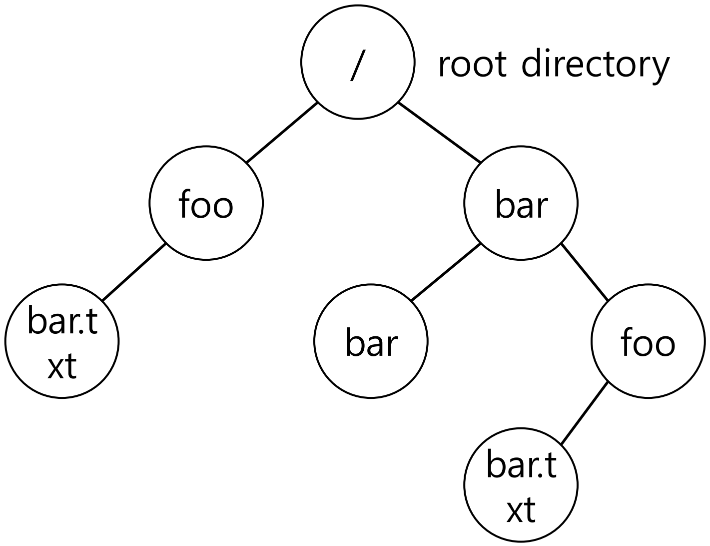
- Each file has low-level name known as an **inode number**
  - The user is not aware of this name.
- Inode numbers are unique across a file system. No 2 files will have the same one.

Directory

- Directory is like a file, and also has a low-level name.
  - It contains a list of (user-readable name, inode)pairs. Remember an inode is a low level name)
  - Each entry in a directory refers to either *files* or other *directories*.

- Example)
  - A directory has an entry (“foo”, “10”)
    - A file “foo” with the low-level name “10”

Directory Tree (Directory Hierarchy)



An Example Directory Tree

Valid files (absolute pathname) :

/foo/bar.txt
/bar/foo/bar.txt

Valid directory :

/
/foo
/bar
/bar/bar
/bar/foo/

Sub-directories

Creating Files

- Use `open()` system call with `O_CREAT` flag.

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

  - `O_CREAT` : create file.
  - `O_WRONLY` : only write to that file while opened.
  - `O_TRUNC` : make the file size zero (remove any existing content).

- `open()` system call returns **file descriptor**.
  - *File descriptor* is an integer, and is used to access files.
  - (The return value of **open**() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process. If open fails -1 is returned and the errno variable is set.)

Reading and Writing Files

- An Example of reading and writing '`foo`' file

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

  - `echo` : redirect the output of echo to the file foo
  - `cat` : dump the contents of a file to the screen

How does the `cat` program access the file foo ?

We can use `strace` to trace the system calls made by a program.

Reading and Writing Files (Cont.)

```
prompt> strace cat foo
…
open("foo", O_RDONLY|O_LARGEFILE)  = 3
read(3, "hello\n", 4096)           = 6
write(1, "hello\n", 6)             = 6 // file descriptor 1: standard out
hello
read(3, "", 4096)                  = 0 // 0: no bytes left in the file
close(3)                           = 0
…
prompt>
```

- `open` (file descriptor, flags)
  - Return file descriptor (3 in example)
  - File descriptor 0, 1, 2, is for standard input/ output/ error.
- `read` (file descriptor, buffer pointer, the size of the buffer)
  - Return the number of bytes it read
- `write` (file descriptor, buffer pointer, the size of the buffer)
  - Return the number of bytes it wrote

Reading and Writing Files (Cont.)

- Writing a file (A similar set of read steps)
  - A file is opened for writing (`open()`).
  - The `write()` system call is called.
    - Repeatedly called for larger files
  - `close()`

Writing Immediately with `fsync()`

- The file system will **buffer** writes in memory for some time.
  - Ex) 5 seconds, 30 seconds…
  - Performance reasons

- At some later point in time, write(s) will **actually be issued** to the storage device.
  - Writes seem to <u>complete quickly</u>.
  - But data can be <u>lost</u> (e.g., the machine crashes).

Writing Immediately with `fsync()` (Cont.)

- However, some applications require more than eventual guarantee.
  - Ex) DBMS requires forced writes to disk to ensure data is correctly serialized and synchronized.

- `off_t fsync(int fd)`
  - Filesystem forces all dirty (i.e., not yet written) data to disk for the file referred to by the file descriptor.
  - `fsync()` returns once all of these writes complete.

Writing Immediately with fsync() (Cont.)

- An Example of `fsync()`.

```
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);
```

Renaming Files

- `rename(char* old, char *new)`
  - Rename a file to different name.
  - Implemented as an **atomic call**.
    - Ex) Change from `foo` to `bar`:

```
prompt> mv foo bar          // mv uses the system call rename()
```

    - Ex) How to update a file atomically:

```
int fint fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Getting Information About Files

- `stat()`, `fstat()`: Show the file metadata
  - **Metadata** is information about each file.
  - Ex) Size, Low-level name, Permission, …
  - `stat` structure is below:

```
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 */
};
```

Getting Information About Files (Cont.)

- To see stat information, use the command line tool `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/ root) 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
```

  - File system keeps this type of information in an `inode` structure.

Removing Files

- `rm` is Linux command to remove a file
  - `rm` call `unlink()` to remove a file.

```
prompt> strace rm foo
…
unlink("foo")                = 0       // return 0 upon success
…
prompt>
```

Why does it call `unlink()`? not "`remove` or `delete`"
(Answer comes later)

Making Directories

- `mkdir()`: Make a directory

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

  - When a directory is created, it is empty.
  - Empty directory have two entries: . (itself), .. (parent)

```
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 ../
```

Reading Directories

- Code to read directory entries (like `ls`).

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");                // open current directory
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL)     // read one directory entry
    {
        // print outthe name and inode number of each file
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);                          // close current directory
    return 0;
}
```

- The information available within `struct dirent`

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

Deleting Directories

- `rmdir()`: Delete a directory.
  - Require that the directory be **empty**.
    - I.e., Only has "." and ".." entries.
  - If you call `rmdir()` on a non-empty directory, it will fail.

Hard Links

- `link(old pathname, new one)`
  - **Link** a new file name to an old one
  - Create another way to refer to *the same file*
  - The command-line link program : `ln`

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

Hard Links (Cont.)

- The way `link` works:
  - **Create** another name in the directory.
  - **Refer** it to the <u>same inode number</u> of the original file.
    - The file is not copied in any way.
  - Then, we now just have two different names (`file` and `file2`) that both refer to the same file.

Hard Links (Cont.)

- The result of `link()`

```
prompt> ls -i file file2
67158084 file  /* inode value is 67158084 */
67158084 file2 /* inode value is 67158084 */
prompt>
```

  - Two files have **same inode** number, but two different names (file, file2).
  - There is **no difference** between file and file2.
    - Both just links to the underlying metadata about the file.

Hard Links (Cont.)

- Thus, to remove a file, we call `unlink()`.

```
prompt> rm file
removed 'file'
prompt> cat file2           // Still access the file
hello
```

- ***reference count***
  - Track how many different file names have been linked to this inode.
  - When `unlink()` is called, the reference count decrements.
  - If the reference count reaches zero, the filesystem frees the inode and related data blocks. → truly "delete" the file

Hard Links (Cont.)

- The result of `unlink()`
  - `stat()` shows the reference count of a file.

```
prompt> echo hello > file                     /* create file*/
prompt> stat file
... Inode: 67158084 Links: 1 ...              /* Link count is 1 */
prompt> ln file file2                         /* hard link file2 */
prompt> stat file
... Inode: 67158084 Links: 2 ...              /* Link count is 2 */
prompt> stat file2
... Inode: 67158084 Links: 2 ...              /* Link count is 2 */
prompt> ln file2 file3                        /* hard link file3 */
prompt> stat file
... Inode: 67158084 Links: 3 ...              /* Link count is 3 */
prompt> rm file                               /* remove file */
prompt> stat file2
... Inode: 67158084 Links: 2 ...              /* Link count is 2 */
prompt> rm file2                              /* remove file2 */
prompt> stat file3
... Inode: 67158084 Links: 1 ...              /* Link count is 1 */
prompt> rm file3
```

Symbolic Links (Soft Link)

- Symbolic link is more **useful** than Hard link.
  - Hard Link cannot point to a directory.
  - Hard Link cannot point to a file on another partition.
    - Because inode numbers are only unique within a file system.

- Create a symbolic link: `ln -s`

```
prompt> echo hello > file
prompt> ln -s file file2  /* option -s : create a symbolic link, */
prompt> cat file2
hello
```

Symbolic Links (Cont.)

- What is different between *Symbolic link* and *Hard Link*?
  - Symbolic links are **a third type of file** the system knows about.

```
prompt> stat file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...          // Actually a file it self of a different type
```

  - The size of symbolic link (`file2`) is **4 bytes**.

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

    - A symbolic link holds the pathname of the linked-to file as the data of the link file, so it size grows as the size of the path/name grows.

Symbolic Links (Cont.)

- If we link to a longer pathname, our link file would be bigger.

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May 3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May 3 19:17 file3 -> alongerfilename
```

Symbolic Links (Cont.)

- **Dangling reference**
  - When remove a original file, symbolic link points to nothing.

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

  - Not the case for hard links (they use reference counts in the inode table)

Making and Mounting a File System

- `mkfs` tool : Make a file system
  - Write an <u>empty file system</u>, starting with *a root directory*, onto a disk partition.
  - Input:
    - A device (such as a disk partition, e.g., `/dev/sda1`)
    - A file system type (e.g., `ext3`)

Making and Mounting a File System (Cont.)

- `mount()`
  - Take an existing directory as a target **mount point**.
  - Essentially paste a new file system onto the directory tree at that point.

  - **Example)**

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

    - The pathname `/home/users/` now refers to the root of the newly-mounted directory.

Making and Mounting a File System (Cont.)

- `mount` program: show **what is mounted** on a system.

```
/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)
```

  - `ext3`: A standard disk-based file system
  - `proc`: A file system for accessing information about current processes
  - `tmpfs`: A file system just for temporary files
  - `AFS`: A distributed file system