Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T29 Files and Directories

Referência principal
Ch. 39 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 12 de novembro de 2018

Arthur João Catto, PhD

2º semestre de 2018

How to Manage Persistent Devices?

How should the OS manage a persistent device?

What are the APIs for that?

What are the most important aspects of the implementation?

Persistent Storage

- A persistent device keeps data intact even if there is a power loss.
  - Hard disk drive
  - Solid-state storage device
- In a Unix-style system there are two key abstractions in the virtualization of persistent storage
  - File
  - Directory
- From a technical point of view there is no real difference between files and directories, although they play different roles in a system.

File

- A file is simply a linear array of bytes, which can be read or written.
- Every file or directory has an **inode**, which contains all the file's metadata (that is, all the administrative data needed to read a file is stored in its inode).
- Each file also has a low-level name, often called its **inode number**.
  - Usually, the user is not aware of this name.
- The file system is responsible for storing files persistently on disk and for making them available upon request.

Inodes

- An **inode** is an entry in an **inode table**, containing information (the metadata) about a regular file or directory.
  - Note that an inode does not store the name of the file.
- The **inode table** contains the **inodes** of all files in that file system.
  - The individual inodes in the inode table have a unique number (unique to that file system), the **inode number**.

- **An inode stores metadata about a file:**
  - Type: regular file, directory, pipe, etc.
  - Permissions: read, write, execute
  - Link count: number of hard links to it
  - User ID: owner
  - Group ID: group owner
  - Size of file: or major/minor number in case of some special files
  - Time stamps: access time, change time and (inode) change time
  - Attributes: ‘immutable’ for example
  - Access control list: permissions for special users/groups
  - Link to location of file
  - . . .

Creating Files

- Creation is the most basic file operation.
- Use `open()` system call with `O_CREAT` flag.

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

  - `O_CREAT`: create file.
  - `O_WRONLY`: writing is the only operation allowed on that file.
  - `O_TRUNC`: make the file size zero (i.e. remove any existing content).
- File descriptor
  - A successful `open()` system call returns a file descriptor.
  - The descriptor is just an integer used to uniquely identify an open file of the process..

Open File Table

- **Open File Table**
  - A table indexed by file descriptors whose elements are pointers to file table entries.
  - One unique open file table is provided by the operating system for each process.
- **Open File Table Entry**
  - An in-memory structure created when a process opens a file and whose components maintain file position and other indicators.

File I/O system calls

- In a Unix-style system, there are basically 5 types of file I/O system calls:
  - Create
    - Create a new empty file.
  - Open
    - Create a new file or open a file for reading, writing or both.
  - Close
    - To tell the OS you are done with a file descriptor and close the file which is pointed by it.
  - Read
    - Read a number of bytes from a file into a memory area.
  - Write
    - Write a number of bytes from a memory area into a file.

Standard file descriptors and calls

- When any process starts, file descriptors 0, 1 and 2 are created automatically.
  - By default the corresponding entries in the process' file table refer to a file named **/dev/tty**.
  - **/dev/tty** is an in-memory surrogate for the terminal, which is a combination of keyboard and video screen.

Standard file descriptors and calls

- System calls to standard files
  - Read from stdin => read from fd 0
    - Whenever we write any character from keyboard, it read from stdin through fd 0 and save to file named /dev/tty.
  - Write to stdout => write to fd 1
    - Whenever we see any output to the video screen, it's from the file named /dev/tty and written to stdout in screen through fd 1.
  - Write to stderr => write to fd 2
    - We see any error to the video screen, it is also from that file write to stderr in screen through fd 2.

Create a file

- Syntax in C language:

```
int creat(char *filename, mode_t mode);
```

- Parameters
  - `filename`: name of file to be created
  - `mode`: access permissions to new file
- Returns
  - first unused file descriptor (generally 3 when first `creat` use in process, because 0, 1 and 2 are reserved)
  - -1 when error
- How it works in OS
  - Create new empty file on disk
  - Create file table entry
  - Set first unused file descriptor to point to file table entry
  - Return file descriptor used, -1 upon failure

Open

- Syntax in C language

```
int open (const char* Path, int flags[, int mode ]);
```

- Parameters
  - Path : path to file which you want to use
  - use absolute path beginning with "/", when you are not working in same directory of file.
  - Use relative path which is only file name with extension, when you are working in the same directory of file.

- Flags: how you would like to use the file
  - `O_RDONLY`: read only
  - `O_WRONLY`: write only
  - `O_RDWR`: read and write
  - `O_CREAT`: create file if it doesn't exist
  - `O_EXCL`: prevent file creation if it already exists

- How it works in OS
  - Find existing file on disk
  - Create file table entry
  - Set first unused file descriptor to point to file table entry
  - Return file descriptor used, -1 upon failure

Close

- Syntax in C language

```c
int close(int fd);
```

- Parameter
  - `fd`: file descriptor
- Return
  - 0 on success
  - -1 on error
- How it works in the OS
  - Destroy open file table entry referenced by element `fd`, as long as no other process is pointing to it!
  - Set element `fd` of file table to `NULL`

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(void) {
    int fd1 = open("foo.txt", O_CREAT | O_RDONLY, 0);
    if (fd1 < 0)
        { perror("c1"); exit(1); }
    printf("opened fd = %d\n", fd1);

    if (close(fd1) < 0)
        { perror("c1"); exit(1); }
    printf("closed fd = %d\n", fd1);

    int fd2 = open("bar.txt", O_CREAT | O_RDONLY, 0);
    printf("opened fd = %d\n", fd2);
    exit(0);
}
```

```
opened fd = 3
closed fd = 3
opened fd = 3
```

Read

- Syntax in C

```
size_t read(int fd, void* buf, size_t cnt);
```

- From the file indicated by `fd`, reads `cnt` bytes into the memory area indicated by `buf`.
  - If successful, it updates the access time for the file.
- Parameters
  - `fd`: file descriptor
  - `buf`: buffer to read data into
  - `cnt`: length of buffer
- Returns
  - Number of bytes read on success
  - 0 on reaching end of file
  - -1 on error
  - -1 on signal interrupt

Read

- Important points
  - `buf` must point to a valid memory location with length not smaller than the specified size to avoid overflow.
  - `fd` should be a valid file descriptor returned from `open()`
    - if `fd` is `NULL`, an error is generated.
  - `cnt` is the number of bytes to be read, while the return value is the actual number of bytes read.
    - Sometimes less than `cnt` bytes may be read.

Example of read system call

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(void) {
    int fd, sz;
    char *c = (char *) calloc(100, sizeof(char));

    fd = open("foo.txt", O_RDONLY);
    if (fd < 0)
        { perror("r1"); exit(1); }

    sz = read(fd, c, 20);
    printf("System call read(%d, c, 20) returned that"
           " %d bytes  were read.\n", fd, sz);
    c[sz] = '\0';
    printf("Those bytes were as follows: %s\n", c);
}
```

```
System call read(3, c, 20) returned that 10 bytes  were read.
Those bytes were as follows: foobar 1 2
```

What is the output of the following program?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    char c1, c2;
    int fd1 = open("foo.txt", O_RDONLY, 0);
    int fd2 = open("foo.txt", O_RDONLY, 0);
    read(fd1, &c1, 1);
    read(fd2, &c2, 1);
    printf("c1 = %c\n", c1);
    printf("c2 = %c\n", c2);
    exit(0);
}
```

What is the output of the following program?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    char c1, c2;
    int fd1 = open("foo.txt", O_RDONLY, 0);
    int fd2 = open("foo.txt", O_RDONLY, 0);
    read(fd1, &c1, 1);
    read(fd2, &c2, 1);
    printf("c1 = %c\n", c1);
    printf("c2 = %c\n", c2);
    exit(0);
}
```

- The descriptors `fd1` and `fd2` each have their own file table entry, so each descriptor has its own file position for `foobar.txt`.
  - Thus, the read from `fd2` reads the first byte of `foobar.txt`, and the output is `c = f`, not `c = o`.

Write file

- Syntax in C

```
size_t write(int fd, void* buf, size_t cnt);
```

- Writes `cnt` bytes from `buf` to the file or socket associated with `fd`.
  - `cnt` should not be greater than `INT_MAX` (defined in the `limits.h` header file).
    - If `cnt` is zero, `write()` simply returns 0 without attempting any other action.

- Parameters
  - `fd`: file descriptor
  - `buf`: buffer to write data to
  - `cnt`: length of buffer

- Returns
  - Number of bytes written on success
  - 0 on reaching end of file
  - -1 on error or signal interrupt

Write file

- Important points
  - The file must be opened for write operations
  - `buf` needs to be at least as long as specified by `cnt` to prevent overflow.
  - `cnt` is the requested number of bytes to write, while the return value is the actual number of bytes written.
    - This happens when `fd` has a smaller than `cnt` number of bytes to write.
  - If `write()` is interrupted by a signal, the effect is one of the following:
    - If `write()` has not written any data yet, it returns -1 and sets errno to `EINTR`.
    - If `write()` has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

Reading And Writing, But Not Sequentially

- An open file has a current offset which determines where the next read or write will start.
- The current offset can be updated
  - Implicitly
    - When a read or write of N bytes takes place, N is added to the current offset.
  - Explicitly
    - By calling `lseek()`.

Reading And Writing, But Not Sequentially

- Syntax in C

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

  - `fildes`: file descriptor
  - `offset`: displacement from a location within the file
  - `whence`: determine how offset is interpreted
- From the manual page…
  - 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.

Writing immediately with `fsync()`

- The file system will buffer writes in memory for some time for performance reasons
- At that later point in time, the writes will actually be issued to the storage device.
  - Write seems to complete quickly but data can be lost, e.g. if the machine crashes.
- However, some applications require more than eventual guarantee.
  - E.g. DBMS requires forced writes to disk from time to time.

```
off_t fsync(int fd)
```

  - File system forces all dirty (i.e., not yet written) data to disk for the file referred to by the file description.
  - `fsync()` returns once all of theses writes are complete.

Writing immediately with fsync()

- An example of fsync()

```c
int main(void) {
    char buffer[12] = "hello world";
    int size = strlen(buffer);
    int fd = open("foo.txt", O_CREAT | O_WRONLY | O_TRUNC);
    assert ( !(fd < 0) );
    int rc = write(fd, buffer, size);
    assert (rc == size);
    rc = fsync(fd);
    assert (rc == 0);
    return 0;
}
```

- In some cases, this may not be enough, and the code will also need to fsync() the directory that contains the file foo.txt.

Renaming Files

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

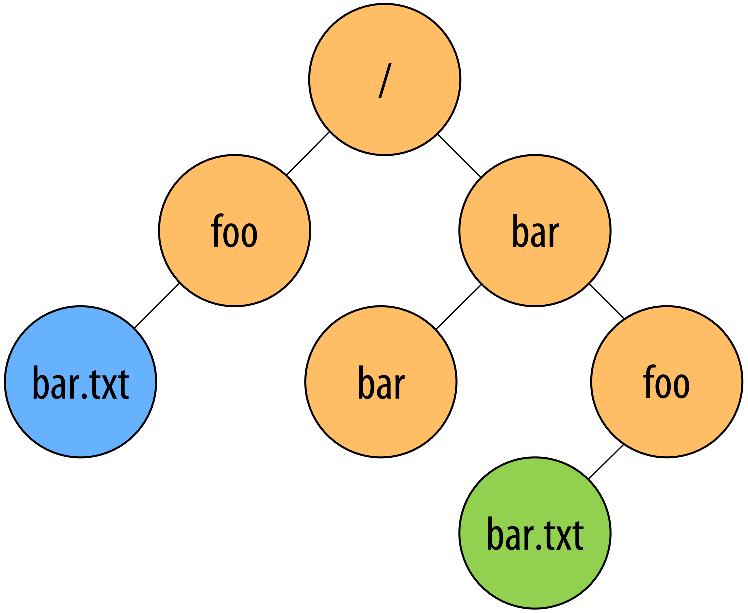
- Rename a file to a new name.
- It is implemented as an atomic call.

- Example

```
int main(void) {
    char buffer[13] = "hello world\n";
    int size = strlen(buffer);
    int fd = open("rentest.tmp", O_WRONLY | O_CREAT | O_TRUNC);
    write(fd, buffer, size); // write out new version of file
    fsync(fd);
    close(fd);
    int renres = rename("rentest.tmp", "rentest.txt");
    printf("rename returned %d\n", renres);
    return 0;
}
```

Directory

- A directory is like a file and also has a low-level name.
  - It contains a list of (user-readable name, low-level name) pairs.
  - Each entry in a directory refers to either a file or another directory.



- By placing directories within other directories, users can build an arbitrary directory tree (or directory hierarchy), under which files and other directories can be stored.

- The directory hierarchy starts at the root directory (named **/** in Unix-based systems) and uses a separator to create a chain of directory names until the desired file is reached.
  - The resulting compound name is called the file's absolute pathname.
  - The last part of the file name, named extension, usually indicates the type of the file.

Creating Directories

```
~$ mkdir tstdir
~$ cd tstdir
tstdir$ mkdir foo
tstdir$ ls
foo
tstdir$ cd foo
foo$ ls
foo$ ls -a
.  ..
foo$ ls -ail
total 0
105312409 drwxr-xr-x  2 arthur.catto  staff  64 Nov 12 17:06 .
105312376 drwxr-xr-x  3 arthur.catto  staff  96 Nov 12 17:06 ..
foo$
```

A directory is created empty.

An empty directory has two entries:
self and parent

Each associated to a different inode

Reading Directories

```c
int main(void) {
    DIR *dp = opendir(".");      // open current directory
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) { // read one directory entry
        // print out the 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;
}
```

while the information available within `struct dirent` is

```c
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 */
}
```

Hard Links

- The **identity** of a file is its inode number, not its name.
- A **hard link** is a name that references an inode.
  - If *file1* has a hard link named *file2*, then both refer to same inode.
- Creating a hard link for a file means adding a new name to an inode.
  - This can be done by the `link` function or the `ln` command.

```c
int link(const char *path1, const char *path2);
```

```
~/tstdir$ cd foo
~/tstdir/foo$ ls
~/tstdir/foo$ echo "hello world" > file1
~/tstdir/foo$ cat file1
hello world
~/tstdir/foo$ ln file1 file2
~/tstdir/foo$ cat file2
hello world
~/tstdir/foo$ rm file1
~/tstdir/foo$ cat file2
hello world
~/tstdir/foo$
```

Hard Links

- This is how `link(file1, file2)` works
  - Create another name in the directory.
  - Refer it to the same inode number of the original file.
    - The file is not copied in any way.
  - Then, we now just have two human names (`file1` and `file2`) that both refer to the same file.

Hard Links

- This is the result of `link()`

```
1.  ~/tstdir/foo$ ls -il file1 file2
2.  105342447 -rw-r--r--  2 arthur.catto  staff  12 Nov 12 19:21 file1
3.  105342447 -rw-r--r--  2 arthur.catto  staff  12 Nov 12 19:21 file2
4.  ~/tstdir/foo$
```

- Two files have the same inode number, but two human names.
  - There is no difference between `file1` and `file2`.
  - Both just link to the underlying metadata about the file.

Symbolic Links

- A **symbolic link** is a separate file whose contents point to the linked-to file.
  - To create a symbolic link, use the `ln` command with the option `-s`.

- Symbolic links are more flexible than hard links.
  - We cannot create a hard link to a directory.
  - We cannot create a hard link to a file in another partition.
  - Can you tell why?
    - Because inode numbers are only unique within a file system.

Symbolic links

```
~/tstdir/foo$ ls
file1 file2
~/tstdir/foo$ ln -s file1 file3
~/tstdir/foo$ ls
file1 file2 file3
~/tstdir/foo$ ls -l
total 16
-rw-r--r--  2 arthur.catto  staff  12 Nov 12 19:21 file1
-rw-r--r--  2 arthur.catto  staff  12 Nov 12 19:21 file2
lrwxr-xr-x  1 arthur.catto  staff   5 Nov 12 19:48 file3 -> file1
~/tstdir/foo$ ls -il
total 16
105342447 -rw-r--r--  2 arthur.catto  staff  12 Nov 12 19:21 file1
105342447 -rw-r--r--  2 arthur.catto  staff  12 Nov 12 19:21 file2
105350240 lrwxr-xr-x  1 arthur.catto  staff   5 Nov 12 19:48 file3 -> file1
```

Symbolic links

```
~/tstdir/foo$ cat file1
hello world
~/tstdir/foo$ cat file2
hello world
~/tstdir/foo$ cat file3
hello world
~/tstdir/foo$ ls -ails
total 16
105312409 0 drwxr-xr-x  5 arthur.catto  staff  160 Nov 12 19:48 .
105312376 0 drwxr-xr-x  3 arthur.catto  staff   96 Nov 12 17:06 ..
105342447 8 -rw-r--r--  2 arthur.catto  staff   12 Nov 12 19:21 file1
105342447 8 -rw-r--r--  2 arthur.catto  staff   12 Nov 12 19:21 file2
105350240 0 lrwxr-xr-x  1 arthur.catto  staff    5 Nov 12 19:48 file3 -> file1
~/tstdir/foo$
```

Symbolic links

```
~/tstdir/foo$ echo "hello world" > alongerfilenameA
~/tstdir/foo$ ln -s alongerfilenameA fileB
~/tstdir/foo$ ls -ails
total 24
105312409 0 drwxr-xr-x  9 arthur.catto  staff  288 Nov 12 20:12 .
105312376 0 drwxr-xr-x  3 arthur.catto  staff   96 Nov 12 17:06 ..
105354762 8 -rw-r--r--  1 arthur.catto  staff   12 Nov 12 20:12 alongerfilenameA
105354350 0 lrwxr-xr-x  1 arthur.catto  staff    5 Nov 12 20:10 areallyveryverylongnewname -> file1
105354151 0 lrwxr-xr-x  1 arthur.catto  staff    5 Nov 12 20:08 averylongnewname -> file1
105342447 8 -rw-r--r--  2 arthur.catto  staff   12 Nov 12 19:21 file1
105342447 8 -rw-r--r--  2 arthur.catto  staff   12 Nov 12 19:21 file2
105350240 0 lrwxr-xr-x  1 arthur.catto  staff    5 Nov 12 19:48 file3 -> file1
105354911 0 lrwxr-xr-x  1 arthur.catto  staff   16 Nov 12 20:12 fileB -> alongerfilenameA
~/tstdir/foo$
```

Removing files

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

```
~/tstdir/foo$ rm fileB
~/tstdir/foo$ rm file2
~/tstdir/foo$ ls -ails
total 16
105312409 0 drwxr-xr-x  7 arthur.catto  staff  224 Nov 12 20:18 .
105312376 0 drwxr-xr-x  3 arthur.catto  staff   96 Nov 12 17:06 ..
105354762 8 -rw-r--r--  1 arthur.catto  staff   12 Nov 12 20:12 alongerfilenameA
105354350 0 lrwxr-xr-x  1 arthur.catto  staff    5 Nov 12 20:10 areallyveryverylongnewname -> file1
105354151 0 lrwxr-xr-x  1 arthur.catto  staff    5 Nov 12 20:08 averylongnewname -> file1
105342447 8 -rw-r--r--  1 arthur.catto  staff   12 Nov 12 19:21 file1
105350240 0 lrwxr-xr-x  1 arthur.catto  staff    5 Nov 12 19:48 file3 -> file1
~/tstdir/foo$
```

  - Reference count
    - Tracks how many different file names have been linked to this inode.
    - When `unlink()` is called, the reference count is decremented.
    - If the reference count reaches zero, the file system frees the inode and related data blocks, actually deleting the file.

Deleting Directories

- To delete a directory we use `rmdir()`.
  - It requires the directory to be empty.
  - If we try to remove a non-empty directory, the call will fail.