39. Files and Directories

Operating System: Three Easy Pieces

Persistent Storage

- Keep a 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

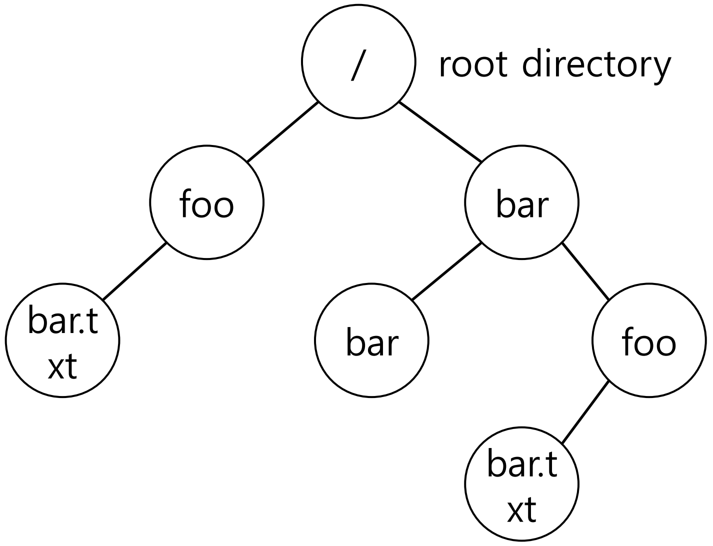
- A linear array of bytes
- Each file has low-level “name” called **i-node number**
  - The user is not aware of this name.
- File-system has a responsibility to store data persistently on disk.

Directory

- Directory is like a file, 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 **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` : if exist, 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, e.g., xv6:

```
struct proc {
  ...
  struct file *ofile[NOFILE]; // Open files
  ...
};
```

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 name, 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 write

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()`

Reading And Writing, But Not Sequentially

- An open file has a **current offset**.
  - Determine **where** the next read (or write) will begin reading from (or writing) to within the file.

- Update the current offset
  - **Implicitly**: A read or write of `N` bytes takes place; `N` is added to the current offset.
  - **Explicitly**: `lseek()`

Reading And Writing, But Not Sequentially (Cont.)

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

- `fildes` : File descriptor
- `offset` : Position the file offset to a particular location within the file
- `whence` : Determine how the seek is performed

From the man 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.
```

```
struct file {  //xv6 open files keep on proc
  int ref; //ref count
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

Shared File Table Entries

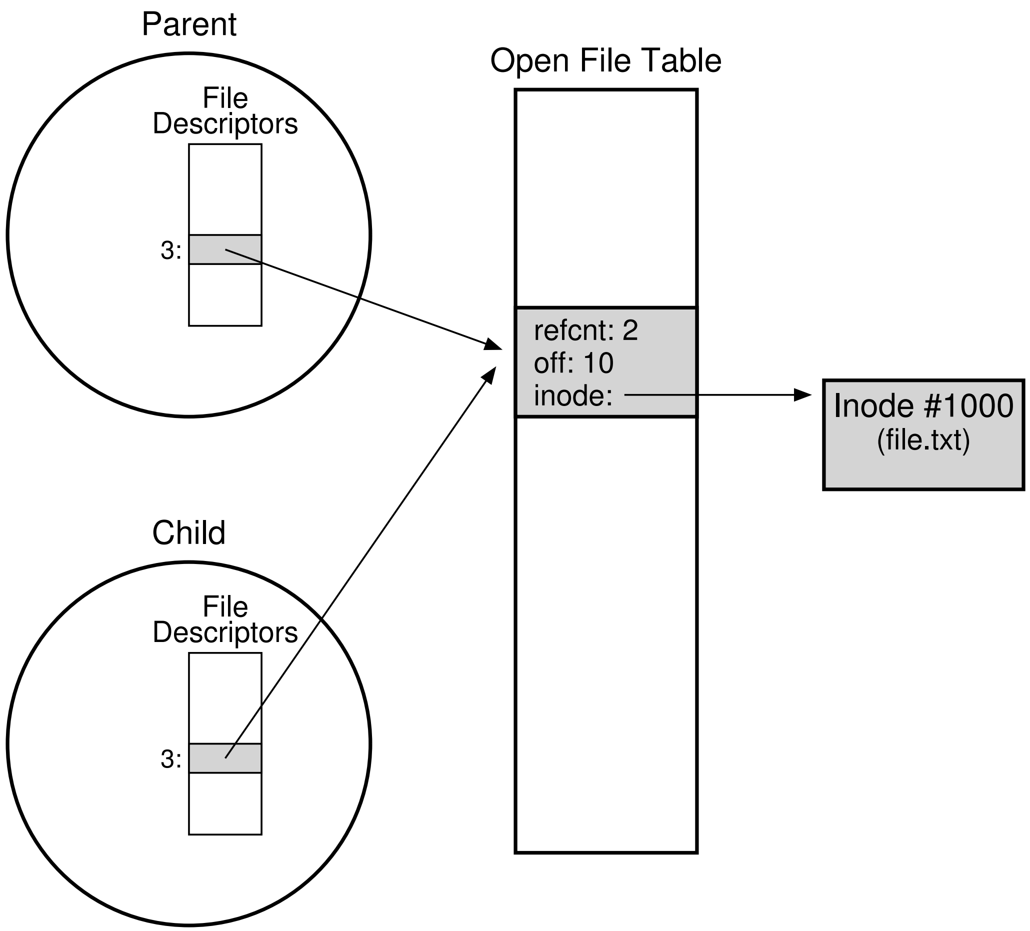
- A file can be opened by many process concurrently and independently?
  - Each one will have a "private" descriptor

- What happens if parent opens a file before the fork()?

```
int main(int argc, char *argv[]) {
 int fd = open("file.txt", O_RDONLY);
 assert(fd >= 0);
 int rc = fork();
 if (rc == 0) {
   rc = lseek(fd, 10, SEEK_SET);
   printf("child: offset %d\n", rc);
 } else if (rc > 0) {
   (void) wait(NULL);
    printf("parent: offset %d\n",(int) lseek(fd, 0, SEEK_CUR));
  }
  return 0;
}
```

Parent entry is affected by children

```
prompt> ./fork-seek
child: offset 10
parent: offset 10
prompt>
```



Sharing descriptors dup/2()

- Usefull to improve performance (e.g., output redirection dup ”1”)

```
int main(int argc, char *argv[])
{
    int fd = open(”OUTFILE.txt", O_APPEND|O_WRONLY);
    assert(fd >= 0);
    dup2(fd,1); // now STDOUT is redirected to OUTFILE.txt
    return 0;
}
```

Writing Immediately with `fsync()`

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

- At that later point in time, the write(s) will **be issued** to the storage device.
  - Write seem to <u>complete quickly</u>.
  - 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 force writes to disk from time to time (ordering constraints)
- `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 description.
  - `fsync()` returns once all theses writes are complete.
- A softer version `flush()`: `fsync()` flush OS buffers into permanent storage, `flush()` user-space level buffers into OS (e.g. glibc)

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);
```

  - In some cases, this code needs to `fsync()` the directory that contains the file `foo`. Application-level bugs can arise otherwise.

Renaming Files

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

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

    - Ex) Sync and update a file name 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, you can 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 it calls `unlink()`? not “`remove` or `delete`”
We can get the answer 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

- A sample 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 outnthe 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[];             /* 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**.
  - If you call `rmdir()` to a non-empty directory, it will fail.
    - I.e., Only has "." and ".." entries.

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 human 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 human name (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 free 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 be created to a directory.
    - Because cycles
  - Hard Link cannot be created to a file to other 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** the file system knows about.

```
prompt> stat file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...         // Actually, a file itself 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.

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 an original file, symbolic link points noting.

```
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
```

Permissions

- Like other abstractions (CPU, mem), we need protections

- File permissions implement such protections

```
prompt> ls -l foo.txt

-rw-r--r-- 1 remzi wheel 0 Aug 24 16:29 foo.txt
```

- `chmod`, `chown`, `chgrp`, allows to change those bits (metadata)

- In modern fs *9-bit* solution is too coarse: Access Control Lists (ACL). e.g. AFS case:

```
prompt> fs listacl private

Access list for private is

Normal rights:
system:administrators rlidwka

remzi rlidwka
```

```
prompt> fs setacl private/ andrea rl
```

l -- Lookup
r -- Read
w -- Write
k -- Lock
i -- Insert
d -- Delete
a -- Administer

Making and Mounting a File System

- `mkfs` tool : Make a file system
  - Write an empty file system, starting with *a root directory*, on to 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

- This lecture slide set is used in AOS course at University of Cantabria by V.Puente. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)