Files and Directories

Chapter 39

Previously in CS212…

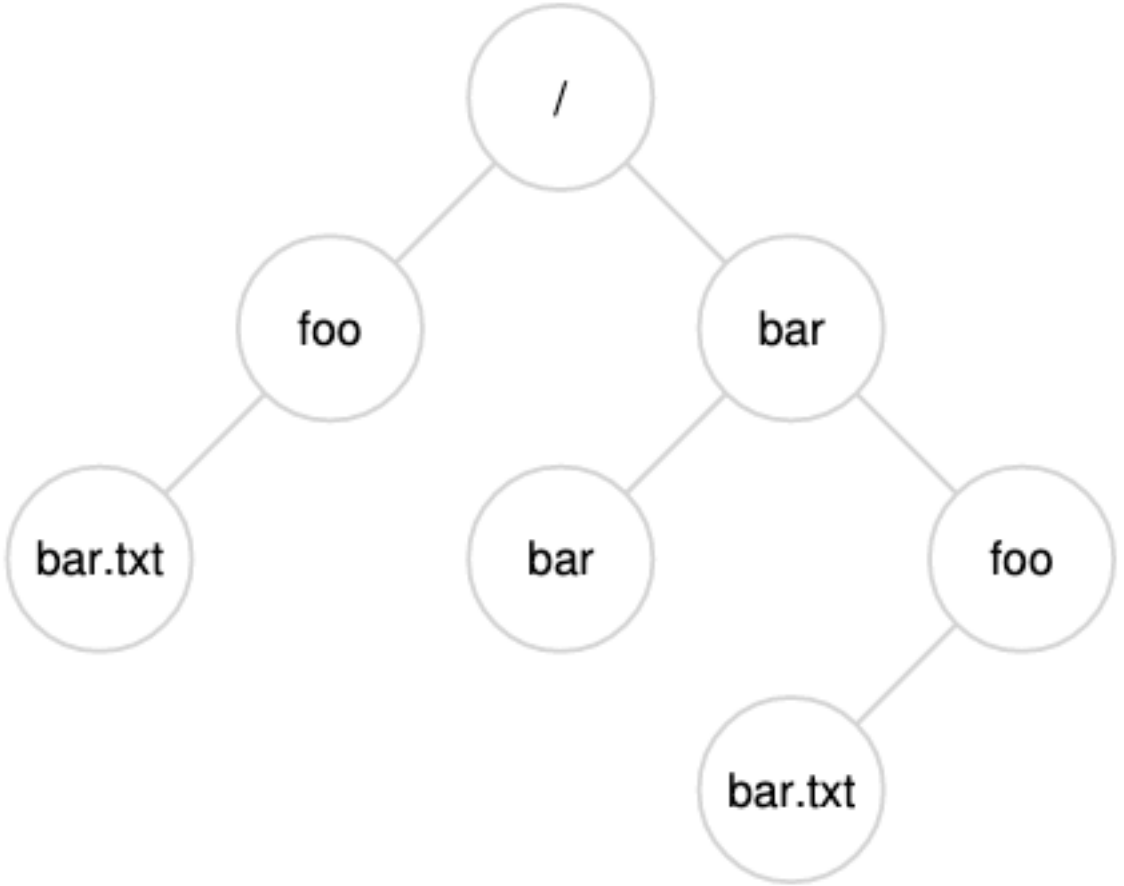
- We talked about Hard disk drives
- We looked at some ways of determining performance of a single disk
- We saw how RAID can help with disk I/O performance, reliability, and storage capacity.
  - Each RAID configuration offers a different balance between these features
- Now we start to look at a higher-level abstraction for our persistent storage

What are files and directories?

- File – linear array of bytes
  - Has a low-level name called its **inode number**
  - Most OSes don’t care about the structure of the file, rather that the data is there and in its proper locaton
- Directory – also like a file, but with a very specific structure
  - Has an **inode number**
  - References files associated (“in”) the directory with a pair:
    (user-given name, inode number)
  - Directories can be nested to create a **directory tree/hierarchy**
  - The ”root” of the directory hierarchy starts at “/” on Unix-based systems

Directory Hierarchy Example

- An approximation of what each director holds:
  - / (0)
    - (foo, 1)
    - (bar, 2)
  - foo (1)
    - (bar.txt, 3)
  - bar (2)
    - (bar, 4)
    - (foo, 5)
  - bar(4)
  - foo(5)
    - (bar.txt, 6)



Paths can be **absolute** (starting with root): /foo/bar.txt or **relative** (based on the current working directory): bar/foo/bar.txt (assuming we are in the root directory)

Creating a File

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

- Can create a file using open
  - “foo” is the name
  - O_CREAT = Creates the file if it doesn’t exist
  - O_WRONLY = The file will be written to only
  - O_TRUNC = If the file exists, clear its contents
  - S_IRUSR | S_IWUSR = permissions for the file (read and write for the user)

- What we get back is a **file descriptor**
  - Per process ”pointer” to a file type object to use for additional operations
  - All programs have three file descriptors to start
    - 0 – standard input
    - 1 – standard output
    - 2 – standard error

File Descriptor Example

- Open “foo” read-only (64-bit mode)
  - Get the file descriptor 3 (process stores all descriptors in an internal structure)
- Read 4K from the file
  - Get 6 bytes of data read
- Write 6 bytes to standard out
- Read and other 4K
  - Get 0 bytes of data (End of File)
- Close the file using descriptor 3

```
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>
```

Non-Sequential Read/Writes

```
struct file {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

***Simplified** File Struct

```
off_t lseek(int fildes, off_t offset, int 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.
```

- It is possible to read a file in a non-linear fashion
- You can provide an offset and provide behavior with respect to that offset
- Why might we do this?

File Offset Examples

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

**Note that all open files are tracked in the open file table kernel structure (along with a lock)

Sharing File Table Entries

- Most times if one or more processes use the same file the open file table has an entry for each process
  - Each read/write is independent with its own offset
- However, if we use fork(), the open file table is shared
  - This will cause the file struct reference count to be incremented (one for each process using the file)
  - When they close their respective file descriptors, the reference count is decremented (and removed at zero)
- Another method is using dup()
  - This will create a new file descriptor, that references the same underlying file struct

Links

- Hard links – Can create an alternate reference to an existing file (updates the reference count for the file)
  - Cannot link files across disks (each file system has its own inode number set)
  - Cannot link to a directory (cyclic path in the directory tree)

- Symbolic Links – Can create an alternative reference to existing files or directories
  - Takes up extra space as it stores the path to the file/directory it references
  - **Dangling references** are possible if you delete the original file/directory the symbolic link does not update

Permissions

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

- By default on a standard Unix file system, we have a simple method for file/directory permissions
- In the image above, we have details about the file foo.txt
- The first character indicates file (–), directory (`d`), or link (`l`)
- Each set of three character after indicates the owner, group, and other user permissions
- Each of these sets can be read (`r`), write (`w`), or execute (`x`) respectively
  - Any position with a hyphen (–) means that permission is not granted
- You can change the permissions of a file/directory with `chmod`
- Some filesystems support access control lists for fine grain permissions

Next Time

- We investigate the implementation of a file system