# File System Interface

04/22/2021

Professor Amanda Bienz

# Persistent Storage

- Keep a data intact, even if there is **power loss**
    - Hard disk drive
    - Solid-state storage device
- **Two key abstractions in the virtualization of storage**
    - File
    - Directory

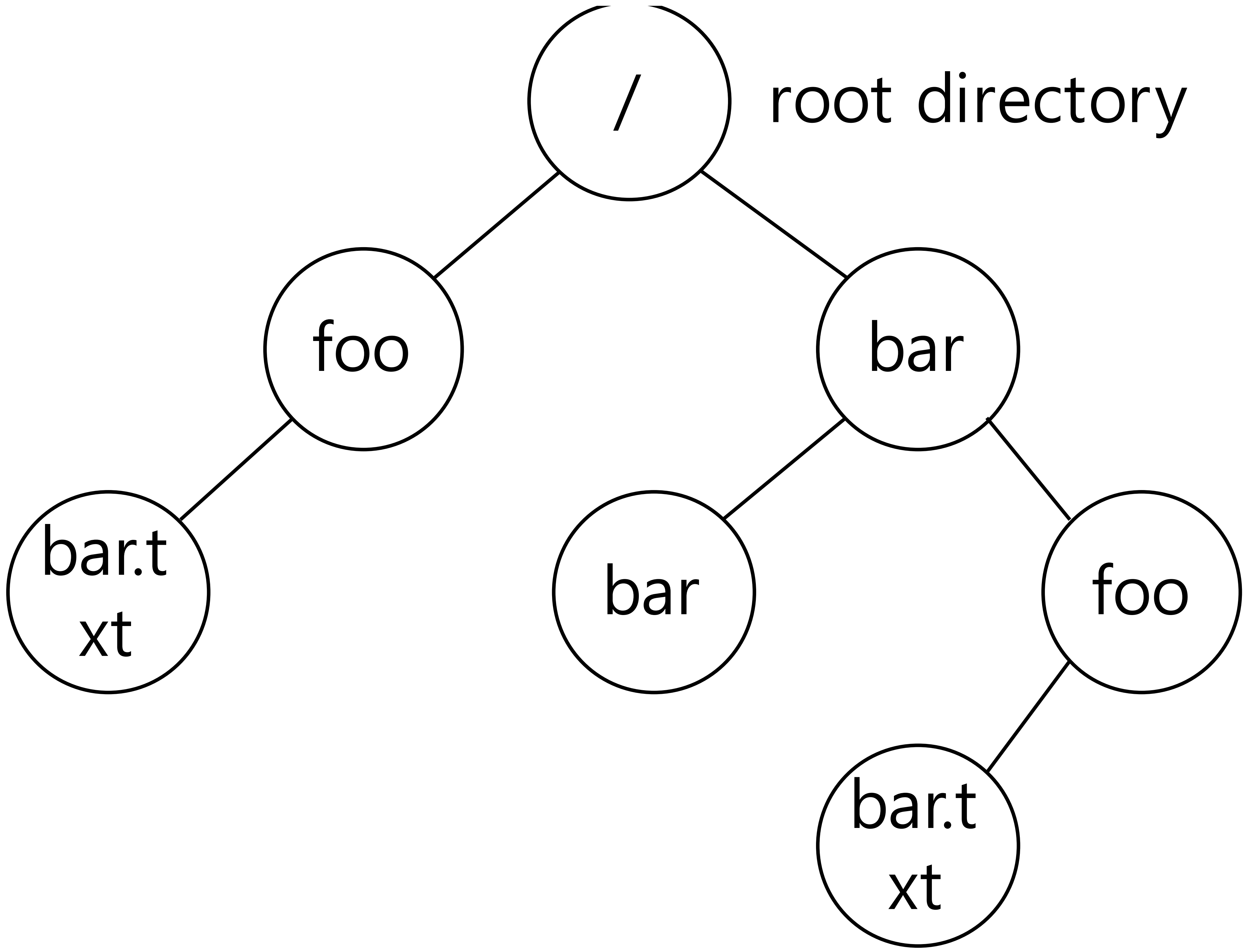
# File

- A linear array of bytes
- Each file has low-level name as inode number
  - 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

# File Operations

- **Create**
- **Write :** at write pointer location
- **Read :** at read pointer location
- **Reposition within file (seek)**
- **Delete**
- **Truncate**
- **Open(Fi) :** search directory structure on disk for entry Fi and move content of entry to memory
- **Close(Fi) :** move content of entry Fi in memory to directory structure on disk

# 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**
  - An integer, used to access 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 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()

# 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

```
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
  - Example : 5 seconds, or 30
  - Performance reasons
- At that later point in time, the write(s) will actually be issued to the storage device
  - Writes seem to complete quickly
  - Data can be lost (e.g. the machine crashes)

# Writing Immediately with fsync() (Cont.)

- However, some applications require more than eventual guarantee
  - Example : database management system requires force writes to disk from time to time
- 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 are complete

# Write 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

# Renaming Files

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

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

    - 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:

```
  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 a `inode` structure.

# Removing Files

- Rm is a Linux command to remove a file
    - Calls unlink() system call to remove the 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.**

# File Systems Reading

- Chapter 14 : Pg 563 - 589