Operating System (OS)
CS232

Persistence: Files and Directories

Dr. Muhammad Mobeen Movania

Outlines

- Files and directories
- File system interface
- Creating, accessing and deleting files
- Sequential and random I/O
- Shared file table entries
- Hard and symbolic links
- Permission bits and access control list (ACL)
- Summary

Files and Directories

- We saw two virtualizations earlier
  - Process (virtualization of CPU)
  - Address space (virtualization of memory)
- We will now look at virtualization of storage
- Two main abstractions
  - File
  - Directory

Files and Directories (2)

- File
  - A linear array of bytes
  - Has a low-level name (which is often a number) called the **inode number**
  - Often files have an extension but its not mandatory to have contents as per extension
- Directory
  - Similar to a file, has a low-level name (inode number)
  - Contains a list of tuples containing (user-readable name, low-level name)
  - Each entry is either a file or another directory
  - Directory within directory allow to create a **directory tree** (**or directory hierarchy**)

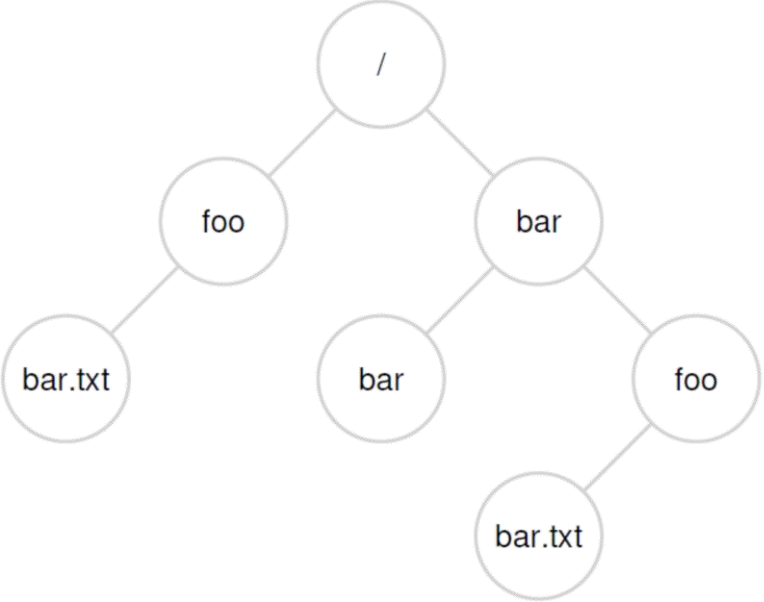


Figure 39.1: **An Example Directory Tree**

File System Interface

- Set of API for creating, accessing and deleting files
- Underneath they may call one or more system calls
- All internal implementation details are hidden from the user

Creating Files

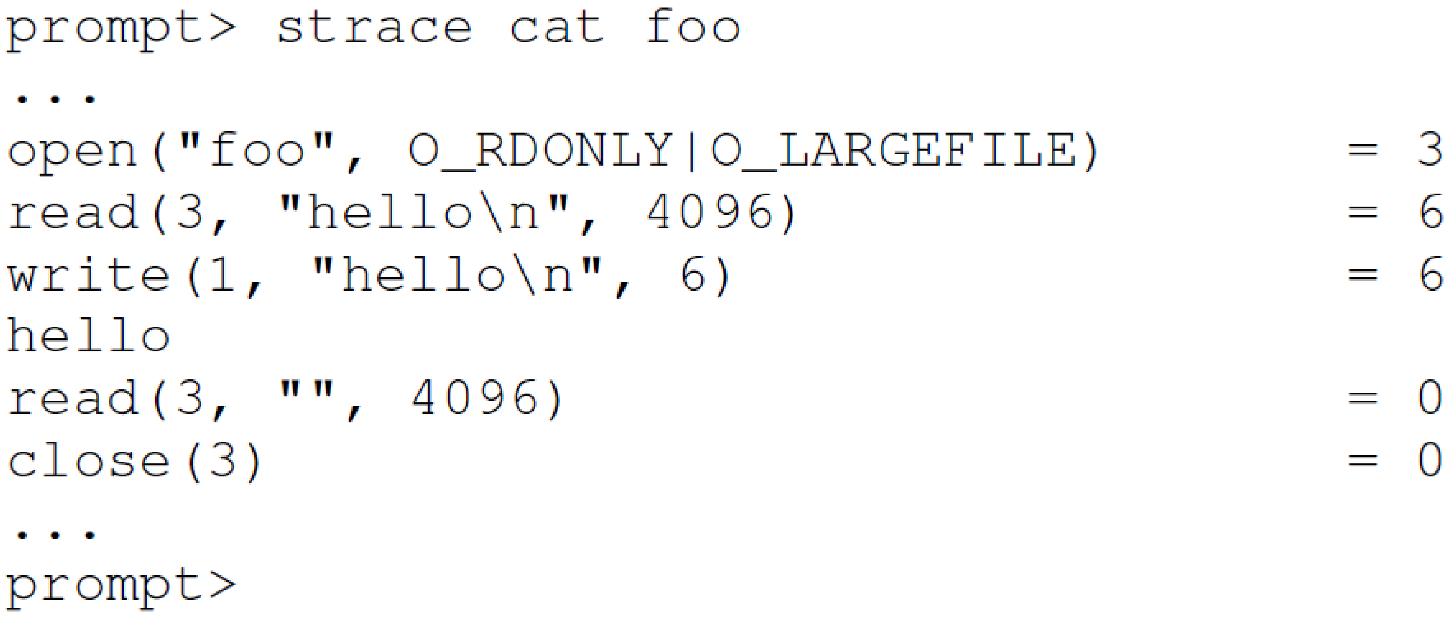
- Uses open system call with O_CREATE flag

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

- The second parameter is a set of flags
  - (O_CREAT) creates the file if it does not exist,
  - (O_WRONLY) ensures that the file can only be written to, and,
  - (O TRUNC) truncates file (if the file already exists) to a size of zero bytes thus removing any existing content.
- The third parameter specifies permissions,
  - (`S_IRUSR|S_IWUSR`) file is readable and writable by the owner
- The return value is a file descriptor that is used to access file in later operations
- File descriptors are stored per-process in **proc** struct

Reading and Writing Files

- We can use **strace** tool to extract system calls made by programs
- Use **`fsync(fd)`** call to force write flushing

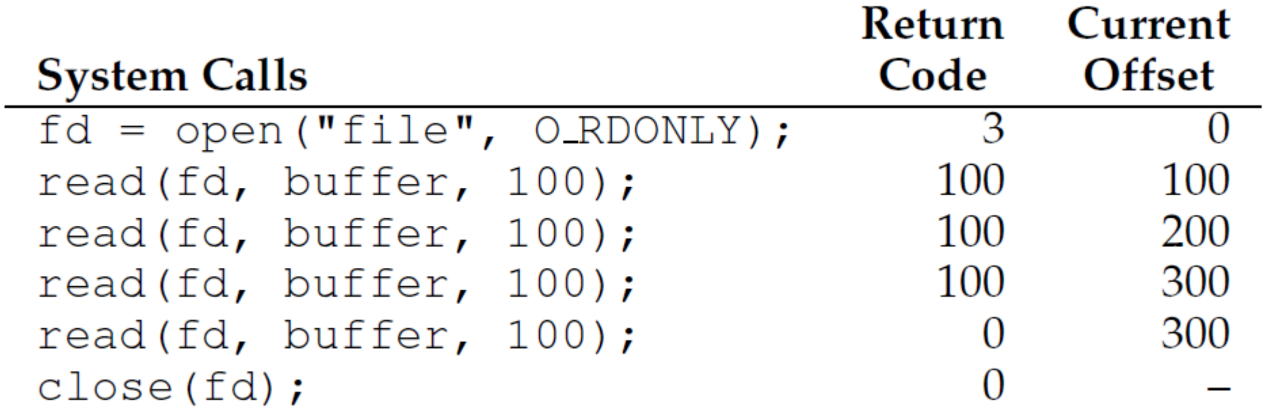


```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)           = 3
read(3, "hello\n", 4096)                    = 6
write(1, "hello\n", 6)                      = 6
hello
read(3, "", 4096)                           = 0
close(3)                                    = 0
...
prompt>
```

Non-sequential read/write

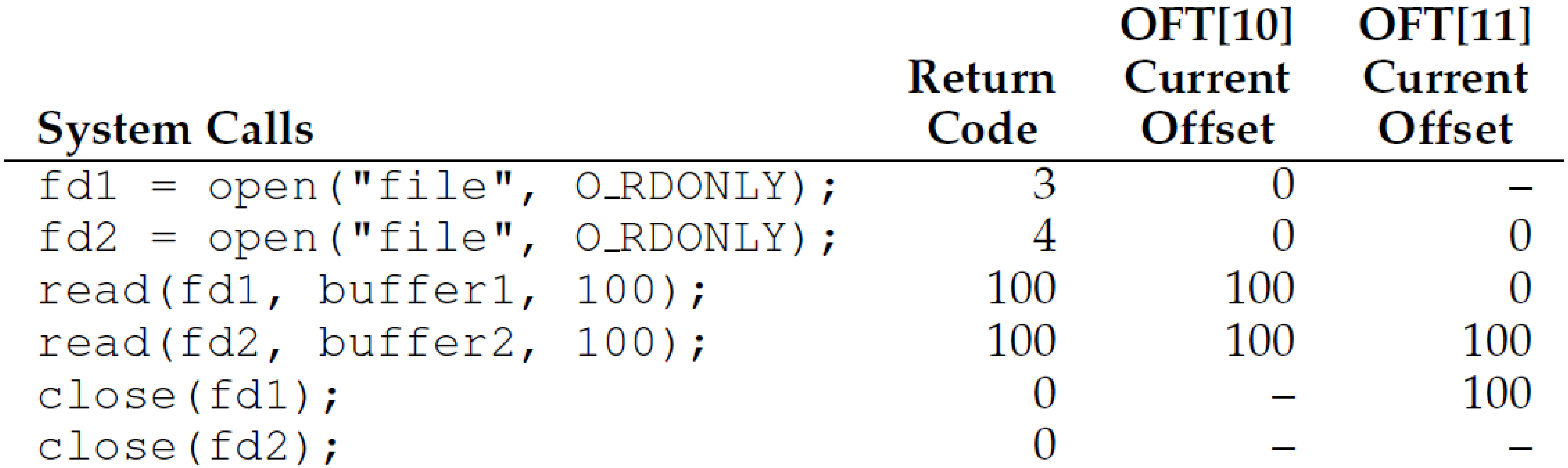
- Use the lseek system call to reposition the read/write pointers in file

```
off_t lseek(int file_des, //file descriptor
            off_t offset, //offset in bytes
            int whence);  //from start (SEEK_SET)
                          //from current (SEEK_CUR)
                          //from end (SEEK_END)
```



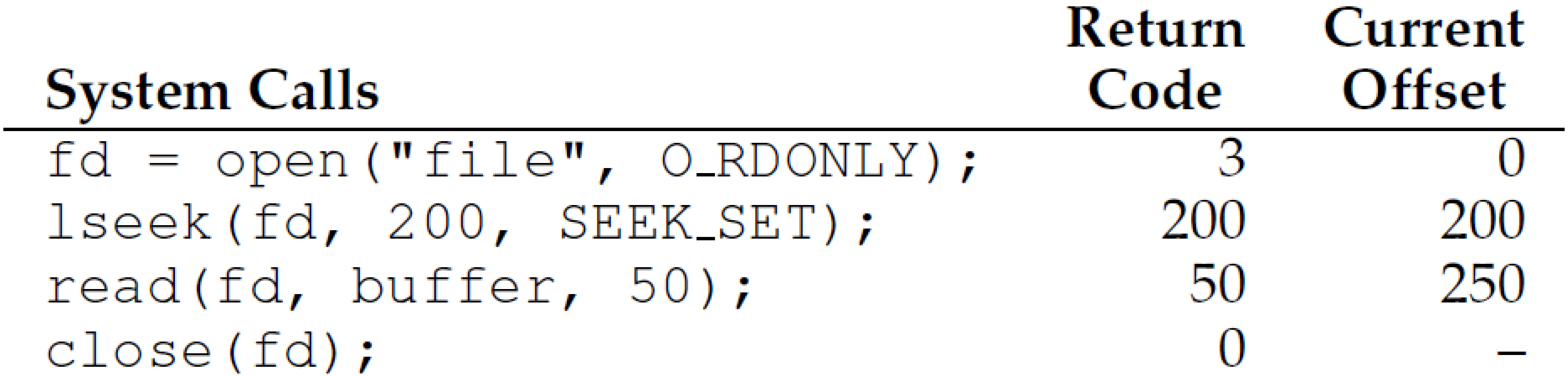
System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

Reading from a single file using two descriptors



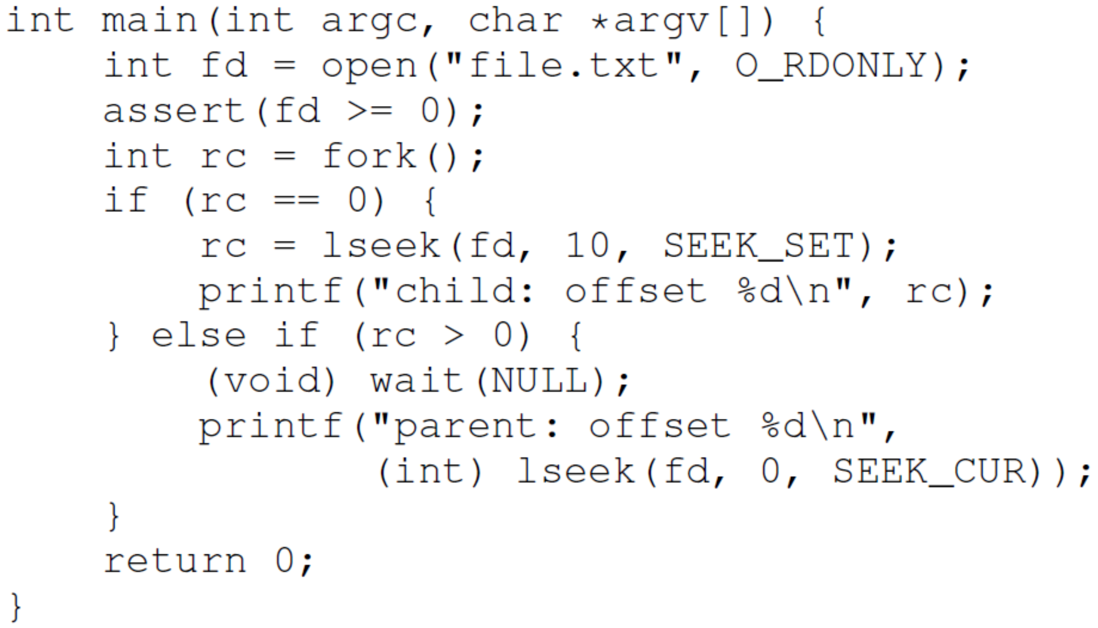
System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

Reading after lseek



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	–

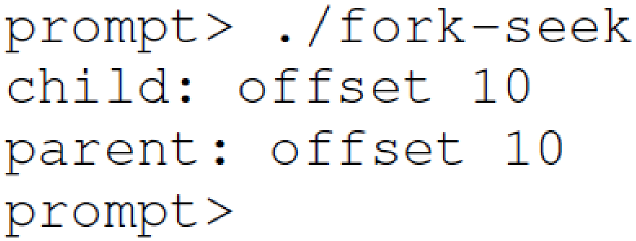
Shared File Table Entries



```
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;
}
```

Figure 39.2: **Shared Parent/Child File Table Entries (`fork-seek.c`)**

- Output



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

Shared File Table Entries

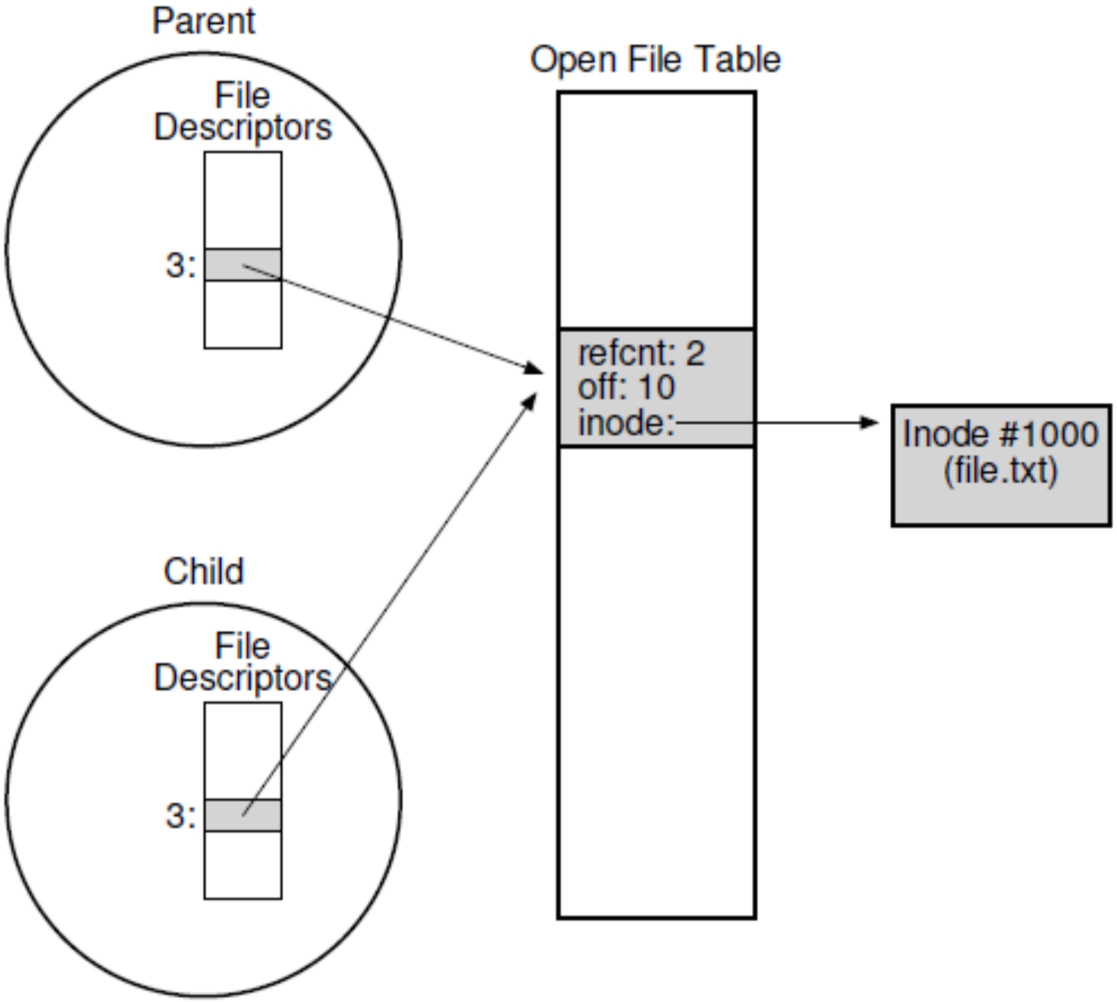
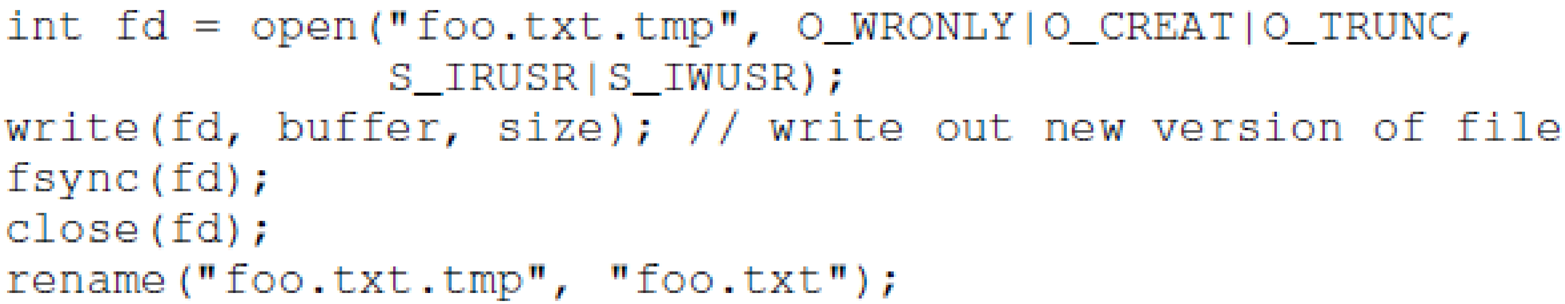


Figure 39.3: Processes Sharing An Open File Table Entry

Renaming a file

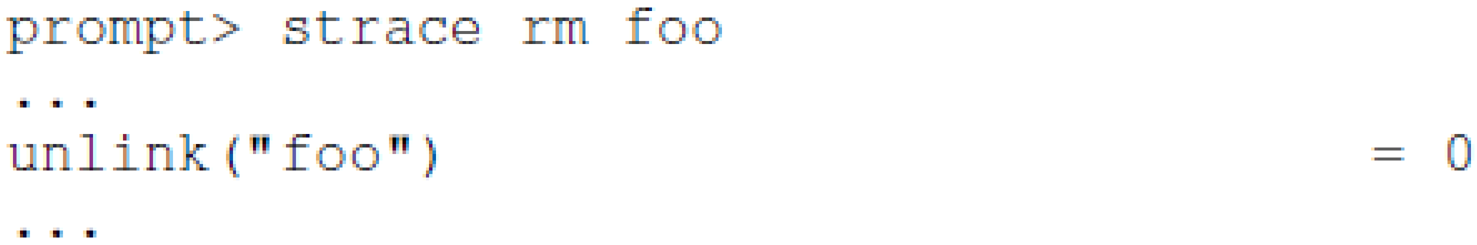
- fsync call flushes writes to ensure persistence
- Rename is achieve through
  - – mv foo bar
- strace call reveals



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

Deleting a file and directory

- Delete file
  - Use rm command
  - Using strace, we see that rm calls unlink

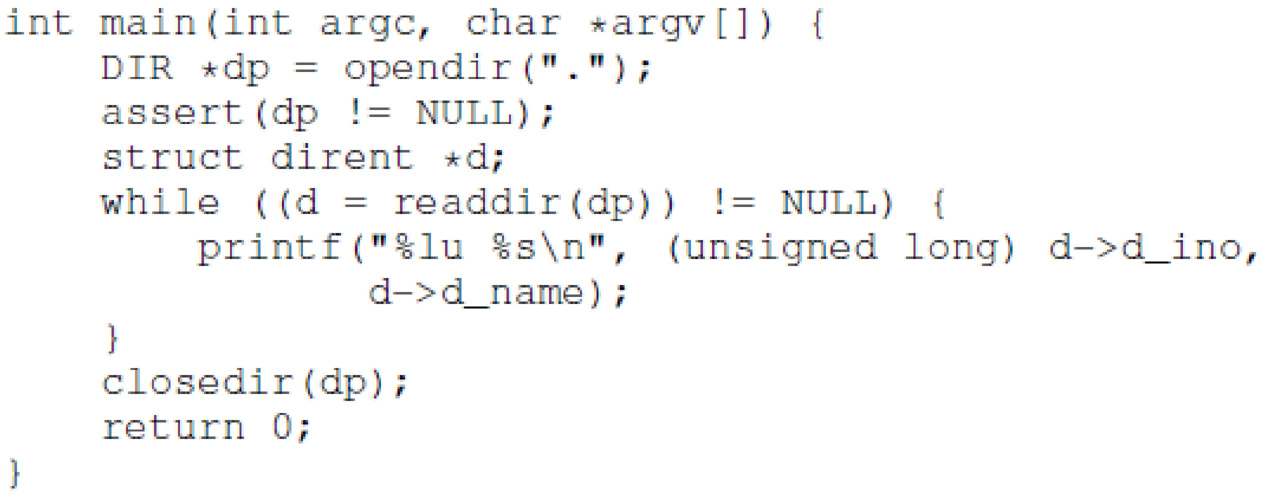


```
prompt> strace rm foo
...
unlink("foo")                                    = 0
...
```

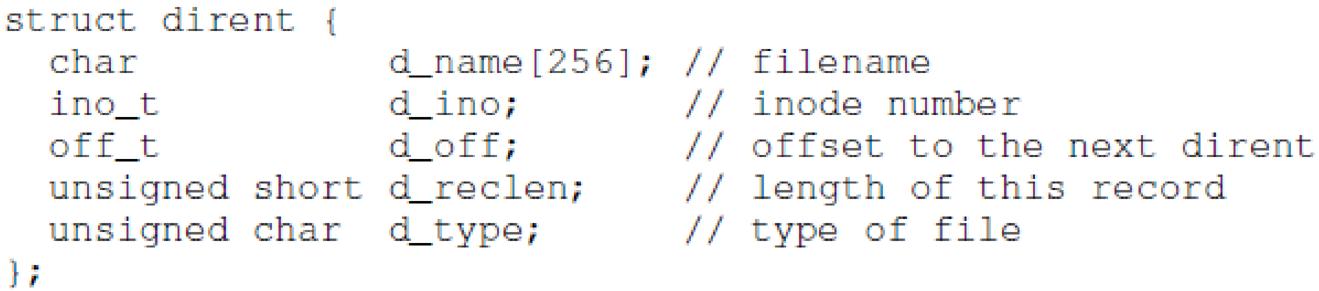
- Delete directory
  - Use rmdir command
  - Directory must be empty or rmdir fails silently

Reading a Directory

- We may simulate the ls command



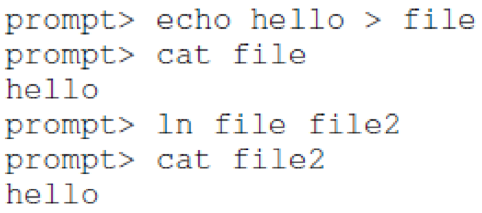
```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino,
               d->d_name);
    }
    closedir(dp);
    return 0;
}
```



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

Hard links

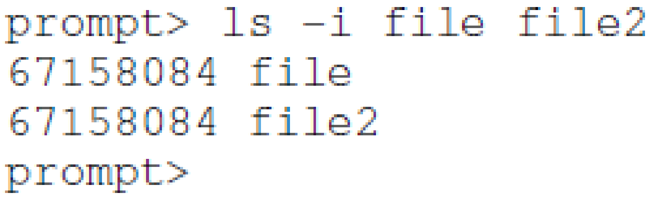
- link() function
  - makes an entry in the file system tree
  - takes two arguments, an old pathname and a new one;
- When you “link” a new file name to an old one, you essentially create another way to refer to the same file.
- Use the ln command as follows



```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

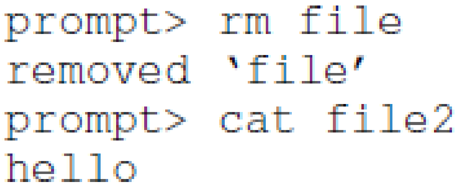
rm command and Unlink

- After creating a hard link to a file, there is no difference between the original file name (file) and the newly created file name (file2)
- They are both just links to the underlying metadata about the file, which is found in inode number 67158084.



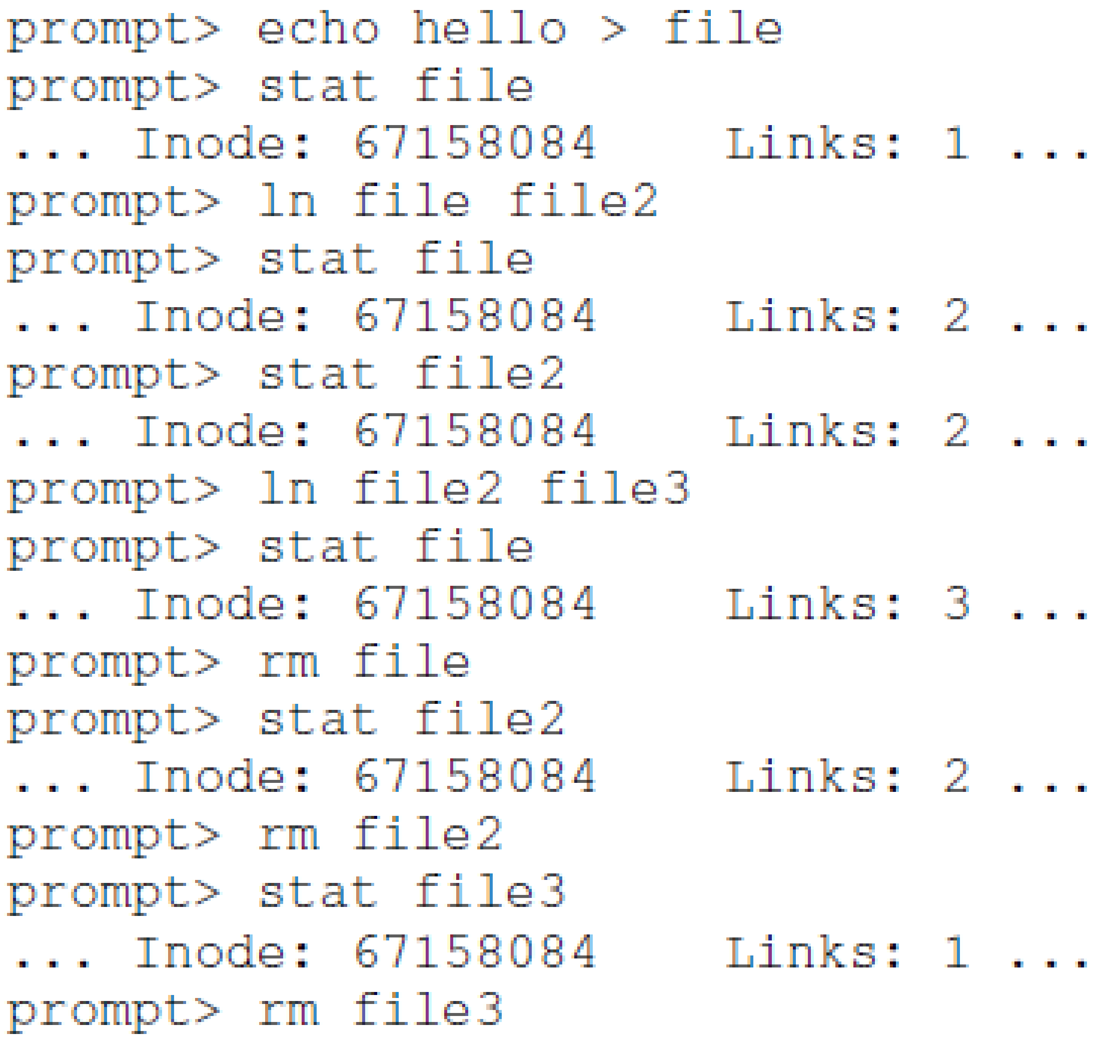
```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

- To remove a file from the file system, we call unlink().



```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

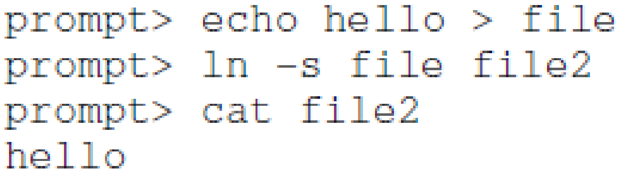
rm command and Unlink (2)



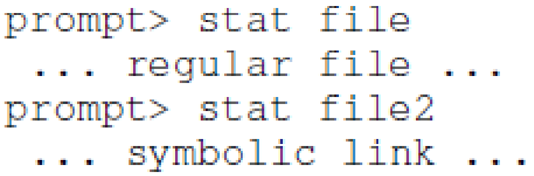
```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084    Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084    Links: 2 ...
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084    Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084    Links: 1 ...
prompt> rm file3
```

Symbolic links

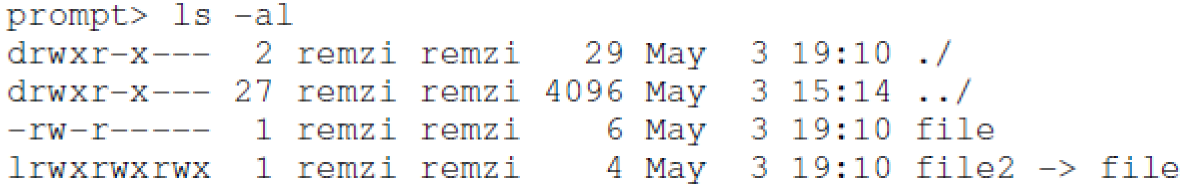
- Hard links have a few restrictions
  - you can’t hard link to a directory (for fear it will create a cycle in the directory tree)
  - you can’t hard link to files in other disk partitions (because inode numbers are only unique within a particular file system)
- Soft link are created with the -s flag to ln



```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```



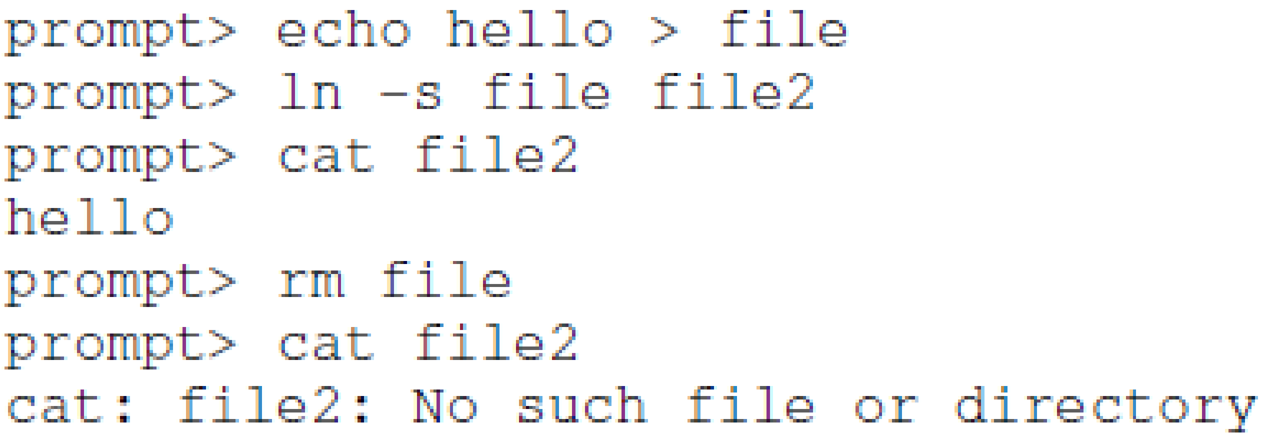
```
prompt> stat file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...
```



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

Symbolic links

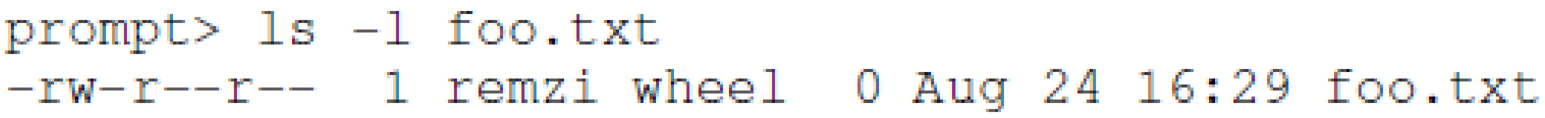
- Soft links might generate dangling pointers



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

Permission bits and access control list (ACL)

- On unix based systems, we can use ls –l command to see the file permission bits



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

  - The first character (-) here just shows the type of the file (- for a regular file), d for a directory, l for a symbolic link)
  - Next nine bits are for permission
    - The permissions consist of three groupings: what the **owner** of the file can do to it, what someone in a **group** can do to the file, and finally, what anyone (sometimes referred to as **other**) can do

Summary

- We saw how file and directories are implemented
- We saw how the OS system calls abstract the low level details during file I/O
- The file system interface in UNIX systems (and indeed, in any system) is seemingly quite rudimentary, but there is a lot to understand