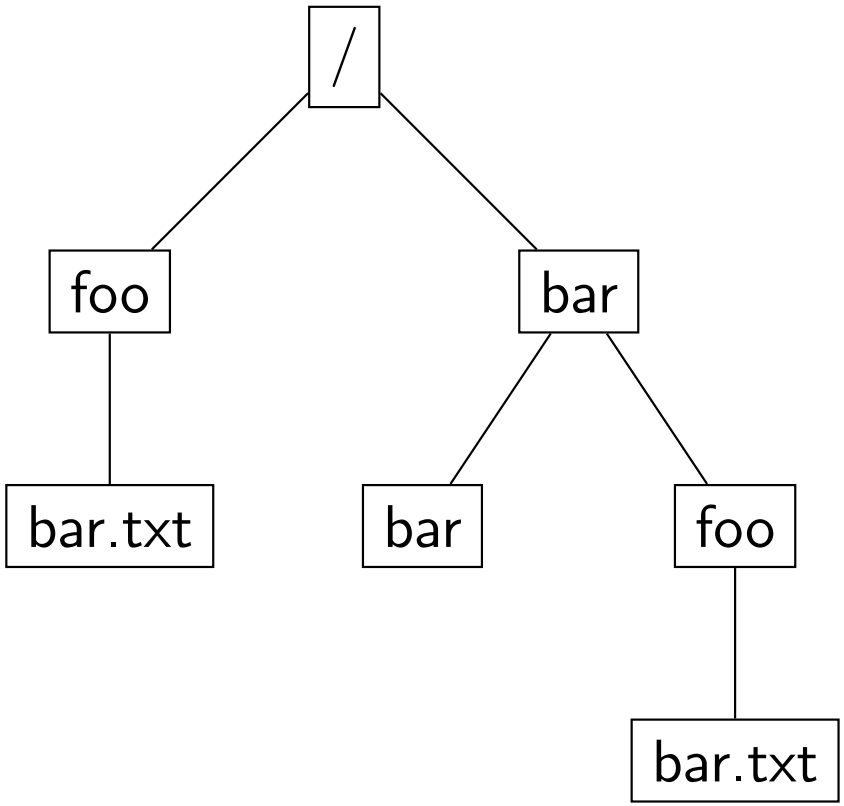


CSL 301

OPERATING SYSTEMS

Lecture 25

Files and Directories

Instructor

Dr. Dhiman Saha

The Missing Piece

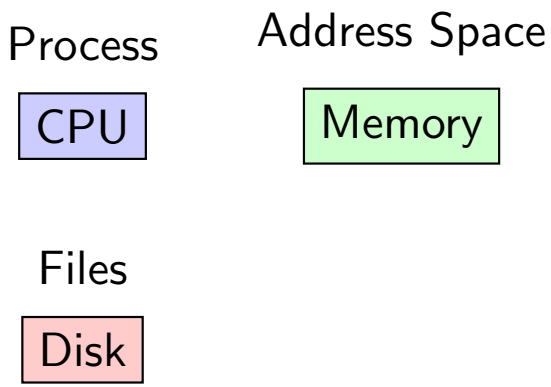
- The **process**: A virtualization of the CPU.
- The **address space**: A virtualization of memory.

The Missing Piece

- The **process**: A virtualization of the CPU.
- The **address space**: A virtualization of memory.

How do we store information permanently?

- Memory is volatile.
- We need **persistent storage**.

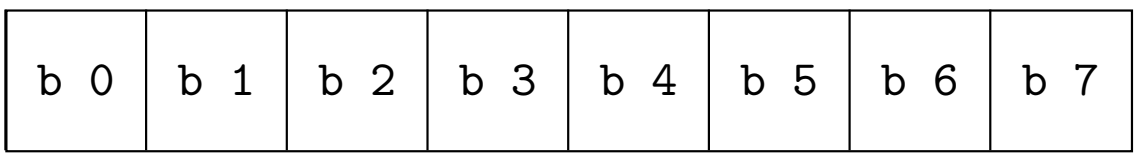


The File Abstraction

A file is a linear array of bytes.

- You can read from it or write to it.
- It has a low-level name, typically a number (e.g., an **inode number**).
- The OS doesn't know the file's structure (e.g., text, image, binary). It just stores the data.

A File

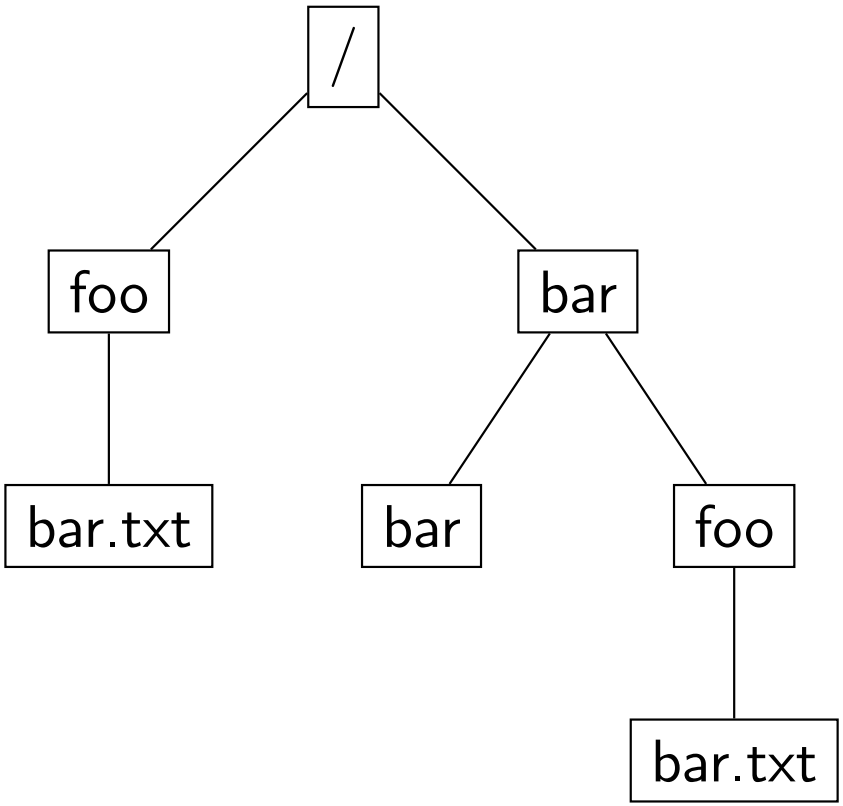


The Directory Abstraction

A directory contains a list of (user-readable name, low-level name) pairs.

- Directories themselves are also files!
- They allow for organizing files into a hierarchy.
- This creates a **directory tree**.

Example Directory Tree



- The hierarchy starts at the **root directory** (/).
- An **absolute pathname** specifies a path from the root.
- Example: `/foo/bar.txt`

File System API: Creating Files

The `open()` system call is used to create and open files.

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

- `O_CREAT`: Create the file if it doesn't exist.
- `O_WRONLY`: Open for writing only.
- `O_TRUNC`: Truncate to zero size if it exists.
- The third argument specifies permissions.

File System API: Creating Files

The `open()` system call is used to create and open files.

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

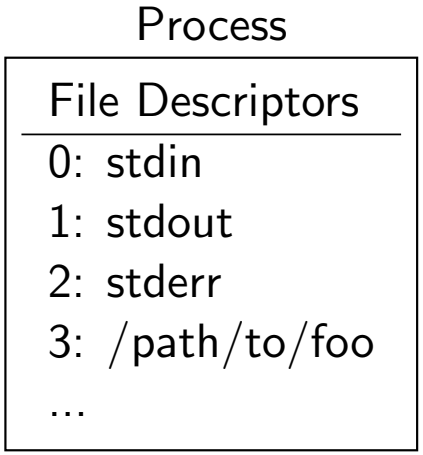
- `O_CREAT`: Create the file if it doesn't exist.
- `O_WRONLY`: Open for writing only.
- `O_TRUNC`: Truncate to zero size if it exists.
- The third argument specifies permissions.

Return Value: A **file descriptor**, which is a small integer.

File Descriptors

A file descriptor is a per-process, private integer used to access a file.

- By default, every process starts with three open file descriptors:
  - 0: Standard Input
  - 1: Standard Output
  - 2: Standard Error
- New file descriptors will usually start at 3.



File System API: Reading and Writing

`read()` and `write()` use file descriptors.

```
// ssize_t read(int fd, void *buf, size_t count);
read(fd, buffer, 4096);

// ssize_t write(int fd, const void *buf, size_t count);
write(STDOUT_FILENO, "hello\n", 6);
```

- The file descriptor `fd` tells the OS which file to operate on.
- Data is read into or written from a buffer in memory.
- The calls return the number of bytes read or written.

File System API: Seeking

The `lseek()` system call changes the current offset of an open file.

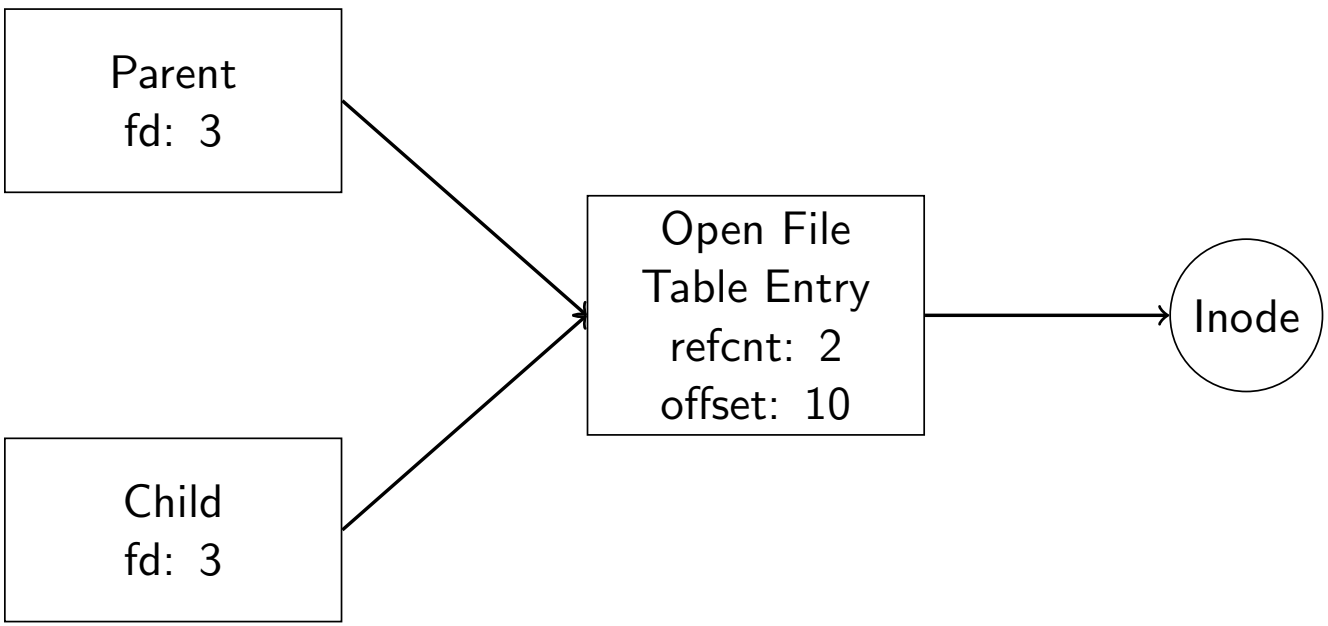
```
// off_t lseek(int fd, off_t offset, int whence);
lseek(fd, 100, SEEK_SET); // Set offset to 100 bytes from s
```

- Each open file has an associated **current offset**.
- `read()` and `write()` update the offset implicitly.
- `lseek()` updates the offset explicitly, allowing for **random access**.
- **Important:** `lseek()` does NOT perform a disk seek! It only changes a variable in the OS.

Sharing: fork()

When a process calls `fork()`, the child inherits the parent's file descriptors.

- Both parent and child descriptors refer to the **same open file table entry**.
- This means they share the same file offset. A read/write in one process affects the other.



Hard Links

A **hard link** creates another name for the same underlying file (inode).

```
prompt> echo "hello" > file1
prompt> ln file1 file2
prompt> ls -i file1 file2
12345 file1
12345 file2
```

- Both `file1` and `file2` point to the same inode.
- There is no "original" file; all links are equal.
- The file system keeps a **reference count** in the inode.

The unlink() Mystery

To remove a file, we use `unlink()`.

```
prompt> rm file1
```

The `rm` command calls `unlink("file1")`.

- `unlink()` removes the link (the name) from the directory.
- It then decrements the inode's reference count.
- The file's data is only truly deleted when the reference count reaches zero.

The unlink() Mystery

To remove a file, we use `unlink()`.

```
prompt> rm file1
```

The `rm` command calls `unlink("file1")`.

- `unlink()` removes the link (the name) from the directory.
- It then decrements the inode's reference count.
- The file's data is only truly deleted when the reference count reaches zero.

So...

After `rm file1`, you can still access the content via `file2`!

Symbolic (Soft) Links

A **symbolic link** is a special file that contains a path to another file.

```
prompt> ln -s file1 symlink
prompt> ls -l
-rw-r--r-- 1 user group 6 Nov  7 10:10 file1
lrwxrwxrwx 1 user group 5 Nov  7 10:11 symlink -> file1
```

- Unlike a hard link, a symlink is a distinct file with its own inode.
- It points to a pathname, not an inode.
- Can link across different file systems.
- Can create a **dangling reference** if the original file is removed.

Summary

- The OS provides two key abstractions for persistent storage: **files** and **directories**.
- A rich set of system calls allows for creating, reading, writing, and managing these objects.
- **File descriptors** are the handles used to access open files.
- **Links** (hard and symbolic) provide powerful ways to structure and share files.
- Understanding the file system interface is crucial for any programmer.