CS111, Lecture 5

File Descriptors and System Calls

Optional reading:

Operating Systems: Principles and Practice (2^{nd} Edition): Sections 13.1-13.2

😷 masks strongly recommended

Topic 1: Filesystems - How can we design filesystems to manage files on disk, and what are the tradeoffs inherent in designing them? How can we interact with the filesystem in our programs?

CS111 Topic 1: Filesystems

Filesystems introduction and design	→	Case study: Unix V6 Filesystem	→	Filesystem System calls and file descriptors	→	Crash recovery
Lecture 1		**Lecture 2-4**		**Today**		**Lectures 6-7**

assign1: implement portions of the Unix v6 filesystem!

Learning Goals

- Evaluate the tradeoffs of the Unix v6 Filesystem design overall
- Learn about the **open**, **close**, **read** and **write** functions that let us interact with files
- Get familiar writing programs that read, write and create files
- Learn what the operating system manages for us so that we can interact with files

Plan For Today

- **Recap**: filesystem design and modern filesystems
- Interacting with the filesystem in user programs
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice:** copying files

```
cp -r /afs/ir/class/cs111/lecture-code/lect5 .
```

Plan For Today

- **Recap: filesystem design and modern filesystems**
- Interacting with the filesystem in user programs
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice:** copying files

```
cp -r /afs/ir/class/cs111/lecture-code/lect5 .
```

Recap From Last Time

- Small files have up to 8 direct block numbers stored in their inode
- Large files have up to 7 singly-indirect and 1 doubly-indirect block number stored in their inode
- Directories are "just files" and are layered on top of files. Directories store directory entries, which contain info about each file/folder directly within that directory. **Note:** name is at most 14 bytes, which may not be null terminated!
- The lookup process traverses through each directory in the path until we reach the file we're looking for (or don't find it)

Hard Links

With the directory entry structure, it's possible for **two different filenames to resolve to the same inumber.** Why is this useful?

- You could have multiple copies of a file without duplicating contents, and if you change one you change all of them.

On linux: `ln originalFile newFile` creates **newFile,** mapping it to the same inumber as **originalFile**

- The **i_nlink** inode field stores the number of directory entries that point to that inode. Files are deleted when **i_nlink** = 0 and no programs are using it.
- This is called a *hard link*. All normal files in Unix (and Linux) are hard links, and there's no way to distinguish which one is the "real" file, since both are real!

Hard Links

- **Example**: you create a library, and others want to use it, so they can make a hard link to it to avoid copying it
- If you delete the library and make a new version of it, the others will still have access to it
- **Downside**: links to directories not allowed (could cause circular references)
- **Downside**: cannot link across filesystems (inumbers not unique)

Soft Links

A soft ("symbolic") link is another way to link one file to another, and allows linking to directories, and allows linking across filesystems.

- Instead of sharing an inode, the link file stores the *path to the original file as its payload data*, and the inode uses a field to track that it is a symbolic link
- When opening, modifying or using that file, it refers to the linked file

On linux: `ln -s originalFile newFile` creates a symbolic link **newFile** linking to **originalFile**

- Soft links can "break" if the file they refer to no longer exists
- Example: **samples/** directory in CS111 assignments

Demo: Hard/Soft Links

Multi-level Indexes

The Unix V6 filesystem (from 1975) is an example of the "multi-level index" filesystem design. There are many alternative designs that could be used – some alterations you could propose might be:

- What if the block size was different?
- What if inodes stored a different number of block numbers?
- What if the file size scheme (small / large) worked differently?

Example: 4.3 BSD Unix filesystem (evolutionary descendent of V6)

- 4KB block size
- Inodes store 14 block numbers
- First 12 block numbers always direct, 13^{th} always singly indirect, 14^{th} always doubly indirect (no small vs. large schemes)

Other Filesystem Design Ideas

Larger block size? Improves efficiency of I/O and inodes but worsens internal fragmentation. Generally: challenges with both large and small files coexisting.

One idea: multiple block sizes

- Large blocks are 4KB, *fragments* are 512 bytes (8 fragments fit in a block)
- The last block in a file can be a fragment (0-7 fragments)
- One large block can hold fragments from multiple files
- Get the time efficiency benefit of larger blocks, but the internal fragmentation benefit of smaller blocks (small files can use fragments)

Filesystem Techniques Today

- Filesystem design is a hard problem! Tradeoffs, challenges with large and small files.
- Even larger block sizes (16KB large blocks, 2KB fragments) – disk space cheap, internal fragmentation doesn't matter as much
- Reallocate files as blocks grow – initially allocate blocks one at a time, but when a file reaches a certain size, reallocate blocks looking for large contiguous clusters
- ext4 is a popular current Linux filesystem – you may notice similarities!
- NTFS (replacement for FAT) is the current Windows filesystem
- APFS ("Apple Filesystem") is the filesystem for Apple devices

Plan For Today

- **Recap**: filesystem design and modern filesystems
- **Interacting with the filesystem in user programs**
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice:** copying files

```
cp -r /afs/ir/class/cs111/lecture-code/lect5 .
```

OS vs. User Mode

- The operating system runs code in a privileged “kernel mode” where it can do things and access data that regular user programs cannot. E.g. only OS can call **readSector**.
- System tracks whether it is in “user mode” or “kernel mode”
- The OS provides public functions that we can call in our user programs – **system calls**. When these functions are called, it switches over to “kernel mode”.

System Calls

Functions to interact with the operating system are part of a group of functions called **system calls**.

- A system call is a public function provided by the operating system.
- The operating system handles these tasks because they require special privileges that we do not have in our programs. When a system call runs, it runs in **kernel mode**, and we switch back to user mode when it's done.
- The operating system *kernel* runs the code for a system call, completely isolating the system-level interaction from your (potentially harmful) program.
- We are going to examine the system calls for interacting with files. When writing production code, you will often use higher-level methods that build on these (like C++ streams or FILE *), but let's see how they work!

open()

Call **open** to open a file:

```
int open(const char *pathname, int flags);
```

- **pathname**: the path to the file you wish to open
- **flags**: a bitwise OR of options specifying the behavior for opening the file
- returns a **file descriptor** representing the opened file, or -1 on error

Many possible flags (see manual page for full list). You must include exactly one of the following flags: **O_RDONLY** (read-only), **O_WRONLY** (write-only), **O_RDWR** (read and write). These say how you will use the file in this program.

Another useful flag: **O_TRUNC** means if the file exists already, truncate (clear) it.

open()

Call **open** to open a file:

```
int open(const char *pathname, int flags, mode_t mode);
```

You can also create a new file if the specified file doesn't exist, by including **O_CREAT** as one of the flags. You must also specify a third **mode** parameter.

- **mode**: the permissions to attempt to set for a created file

open()

Call **open** to open a file:

```
int open(const char *pathname, int flags, mode_t mode);
```

You can also create a new file if the specified file doesn't exist, by including **O_CREAT** as one of the flags. You must also specify a third **mode** parameter.

- **mode**: the permissions to attempt to set for a created file

Another useful flag: **O_EXCL**, which says the file must be created from scratch, and to fail if the file already exists.

Aside: how are there multiple signatures for **open** in C? See here.

File Descriptors

A **file descriptor** is like a "ticket number" representing your currently-open file.

- It is a unique number assigned by the operating system to refer to that file in this program.
- Each program has its own file descriptors
- When you wish to refer to the file (e.g. read from it, write to it) you must provide the file descriptor.
- file descriptors are assigned in ascending order (next FD is lowest unused)
- The OS remembers information associated with each of your file descriptors, like where in the file you currently are (if reading/writing)

close()

Call **close** to close a file when you're done with it:

```
int close(int fd);
```

- **fd:** the file descriptor you'd like to close.

It's important to close files when you are done with them to preserve system resources.

- You can use **valgrind** to check if you forgot to close any files. (--track-fds=yes)

Example: Creating a File (touch)

```
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

touch.c

Example: Creating a File (touch)

```
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

Open the file to be written to

touch.c

Example: Creating a File (touch)

```
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

If the file doesn't exist, create it

touch.c

Example: Creating a File (touch)

```c
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

If it does exist, throw an error

touch.c

Example: Creating a File (touch)

```c
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

If we create a new file, it should have these permissions (don't worry about specifics for now)

touch.c

Example: Creating a File (touch)

```
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

touch.c

Recap

- **Recap**: filesystem design and modern filesystems
- Interacting with the filesystem in user programs
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice:** copying files

Next time: how can we design a filesystem that is resilient in the event of a system crash?

Lecture 5 takeaway: System calls are functions provided by the operating system to do tasks we cannot do ourselves. open/close/read/write are system calls that work via file descriptors to create, read from and write to files.