CS111, Lecture 5
File Descriptors and System Calls

Optional reading:

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

How is assign1 going for you so far?

- Great!
- Good
- Ok
- Not so great
- Going to start soon! :)

How is assign1 going for you so far?

Great!

Good

Ok

Not so great

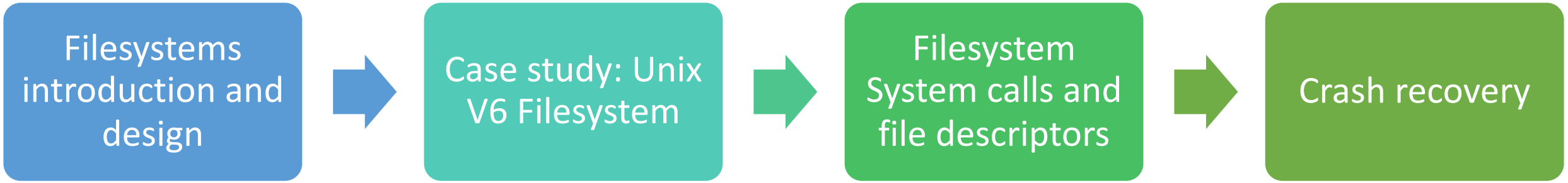
Going to start soon! :)

Announcements

- Sections start in person tomorrow! Check the course website for your section assignment and location.
- assign1 YEAH hours recording / slides posted to course website – thanks to everyone who came out!
- Adding helper hours Fri. 2-4PM (at Nick's office, Durand building room 331A)

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



Lecture 1/ lecture 2

Lectures 2-4

This lecture

Lectures 6-7

Learning Goals

- 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 as users
- Interacting with the filesystem as programmers
  - 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 as users
- Interacting with the filesystem as programmers
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice:** copying files

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

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, 13th always singly indirect, 14th 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 as users**
- Interacting with the filesystem as programmers
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice:** copying files

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

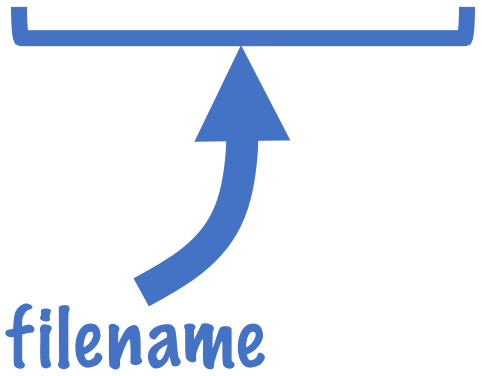
Filesystem: User Perspective

- Studying how we interact with the filesystem as users will inform how we interact with it as programmers.
- As users, we can run **ls** to get details about particular files. Using -a shows all files (even hidden ones), -l shows more info about each file

```
troccoli@myth60:~/assign1$ ls -al
total 60
drwx------  5 troccoli operator  4096 Oct  4 22:42 .
drwxr-xr-x 58 troccoli operator  6144 Oct  1 23:45 ..
-rw-------  1 troccoli operator  1920 Sep 30 14:32 chksumfile.c
-rw-------  1 troccoli operator  1063 Sep 30 14:32 chksumfile.h
-rw-------  1 troccoli operator   839 Sep 30 14:32 custom_tests
-rw-------  1 troccoli operator   509 Sep 30 14:32 directory.c
-rw-------  1 troccoli operator   476 Sep 30 14:32 directory.h
-rw-------  1 troccoli operator   499 Sep 30 14:32 direntv6.h
-rw-------  1 troccoli operator 11426 Sep 30 14:32 diskimageaccess.c
```

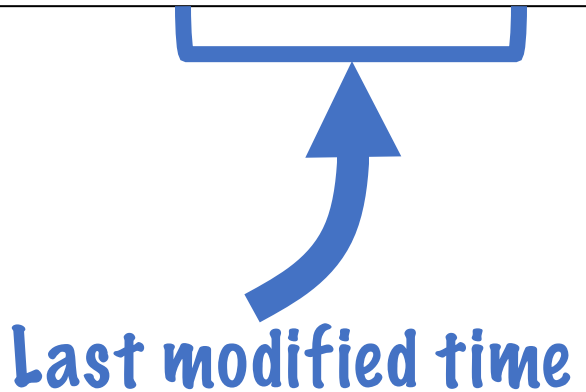
Filesystem Information

```
troccoli@myth60:~/assign1$ ls -al
total 60
drwx------  5 troccoli operator  4096 Oct  4 22:42 .
drwxr-xr-x 58 troccoli operator  6144 Oct  1 23:45 ..
-rw-------  1 troccoli operator  1920 Sep 30 14:32 chksumfile.c
-rw-------  1 troccoli operator  1063 Sep 30 14:32 chksumfile.h
-rw-------  1 troccoli operator   839 Sep 30 14:32 custom_tests
-rw-------  1 troccoli operator   509 Sep 30 14:32 directory.c
-rw-------  1 troccoli operator   476 Sep 30 14:32 directory.h
-rw-------  1 troccoli operator   499 Sep 30 14:32 direntv6.h
-rw-------  1 troccoli operator 11426 Sep 30 14:32 diskimageaccess.c
```



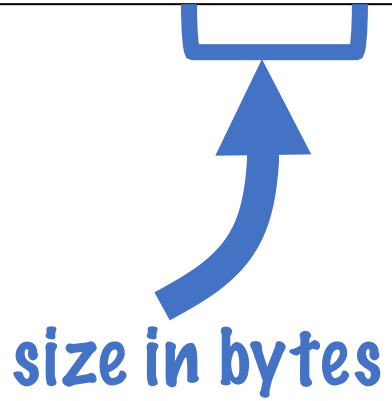
Filesystem Information

```
troccoli@myth60:~/assign1$ ls -al
total 60
drwx------  5 troccoli operator  4096 Oct  4 22:42 .
drwxr-xr-x 58 troccoli operator  6144 Oct  1 23:45 ..
-rw-------  1 troccoli operator  1920 Sep 30 14:32 chksumfile.c
-rw-------  1 troccoli operator  1063 Sep 30 14:32 chksumfile.h
-rw-------  1 troccoli operator   839 Sep 30 14:32 custom_tests
-rw-------  1 troccoli operator   509 Sep 30 14:32 directory.c
-rw-------  1 troccoli operator   476 Sep 30 14:32 directory.h
-rw-------  1 troccoli operator   499 Sep 30 14:32 direntv6.h
-rw-------  1 troccoli operator 11426 Sep 30 14:32 diskimageaccess.c
```



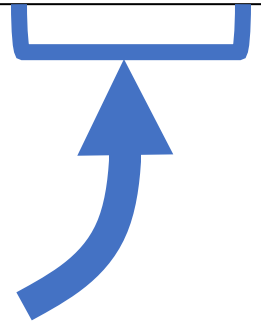
Filesystem Information

```
troccoli@myth60:~/assign1$ ls -al
total 60
drwx------  5 troccoli operator  4096 Oct  4 22:42 .
drwxr-xr-x 58 troccoli operator  6144 Oct  1 23:45 ..
-rw-------  1 troccoli operator  1920 Sep 30 14:32 chksumfile.c
-rw-------  1 troccoli operator  1063 Sep 30 14:32 chksumfile.h
-rw-------  1 troccoli operator   839 Sep 30 14:32 custom_tests
-rw-------  1 troccoli operator   509 Sep 30 14:32 directory.c
-rw-------  1 troccoli operator   476 Sep 30 14:32 directory.h
-rw-------  1 troccoli operator   499 Sep 30 14:32 direntv6.h
-rw-------  1 troccoli operator 11426 Sep 30 14:32 diskimageaccess.c
```



Filesystem Information

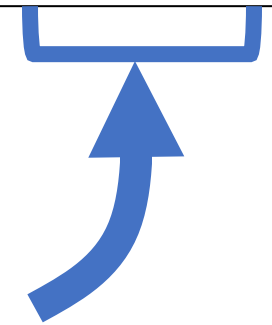
```
troccoli@myth60:~/assign1$ ls -al
total 60
drwx------  5 troccoli operator  4096 Oct  4 22:42 .
drwxr-xr-x 58 troccoli operator  6144 Oct  1 23:45 ..
-rw-------  1 troccoli operator  1920 Sep 30 14:32 chksumfile.c
-rw-------  1 troccoli operator  1063 Sep 30 14:32 chksumfile.h
-rw-------  1 troccoli operator   839 Sep 30 14:32 custom_tests
-rw-------  1 troccoli operator   509 Sep 30 14:32 directory.c
-rw-------  1 troccoli operator   476 Sep 30 14:32 directory.h
-rw-------  1 troccoli operator   499 Sep 30 14:32 direntv6.h
-rw-------  1 troccoli operator 11426 Sep 30 14:32 diskimageaccess.c
```



Group name

Filesystem Information

```
troccoli@myth60:~/assign1$ ls -al
total 60
drwx------  5 troccoli operator  4096 Oct  4 22:42 .
drwxr-xr-x 58 troccoli operator  6144 Oct  1 23:45 ..
-rw-------  1 troccoli operator  1920 Sep 30 14:32 chksumfile.c
-rw-------  1 troccoli operator  1063 Sep 30 14:32 chksumfile.h
-rw-------  1 troccoli operator   839 Sep 30 14:32 custom_tests
-rw-------  1 troccoli operator   509 Sep 30 14:32 directory.c
-rw-------  1 troccoli operator   476 Sep 30 14:32 directory.h
-rw-------  1 troccoli operator   499 Sep 30 14:32 direntv6.h
-rw-------  1 troccoli operator 11426 Sep 30 14:32 diskimageaccess.c
```



Owner name

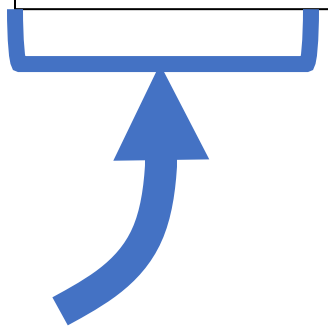
Filesystem Information

```
troccoli@myth60:~/assign1$ ls -al
total 60
drwx------  5 troccoli operator  4096 Oct  4 22:42 .
drwxr-xr-x 58 troccoli operator  6144 Oct  1 23:45 ..
-rw-------  1 troccoli operator  1920 Sep 30 14:32 chksumfile.c
-rw-------  1 troccoli operator  1063 Sep 30 14:32 chksumfile.h
-rw-------  1 troccoli operator   839 Sep 30 14:32 custom_tests
-rw-------  1 troccoli operator   509 Sep 30 14:32 directory.c
-rw-------  1 troccoli operator   476 Sep 30 14:32 directory.h
-rw-------  1 troccoli operator   499 Sep 30 14:32 direntv6.h
-rw-------  1 troccoli operator 11426 Sep 30 14:32 diskimageaccess.c
```

hard links

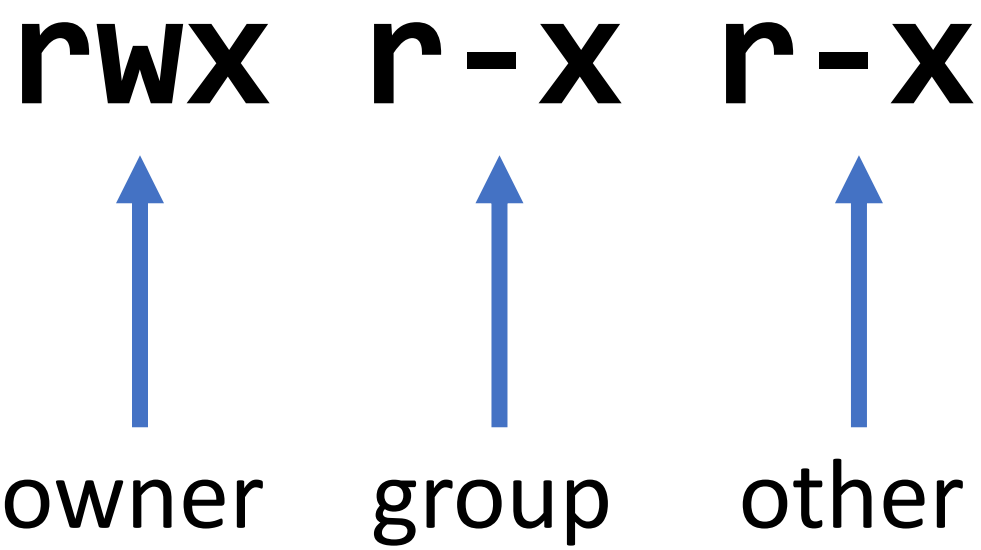
Filesystem Information

```
troccoli@myth60:~/assign1$ ls -al
total 60
drwx------  5 troccoli operator  4096 Oct  4 22:42 .
drwxr-xr-x 58 troccoli operator  6144 Oct  1 23:45 ..
-rw-------  1 troccoli operator  1920 Sep 30 14:32 chksumfile.c
-rw-------  1 troccoli operator  1063 Sep 30 14:32 chksumfile.h
-rw-------  1 troccoli operator   839 Sep 30 14:32 custom_tests
-rw-------  1 troccoli operator   509 Sep 30 14:32 directory.c
-rw-------  1 troccoli operator   476 Sep 30 14:32 directory.h
-rw-------  1 troccoli operator   499 Sep 30 14:32 direntv6.h
-rw-------  1 troccoli operator 11426 Sep 30 14:32 diskimageaccess.c
```



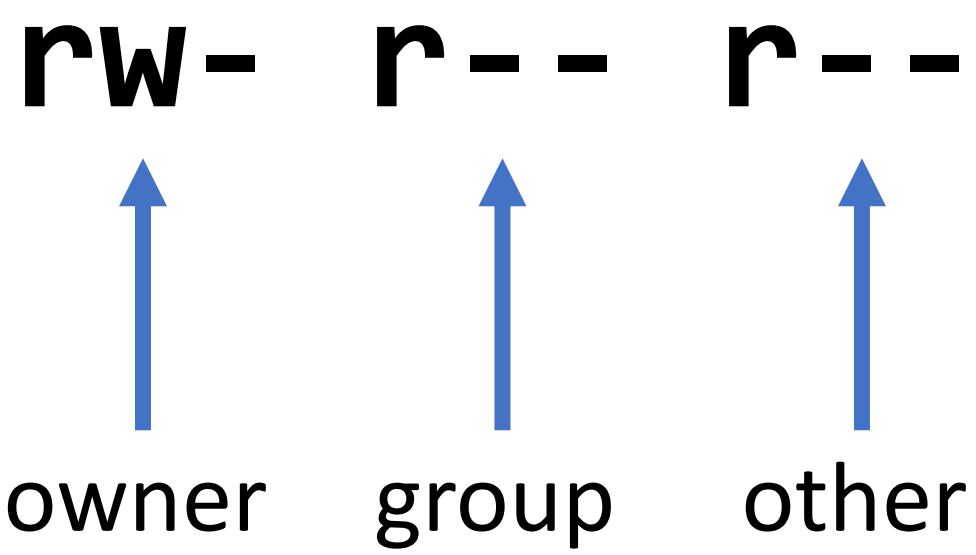
Type and permissions

Unix File Permissions



Here, the owner has read, write, and execute permissions, the group has only read and execute permissions, and the user also has only read and execute permissions.

File Permissions



We can represent permissions in binary (1 or 0 for each permission option):

- eg. for permissions above: 110 100 100
- we can further convert each group of 3 into one *base-8 digit*
  base 8 ("octal"): **6 4 4**
- So, the permissions for the above file would be **644**

Plan For Today

- **Recap**: filesystem design and modern filesystems
- Interacting with the filesystem as users
- **Interacting with the filesystem as programmers**
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice:** copying files

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

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.
- The operating system *kernel* actually 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).

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)

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.

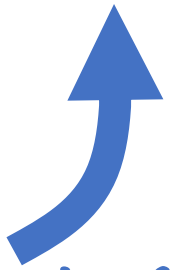
It's important to close files when you are done with them to preserve system resources.

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

Example: Creating a File

```
// Create the file
int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_EXCL, 0644);

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



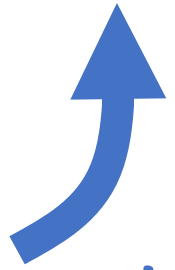
Open the file to be written to

touch.c

Example: Creating a File

```
// Create the file
int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_EXCL, 0644);

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



Create the file if it doesn't exist

touch.c

Example: Creating a File

```
// Create the file
int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_EXCL, 0644);

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

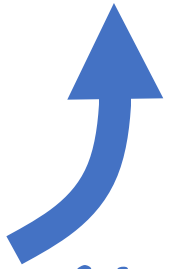
If it does exist, throw an error

touch.c

Example: Creating a File

```
// Create the file
int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_EXCL, 0644);

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



Attempt to set this file's permissions to rw for owner, r for all others. (note: octal numbers start with leading 0)

touch.c

Example: Creating a File

```
// Create the file
int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_EXCL, 0644);

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

touch.c

Plan For Today

- **Recap**: filesystem design and modern filesystems
- Interacting with the filesystem as users
- Interacting with the filesystem as programmers
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice:** copying files

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

read()

Call **read** to read bytes from an open file:

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

- **fd**: the file descriptor for the file you'd like to read from
- **buf**: the memory location where the read-in bytes should be put
- **count**: the number of bytes you wish to read
- returns -1 on error, 0 if at end of file, or nonzero if bytes were read

Key idea: read may not read all the bytes you ask it to! The return value tells you how many were actually read.

Key idea #2: the operating system keeps track of where in a file a file descriptor is reading from. So the next time you read, it will resume where you left off.

write()

Call **write** to write bytes to an open file:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- **fd**: the file descriptor for the file you'd like to write to
- **buf**: the memory location storing the bytes that should be written
- **count**: the number of bytes you wish to write from buf
- returns -1 on error, or otherwise the number of bytes that were written

Key idea: write may not write all the bytes you ask it to! The return value tells you how many were actually written.

Key idea #2: the operating system keeps track of where in a file a file descriptor is writing to. So the next time you write, it will write to where you left off.

Plan For Today

- **Recap**: filesystem design and modern filesystems
- Interacting with the filesystem as users
- **Interacting with the filesystem as programmers**
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice: copying files**

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

Example: Copy

Let's write an example program **copy** that emulates the built-in **cp** command. It takes in two command line arguments (file names) and copies the contents of the first file to the second.

E.g. **`./copy source.txt dest.txt`**

1. Open the source file and the destination file and get file descriptors
2. Read each chunk of data from the source file and write it to the destination file

`copy-soln.c` and **`copy-soln-full.c`** (with error checking)

File descriptors are a powerful abstraction for working with files and other resources. They are used for files, networking and user input/output!

Recap

- **Recap**: filesystem design and modern filesystems
- Interacting with the filesystem as users
- Interacting with the filesystem as programmers
  - System calls
  - **open()** and **close()**
  - **read()** and **write()**
  - **Practice:** copying files

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.

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