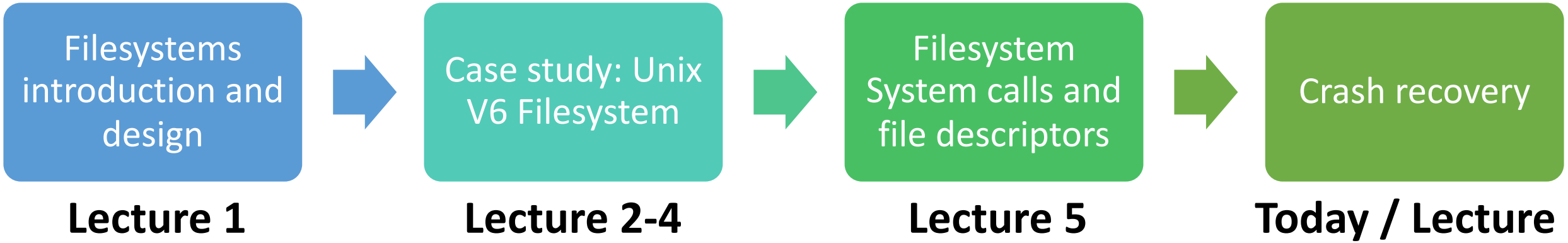
# CS111, Lecture 6
## Filesystem System Calls and Crash Recovery

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 14 through 14.1

**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




**assign1:** implement portions of the Unix v6 filesystem!

# Learning Goals

- Get practice working with file descriptors in programs
- Learn about the role of the free map and block cache in filesystems
- Understand the goals of crash recovery and potential tradeoffs

# Plan For Today

- **Recap and continuing**: file descriptors and system calls
- Announcements
- Free space management

```
cp -r /afs/ir/class/cs111/lecture-code/lect6 .
```

# Plan For Today

- **Recap and continuing**: file descriptors and system calls
- Announcements
- Free space management

```
cp -r /afs/ir/class/cs111/lecture-code/lect6 .
```

# 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.  They are tasks the operating system can do for us that we can't do ourselves.
- **open()** and **close()**, are 2 examples of system calls we use to interact with files.

# open()

A function that a program can call to open a file, and potentially create a file:

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

- **pathname**: the path to the file you wish to open
- **flags**: a bitwise OR of options specifying the behavior for opening the file
- **mode** (if applicable): the permissions to attempt to set for a created file
- returns a **file descriptor** representing the opened file, or -1 on error

Many possible flags (see man page). You must include exactly one of **O_RDONLY, O_WRONLY, O_RDWR**, which specifies how you'll use the file in this program.

- **O_TRUNC:** if the file exists already, clear it ("truncate it").
- **O_CREAT:** if the file doesn't exist, create it
- **O_EXCL:** the file must be created from scratch, fail if already exists

# 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 instance of that file in this program.
- Each program has its own file descriptors
- You can have multiple file descriptors for the same file
- 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 (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 we're done with it
  close(fd);
  return 0;
}
```

**touch.c**

Specify how we are going to use this file in *this program*

# 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**

Specify permissions for everyone on disk if this call creates a new file

# 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. E.g. if there aren't that many bytes, or if interrupted)

**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. E.g. if not enough space, or if interrupted)

**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.

# 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)

# Example: Copy

The **copy** program emulates **cp**; it copies the contents of a source file to a specified destination.

```
int main(int argc, char *argv[]) {
    int sourceFD = open(argv[1], O_RDONLY);
    int destinationFD = open(argv[2],
         O_WRONLY | O_CREAT | O_EXCL, kDefaultPermissions);

    copyContents(sourceFD, destinationFD);

    close(sourceFD);
    close(destinationFD);
    return 0;
}
```

# Example: Copy

The **copy** program emulates **cp**; it copies the contents of a source file to a specified destination.

```
int main(int argc, char *argv[]) {
    int sourceFD = open(argv[1], O_RDONLY);
    int destinationFD = open(argv[2],
         O_WRONLY | O_CREAT | O_EXCL, kDefaultPermissions);

    copyContents(sourceFD, destinationFD);

    close(sourceFD);
    close(destinationFD);
    return 0;
}
```

”create the file to write to, and it must not already exist”

# Example: Copy

The **copy** program emulates **cp**; it copies the contents of a source file to a specified destination.

```
void copyContents(int sourceFD, int destinationFD) {
      // Goal: while there's more data from source, read the next
      // chunk and write it to the destination.
}
```

# Example: Copy

The **copy** program emulates **cp**; it copies the contents of a source file to a specified destination.

```
void copyContents(int sourceFD, ir
    char buffer[kCopyIncrement];
    ...
}
```

Read in chunks of **kCopyIncrement** bytes

# Example: Copy

The **copy** program emulates **cp**; it copies the contents of a source file to a specified destination.

```
void copyContents(int sourceFD, int destinationFD) {
    char buffer[kCopyIncrement];
    while (true) {
        ssize_t bytesRead = read(sourceFD, buffer, sizeof(buffer));
        if (bytesRead == 0) break;

        ...
    }
}
```

Read a chunk of bytes. It may not be **kCopyIncrement** bytes! If **read** returns 0, there are no more bytes to read.

# Example: Copy

The **copy** program emulates **cp**; it copies the contents of a source file to a specified destination.

```
void copyContents(int sourceFD, int destinat
    char buffer[kCopyIncrement];
    while (true) {
        ssize_t bytesRead = read(sourceFD, b
        if (bytesRead == 0) break;
        size_t bytesWritten = 0;
        while (bytesWritten < bytesRead) {
            ...
        }
    }
}
```

Now we write this chunk of bytes to the destination file. We must loop until **write** writes them all.

# Example: Copy

The **copy** program emulates **cp**; it copies the contents of a source file to a specified destination.

```
void copyContents(int sourceFD, int destinationFD) {
    char buffer[kCopyIncrement];
    while (true) {
        ssize_t bytesRead = read(sourceFD, bu
        if (bytesRead == 0) break;
        size_t bytesWritten = 0;
        while (bytesWritten < bytesRead) {
            ssize_t count = write(destinationFD, buffer + bytesWritten,
                                  bytesRead - bytesWritten);
            bytesWritten += count;
        }
    }
}
```

Since **write** may write only some of the bytes, we need to just give it the *rest* of the bytes that it hasn't written yet.

# Example: Copy

The **copy** program emulates **cp**; it copies the contents of a source file to a specified destination.

```
void copyContents(int sourceFD, int destinationFD) {
    char buffer[kCopyIncrement];
    while (true) {
        ssize_t bytesRead = read(sourceFD, buffer, sizeof(buffer));
        if (bytesRead == 0) break;
        size_t bytesWritten = 0;
        while (bytesWritten < bytesRead) {
            ssize_t count = write(destinationFD, buffer + bytesWritten,
                                  bytesRead - bytesWritten);
            bytesWritten += count;
        }
    }
}
```

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

# File Descriptors and I/O

There are 3 special file descriptors provided by default to each program:

- 0: standard input (user input from the terminal) - STDIN_FILENO
- 1: standard output (output to the terminal) - STDOUT_FILENO
- 2: standard error (error output to the terminal) - STDERR_FILENO

**Programs always assume that 0,1,2 represent STDIN/STDOUT/STDERR.** Even if we change them! (eg. we close FD 1, then open a new file).

# Example: Copy

What is the smallest 1 line change/hack we could make to this code to make it print the contents of the source file to the terminal instead of copying it to the destination file?

```
int main(int argc, char *argv[]) {
    int sourceFD = open(argv[1], O_RDONLY);
    int destinationFD = open(argv[2],
         O_WRONLY | O_CREAT | O_EXCL, kDefaultPermissions);

    copyContents(sourceFD, destinationFD);

    close(sourceFD);
    close(destinationFD);
    return 0;
}
```

# Example: Copy

What is the smallest 1 line change/hack we could make to this code to make it print the contents of the source file to the terminal instead of copying it to the destination file?

```
int main(int argc, char *argv[]) {
    int sourceFD = open(argv[1], O_RDONLY);
    int destinationFD = open(argv[2],
         O_WRONLY | O_CREAT | O_EXCL, kDefaultPermissions);

    copyContents(sourceFD, STDOUT_FILENO);

    close(sourceFD);
    close(destinationFD);
    return 0;
}
```

# Example: Copy

What is the smallest 1 line change/hack we could make to this code to make it print the contents of the source file to the terminal instead of copying it to the destination file?

```
int main(int argc, char *argv[]) {
    int sourceFD = open(argv[1], O_RDONLY);
    int destinationFD = open(argv[2],
            O_WRONLY | O_CREAT | O_EXCL, kDefaultPermissions);

    copyContents(sourceFD, STDOUT_FILENO);

    close(sourceFD);
    close(destinationFD);
    return 0;
}
```

# Plan For Today

- **Recap and continuing**: file descriptors and system calls
- **Announcements**
- Free space management

# Announcements

- Lecture 5 hard links demo: Emacs was the culprit! ☺
  - Emacs default behavior is funky with hard links. But changes should be visible across all links to that payload data
- Section 1 grades posted to gradebook; solutions posted on course website
- assign1 updates:
  - printing error messages won't be required for grading, but we still highly encourage them to help with debugging! (Note: our solution does not print an error message if **`directory_findname`** can't find an entry)
  - No need to use heap allocation on this assignment

# Plan For Today

- **Recap and continuing**: file descriptors and system calls
- Announcements
- **Free space management**

# Crash Recovery

- Next, we're going back to filesystem design to talk about how filesystems can recover from crashes.
- To learn about this, there are two last features of filesystems that are important for us to know about: the **free list** (how free blocks on disk are tracked) and the **block cache** (a space in memory that stores frequently-used disk blocks).

# Free Space Management

- Early Unix systems (like Unix v6) used a linked list of free blocks
  - Initially sorted, so files allocated contiguously, but over time list becomes scrambled

More common: use a **bitmap**

- Array of bits, one per block: 1 means bock is free, 0 means in use
- Takes up some space – e.g. 1TB capacity -> $2^{28}$ 4KB blocks -> 32 MB bitmap
- During allocation, search bit map for block close to previous block in file
  - Want *locality* – data likely used next is close by (linked list not as good)

**Problem:** slow if disk is nearly full, and files become very scattered

# Free Space Management

More common: use a **bitmap** – an array of bits, one per block, where 1 means bock is free, 0 means in use.

- During allocation, search bit map for block close to previous block in file

**Problem:** slow if disk is nearly full, and blocks very scattered

- Expensive operation to find a free block on a mostly full disk
- Poor *locality* – data likely to be used next is not close by

**Solution:** don't let disk fill up!

- E.g. Linux pretends disk has less capacity than it really has (try **df** on myth!)
- Increase disk cost, but for better performance

# Recap

- **Recap and continuing**: file descriptors and system calls
- Announcements
- Free space management

**Next time:** more about crash recovery

**Lecture 6 takeaway:** File descriptors and **read/write** let us read and write files. The free list tracks free blocks on disk, and is commonly implemented using a bitmap.