CS 521: Systems Programming

File I/O

Lecture 9

Today's Schedule

- Reading Files
- Writing Files

Today's Schedule

- **Reading Files**
- Writing Files

Opening a File: fopen

C provides a function for opening files: `fopen()`

```
/* This opens the file specified by the
 * first command line argument: */
printf("Opening file: %s\n", argv[1]);
FILE *file = fopen(argv[1], "r");
if (file == NULL) {
    perror("fopen");
    // error handling
}
/* Note the "r": open for reading */
```

It returns a `FILE *`, which represents an open file

Open Modes

- The basics:
  - r -– read
  - w -– write
  - a -– append
- This isn't the full story, however: each mode can be followed by a '+'
  - r+ – open for read and write, file must exist
  - w+ – open for read and write, file is created if not present
- There are more details in the man page for `fopen()`

Reading Line by Line: fgets

- Once we have opened a file, we need to read it
- A common approach is reading line by line via `fgets`:

```
char line[500];
while (fgets(line, 500, file) != NULL) {
        /* Process the line */
}
```

- This uses a 500-character buffer to store the line
- `fgets` will also stop once it finds a newline (`\n`) character

Rewinding a File

- When you reach the end of a file, you'll usually get either `NULL` or `EOF` for your return value
  - In the case of fgets, `NULL`
- This tells you that you've reached the **E**nd **O**f **F**ile
- If you want to loop through the file again, go back to the start:
  - `fseek(file, 0L, SEEK_SET);`
  - `rewind(file); /* Note: old, deprecated */`
  - (You can also re-open the file 💩)

Cleaning Up

- It's good practice to also close your files when you're done with them:
  - `fclose(file)`
- Each file you open uses up a **file descriptor**
- The operating system imposes limits on how many file descriptors can be open per program
- When you open several files, don't forget to close them when you're done!

Today's Schedule

- Reading Files
- **Writing Files**

puts and fputs

- If you don't need formatting functionality, you can use `puts` to "put a string" to your terminal
- `fputs` is similar, but lets you specify a file:
  - `FILE *file = fopen("my_file.txt", "w");`
  - `fputs("Hi there, file!\n", file);`
- **Note**: we need to fopen the file with 'w' mode.
- Want to write a single character? `fputc`.

fprintf

- You can print to `stderr` with `fprintf`
  - "File printf"
- `stderr` is represented as a file that is automatically opened for you
- So if we want to write data to a file, just pass it to `fprintf` after opening it:
  - `fprintf(file, "My name is: %s", "Bob");`

Cleaning Up (again!)

- Make sure you close the files that you are writing!
  - `fclose(file)`
- If not closed, there is no guarantee that the output will actually be written to the destination file
  - Generally files on the disk are buffered ***more*** than, say, `stdout`
- You can also use `fflush()` on a file you've opened

Flushing Files

- `fflush()` empties a file's **buffer**
- One of the most common places you'll see `fflush` used is after printing text to the terminal that you want displayed **NOW**
- ...but it can also be used to ensure data has been written to a file
  - Note: the OS may still buffer some data
  - (we can compare flushing, not flushing, and flushing in Python!)

Unix Utility of the Week: cat

- Let's take a look at how another utility works!
- `cat` is useful for *concatenating files*
  - Sounds like something we can definitely do now, right?