CS 521: Systems Programming

Input/Output (I/O)

Before we Start…

- My office hours
- Li's office hours
- What am I going to name my VM again??
- We'll look at ssh aliases and VSCode in a minute…

Input/Output

- *Most* useful programs will provide some type of input or output
- Our main approach thus far is printing via `printf`
  - Whooo! 💩
- We can read information with `scanf`:

```
printf("Please enter your age: ");
int age;
scanf("%d", &age);
printf("You are %d years old, huh? Wow!\n", age);
```

I/O Streams

- By default, we print data to `stdout` and read data from `stdin`
- A few other ways to read input:
  - `getc` – get a character
  - `fgets` – read a line
    - Why not `gets`? Check the man pages: `man gets`

Working with I/O Streams

- Since we've been using **stdin**, we can also use file redirection to effectively simulate the user typing:
  - `./program < input_file.txt`
    - Makes it seem like the user is typing the commands in `input_file.txt`
- There are more options:
  - Command line arguments
  - File I/O

Utility of the week: cat

- Meow!
- As a case study for I/O, we are going to look at the `cat` command
- Then you're going to build your own version of `cat` for Lab 2!
- Let's check out how `cat` works and come up with a strategy for implementing it.

Today's Schedule

- I/O Streams
- Command Line Arguments
- Reading Files
- Writing Files

Today's Schedule

- **I/O Streams**
- Command Line Arguments
- Reading Files
- Writing Files

Input/Output Streams

- Each program gets allocated three I/O streams by default:
  - `stdout` (standard output)
  - `stderr` (standard error)
  - `stdin` (standard input)
- These streams have different functions…

stdout

- When you call `printf`, you are writing to `stdout`
- This stream is designed for general program output; for example, if you run `ls` then the list of files should display on `stdout`
- You can pipe stdout to other programs:
  - `ls -l / | grep 'bin'`
- …or redirect to a file:
  - `ls -l / > list_of_files.txt`

Special Characters

- `|` – pipe: sends stdout of the program on the left to the stdin of the program on the right
  - `cat something.txt | sort`
- `>` – output redirection: send stdout to a file instead of the terminal
  - `cat something.txt > something_else.txt`
- `>>` – output redirection, but will append to the file instead of overwriting
- `<` – input redirection: read from file instead of stdin

stderr

- The standard error stream is used for diagnostic information
  - Log messages often print to `stderr`
  - Program “usage” messages often go there too
- This way, program output can still be passed to other programs/files but we’ll still see diagnostics printed to the terminal
- Helps us know when something went wrong
- Unlike stdout, stderr is not *buffered*
  - Will be flushed to the terminal immediately

stdin

- The final stream, stdin, is how we provide program input (via `scanf`, for example)
- This can be entered by the user, or we can redirect input directly into a program:
  - `./my_prog < ./test_file.txt`
  - `ls -l / | grep 'bin'`
    - (`grep` receives the input from the pipe rather than the user) – wait, have we seen `grep` before?

“printf debugging”

- Let’s say we’re working on a bug and want to determine what’s wrong… `printf` to the rescue!
  - *cough*, *cough*, don’t do that, use logging… we’ll talk about this later!
- Unfortunately, sometimes printing to the terminal can be misleading
- The `printf()` may execute, but the program crashes before any output is displayed
- This occurs due to **Input/Output (I/O) Buffering**

I/O Buffering

- Input/output operations are slow: they have high **latency**
  - Printing to the terminal outputs to `stdout`
  - Writing to disk or controlling an external hardware device are also I/O operations
- These devices generally operate on **buffers**
  - Example: our terminal has a 8-byte buffer; we fill up the buffer before asking it to print the text
- You may have used buffered streams in Java to get better performance
- Buffered I/O collects multiple I/O operations, combines them, and then executes them as one big operation

Flushing the Output Stream

- Sometimes when debugging your program crashes before the buffer gets cleared
  - Data is lost before the buffer is flushed
- To make the print operation happen *now*, we need to flush the output stream:
  - `fflush(stdout);`

Why not Always Flush?

- Flushing the buffer when it's not full or at inopportune times for the OS incurs more latency
- Performing the print operation takes several steps, and that takes time
- We can compare the performance of two C programs, one that flushes I/O and one that does not
  - Demo: `flush.c`

Today's Schedule

- I/O Streams
- **Command Line Arguments**
- Reading Files
- Writing Files

Input from the Command Line

- Passing command line arguments is a common form of input:
  - `./my_program testmode ./file.txt`
- We see this often with Unix utilities:
  - `ls -l /my/directory`
- This makes providing input to a program easier, and allows for scripting as well:
  - `./my_program ${MODE} ./file.txt`

Command Line Arguments

- You may have noticed an alternative version of our main(void) function:
  - `int main(int argc, char *argv[])`
  - `int main(int argc, char **argv)`
- This allows us to accept and process command line arguments
  - For example, when you run `git status` the string ‘status’ is passed to the main method
  - In fact, so is the name of the program, ‘git’

Argument Attributes

- We receive two parameters:
  - `argc` – the number of command line arguments
  - `argv` – the arguments themselves
- Some notes:
  - argc will always be at least 1
  - argv[0] will always be the name of your program
- So if we want one argument from the user, we test whether argc == 2

Numeric Conversions

- You may want to accept more than just strings as your command line arguments
- To get the numeric value of a string, there are a couple basic functions:
  - `atoi()`, `atol()`, `atof()`, ...
  - (string to X, where X is the data type)
- However, these have a weakness: if a conversion error occurs, they just return 0 (a valid number!)
- `strtol`, `strtof`, and `strtod` allow error checking

Converting with Error Checking

- Let's take a look at the `convert.c` example to see how to do safe conversions

Converting Back

- One question you might ask is: “if `atoi()` exists, is there an `itoa()` ?”
- The answer is: it depends
  - It’s not part of the standard, but many C libraries include it
- My favorite way to convert numbers to strings is with `sprintf` :
  - `sprintf(buf, "%i", some_int);`

Demo: Command Line Arguments

- Let's experiment with command line arguments…

Today's Schedule

- I/O Streams
- Command Line Arguments
- **Reading Files**
- Writing Files

Opening a File: fopen

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

```c
/* 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

- I/O Streams
- Command Line Arguments
- 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