CS 220: Introduction to Parallel Computing

Input/Output II

Lecture 8

Today's Agenda

- Debugging and I/O Buffering
- I/O Streams
- Writing files

Today's Agenda

- **Debugging and I/O Buffering**
- I/O Streams
- Writing files

printf Debugging

- One of the simplest ways to debug a program is printing information to the terminal
- For example, HW1:
  - If your min/max functionality isn't working, then you can print the value before and after the change
  - You can also print the values as you loop through the arrays
- This is fairly effective, and is used in production code when combined with preprocessor directives

Smarter printf Debugging

- From stackoverflow (https://stackoverflow.com/questions/5765175/macro-to-turn-off-printf-statements):

```
#define LOG(fmt, ...) \
do { \
  if (DEBUG) fprintf(stderr, fmt, __VA_ARGS__); \
} while (0)
```

- What is the benefit of this approach?

Demo: Preprocessor Debug

printf Debugging: Problems

- Let's say we're working on a bug and want to determine what's wrong… printf to the rescue!
- 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 (1/2)

- Input/output operations are expensive: they have high **latency**
  - Printing to the terminal is **outputting** data to the standard output stream
  - Writing to disk or controlling an external hardware device are also I/O operations
- These devices generally operate on **buffers**
  - Example: our serial console has a 2-byte buffer; we fill up the buffer before asking it to print the text

I/O Buffering (2/2)

- You may have used **buffered** streams in Java to get better performance
- You might have also noticed what happens when you forget to close a stream…
- Buffered I/O collects multiple I/O operations, combines them, and **then** executes them
  - Important: matching up with hardware capabilities

Why is I/O Buffering Used? (1/2)

- Printing to your terminal accesses a virtual device, a **pseudoterminal**
  - Back in the days of mainframes, everybody had a physical terminal on their desk
- Accessing a device, virtual or otherwise, requires a **system call**
  - Privileged operations executed by the OS that interact with the hardware
  - These take time

Why is I/O Buffering Used? (2/2)

- I/O buffering **batches** the writes into groups to reduce the amount of system calls
- Rather than printing:
  - 'H', 'e', 'l', 'l', 'o'
- It's faster to just print:
  - 'Hello'
- Unfortunately, if your program crashes before the buffer is flushed, everything is lost

Sizing our Buffers

- Large buffers are expensive
  - See: high-end networking hardware
  - If they're too large, we're just wasting space
- Small buffers may increase latency
  - More I/O operations
    - More round trips
    - More context switching at the OS level

Flushing the Output Stream

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

Why not always call fflush()?

- 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
  - Example on schedule: flush.c

Demo: fflush

Another Tip

- I/O buffers are often flushed when a newline (\n) is encountered
- If you make sure to add a newline character to your print statements, they **generally** will be flushed to the display and useful for debugging
  - Not a guarantee though – the only way to know for sure is to call fflush each time!

Today's Agenda

- Debugging and I/O Buffering
- **I/O Streams**
- Writing files

Input/Output Streams

- I mentioned the standard output stream earlier… **stdout**
- Each program gets allocated three 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 type ‘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`

stderr

- The standard error stream is used for diagnostic information
- You may recall our LOG macro was printing to stderr
- 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

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 pipe input directly into a program:

```
ls -l / | grep 'bin'
```

Writes file list to **stdout**

Reads file list from **stdin**,
Writes matching files to **stdout**

Today's Agenda

- Debugging and I/O Buffering
- I/O Streams
- **Writing files**

fprintf

- As we saw earlier, we can print to stderr with **fprintf**
  - File printf
- stderr is actually a file – in fact, on Unix systems most devices are represented as files
  - Try `ls /dev` to view the devices on your machine (includes macOS)
- So if we want to write data to a file, just pass it in:
  - `fprintf(file, "My name is: %s", "Bob");`

puts & 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.
- Why does the file arg come **after**? Well, it **is** C…

sprintf

- One last thing we can do: print to a buffer

```
char buffer[100];
sprintf(buffer,"Hi, my favorite number is %d!", 42);
printf("Buffer contents ->%s\n", buffer);
```