Assignment 4
XXD

by Jess Srinivas
CSE 13S, Spring 2024
Document version 1 (changes in Section 4)

Due Wednesday May 8th, 2024, at 11:59 pm
Draft Due Monday, May 6th, 2024, at 11:59 pm

1 Introduction

In past assignments, you were shown some files that had weird, sometimes invisible differences. One challenge with files like these is finding an appropriate way to make them human readable, as not every character is printable. To make this easier, we can use tools like `xxd`. `xxd` on Unix systems is a very complicated program with many options, but in its simplest form, it displays a binary file in its hexadecimal representation. Feel free to play around with the `xxd` program. In this assignment, you will start to build your own version of this program, with all of the basic functionality of `xxd`.

2 Helpful information

Buffered Reading

This assignment is your first foray into reading files. Many programs use functions such as `fread`, which is a function that performs buffered reading. This function will block (wait) until all the requested data is ready or until `EOF` is reached. This behavior is caused by buffering, which loads data into an array (buffer) as it's read, and then returns all the requested data at once. In CSE 130 you will see why the blocking behavior can be detrimental to performance, and how creating your own buffer may be advantageous.

Instead of using `fopen()` and `fread()`, which do the buffering for you, you will be using `open()` and `read()`. These functions are "System Calls" that are included in `<fcntl.h>` and `<unistd.h>` respectively. A system call, or "syscall" for short, is a function that engages directly with the operating system, and in these cases requests it to read files from the disk. This menas that there are no abstractions to help you! These functions will return data as soon as it is available, but they may not return all of the requested data. To remedy this, we can repeatedly request data and load it into our own buffer until we have enough to process. We can then repeat this step until we have processed all of our data.

Another critical difference between `fopen()` and `open()` is that while `fopen` returns a file stream pointer (`FILE*`), `open()` returns a file descriptor, which is just an `int` corresponding to the file's index on your program's file descriptor table. For more information about these functions, you may consult the `man` pages.

3 Your Task

The output of the `xxd` program

`xxd` has a lot of arguments, most of which we will not use. Your job will be to replicate the following behaviors:

- If you supply no arguments, `xxd` will read in from `stdin` and print to `stdout`.

- If you supply one argument, it is treated as a filename, and `xxd` will read from that file, if it exists. It still will print to `stdout`.

The `xxd` program (and thus, our example program) has a well defined structure for its output. Here is the output for `xxd` on a file with the contents `0123456789abcdef\n`.

```
00000000: 3031 3233 3435 3637 3839 6162 6364 6566  0123456789abcdef
00000010: 0a                                       .
```

In the leftmost column is the index of the first byte. This is in hexadecimal, padded to 8 digits. It is followed by a colon and a space.

In the middle is the hex values of the bytes passed in. They are padded to 2 hex digits and are in groups of two bytes. There are 8 groups (16 bytes) one space between each group, and 2 spaces at the end of the column. If EOF (end of file) is reached before we print all 8 groups, we print 2 spaces instead of the hex digits.

In the final column is the ASCII representation. Any character between ASCII 32 and ASCII 126 (inclusive) is printed. Any character outside of that range is replaced with a `.` character. If EOF is reached before we print all 16 bytes, we do not print anything else. There is a newline at the end of this column.

If EOF is not reached on any given read, it waits until 16 bytes are entered. You can try this behavior by using `stdin` as your input.

Errors

In this assignment, the only way you will handle errors is by returning a non-zero error code. The only errors you *must* handle are if a filename is invalid or more than one argument is supplied. In all other error circumstances, your program must exit cleanly. This includes cleaning up memory before exit.

Workflow

1. Complete and submit your `design.pdf` draft by Monday, May 6$^{\text{th}}$.
2. Create a buffered reader that reads 16 bytes at a time.
3. Create tests for your program. We provide a file, `delayinput.sh`, to help you with this. This script reads each line of a file (whose name is provided in a command-line argument) with a one-second delay between lines.
4. Build your program, naming it `xd` (to avoid confusion with the existing program `xxd`).
5. If you have time, make a copy of `xd.c` called `bad_xd.c`. You can then modify your Makefile to add a target for `bad_xd`. (See more about `bad_xd` below.)
6. Submit your final commit ID by Wednesday May 8$^{\text{th}}$

Assignment specifics

Submission

These are the files we require from you:

- `Makefile`: This should have rules for `all`, `xd`, `xd.o`, `clean`, and `format`. We also require the same flags as usual, namely

  `-Werror -Wall -Wextra -Wconversion -Wdouble-promotion -Wstrict-prototypes -pedantic`

- `xd.c`: This file can contain all your code. You may also choose to make more files if it helps your code be more readable.
- `design.pdf` This file should answer the questions given in the template.

- `runner.sh` this file should run all your tests. You are welcome to use previous iterations of this file as a resource.
- `tests`: This folder is where you will write tests.
- `bad_xd.c`: **OPTIONAL** This file should function identically to `xd.c`.

Do's and Don'ts

You must do the following:

- Format *all* of your C files with the `clang-format` file we provide.
- If you choose to write `bad_xd`, be sure to make sure it functions identically to `xd.c`.
- If you choose to write `bad_xd`, detail how you shortened it in your design.
- Output only to `stdout`
- Read into a buffer of at least size 16.

You may not do any of the following:

- Have a comment with `clang-format off`.
- Use `fopen`, `fread`, or any other buffered reader.
- Have memory leaks or errors
- Use any `#define`s that would cause your program to have mismatched braces, parenthesis, or quotes.
- Adding anything to your makefile that helps reduce the size of your program
- Run any external executibles
- Use `#include` with any `bad_xd.c` code that you write
- Attempt to read fewer than 16 bytes at a time.
- Use `#define` to make anything equal a semicolon or curly brace.
- Use any other tricks that don't let clang-format work as expected.

Extra Credit

You can get extra credit if your program, `bad_xd` is less than 1000 characters. For every 15 characters (rounded up) less than 1000, you will get one extra credit point. For example, there is a known working solution which is 688 characters, which would receive 21 points of extra credit.

4 Revisions

Version 1 Original.