Assignment 7 – XXD

Asaveri Rao

CSE 13S – Fall 2023

Purpose

The major purpose behind this program is to be able to run xxd where it outputs a binary file into hexadecimal form. Traditionally the fopen, and fread would be implemented to do so, as they act as the buffer. However, the logic which these functions utilize versus functions like open and read vary greatly. In this program, the open and read functions are utilized to request data over and over again and uploaded into a customized buffer.

How to Use the Program

This program is fairly simple to maneuver and use. Due to the purpose of this program being simple and less on the complex side, to run or execute this program, all the user has to enter is, ./xd and the file name of the binary file that they want to format in hexadecimal format. For example if I had binary file called "testfile", to run the code on this file, I would enter ./xd. testfile. The programmer should clang format the files, and run make, make format, make clean to make sure it is properly formatted.

Program Design

The program design for this assignment is on the simpler side, it mainly consists of a custom buffer reader within the xd program. Due to the simplicity of the code, I added psuedocode of the xd.c file. There are a few errors that could occur as a result of something not executing correctly in this program. One of them being an error with the command-line and how many arguments are entered, if not entered correctly it would return EXITFAILURE. There could also be an error with opening the file entered in the command line, would also return EXITFAILURE. Due to 'read' being used, it could also result in -1 being printed if it wasn't read correctly. To run the program you have to run some make commands like make, make format, make clean to make sure it compiles properly. Now more on the inner aspect of the file, if the buffer size does not match what is expected in the file, or if there is an overflow, it could result in issues within BUFFERSIZE and it will not execute properly.

psuedocode of custom buffered reader:

```
function buffered_reader takes a fileDescriptor:
    declare a byte BUFFER_SIZE
    initialize bytesRead, offset, and totalBytes to 0

    while true:
        if bytesRead is positive:
            increment totalBytes by the # of bytesRead
            if totalBytes is equal to or greater than BUFFER_SIZE:
                call function print_hex_ascii_line
                increment offset by BUFFER_SIZE
                move remaining bytes in buffer to the beginning
                reduce totalBytes by BUFFER_SIZE
        else if bytesRead is -1 and error is not an interrupt:
            print error message
            break;

    if totalBytes is greater than 0 after the loop:
        function print_hex_ascii_line
```

psuedocode of xd program:

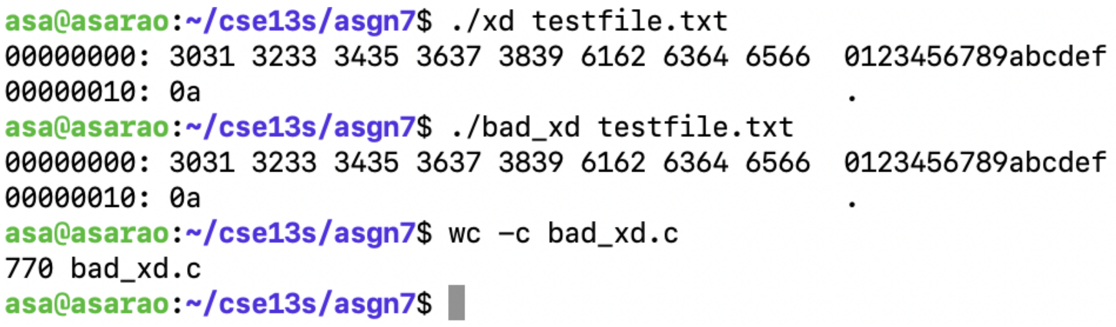
```
function print_hex_ascii_line takes a byte buffer, its size, and an offset:
    initialize loop index i
    print the offset in hexadecimal followed by colon & space
    loop from i equals to 0 to size:
        print each byte in buffer to hexadecimal format
        after every two bytes, print a space
    for the remaining bytes:
        print two spaces and additional space after two bytes
    print a space
    loop from i equlas to 0 to size:
        if the byte is a printable ASCII character:
            print the character
        else:
            print dot
    print newline

custom buffered_reader pseudocode included above

main function execution starts here:
    variable for fileDescriptor declared
    if two arguments are passed:
        open the file for reading and store the file descriptor
    else:
        set fileDescriptor to standard input
    if opening the file fails:
        print error message and return exit failure
    call buffered_reader with the fileDescriptor
    close the file descriptor
    return exit success
end of program
```

Results

. My results followed what the expected output was supposed to be. After running ./xd ¡filename¿ it was supposed to produce the hexadecimal format of the binary file. Thus the results are executed properly and the code overall is working correctly. I did partake in the extra credit, and try to shorten my code under a 1000 characters. I am currently at 770 characters which includes white spaces. I shortened the code with these following mechanisms. The bad_xd.c file is significantly shorter than my xd.c file. One of the first techniques I used was header inclusions. I used a few more headers in my xd.c file, but got rid of three in bad_xd.c, I only used essential headers like fcntl.h, etc. This definitely helped decrease the character count. One of the major differences between xd.c and bad_xd.c was how functions were defined and formatted. In xd.c I had a very organized and easy to follow format of the various functions. For example, in xd.c I had print_hex_ascii as a function, buffered_reader as a function, and then my main function to perform the actual



```
asa@asarao:~/cse13s/asgn7$ ./xd testfile.txt
00000000: 3031 3233 3435 3637 3839 6162 6364 6566  0123456789abcdef
00000010: 0a                                       .
asa@asarao:~/cse13s/asgn7$ ./bad_xd testfile.txt
00000000: 3031 3233 3435 3637 3839 6162 6364 6566  0123456789abcdef
00000010: 0a                                       .
asa@asarao:~/cse13s/asgn7$ wc -c bad_xd.c
770 bad_xd.c
asa@asarao:~/cse13s/asgn7$ 
```

Figure 1: Screenshot of xd and bad_xd outputs

code. In bad_xd.c, I had just a main function that contained the same functionality as my printascii and buffered_reader functions but it was combined into one to cut down the characters. The output logic was separated in xd.c, but also combined into the main function in bad_xd.c to help cut down on characters. Another technique that helped me cut down a lot of character was the shortened variable names. In xd.c I used full words like buffered_reader, and so on, however in bad_xd.c to conserve on character count, I shortened it to a single letter like 'b' for buffer or 'f' for file descriptor. In terms of readability, it isn't the most effective, however the purpose of the extra credit was to achieve a lower character count. There was also in-depth error handling in xd.c, however in bad_xd.c it was much more simplified. In terms of logic, loops and conditions were much shorter in bad_xd.c which is where I also used ternary operators to save characters. I also cut down generally on any repetitive code that seemed unnecessary and not helpful for functionality in bad_xd.c.