

Assignment 5 – Buffered I/O

Description:

This project implements the `b_open`, `b_read`, and `b_close` set of functions that help manage file control blocks to allow users to open, read, and close files. When called `b_open` will retrieve a file descriptor and prepare a file control block/member of the `fcblArray` at its respective position. The `b_read` function when called will read data to a buffer within a file control block and will then copy to a buffer provided by a user and supply them with their requested amount of bytes. Finally, `b_close` will free the allocated buffer within a file control block and reset other values for future reads if a user accesses a new file.

Approach:

To begin implementing the three functions `b_open`, `b_read`, and `b_close`, I intend to start by modifying the file control block array with variables that will help manage reading from a file control block buffer to a user buffer. I will do this by creating the buffer within the file control block followed by an integer that will act as its index to track the amount of data occupying the buffer. I will also create two integers that will be for tracking the position in the file in terms of bytes and another in terms of blocks. After doing this I intend to work on implementing `b_open` and `b_close` prior to implementing `b_read` since these would be able to work independently of `b_read`.

`b_open`

For `b_open` I intend to start by creating a file descriptor and checking its value to ensure it is valid for use, from here I will then gather the information of the file being opened using the name supplied to `b_open` and after checking that the information is present I will allocate the buffer of the current file control block and upon successful allocation, I will set the other variables I will create in the file control block array to zero and return the file descriptor I create.

`b_close`

In `b_close` I will deallocate the file control block buffer using `free` and reset the values of the rest of the file control block array values for future use. I will do this by setting all the integers within the file control block to 0 and reverting the file info to a `NULL`.

`b_read`

For `b_read` I intend on implementing the remaining portion of the function using `if` statements in the following order, one that checks whether there is excess data in the current file control block buffer from a previous read, another that checks if the count being requested exceeds the size of a single block, and lastly a base case that works by checking if the count is greater than 0 which I plan for it to work when during the first read call and if there is still bytes being requested after the other two cases. I will implement the second and third cases in similar manners by setting the number of blocks that need to be read, finding the current location in the file, and reading to a buffer by calling `LBAread`. The main difference between these two cases is that if there is a request for more than one block the user's buffer is the one I plan to read to using `LBAread` instead of the current file control block's buffer. The only other difference

I intend to make is by incrementing a bytesCopied integer using the number of blocks that have been read multiplied by the size of a block in bytes in the case that the count exceeds a single block. The base case will only read a single block to the current file control block buffer and I will then have it copied into the user's buffer, the bytesCopied in this case will be incremented using the count itself. I plan to have all three of these cases end with the bytePosition integer being incremented and the count itself being decremented. in the case of the current file control block buffer still containing bytes I will create a temporary count integer and set it to the count itself or the space available in the buffer depending on which of the two is smaller. I will then follow this by copying the contents of the current buffer to the user's buffer, and increment the bytesCopied with the temporary count.

Issues and Resolutions:

Now that I have completed the assignment majority of the issues I had throughout the creation of the b_read implementation turned out to be linked to a single overarching problem within my logic for the buffering. I was not taking into account the order of operations before and after reading using LBAreadd and copying using memcpy. I originally was checking if the buffer had data, and then if the count was larger, but instead of decrementing the count I was leaving it as is and checking the buffer if it was empty. This was bound to fail since the buffer index would never be updated in a manner where it would trigger this case. From here I was able to correct my cases after I began decrementing the count at the end of each case after bytes were copied to the user and checking if it was still greater than zero.

Before Correction

```
if(fcbArray[fd].spaceUsed > 0){}  
if(count > B_CHUNK_SIZE){}  
if(fcbArray[fd].spaceUsed == 0){}
```

After Correction

```
if(fcbArray[fd].spaceUsed > B_CHUNK_SIZE){}  
if(count > B_CHUNK_SIZE){}  
if(count > 0){}
```

The other smaller issues I had were due to how I was calculating the current position in the file as I was not using the return value of LBAreadd as I did prior so after having it saved to an integer and using this to increment my block position it helped me get closer to a final solution. I was also incrementing the block position before reading using

```
fcbArray[fd].blockPosition = fcbArray[fd].bytePosition / B_CHUNK_SIZE;
```

which should have been done after reading with LBAreadd with

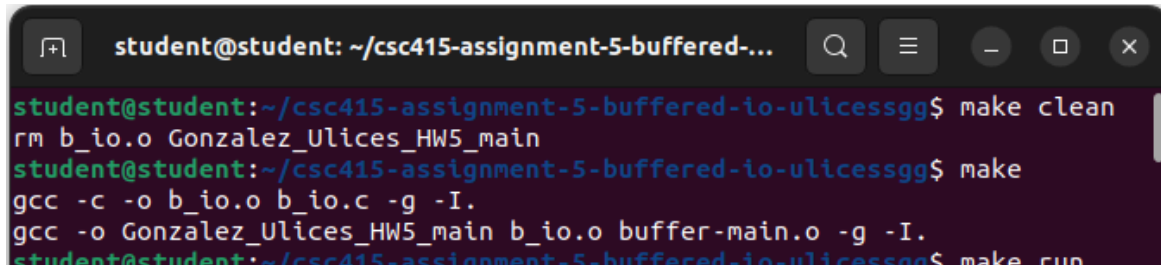
```
fcbArray[fd].blockPosition += blocksCopied;
```

Lastly, I had not been handling data from prior reads properly as I was under the impression I had properly offset the buffer but in reality, I was doing it incorrectly and I was able to fix it by not relying on pointer arithmetic and just checking what could fit within the user's buffer which I was able to fix using a temporary variable and the following if-else statement

```
if(count < (B_CHUNK_SIZE - fcbArray[fd].bufferUsed)){tempCount = count;}  
else{tempCount = B_CHUNK_SIZE - fcbArray[fd].bufferUsed;}
```

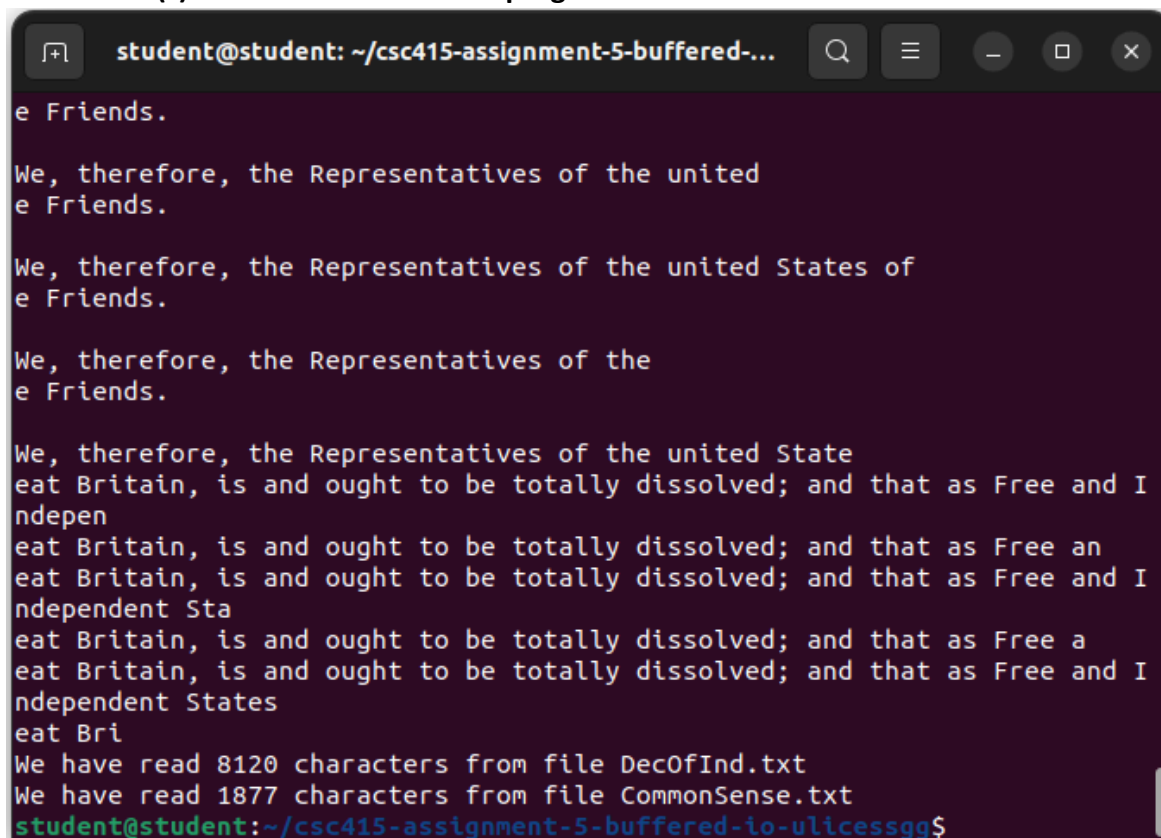
Analysis: N/A

Screen shot of compilation:



```
student@student: ~/csc415-assignment-5-buffered-io-ulicessgg$ make clean
rm b_io.o Gonzalez_Ulices_HW5_main
student@student:~/csc415-assignment-5-buffered-io-ulicessgg$ make
gcc -c -o b_io.o b_io.c -g -I.
gcc -o Gonzalez_Ulices_HW5_main b_io.o buffer-main.o -g -I.
student@student:~/csc415-assignment-5-buffered-io-ulicessgg$ make run
```

Screen shot(s) of the execution of the program:



```
student@student: ~/csc415-assignment-5-buffered-io-ulicessgg$ ./Gonzalez_Ulices_HW5_main
e Friends.

We, therefore, the Representatives of the united
e Friends.

We, therefore, the Representatives of the united States of
e Friends.

We, therefore, the Representatives of the
e Friends.

We, therefore, the Representatives of the united State
eat Britain, is and ought to be totally dissolved; and that as Free and I
ndepen
eat Britain, is and ought to be totally dissolved; and that as Free an
eat Britain, is and ought to be totally dissolved; and that as Free and I
ndependent Sta
eat Britain, is and ought to be totally dissolved; and that as Free a
eat Britain, is and ought to be totally dissolved; and that as Free and I
ndependent States
eat Bri
We have read 8120 characters from file DecOfInd.txt
We have read 1877 characters from file CommonSense.txt
student@student:~/csc415-assignment-5-buffered-io-ulicessgg$
```