I. Documentation

Environment and Overview

This assignment involved creating a user-made library of file open, close, input and output methods which simulate the function of the <stdio.h> library included with C and C++. The program was written with Eclipse IDE for use in a Linux environment, as the implementation hinges on several UNIX system calls including read(), write(), lseek(), open() and close(). Additionally, UNIX flags for the mode of the open file (r, r+, w, w+, a, a+) are used when calling fopen().

driver.cpp runs through all the user-implemented methods to assess their correct function, while eval.cpp calculates the speed of these user-defined methods compared to UNIX system calls and the UNIX-original <stdio.h>.

Technique to Implement <stdio.h>

In order to build a "stdio.h" that operates at a speed approximating the actual UNIX version of C++ <stdio.h>, each opened file receives a user-created buffer (an array) which can be sized manually by calling the setvbuf() method. Otherwise, the buffer has a default value of 8,192 (2^13) chars, which is a likely "block size" when reading from or writing to a file.

This user created buffer speeds up file operations by eliminating additional system calls that would be used to read or write, as system calls are significantly more expensive than reading from an array. The buffer is filled to its maximum size each time before a read() or write() system call is issued to maximize efficiency. The exception being if a read reaches end-of-file without the buffer being completely filled or a partially-filled write buffer is flushed due to the file being closed.

The implementation also is designed for an open file to have fully-buffered, unbuffered and line buffered input + output modes. While Dr. Parsons said we need only implement fully-buffered mode, I've included unbuffered in mine as well, though neither driver.cpp nor eval.cpp make use of this mode.

Program Structure

The only user-defined class is FILE, and it has only public data members which represent the current state of the file and that file's buffer, plus a default constructor to initialize all these members to 0 or false. The implementation of the FILE class is similar to a template class, with all the method declarations and definitions in stdio.cpp, and stdio.h has a #include "stdio.cpp".

The class has the following FILE class methods implemented by Professor Munehiro Fukuda, which operate with the same signature, outputs and return values as UNIX-original C++ <stdio.h>. As they were written by Dr. Fukuda, I will not discuss their implementation, only their function:

- **printf():** Takes a void* and uses that to write to the console, along with any additional parameters to represent different values for output.
- **itoa() + recursive_itoa():** Converts an integer type to string.

    - **setvbuf() + setbuf():** Allocates the buffer for the file being opened.
    - **fopen():** Opens a file and sets all relevant flags and modes.
    - **feof():** Returns true if FILE->eof is true, false otherwise.

The rest of the methods discussed were written by me, and therefore, I will elaborate on their exact implementation:

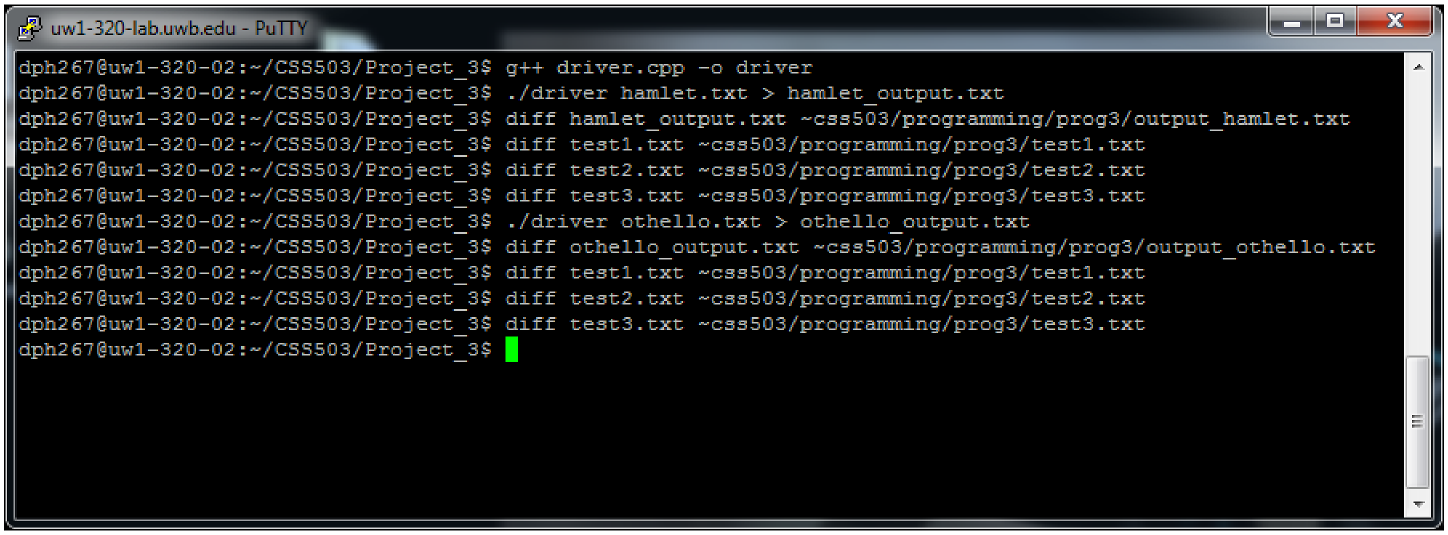
- **fpurge():** Uses memset() to quickly set all chars in the FILE->buffer to 0 and resets FILE->pos (position in the FILE buffer) to 0.
- **fflush():** If the last operation performed on the FILE was a read, it calls fpurge(). If the last operation was a write, it flushes the buffer to the file descriptor provided as parameter and outputs the size of the flushed buffer to console.
- **fread():** This method uses a while loop to operate until the number of blocks to read passed as parameter is met, or end-of-file is reached. The contents of the file are read into FILE->buffer and memcpy() is used to efficiently copy the file's buffer to the buffer passed as parameter.
- **fwrite():** Like fread(), this method uses a loop until the number of blocks is written. Contents of the file are written to FILE->buffer using memcpy(), and flushed when the buffer is full.
- **fgetc():** This method calls fread with the parameter of a nmemb == 1 (size of blocks to read is 1), then returns the char placed in the buffer (ptr parameter) by fread().
- **fputc():** This places a single char provided as parameter into FILE->buffer using an fwrite() call of buffer size 1. fwrite() will flush the buffer if it is full.
- **fgets():** Reads from the FILE->buffer using fgetc() in a loop. It would be more efficient to use an fread() call hear, but since fgets() has to terminate when encountering a '\n' (end of line) character, fgetc() is used so each character can be evaluated to not be '\n'.
- **fputs():** Like fgets(), fputs() uses fputc() to read from an array to FILE->buffer. Each character must be examined because fputs() terminates upon reaching a '\0' (NULL) char, so fwrite() cannot be used.
- **fseek():** Uses a UNIX lseek() system call to move the FILE pointer. As with <stdio.h>, FILE->eof is reset and FILE->pos + FILE->actual_size are both set to 0.
- **fclose():** Calls fflush(), uses a close() UNIX system call and deletes FILE->buffer if it was allocated by the program automatically. If a user passed a char* buffer to act as FILE->buffer, that buffer is not deleted.

Error checking was added to the program by including errno.h, setting error numbers and outputting those errors to strerror(), similar to how they are set by the STL <stdio.h>. The two error numbers used are:

1. **EBADF**, which is bad file descriptor. Typically NULL FILE object pointers or FILE pointers not representing an open file.
2. **EBADFD**, which is file descriptor in bad state. NULL buffer, bad parameters for the FILE.

III. Execution Output

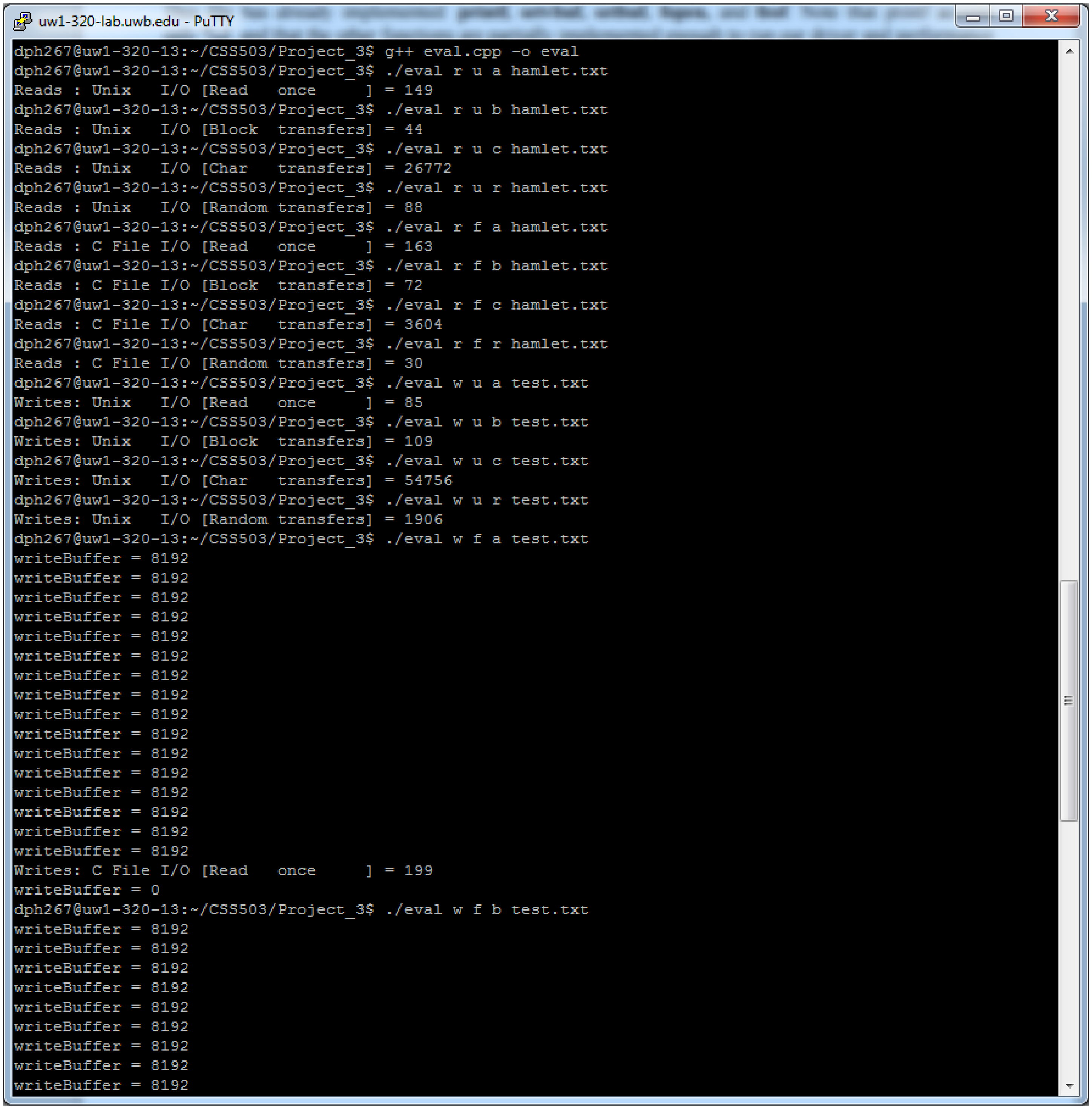
Output using **driver.cpp** to show that **all file input and output are correct:**



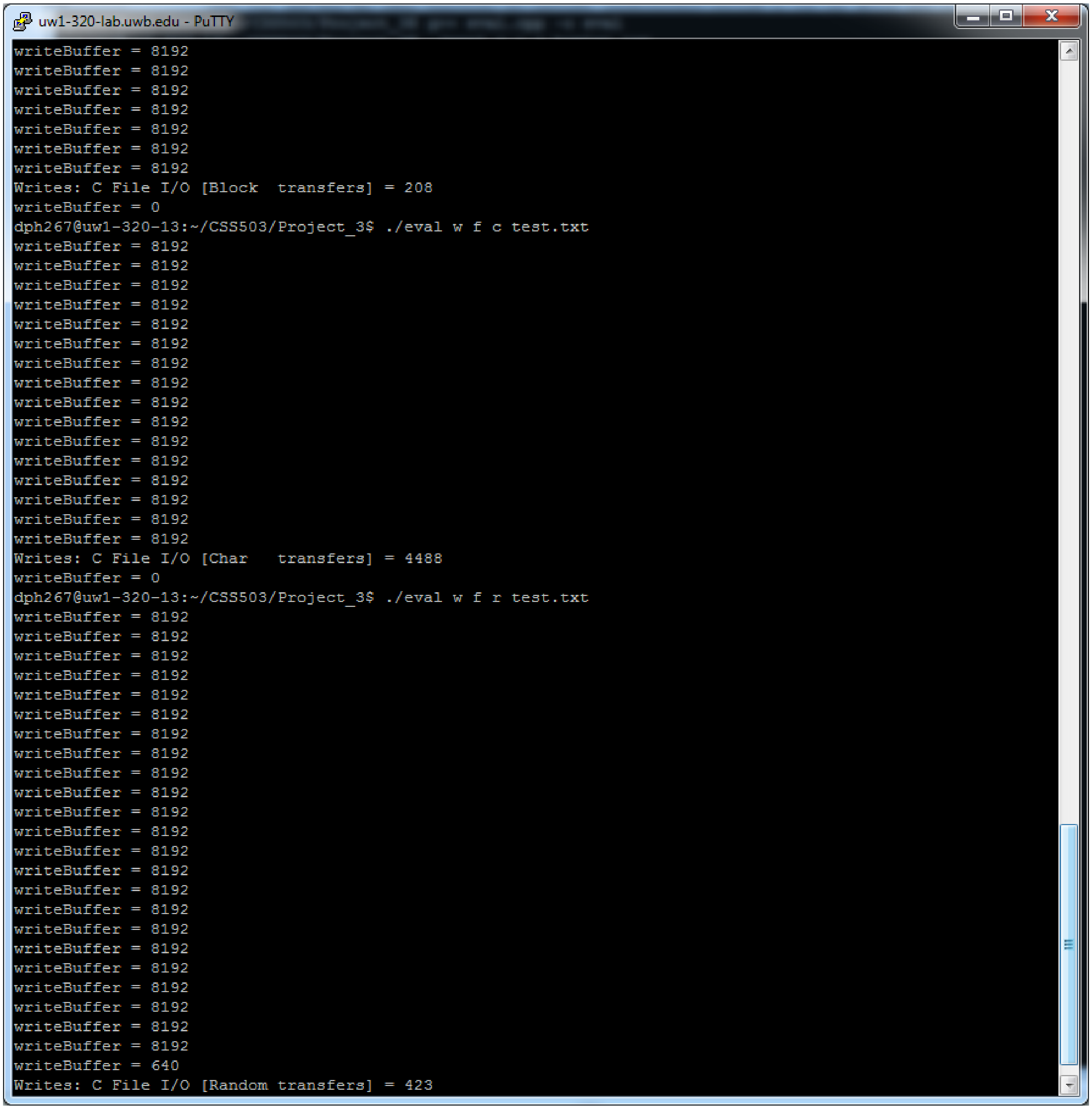
```
dph267@uw1-320-02:~/CSS503/Project_3$ g++ driver.cpp -o driver
dph267@uw1-320-02:~/CSS503/Project_3$ ./driver hamlet.txt > hamlet_output.txt
dph267@uw1-320-02:~/CSS503/Project_3$ diff hamlet_output.txt ~css503/programming/prog3/output_hamlet.txt
dph267@uw1-320-02:~/CSS503/Project_3$ diff test1.txt ~css503/programming/prog3/test1.txt
dph267@uw1-320-02:~/CSS503/Project_3$ diff test2.txt ~css503/programming/prog3/test2.txt
dph267@uw1-320-02:~/CSS503/Project_3$ diff test3.txt ~css503/programming/prog3/test3.txt
dph267@uw1-320-02:~/CSS503/Project_3$ ./driver othello.txt > othello_output.txt
dph267@uw1-320-02:~/CSS503/Project_3$ diff othello_output.txt ~css503/programming/prog3/output_othello.txt
dph267@uw1-320-02:~/CSS503/Project_3$ diff test1.txt ~css503/programming/prog3/test1.txt
dph267@uw1-320-02:~/CSS503/Project_3$ diff test2.txt ~css503/programming/prog3/test2.txt
dph267@uw1-320-02:~/CSS503/Project_3$ diff test3.txt ~css503/programming/prog3/test3.txt
dph267@uw1-320-02:~/CSS503/Project_3$
```

The output file in the CSS503 folder at UW used for the diff is output_hamlet.txt, not the out_hamlet_new.txt provided by Dr. Parsons. The only difference between the two is that the original output_hamlet.txt, when closing the 3 files open for writing, it only displays “d = %d”, while in out_hamlet_new.txt, it’s displayed as “fd = %d”. Since matching out_hamlet_new.txt required changing driver.cpp and I wasn’t sure which driver.cpp you the grader would be using (original or my modified version), I opted to leave driver.cpp as-is and compare to output_hamlet.txt.

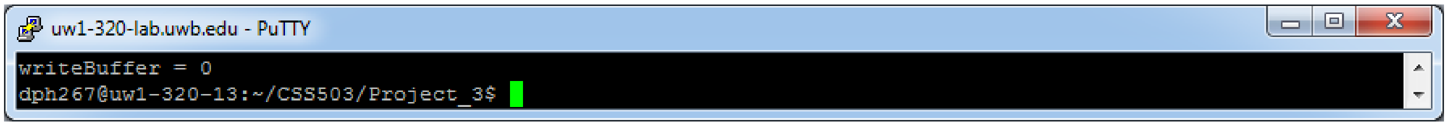
Here is the output from **eval.cpp** showing speeds of all read and write modes (whole file, blocks, by character, random block size) of UNIX I/O and my “stdio.h”. Notice speeds for both are relatively similar, except by character, where my “stdio.h” is far, far faster than standard UNIX I/O. **This output includes the calls to printf() during all write methods, which does slow down my “stdio.h”** methods compared to UNIX I/O. These calls to printf() were required by the program rubric, however.



```
dph267@uw1-320-13:~/CSS503/Project_3$ g++ eval.cpp -o eval
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r u a hamlet.txt
Reads : Unix   I/O [Read   once     ] = 149
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r u b hamlet.txt
Reads : Unix   I/O [Block  transfers] = 44
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r u c hamlet.txt
Reads : Unix   I/O [Char   transfers] = 26772
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r u r hamlet.txt
Reads : Unix   I/O [Random transfers] = 88
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r f a hamlet.txt
Reads : C File I/O [Read   once     ] = 163
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r f b hamlet.txt
Reads : C File I/O [Block  transfers] = 72
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r f c hamlet.txt
Reads : C File I/O [Char   transfers] = 3604
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r f r hamlet.txt
Reads : C File I/O [Random transfers] = 30
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u a test.txt
Writes: Unix   I/O [Read   once     ] = 85
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u b test.txt
Writes: Unix   I/O [Block  transfers] = 109
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u c test.txt
Writes: Unix   I/O [Char   transfers] = 54756
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u r test.txt
Writes: Unix   I/O [Random transfers] = 1906
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f a test.txt
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
Writes: C File I/O [Read   once     ] = 199
writeBuffer = 0
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f b test.txt
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
```

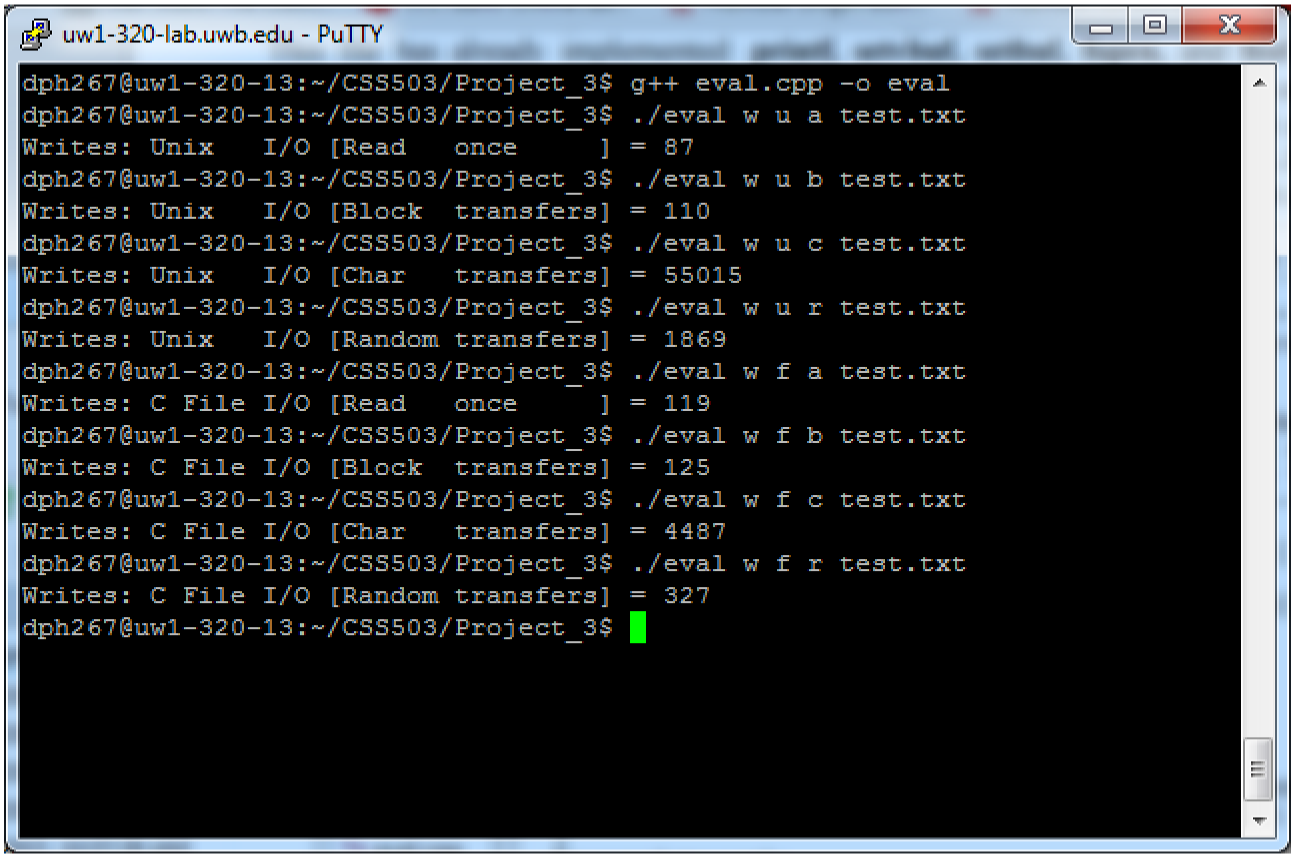


```
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
Writes: C File I/O [Block  transfers] = 208
writeBuffer = 0
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f c test.txt
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
Writes: C File I/O [Char   transfers] = 4488
writeBuffer = 0
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f r test.txt
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 8192
writeBuffer = 640
Writes: C File I/O [Random transfers] = 423
```



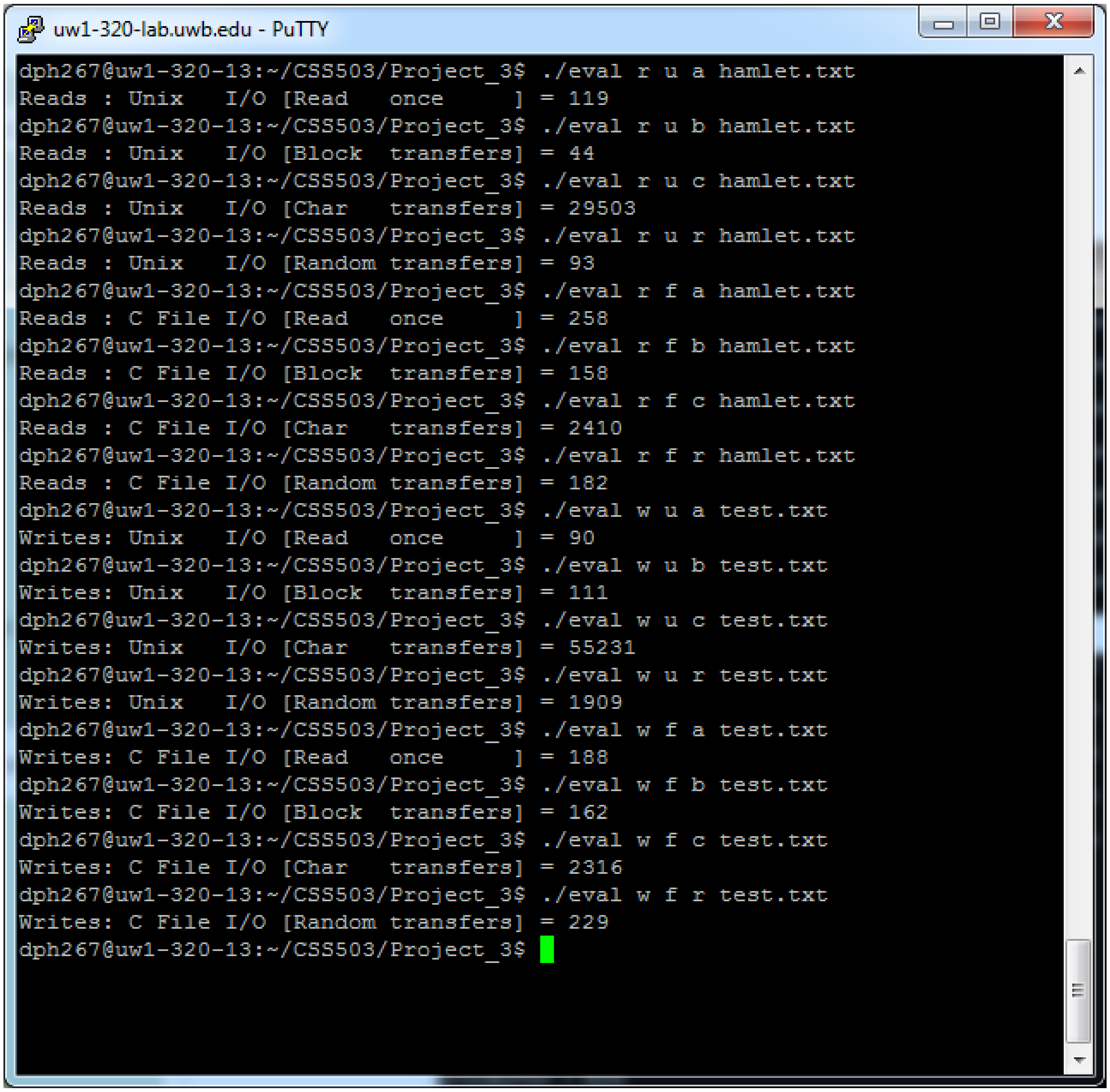
```
writeBuffer = 0
dph267@uw1-320-13:~/CSS503/Project_3$ 
```

To compare the write methods of my “stdio.h” to UNIX I/O more fairly, I eliminated the printf() output to console in stdio.cpp that gives the size of the writeBuffer. Here is **eval.cpp** with UNIX file output and my “stdio.h” without console output. “stdio.h” **does show faster write speeds after removing printf() calls compared to write speeds where “writeBuffer = 8192” is output to console.**



```
dph267@uw1-320-13:~/CSS503/Project_3$ g++ eval.cpp -o eval
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u a test.txt
Writes: Unix   I/O [Read   once     ] = 87
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u b test.txt
Writes: Unix   I/O [Block  transfers] = 110
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u c test.txt
Writes: Unix   I/O [Char   transfers] = 55015
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u r test.txt
Writes: Unix   I/O [Random transfers] = 1869
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f a test.txt
Writes: C File I/O [Read   once     ] = 119
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f b test.txt
Writes: C File I/O [Block  transfers] = 125
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f c test.txt
Writes: C File I/O [Char   transfers] = 4487
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f r test.txt
Writes: C File I/O [Random transfers] = 327
dph267@uw1-320-13:~/CSS503/Project_3$
```

Lastly, here is the output from **eval.cpp** using UNIX I/O and **actual UNIX-original <stdio.h>:**


uw1-320-lab.uwb.edu - PuTTY

```
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r u a hamlet.txt
Reads : Unix   I/O [Read   once     ] = 119
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r u b hamlet.txt
Reads : Unix   I/O [Block  transfers] = 44
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r u c hamlet.txt
Reads : Unix   I/O [Char   transfers] = 29503
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r u r hamlet.txt
Reads : Unix   I/O [Random transfers] = 93
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r f a hamlet.txt
Reads : C File I/O [Read   once     ] = 258
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r f b hamlet.txt
Reads : C File I/O [Block  transfers] = 158
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r f c hamlet.txt
Reads : C File I/O [Char   transfers] = 2410
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval r f r hamlet.txt
Reads : C File I/O [Random transfers] = 182
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u a test.txt
Writes: Unix   I/O [Read   once     ] = 90
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u b test.txt
Writes: Unix   I/O [Block  transfers] = 111
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u c test.txt
Writes: Unix   I/O [Char   transfers] = 55231
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w u r test.txt
Writes: Unix   I/O [Random transfers] = 1909
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f a test.txt
Writes: C File I/O [Read   once     ] = 188
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f b test.txt
Writes: C File I/O [Block  transfers] = 162
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f c test.txt
Writes: C File I/O [Char   transfers] = 2316
dph267@uw1-320-13:~/CSS503/Project_3$ ./eval w f r test.txt
Writes: C File I/O [Random transfers] = 229
dph267@uw1-320-13:~/CSS503/Project_3$
```

IV. Discussion

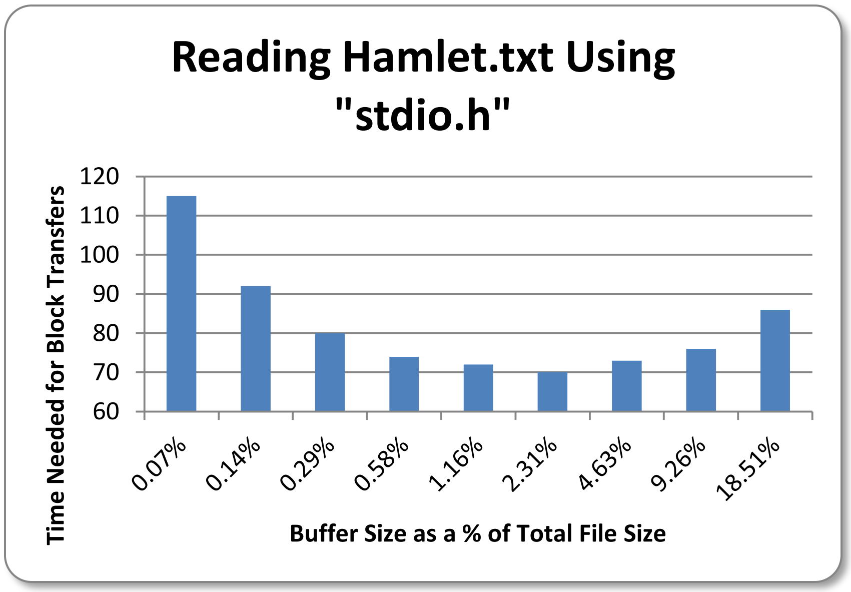
Limitations and Potential Improvements/Extensions

It’s not easy to come up with limitations to a class that simulates part of the C/C++ STL (<stdio.h>), as the STL has been written and improved upon repeatedly since its inception. That said, there are **a small few noticeable limitations** due more to our (mine and Dr. Fukuda’s) implementation of “stdio.h”.

- There is a default buffer size for a FILE object (8,192), or the user can call setvbuf() to manually choose a buffer size. However, relying on the user to choose the best size or using a fixed default value seems inefficient when dealing with files of varying sizes, not to mention that the optimal buffer size may be different for reading from a file than it is for writing to a file.
- The same is true of fread() and fwrite(), where the user utilizes a method call to read or write, and specifies the number of bytes and blocks to read. This is a flexible method because the additional parameters lets the user control how much data is read/written, but the buffer size chosen may have no correlation to the actual FILE buffer size, leading to inefficiencies in speed.
- Professor Fukuda does not delete dynamically allocated memory in driver.cpp or eval.cpp.
- The printf() function, while useful, doesn’t not allow char* or char arrays to be output to console using “%s”, which would be valuable in a number of areas of the program.
- One issue with <stdio.h> (and “stdio.h”) fgets() and fputs() is that they cannot actually be completed as block reads or writes, because each character must be examined to look for ‘\n’ (fgets) or ‘\0’ (fputs). So they are effectively a glorified fgetc() and fputc() that operate with a greater number of characters at once, but do so no faster than the by character functions.

Potential improvements/extensions:

- Instead of relying on a user-determined or a fixed default buffer size when instantiating a FILE object, we could read from the metadata contained in the header of the file we are opening, and determine an optimal buffer size. This can be accomplished with the “stat” or “fstat” structs in C/C++, as is implemented in eval.cpp, to discover the size of the file in bytes.



In the graph at left, you can see the average speed of fread() based on several buffer sizes as a percentage of the total file size (hamlet.txt is about 177,000 bytes). From my data, it looks like a buffer about 2.5% of the file size achieves the optimal speed, so fopen() could create buffers roughly 2.5% of file size to generate maximum block read speed.

- While I delete the dynamically allocated buffer in fclose(), the allocated memory in driver.cpp and eval.cpp could also be deleted to avoid memory leaks.
- printf() could be expanded to allow char* and char array parameters with a %s marker, which would allow me to print the actual error encountered using strerror() and errno.h, not just the error number with a %d.

Performance Considerations | UNIX I/O vs. "stdio.h"

The results of UNIX I/O and "stdio.h" using eval.cpp are compared below. As expected, as the number of UNIX system calls increases the time required for UNIX I/O to operate. The speed of both are relatively similar, with the exception of By Single Character, where my "stdio.h" is much, much faster, and to a lesser extent, random block size. Please note, "stdio.h" write speeds are those with the printf() call to display writeBuffer size commented out. This was done to make a more accurate comparison between write speeds.

	UNIX I/O	"stdio.h"	Diff	% Diff
Read from File				
Whole File at Once	149	163	14	8.59%
By Blocks	44	72	28	38.89%
By Single	26,772	3,604	-23,168	642.84%
By Random-sized	88	30	-58	193.33%
Write to File				
Whole File at Once	87	119	32	26.89%
By Blocks	110	125	15	12.00%
By Single	55,015	4,487	-50,528	1126.10%
By Random-sized	1,869	327	-1,542	471.56%

Whole file at once, UNIX I/O is faster than "stdio.h". "stdio.h" still needs to make multiple system calls to fill the buffer of 8,192 bytes, while a single read() or write() system call is made by UNIX I/O.

UNIX I/O is also faster in **block sizes**, as hamlet.txt is roughly 177,000 bytes, so only about 43 system calls are made (4,096 block size). However, since "stdio.h" operates on a block size of 8,192, it needs only 22 system calls to fill the file buffer and operate on it. However, "stdio.h" operating slower than UNIX I/O in block read and block write is pure overhead where "stdio.h" has to operate on the file's buffer, but UNIX I/O only writes straight to the file.

By Single Character is where "stdio.h" really shines compared to UNIX I/O, since "stdio.h" still only requires the same number of system calls as block read (roughly 22), while for UNIX I/O, a system call is made for each character (about 177,000), bogging down performance tremendously. If **Random-sized** blocks were truly random, we would not be able to compare them. However, since eval.cpp uses the rand() function with no "% time()", the result of the random-blocks is the same every time, and "stdio.h" is faster.

Performance Considerations | UNIX Original <stdio.h> vs "stdio.h"

The results of UNIX-original <stdio.h> and "stdio.h"are displayed below. As you can see, the opposite is true of <stdio.h> speeds compared to UNIX I/O: <stdio.h> is slower than "stdio.h" in whole file and block operations, but faster in Single Character and Random Sizes (faster Random in write, but not read).

	<stdio.h>	"stdio.h"	Difference	%Difference
Read from File				
Whole File at Once	258	163	-95	58.28%
By Blocks	158	72	-86	119.44%
By Single	2,410	3,604	1,194	33.13%
By Random-sized	182	30	-152	506.67%
Write to File				
Whole File at Once	188	119	-69	57.98%
By Blocks	162	125	-37	29.60%
By Single	2,316	4,487	2,171	48.38%
By Random-sized	229	327	98	29.97%

Honestly, I'm surprised that my "stdio.h" is faster in **whole file and block reads** than <stdio.h>, but it makes sense <stdio.h> is faster in **single characters**, because being part of the STL, it should be much more optimized than my simple "stdio.h" implementation of fgetc().

<stdio.h> also appears to be faster at writing than reading, at least relative to my "stdio.h". My "stdio.h" is 119% faster than <stdio.h> in block read, but only 29.6% faster in block write. Plus, random-sized write in <stdio.h> is faster than "stdio.h", while the opposite is true for random-sized read.