Documentation

1. Purpose

This programming assignment designs and implements my own version of standard I / O library. This libray includes some core input and output functions, including fread, fwrite, fflush, fpurge, fgetc, fgets, fputc and some other functions.

Unix based operating systems such as Linux and Mac OS / X provide system calls for file I / O: open(), read(), write(), and lseek(). However, there are some drawbacks with these system calls, Which are block - based and not able be used in other platforms including Windows. Fixing these drawbacks are basic motivation for designing and implementation of the C / C++ standard I / O library, as well as the main purpose for this assignment.

2. Implementation:

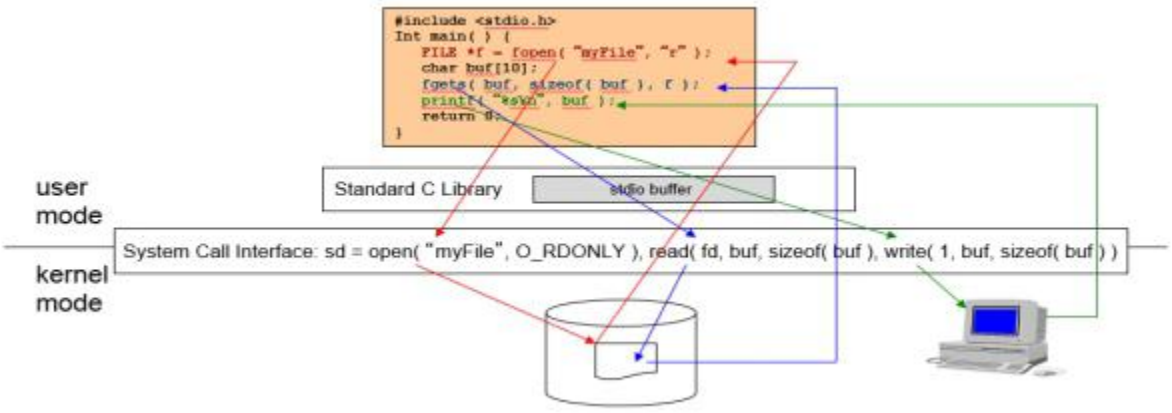


Figure1.

Using our version of library, the users are able to read and write files through a file stream (FILE *) instead of directly calling the underlying OS system calls. Like the process showing in figure, there is a buffer exist inside the file stream, and the fread and fwrite functions use this buffer to try to reduce the number of times transfering data from disk to main memory in order to improv efficiency. Generally, when reading/writing small byte-counts (e.g., reading one line at a time or even on character a time), our fread and fwrite functions are faster because of the overhead of calling into the kernel.

Fread()

This function has four parameters, *buffer, size, nmemb and *stream. Apparently, this function reads size*nmemb of data from file which specified by stream, and gives this amount of data to the user(*buffer). However, from implementation point of view, the fread() often reads more data from file to fullfill the whole buffer in order to improve efficiency.

This reading process is like moving in to a new house. Since all the furniture(file or hamlet in our program) is in downstairs, which is very far from the bedroom in upstairs, even the user just wants to move a pillow to bedroom, it is better to use a big box(buffer in file stream) to carry more stuff(which include that pillow the user wanted) to upstairs first at once. In this way, the user can get that pillow; and then, when the user want another pillow and blanket, he/she can get them easily from the box without going to downstairs again.

This is more efficient mainly because that the file is saved in disk, and data transfers from disk to main memory is much slower than data transfer with in main memory. We improve the efficiency by moving more data than the user need, therefore transfer from disk less times.

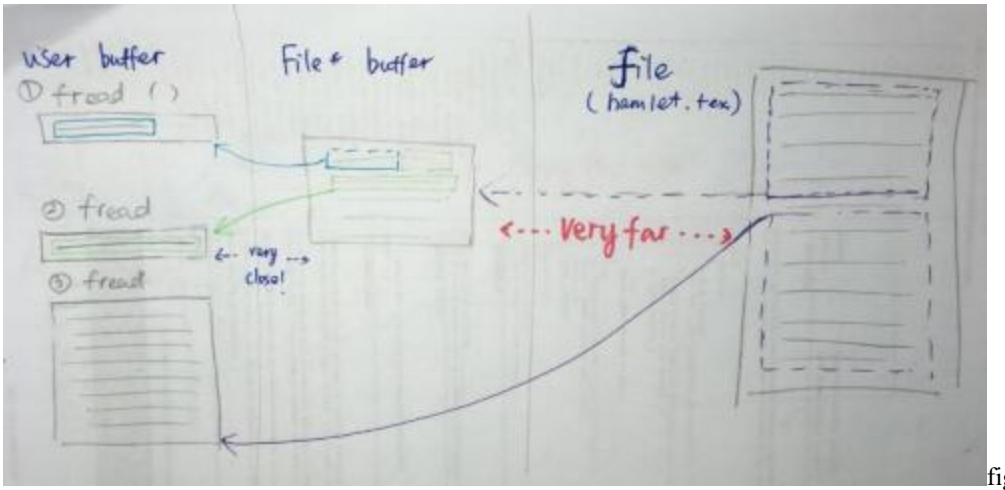


figure 2

Whenever a user opens a file and makes a request to read certain amount of data from this file, fread() check conditions to decide what to do. Like the figure 2, if the request size is smaller than the size of data left in buffer, the buffer gives the request data to user easily, like fread 1 and fread 2 in figure2. However, when the request size is bigger than the buffer, like read 3, it won't be necessary to read from buffer in the case. Therefore, the buffer will give the data left in buffer to user fist; then, let the use read remain size of data directly from file.

Fwrite()

Apparently, this function write size*nmemb of data from user buffer to the file which specified by stream. However, from implementation point of view, The fwrite() might not necessarily write the data directly to the file yes depends on different conditions.

The writing process is like moving out from a house. Even the user request to move a pillow (user buffer) from upstairs to the truck in downstairs (file or rest.txt in our program), it doesn't worth to run that far just for this pillow. The same idea, we use a big box (buffer) to take the pillow, and wait until the buffer is full, then move the whole box to the truck.

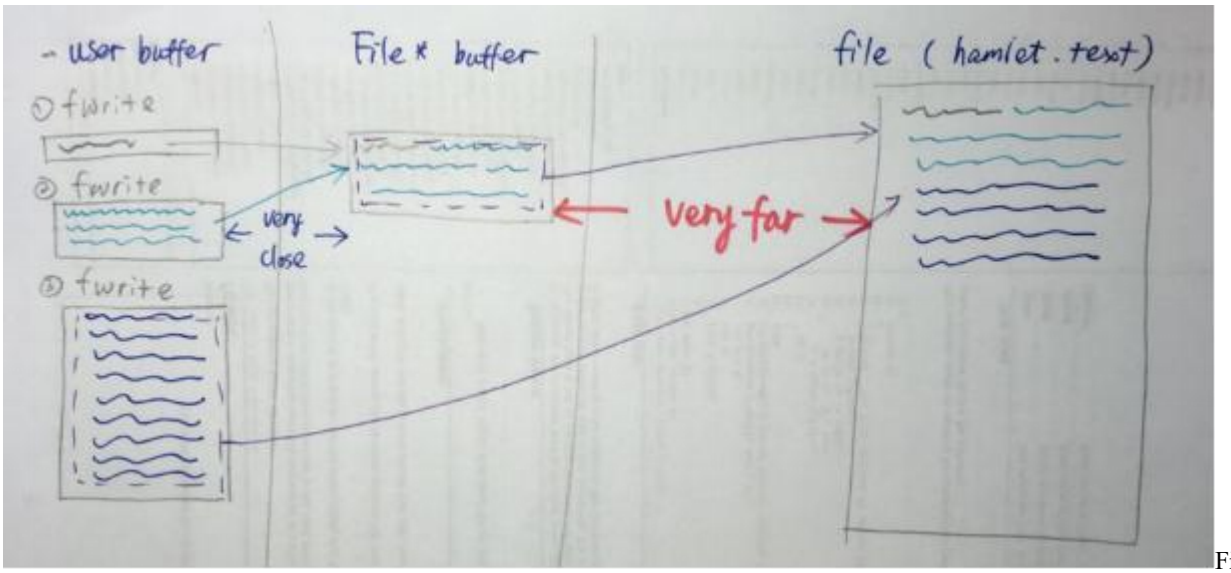
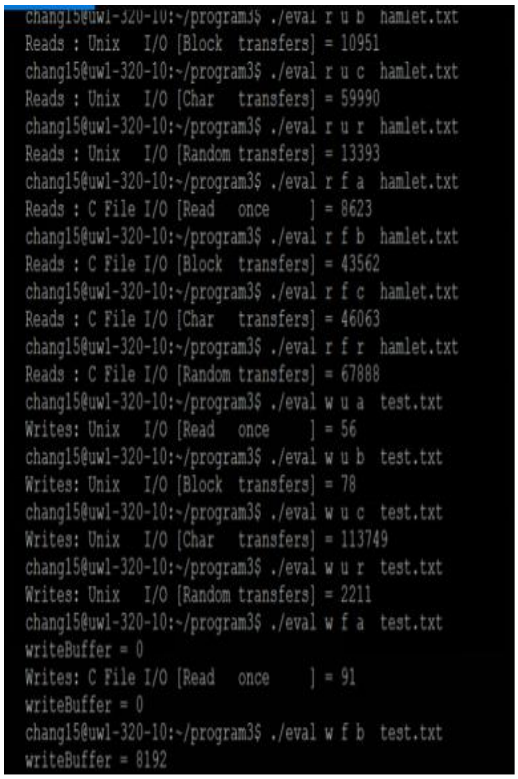


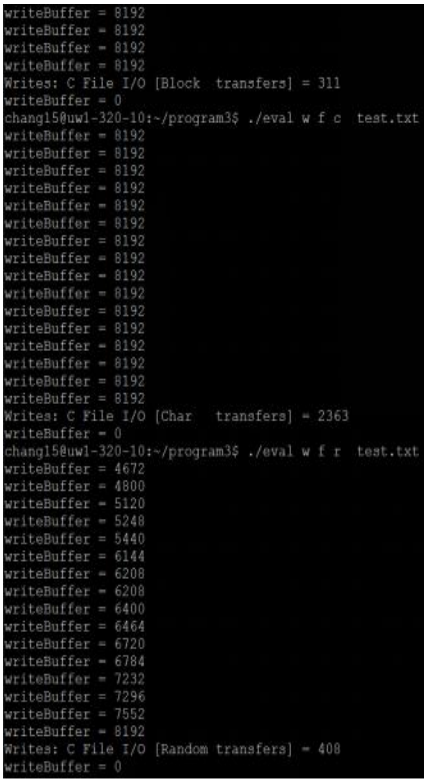
Figure 3

The fwrite function also compares the size the data user request to write with the remain size in buffer. If the request write size is too big, use the system all write to the file directly.

Discussion

- Performance consideration between your own stdio.h and Unix I/O (+5pts)



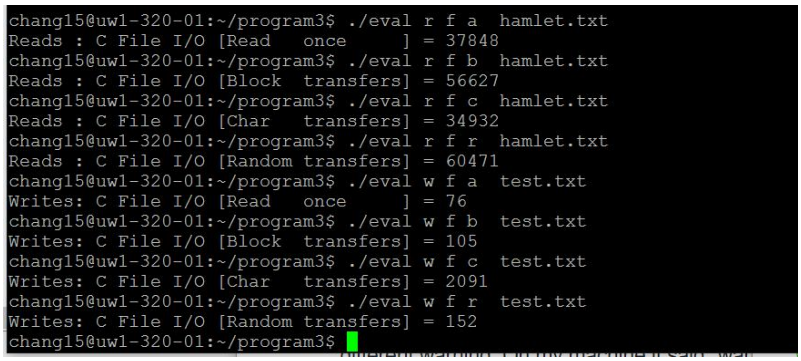


From the output, we can see that when we read and write very small amount of data, especially character by character, my stdio library is much faster than Unix I/O which is use system call and read and write one by one character from dick to main memory.

Other than that, the w f r and r f r is sometimes faster since the size that user request depends. There is not very big difference between read and write 4086 bytes each time and using my stdio sometimes is ever slower because my buffer is size of 8192. Since the buffer is not much too bigger than the request size and checking conditions and transfer from buffer to user also take a little excursion time, the performance of my stdio is sometimes worse than Unix I/O.

Reading and writing the whole file at once is also slower than the Unix I/O system call. I guess it's because of the same reason that my stdio might take more excursion time on checking conditions and other excursions.

- Performance consideration between your own stdio.h and the Unix-original stdio.h



The Unix-original stdio.h and my stdio has very similar performance. I guess that is because we using the same concept to build the stdio library architecture.
Generally the Unix-original stdio.h always does a little better job than my own stdio. I guess the Unix-original stdio.h might have a more advance implementation. If I have to guess, I think the the Unix-original stdio.h might handle the buffer size in a better way, therefor using the buffer more efficiently.

- Limitation and possible extension of your program

1. My program doesn't handle the race condition problem

   Due to this program is used for reading from and writing to file, is very likely that the race condition problem would occur. Normally, the user always need to call fopen() to open a file descriptor in order to read from or write to a file. However, if the read and write occur in the same process but different threads that share same data. The different thread might racing for the same resource file.

   My program doesn't use any lock to protect the using the resource in a critical section, and assume that only one program is running at a time. This is a big limitation.

2. I guess a better implementation might be using muti-process concurrency that can bring a big improvement on efficiency, especially for this program that requires a lot of I/O. I/O is much slower than CPU calculation. Muti-thread might improve the performance obviously. The c library might use this technic.
   However, muti-thread increases the implementation difficulty to a large extent. This program won't consider to use it.