University of Washington Bothell

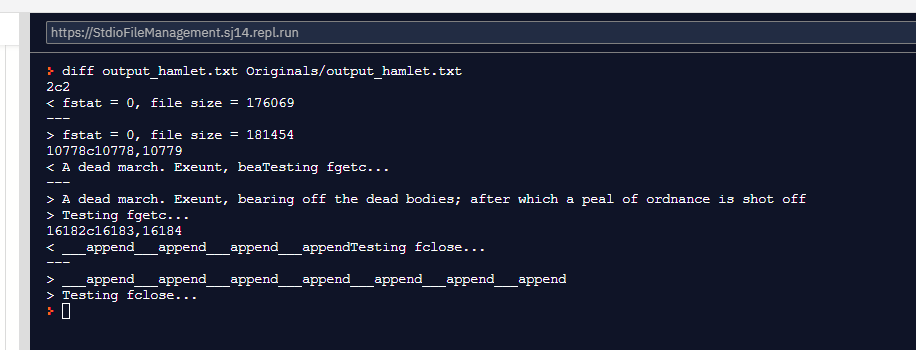
CSS503: System Programming

Program 3: C++ Standard I/O Library

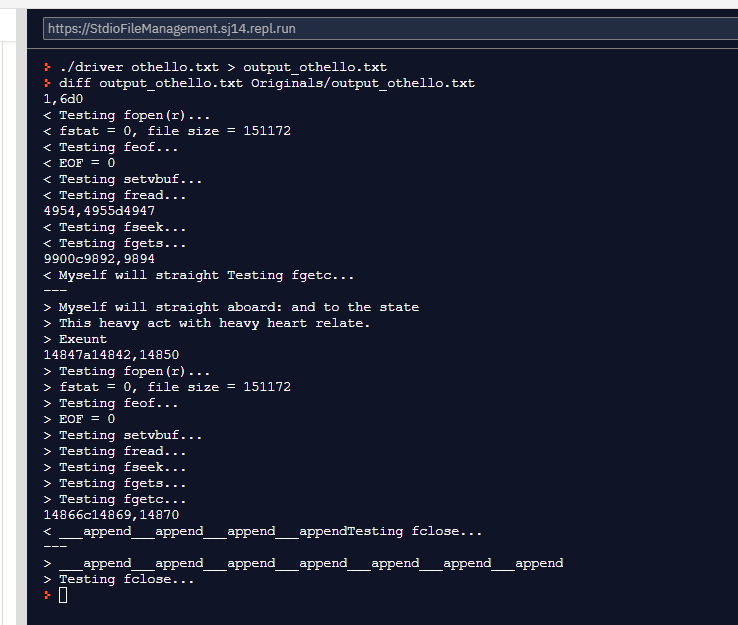
*Submitted by Snehal Jogdand*

**Execution output**

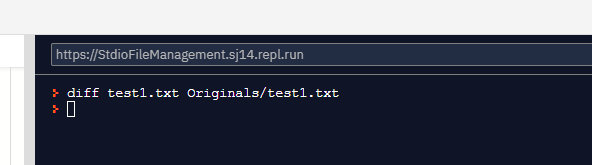
diff output\_hamlet.txt Originals/output\_hamlet.txt



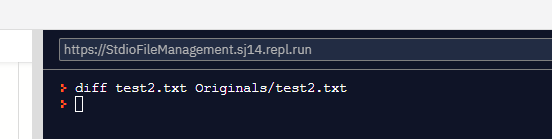
diff output\_othello.txt Originals/output\_othello.txt



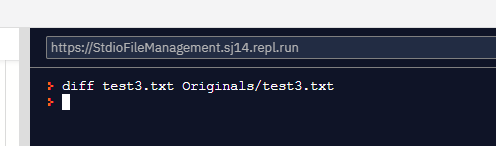
diff test1.txt Originals/test1.txt



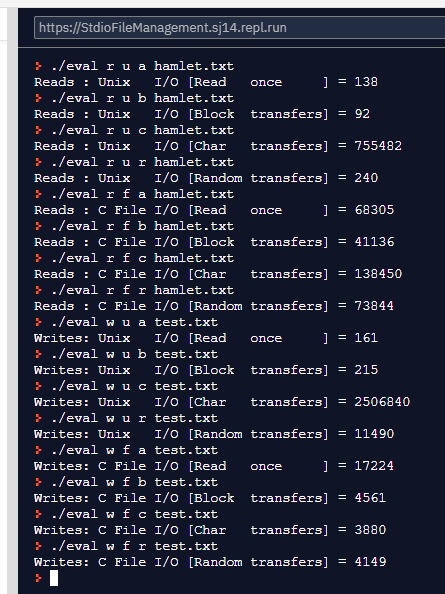
diff test2.txt Originals/test2.txt



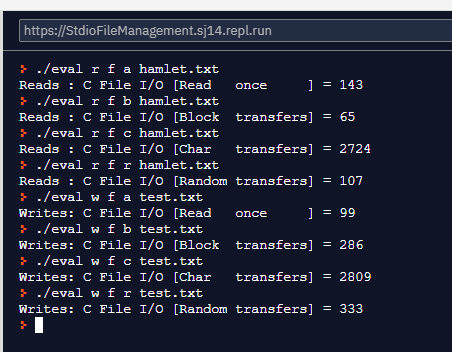
diff test3.txt Originals/test3.txt



Screenshots of all 16 test cases in Step 5



Screenshots of all 8 test cases in Step 6



**Implementation Details**

In this program, I have implemented a stdio functions that allows to read and write files instead of directly calling the underlying OS system calls. It provides buffering and a user-friendly file stream interface (FILE \*). The file stream interface is implemented utilizing the underlying system calls.

When reading and/or writing to a file, instead of directly calling OS system calls, we read and/or write from buffer. The program ensures the buffer is correctly re-filled only when needed minimizing the number of OS system call being made.

Figure 1 below shows how the standard I/O library utilizes a buffer to reduce the number of read and write system calls with file streams.

|  |
| --- |
|  |
| Figure 1. This figure shows Standard I/O functions using FILE instead of file descriptor. |

Following functions are implemented:

* int fgetc(FILE \*stream)

The function gets the next character from the specified stream and advances the position indicator for the stream. A buffer of size defaulted to 8192 bytes or as specified by user is used to read the file. The first call to fgetc reads entire buffer size of file contents. The next character now is returned from the buffer. The pos variable maintains the current position of pointer in buffer. On every read we advance the pos pointer. Once the pointer has reached end of buffer, we make another system call to read the next set of buffer size characters.

* size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)

The function reads an array of count elements, each one with a size of size bytes, from the stream and stores them in the block of memory specified by ptr. The position indicator of the stream is advanced by the total amount of bytes read. The total amount of bytes read if successful is (size\* nmemb). Internally, the fread calls fgetc to read the stream of contents from buffer.

* int fputc(int c, FILE \*stream)

The function writes a character to the stream and advances the position indicator. The character is written at the position indicated by the internal position indicator of the stream, which is then automatically advanced by one. A buffer of size defaulted to 8192 bytes or as specified by user is used to write the file. Every call to fputc writes to the buffer. Once the buffer is full, a single OS write call is made to write all the contents of the buffer to the file. The buffer is then made empty and reset to be available for next set of fputc.

* size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)

The function writes an array of count elements, each one with a size of size bytes, from the block of memory pointed by ptr to the current position in the stream. The position indicator of the stream is advanced by the total number of bytes written. Internally, the function interprets the block pointed by ptr as if it was an array of (size\* nmemb) elements of type unsigned char, and writes them sequentially to stream as if fputc was called for each byte.

* char \*fgets(char \*str, int size, FILE \*stream)

The function reads characters from stream and stores them as a string into str until (num-1) characters have been read or either a newline or the end-of-file is reached, whichever happens first. A newline character makes fgets stop reading, but it is considered a valid character by the function and included in the string copied to str. A terminating null character is automatically appended after the characters copied to str. Internally, the fgets calls fgetc to read the stream of contents from buffer

* int fputs(const char \*str, FILE \*stream)

The function begins copying from the address specified (str) until it reaches the terminating null character ('\0'). This terminating null character is not copied to the stream. Internally, the function interprets the block pointed by str as if it was an array of elements of type unsigned char, and writes them sequentially to stream as if fputc was called for each byte.

* int fflush(FILE \*stream)

If the given stream was open for writing (or if it was open for updating and the last i/o operation was an output operation) any unwritten data in its output buffer is written back to the file. We then purge the stream buffer to allow it to be used again after.

* Limitation and possible extension of your program

The program is limited by the buffer size. OS system calls are expensive, and to reduce the number of calls, a larger size buffer would greatly benefit the performance. The possible extension to the program is to compute the buffer size based on the file size or have it dynamic. As file size grows, a dynamic buffer that grows in size as well would help improve read and write to the files.

The program is prone to data loss as well. If the program crashes, all the writes that were made to the buffer would be lost. A possible extension to the program would be to have some auto save functionality to allow unwritten buffer contents to be saved back to the file periodically after a certain time period, instead of just saving on file close or when the buffer is full.

* Performance consideration between your own stdio.h and Unix I/O

Our own stdio.h utilizes a buffer which definitely provides a higher performance. The OS system calls are expensive and having a buffer to minimize the number of calls is a great improvement over Unix I/O. With a larger buffer size, the read/write speed is greatly improved.

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

The Unix-original stdio.h is extremely performant compared to my own stdio.h. I believe the reason is that we are using a fixed size buffer. Our program follows a very simplistic approach to read and write to that shared buffer of fixed size. Having a dynamic buffer size and optimizing read/write system calls only when needed would help improve performance.