

## CSE344 System Programming HW1 Report

### Program A

In programA, it reads the read1.txt or read2.txt with read only mode then convert the characters to integer values respect to ASCII values and formats the “97+i53” form.

After that it reads the read3.txt with read and write mode then look for the empty line ('\n' or real empty) when it finds, it writes the determined line. If there are no empty line in file, writes the end of file.

### Program A's functions

**write\_lock(...):** It runs like a normal write system call but this function also provides write lock and unlock by using flock structure. So I use this function everywhere instead of write.

**read\_lock(...):** It runs like a normal read system call but this function also provides read lock and unlock by using flock structure. So I use this function everywhere instead of read.

**write\_with\_no\_overwrite(...):** It writes a file by extending. Basically it avoids the write system call's overwriting.

**convert\_char\_to\_complex(...):** It converts the normal string to complex number string by using ASCII values and returns it.

**read\_and\_write(...):** All of the operations are done in this function. First it opens the read1.txt and continuously reads 32 bytes to end of the file in while loop. In the while loop It reads the other read3.txt file and If there are valid “empty” line then writes the string that read previously in read1.txt. To find the empty line, I looked at the event of encountering a new line in a loop.

**msleep(...):** Provides the millisecond sleep by using usleep function. (normal sleep did not run in parameter/1000 issue.)

### Program B

In programB, it finds a random number after that it begin the reading from that number. It takes the complex number it read from read3.txt by writing a newline on it. To delete the specified line, it does the following: Saves back the all characters from the line after the line to delete until the file is finished along the length of the line.

After the undo is complete, the file shortens the length of the line it will delete. After doing this, it converts the complex number it deletes by applying FFT algorithm. It writes the transformed complex number to file4.txt. programB repeats them until file3.txt is finished.

## Program B's functions

**write\_lock(...):** It runs like a normal write system call but this function also provides write lock and unlock by using flock structure. So I use this function everywhere instead of write.

**normal\_read\_lock(...):** It runs like a normal read system call but this function also provides read lock and unlock by using flock structure. So I use this function everywhere instead of read.

**read\_lock(...):** It runs like a normal read system call but this function also provides read lock and unlock by using flock structure and lseek. Basically it reads current place + offset's string.

**write\_with\_no\_overwrite(...):** It writes a file by extending. Basically it avoids the write system call's overwriting.

**FFT\_inner(...):** FFT calculator. It implements fast fourier transform algorithm.

**FFT(...):** FFT wrapper. It implements fast fourier transform algorithm for given double complex array.

**FFT\_out(...):** It convert the actual transformed double complex numbers to char array formed by "a+bi," using complex.h library.

**split\_and\_calculate\_FFT(..):** It gets the char array as input parameter. Then it splits the given char array by using strtok() function. After that it saves the splitted and converted(by atof()) to complex double array. At the end applies the FFT algorithm by using FFT and return the transformed char array by using FFT\_out.

**Is\_empty(...):** Controls the given file is empty.

**number\_of\_line\_file(...):** Returns the total number of line in the given file.

**read\_line(...):** Reads line after the given offset and if remove\_line parameter is 1 then deletes the determined line with the algorithm mentioned above.

**write\_output\_file(...):** After the read. It writes the given character array to file4.txt on first empty line. To find the empty line, I looked at the event of encountering a new line in a loop.

**msleep(...):** Provides the millisecond sleep by using usleep function. (normal sleep did not run in parameter/1000 issue.)

## Program C

In programC, it implies basic in place merge sort algorithm by using cumulative magnitude of given complex numbers in a line. To use this algorithm. I design the setter and getter functions for the file. So I assume this file like an array. After that only use instead of “arr[i]”, get\_the\_nth\_line(...); function and instead of “=arr[j]” set\_the\_nth\_line(...); function in merge sort algorithm.

## Program C's functions

**write\_lock(...):** It runs like a normal write system call but this function also provides write lock and unlock by using flock structure. So I use this function everywhere instead of write.

**read\_lock(...):** It runs like a normal read system call but this function also provides read lock and unlock by using flock structure. So I use this function everywhere instead of read.

**write\_with\_no\_overwrite(...):** It writes a file by extending. Basically it avoids the write system call's overwriting.

**number\_of\_line\_file(...):** Returns the total number of line in the given file.

**cumulative\_magnitude\_of\_complex\_numbers(...):** Calculates the total magnitude of given complex numbers in a line. In the function it splits the given parameter char array and sums the magnitudes. (*magnitude of  $a+bi = \sqrt{a^2+b^2}$* )

**set\_the\_nth\_line(...):** Writes the string given as a parameter to n.element in the file. In writing I use write() and write\_with\_no\_overwrite() function for providing essential character changes.

**get\_the\_nth\_line(...):** Gets the nth line complex character array.

**merge(...):** Ordinary in-place merge function.

**mergeSort(...):** Ordinary recursive merge sort.

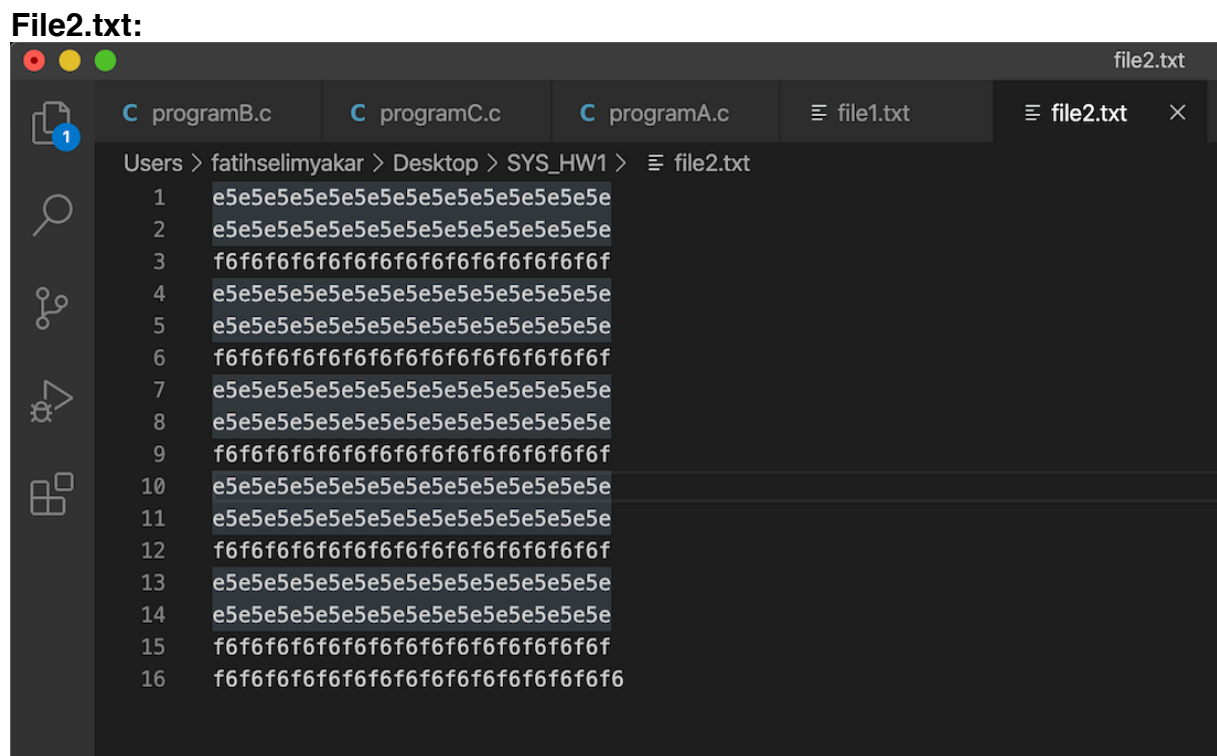
## Sample Execution Screenshots

This screenshots run with “./programB -i file3.txt -o file4.txt -t 1 & ./programB -i file3.txt -o file4.txt -t 1 & ./programA -i file1.txt -o file3.txt -t 10 & ./programA -i file2.txt -o file3.txt -t 5” command.

+

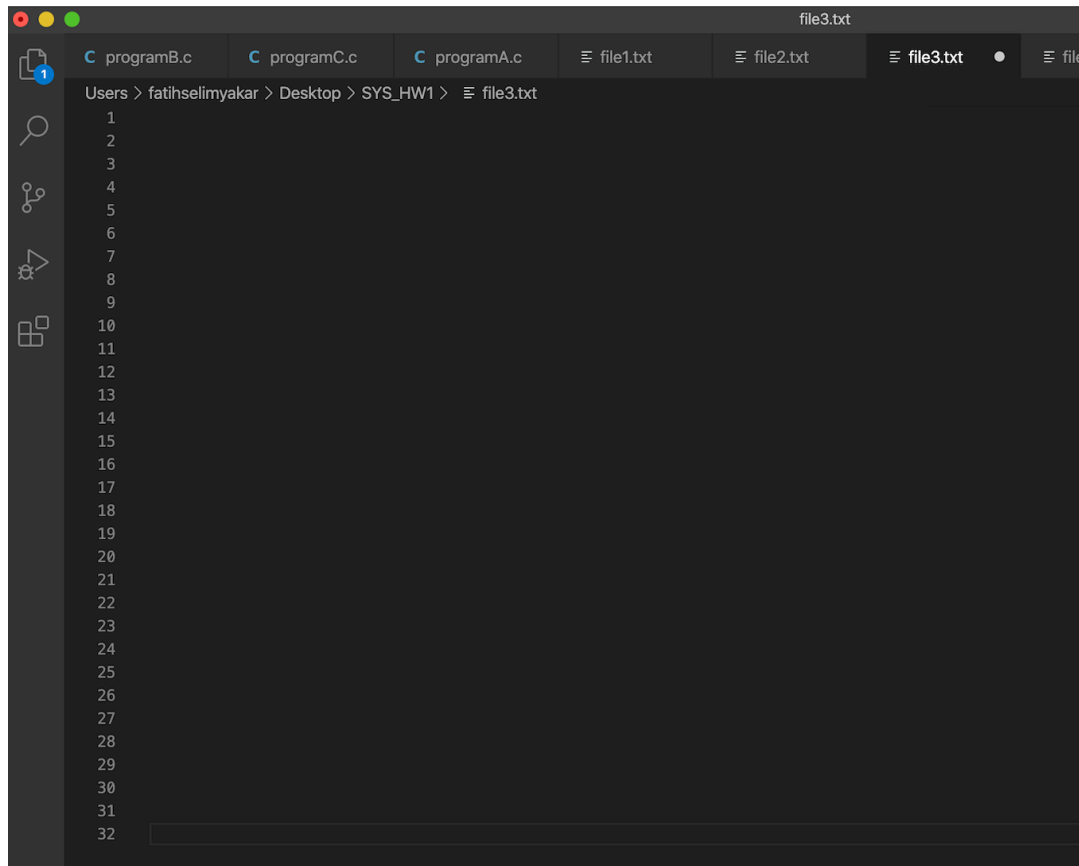
./programC -i file4.txt

**File1.txt:**



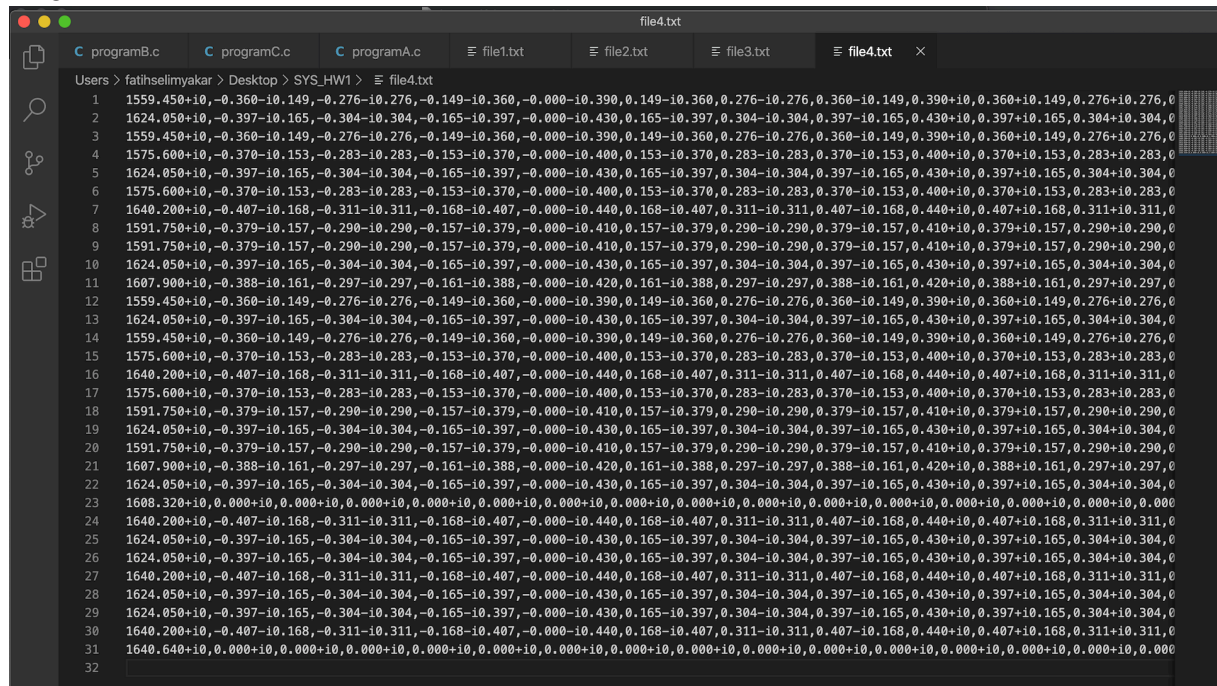
After the parallel execution:

File3.txt:



The screenshot shows a code editor window titled 'file3.txt'. The editor has a dark theme and a sidebar on the left with icons for file explorer, search, and other functions. The main area displays 32 empty lines, numbered 1 to 32. The top of the editor shows a tab bar with several open files: 'programB.c', 'programC.c', 'programA.c', 'file1.txt', 'file2.txt', 'file3.txt' (selected), and 'file4.txt'.

File4.txt:



The screenshot shows a code editor window titled 'file4.txt'. The editor has a dark theme and a sidebar on the left with icons for file explorer, search, and other functions. The main area displays 32 lines of numerical data, numbered 1 to 32. The data consists of long strings of numbers separated by commas, representing a large dataset. The top of the editor shows a tab bar with several open files: 'programB.c', 'programC.c', 'programA.c', 'file1.txt', 'file2.txt', 'file3.txt', and 'file4.txt' (selected).

### File4.txt

[illegible]