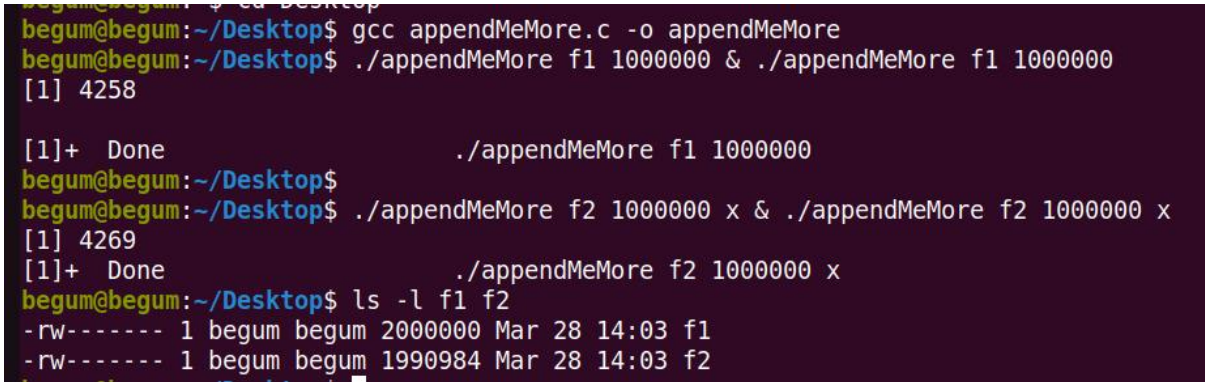
CSE344 HOMEWORK 1 REPORT

Ayşe Begüm NUR

1901042613

Part – 1:

For the first part of the homework, I have provided a source file named appendMeMore.c that takes 3 or 4 command line arguments. We were supposed to write to a file one byte at a time and see the difference in the file sizes when we use the O_APPEND flag versus setting the file offset to the end by using lseek(). As intended, my tests gave the result exactly as follows:



```
begum@begum:~/Desktop$ gcc appendMeMore.c -o appendMeMore
begum@begum:~/Desktop$ ./appendMeMore f1 1000000 & ./appendMeMore f1 1000000
[1] 4258

[1]+  Done                    ./appendMeMore f1 1000000
begum@begum:~/Desktop$
begum@begum:~/Desktop$ ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
[1] 4269
[1]+  Done                    ./appendMeMore f2 1000000 x
begum@begum:~/Desktop$ ls -l f1 f2
-rw------- 1 begum begum 2000000 Mar 28 14:03 f1
-rw------- 1 begum begum 1990984 Mar 28 14:03 f2
```

The reason for the difference between file sizes is a feature called atomicity. In programming, "atomic" refers to an operation or a piece of code that is indivisible and cannot be interrupted or altered by other processes or threads running concurrently.

In other words, an atomic operation is guaranteed to be completed without interference from other processes or threads, ensuring that the operation is completed in a consistent state. This is important in concurrent programming where multiple processes or threads may attempt to access the same resource simultaneously, potentially causing data race conditions or other synchronization issues.

Since O_APPEND flag ensures us that the process will be atomic, we get a file with the size 2 million when we execute the command:

$ appendMeMore f1 1000000 & appendMeMore f1 1000000

On the other hand, lseek() makes no such promises when it comes to atomicity. That is why we get a file size less than 2 million when we run the command:

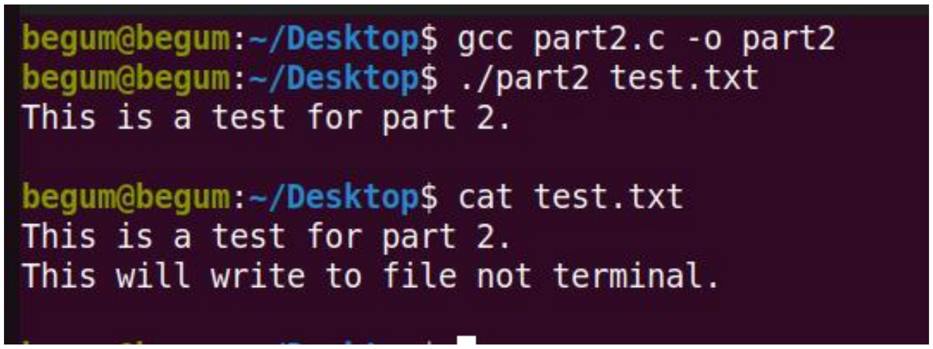
$ appendMeMore f2 1000000 x & appendMeMore f2 1000000 x

Part – 2: For this part, we were asked to write clones of dup () and dup2 () functions using fcntl(). For dup () function, it was as easy as checking if the old file descriptor is open or not and using fcntl(old_fd, F_GETFD, 0) to handle the rest.

Writing dup2() was trickier because there were more things that could go wrong. First, if the old file descriptor is the same as the new one, making sure that the old descriptor is open and returning the function was enough.

Dup() function returns the least file descriptor available. In the end, you have two file descriptors that refer to the same file. I have tested my dup() function in two steps. First, I have opened the said file with the “original” file descriptor and write some text to the file. Then, I duplicated the file descriptor using dup() and printed the file contents using the duplicated file descriptor. If the contents written to the file matches the data written to terminal, dup() function works.

To test my dup2() function, I provided the new file descriptor as STDOUT and used printf() after. As intended, since the file descriptor is duplicated as the STDOUT, any function that intends to write to STDOUT now writes to the file.

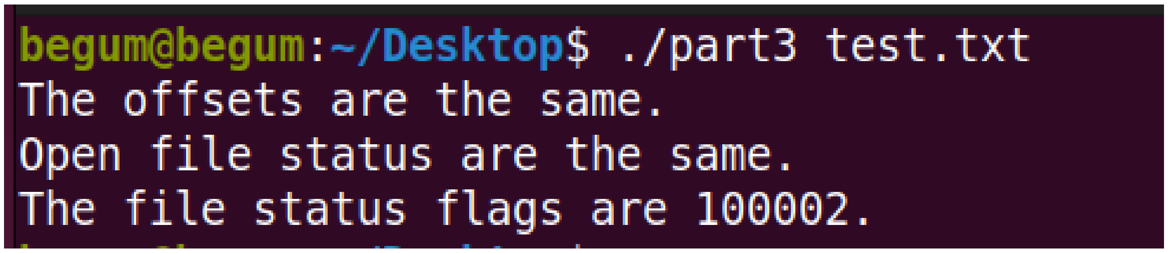

```
begum@begum:~/Desktop$ gcc part2.c -o part2
begum@begum:~/Desktop$ ./part2 test.txt
This is a test for part 2.

begum@begum:~/Desktop$ cat test.txt
This is a test for part 2.
This will write to file not terminal.
```

Part – 3: This part was about proving that duplicated file descriptors share a file offset value and open file status flags. To prove as such, after opening the file with ‘the original file descriptor’, I have duplicated the file descriptor using dup (). After that I have used lseek(), giving it the original file descriptor as the first argument and set the file offset to same place. Then, I got the file offset values for both file descriptors by using lseek() with SET_CURRENT. When we compare these two offsets, they were equal.

Similarly, for the open file status flags, I got help from the fcntl() function and used it with both file descriptors with F_GETFL, as expected these values were the same as well.

My results for this part are as follows:



```
begum@begum:~/Desktop$ ./part3 test.txt
The offsets are the same.
Open file status are the same.
The file status flags are 100002.
```