Esra Eryılmaz
171044046

CSE 344 – System Programming

Homework 1 Report

30.03.2023

General Information

- First of all, I handle 3 problems in the homework independently of each other and created a folder for each of them and solved it that way.
- Each part has its own makefile, for compilation it is enough to go to the each path and running make command.
- In general, I made some error checks in all parts. I checked for errors in system calls such as open and write as much as possible.

Part 1

- **How I solved the problem?**
  - Firstly, I check the command line arguments so that the user enter it correctly. 3 or 4 arguments must be entered.
  - After that with 2 functions, I solved the problem :
    1) The first function named ***"int writeSyscall(char *filename, int byteNumber)"*** works without x argument.
       It opens a file (creating if it does not exist) with O_APPEND flag in the open() system call.
       Then writes with system call write(), the letter "e" (my choice, just for the test purposes) up to the given byte number to the file.
       Finally closes the file.
    2) The second function named ***"int lseekSyscall(char *filename, int byteNumber)"*** works with the x argument.
       It opens the file but does not use the O_APPEND flag.
       Then it writes the file but each write system call it calls other system call *"lseek(fd, 0, SEEK_END)"* (This lseek() indicates 0 bytes after the end of the file.)
  - In summary, these two functions actually trying to do similar things. The first one always writes to the end of the file using append flag. The second one is using lseek for finding the end of the file and adjust the offset accordingly and try to write to the end of the file again.

- **Explanation of the test results.**
  - In the screenshot below, I have shown both compile and run.

```
esoo@esra:~/Desktop/eryilmaz_esra_171044046/part1$ make
gcc -c appendMeMore.c
gcc -o appendMeMore appendMeMore.o -lm
esoo@esra:~/Desktop/eryilmaz_esra_171044046/part1$ ./appendMeMore
Usage: 2 or 3 command line arguments are required!
EXAMPLES :
./appendMeMore f1 1000000 & ./appendMeMore f1 1000000
./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
esoo@esra:~/Desktop/eryilmaz_esra_171044046/part1$ ./appendMeMore f1 1000000 & ./appendMeMore f1 1000000
[1] 11853
[1]+  Done                    ./appendMeMore f1 1000000
esoo@esra:~/Desktop/eryilmaz_esra_171044046/part1$ ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
[1] 11892
[1]+  Done                    ./appendMeMore f2 1000000 x
esoo@esra:~/Desktop/eryilmaz_esra_171044046/part1$ ls -l f1 f2
-rwxr--r-- 1 esoo esoo 2000000 Mar 30 00:08 f1
-rwxr--r-- 1 esoo esoo 1046170 Mar 30 00:08 f2
```

  - I tried the tests in the homework pdf, you can see the outputs in the above and below images.
  - Above image shows the dual boot ubuntu output and the below shows virtual machine ubuntu output.

```
esra@esoo:~/Desktop/eryilmaz_esra_171044046/part1$ ./appendMeMore f1 1000000 & ./appendMeMore f1 1000000
[1] 18124
[1]+  Done                    ./appendMeMore f1 1000000
esra@esoo:~/Desktop/eryilmaz_esra_171044046/part1$ ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
[1] 18136
[1]+  Done                    ./appendMeMore f2 1000000 x
esra@esoo:~/Desktop/eryilmaz_esra_171044046/part1$ ls -l f1 f2
-rwx------ 1 esra esra 2000000 Mar 29 01:25 f1
-rwx------ 1 esra esra 1999822 Mar 29 01:25 f2
```

  - In the above tests, I tested them to write to a file by running two programs at the same time. With x arguments; write() syscall works after lseek() and in the other just write() syscall works with append flag.

- **So why did the file size of the written file different while the two tests were actually trying to do the same thing?**
  - Because when two processes write to the same file using lseek() syscall, the action depends on the system. In general, if two processes write to the same file at the same offset using lseek(), the data from the process that writes last will overwrite the data written by the first process. The reason is the lseek() syscall sets the file offset for the write operation, and the second process will overwrite the data written by the first process at the same offset.
  - But, if the two processes write to different offsets in the same file using lseek() syscall, their writes will not affect each other. However in our test, I used the same lseek() so the offsets are the same and their writes will affect each other. Also the way in which they will be overwritten with each other

is different in each operation, those with lseek() syscall may always give different byte results.

- The lseek function is used to determine the location of a file to read or write to and that is why it is not a synchronization function.
- The synchronization problems can occur if multiple processes or threads are writing or reading to the same file. This happened in our test. We should use additional syscalls for synchronization.

But why was the other one able to write 2000000 bytes?

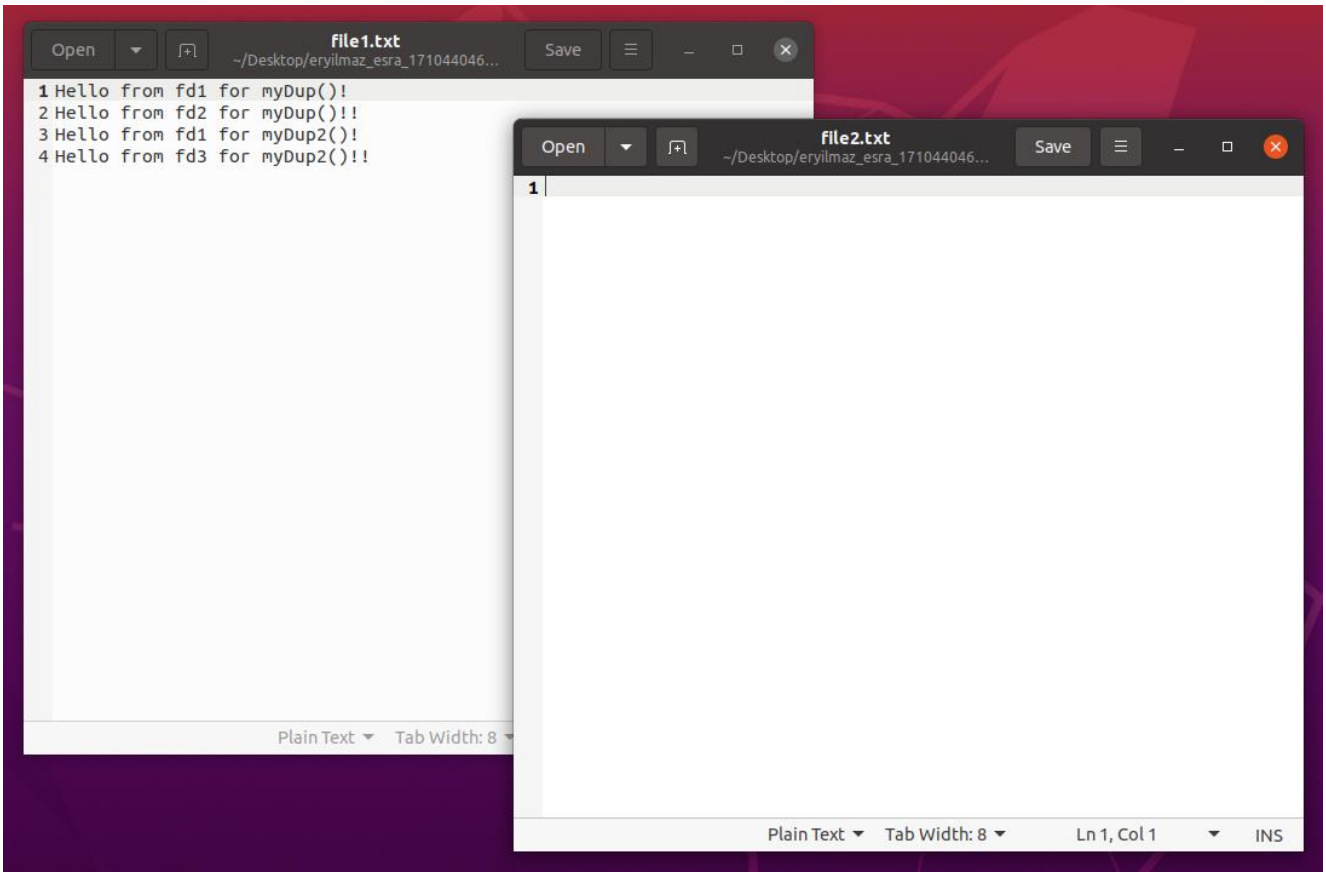
- Because both processes of the program are using the O_APPEND flag to append data to the file, the writes should be performed atomically and coordinated. As a result, it is unlikely that we will see any unexpected behavior or data corruption.
- However, in 2000000 bytes the order in which the data is written to the file may not be deterministic or predictable, since the two processes of the program are running concurrently and may be scheduled by the operating system in different orders.

Part 2

- **How I solved the problem?**
  - With 2 functions, I solved the problem :
    1) The first function named ***"int myDup(int oldfd)"*** works like dup() on unix but I implemented it with fcntl() syscall.
       It calls fcntl() with the F_DUPFD command (F_DUPFD: Return a new file descriptor which shall be the lowest numbered available).
       If succeed, function returns new duplicated file descriptor.
    2) The second function named ***"int myDup2(int oldfd, int newfd)"*** works like dup2() on unix but I implemented it with fcntl() syscall.
       Firstly it controls the special case where the oldfd equals newfd. If they are equal than I check whether oldfd is valid or not with fcntl(oldfd, F_GETFL). If oldfd is not valid (if above fcntl syscall returns -1) then I return -1 and set errno to EBADF.
       But if oldfd == newfd and oldfd is valid then no need to duplicate, I just return oldfd.
       After that I close newfd, because one of dup2()'s tasks is to turn it off.
       But if the function did not return from any of what I wrote above, we move on to the actual dup2() part.
       I create newfd which is a duplicate of oldfd with "newlyCreatedfd = fcntl(oldfd, F_DUPFD, newfd);" syscall and return newly created fd.

- **Explanation of the test results.**
  - In the screenshots below, I have shown both compile and run.

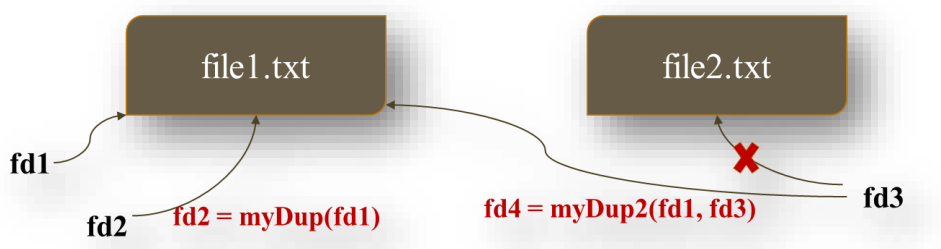
```
esoo@esra:~/Desktop/eryilmaz_esra_171044046/part2$ make
gcc -c part2.c
gcc -o part2 part2.o -lm
esoo@esra:~/Desktop/eryilmaz_esra_171044046/part2$ ./part2
Test myDup()  =>  int fd2 = myDup(fd1);
fd1 = 3,
fd2 = 65

Test myDup2()  =>  int fd4 = myDup2(fd1, fd3);
fd1 = 3,
fd3 = 4,
fd4 = 4

And also you can look at the created files in the part2 directory for the writing test results.
```



For the testing ;

- First of all I create a file descriptor **fd1** and it shows the **file1.txt**.
- After that **myDup()** returns newly created (duplicated) fd. In our case it is **fd2**.
- Then I create a new file descriptor **fd3** for pointing to another file which name is **file2.txt**.

- Then **myDup2()** returns the duplicated fd; if the duplication was successful, in our case it is **fd4**. (Of course, after myDup2(), fd3 closed before being reused; the close is performed silently.)
- So I can summarise my tests with the diagram I have drawn below.



- So in the end fd1, fd2 and fd3 are file descriptors pointing to the same file.
- And that's why all the writing were written in the file1.txt file.

Part 3

- **How I solved the problem?**
  - I made this third part using the unix system call dup(). I did not use myDup() because I'm testing the myDup() in the second part of the homework.
  - With one function, I solved the problem :
    1) The function named ***"int offsetCheck()"*** verifies that duplicated file descriptors share a file offset value and they open a same file.
       - **For verify same offset value :**

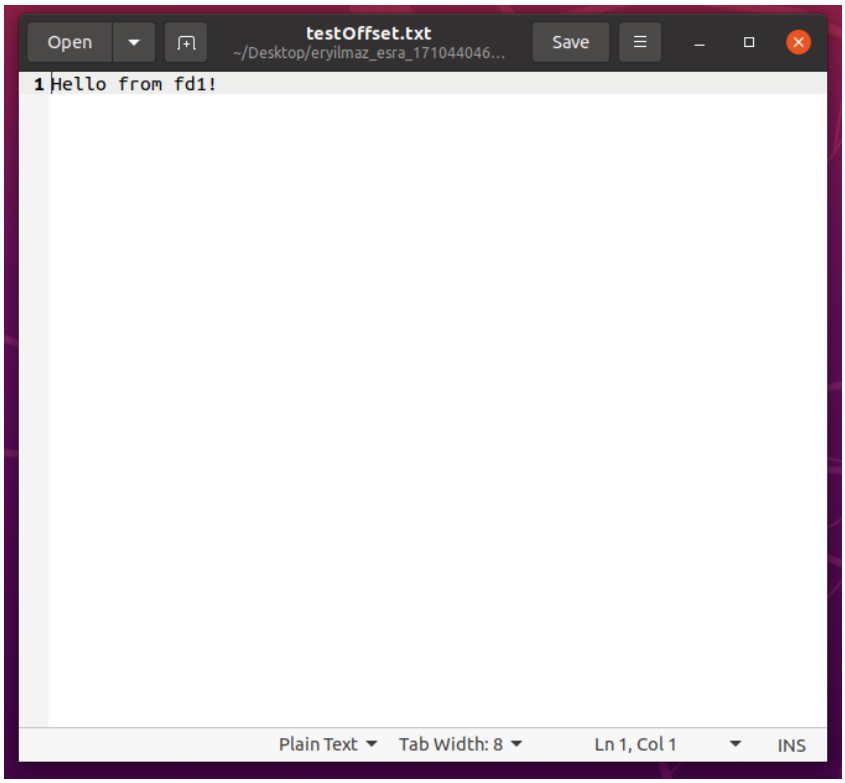
       Firstly it opens testOffset.txt file for verification purposes. And file descriptor fd1 points to this file.
       After that I write something to the file with fd1. Then I get the current offset of the file with **lseek(fd1, 0, SEEK_CUR)** syscall. So this is my first offset value.
       After that I duplicated this fd1 with **dup()** syscall on unix. I named new duplicated fd as a fd2.
       Then I get the current offset of the duplicated fd2 with **lseek(fd2, 0, SEEK_CUR)**.
       And I printed fd1 and fd2 offset values for testing purposes, you can see them in the test results below.
       Finally I compared these two offset values to verify that duplicated file descriptors share a file offset value or not. And it turns out that they shared same offset value.

- **For verify open a same file :**

  I used **fstat()** syscall for that purpose. Also I used fd1 and fd2 file descriptors that I used above.
  After that with **fstat(fd1, &stat1)** and **fstat(fd2, &stat2)** syscalls I obtained struct stat objects of the file descriptors. These stat objects hold a lot of information about the files that file descriptors points.
  Finally the fact that the stat objects have the same **"st_ino"** and **"st_dev"** (taken together, uniquely identify the file) values is proof that those descriptors open the same file.
  I compared these stat object values to verify that duplicated file descriptors open a same file or not.
  And it turns out that they opened a same file.

- **Explanation of the test results.**
  - In the screenshots below, I have shown both compile and run .

```
esoo@esra:~/Desktop/eryilmaz_esra_171044046/part3$ make
gcc -c part3.c
gcc -o part3 part3.o -lm
esoo@esra:~/Desktop/eryilmaz_esra_171044046/part3$ ./part3
offset1 : 16, offset2 : 16
DUPLICATED FILE DESCRIPTORS SHARE A FILE OFFSET VALUE !!!!
DUPLICATED FILE DESCRIPTORS OPEN A SAME FILE!!!!
```



- You can see the output of what I explained in the above part here.
- You can see in the output that the file descriptors have the same offset values.
- You can also see the output of the if conditions in the implementation; if they share same offset and if they open the same file.
- Finally, you can see that I created a file named testOffset.txt and wrote something into it to get some offset value for verification and testing.