

GTU Department of Computer Engineering
CSE 344 - Spring 2023
Homework 1 Report

Mehmet Mete Şamlıoğlu
200104004093

1-) PART 1 (Question1)

In part 1, it was wanted to run the code simultaneously. However, the difference was the run commands. There were two ways to run the code. The first version takes three arguments, the first one is the name of the executable file which is appendMeMore, the second argument is the filename, and the third one is the number of bytes. In the second version, on the other hand, there is another argument which is a single 'x' which means that the way of appending will be different from the first version.

The first version is to create a file and write 1 million bytes to the same file simultaneously.

```
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ gcc appendMeMore.c -o appendMeMore
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ ./appendMeMore f1 1000000 & ./appendMeMore f1 1000000
```

The first version is to create a file and write 1 million bytes to the same file simultaneously.

```
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
```

What is the difference between these two running commands?

- The main difference between these two run commands is, in the first version it opens the file f1 with O_APPEND mode, and in the second version it opens the file with only O_WRONLY mode and uses lseek to write 1000000 bytes to file f2.

After we write 1000000 bytes simultaneously to files f1 and f2 by using these two commands, I had different sizes of files even though I wrote 1000000 bytes to both files f1 and f2 **their size was different**. Below you can see file sizes f1 and f2 after running these two commands.

```
-rwxrwxrwx 1 meterose meterose 2000000 Mar 30 04:58 f1
-rwxrwxrwx 1 meterose meterose 1003437 Mar 30 05:02 f2
```

Why we had different sizes of files?

In the first version (\$ appendMeMore f1 1000000 & appendMeMore f1 1000000), the file has been **opened with the mode O_APPEND flag**.

```
else if(argc == 3)
{
    for(i = 0 ; i < bytes_num; i++)
    {
        if(write(fd,"",1) == -1)
        {
            perror("write\n");
            exit(EXIT_SUCCESS);
        }
    }
}
```

In each iteration of for loop write the () function appends the byte at the end of the function **without changing the file offset**.

Therefore, in this way **in each write() the write function starts where the previous write function() left**. Thanks to this, at each iteration of for loop, bytes will be written to the file **without overlapping and overriding each other**. Therefore, this operation will result in a file that is sized 2 million bytes.

```
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ ls -l f1
-rwxrwxrwx 1 meterose meterose 2000000 Mar 30 04:58 f1
[2]- Done ./appendMeMore f2 1000000 x
```

In the second version (\$./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x), the file has been opened **without the O_APPEND flag** and performs this operation differently.

In each iteration, the lseek(fd, 0, SEEK_END) call is made before each write() operation. Before each byte is written, **the file offset is set to the end of the file**.

```
if(argc == 4){
    for(i = 0 ; i < bytes_num; i++)
    {
        if(lseek(fd, 0, SEEK_END) == -1)
        {
            perror("lseek\n");
            exit(EXIT_SUCCESS);
        }
        if(write(fd,"",1) == -1)
        {
            perror("write\n");
            exit(EXIT_SUCCESS);
        }
    }
}
```

Therefore, each write operation will start **at the end of the previous write operation**. The bytes will be written to the file one after the other and in this way, it writes bytes to file f2. However, if two write process tries to write the same location in the file at the same time, **that particular part of the file might be corrupted**.

Therefore, this operation will result in a file that is sized approximately 1 million bytes long **due to the overlapping writes**. This is basically the difference between the sizes file f1 and f2.

```
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ ls -l f2
-rwxrwxrwx 1 meterose meterose 1003437 Mar 30 05:02 f2
```

How to run (appendMeMore.c) with correct inputs?

There are two ways to run this file and create a file. One of them takes 3 arguments and the other one takes 4 arguments

- (\$./appendMeMore f1 1000000 & ./appendMeMore f1 1000000 (3 arguments)
- (\$./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x (4 arguments)

Any other argument which is less than 3 or more than 4 will end up with an error. I checked that error as follows.

```
else
{
    perror("At least 3 arguments is needed but can not be more than 4.\n");
    exit(EXIT_SUCCESS);
}
```

If the user does not give a byte number to the second argument or puts any other sign other than 'x' in the second version, he/she will also end up with an error. I controlled that parts as follows.

```
if(argc == 3 || argc == 4)
{
    if(argc == 4 && strcmp(argv[3], "x") == 0)
        fd = open(filename, O_WRONLY | O_CREAT, modes);
    else if(argc == 3)
        fd = open(filename, O_WRONLY | O_APPEND | O_CREAT, modes);
    else{
        perror("This command does not exist!");
        exit(EXIT_SUCCESS);
    }
    if(fd == -1)
    {
        perror("fd\n");
        exit(EXIT_SUCCESS);
    }
    int bytes_num = atoi(argv[2]);
    if(bytes_num == 0)
    {
        perror("The command must include the number of bytes!\n");
        exit(EXIT_SUCCESS);
    }
}
```

2-) PART 2 (Question 2)

In this part, I implemented my own dup() and dup2() functions. Below you can see the implementations of my own dup functions.

```
int dup(int oldfd)
{
    int newfd = fcntl(oldfd, F_DUPFD, 0); //Create the duplicate of oldfd by using F_DUPFD
    if(newfd == -1) // An error occurred
    {
        perror("newfd");
        return -1;
    }
    else
        return newfd;
}

int dup2(int oldfd, int newfd)
{
    if(oldfd == newfd) //Special case
    {
        if(fcntl(oldfd, F_GETFL) == -1) //Check whether oldfd is valid, if it is not valid set errno to EBADF and return -1
        {
            errno = EBADF;
            return -1;
        }
        else
            return oldfd;
    }
    else
    {
        /* Close newfd if it is still open. */
        close(newfd);
        int duplicate = fcntl(oldfd, F_DUPFD, newfd);
        if(duplicate == -1)
        {
            perror("duplicate");
            return -1;
        }
        return duplicate; //Return the new file
    }
}
```

- dup() function basically takes an integer argument oldfd as a file descriptor and makes a duplicate of it by using the fcntl function and returns the duplicate.
- dup2() a little more complex than the dup() function. It takes two integer arguments as file descriptors. As it was mentioned in the PDF, it has a special case where newfd equals to oldfd. In this case, it checks whether oldfd is a valid file descriptor by using fcntl(oldfd, F_GETFL). If it succeeds, it returns oldfd, if not it returns -1 and set errno to EBADF. If newfd is not equal to oldfd, firstly it closes newfd in case of it is still. After that it creates a duplicate of oldfd by using the fcntl function, if the duplicate's value is not -1 it returns duplicate, otherwise it returns -1.

Test of PART2

- In this part starting from test1, I will test each dup() and dup2() function and possible error cases.

```
#include<fcntl.h>
#include <unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include <errno.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);

int main( )
{
    mode_t modes = S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IWOTH | S_IROTH;
    char *filename1 ="test1.txt";
    char *filename2 ="test2.txt";
    /* TEST1 DUP() Function */
    /* ----- */
    int fd1 = open(filename1, O_CREAT | O_WRONLY | O_APPEND, modes);
    if(fd1 == -1)
    {
        perror("open");
        exit(EXIT_SUCCESS);
    }

    if( write(fd1,"fd1 file descriptor\n",20) == -1)
    {
        perror("write");
        exit(EXIT_SUCCESS);
    }

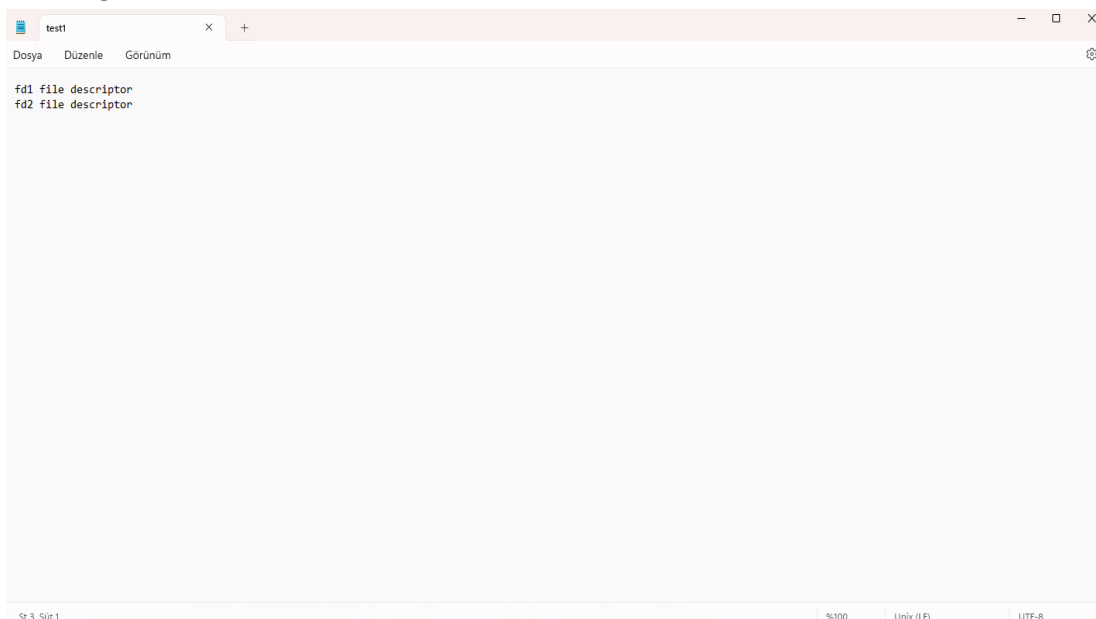
    int fd2 = dup(fd1); /* Create the duplicate of fd1 and assign it to fd2 */

    if( write(fd2,"fd2 file descriptor\n",20) == -1)
    {
        perror("write");
        exit(EXIT_SUCCESS);
    }

    printf("test1 dup() fd1=%d , fd2=%d\n",fd1, fd2); /* The file descriptor number of fd1 and fd2 will be different but
                                                         they will write to the same file which is filename1*/

    /* ----- End of test1 ----- */
}
```

I tested my own dup() function in TEST1. It creates a duplicate of fd1 and assigns it to fd2. Right now both fd1 and fd2 writes to “test1.txt”.



As you can see both fd1 and fd2 file descriptor, wrote to test1.txt. Test is completed.

```
meterose@DESKTOP-3HDDHUD: /mnt/c/homework1
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ gcc part2.c -o part2
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ ./part2
test1 dup() fd1=3 , fd2=4
```

You can see the values of fd1 and fd2 above. Their file descriptor values are different but fd2 is a duplicate of fd1 so they write to the same file which is test1.txt.

Test2 of dup2(oldfd, newfd)

```
/*----- TEST2 DUP2(oldfd, newfd)-----*/
int fd3 = open(filename2, O_CREAT | O_WRONLY | O_APPEND, modes);
int fd5 = 6; // Create a new file descriptor, with value 6.
if(fd3 == -1)
{
    perror("fd3 could not open.");
    exit(EXIT_SUCCESS);
}

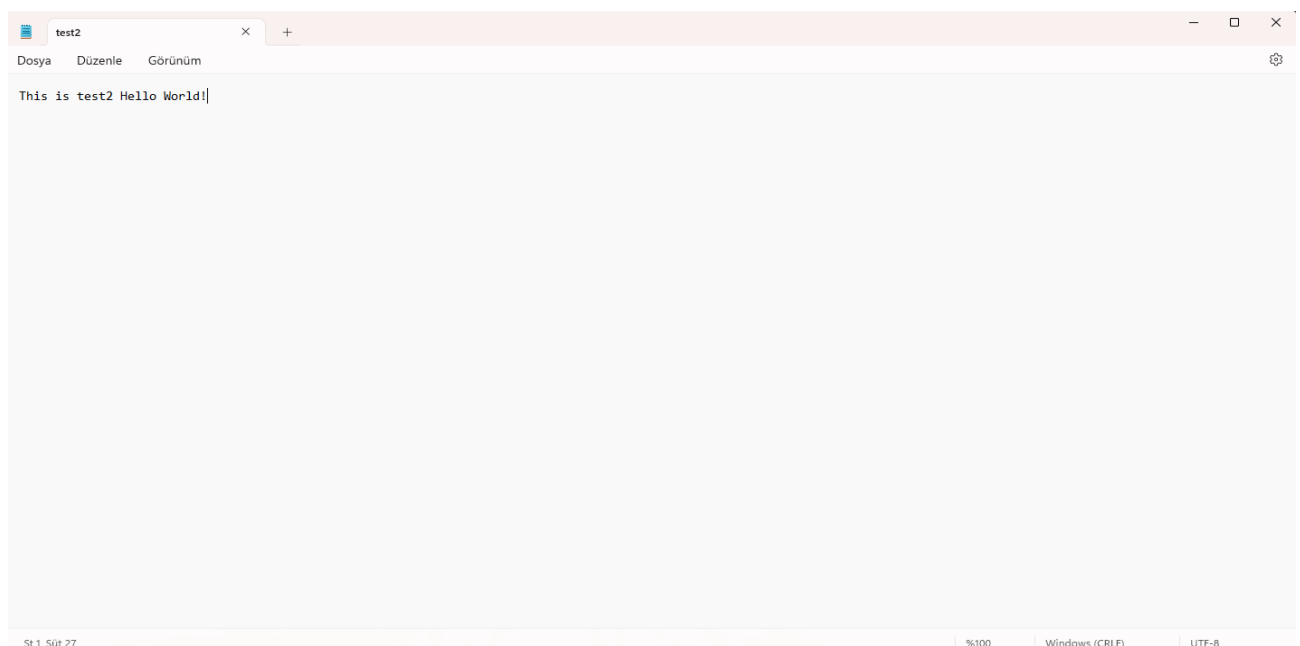
int fd4 = dup2(fd3, fd5); //fd4 will be assigned to 6 which is file descriptor number of fd5, so both f4 and fd5 will be duplicates of fd3

if(fd4 == -1)
{
    perror("fd4");
    exit(EXIT_SUCCESS);
}

write(fd3, "This is test2 ",14);
write(fd4,"Hello ",6); /* fd4 writes Hello to filename2.txt */
write(fd5,"World!",6); /* fd5 writest World! to filename2.txt */

printf("test2 dup2() fd3=%d fd4=%d fd5=%d\n",fd3,fd4,fd5);
```

- int fd3 file descriptor opens a file name “test2.txt”, dup2 function makes a duplicate of fd3 by using the file descriptor value of fd5. So that the returning value of dup2() function will have the file descriptor value of fd5. **In the end, fd4 and fd5 will have the same file descriptor value different than fd3, but each one of them will write to “test2.txt”.**



The screenshot shows a text editor window with the title 'test2'. The content of the file is 'This is test2 Hello World!'. The editor has a menu bar with 'Dosya', 'Düzenle', and 'Görünüm'. The status bar at the bottom indicates 'St 1, Süt 27', '%100', 'Windows (CRLF)', and 'UTF-8'.

As you can see above, each one of them wrote to the file test2.txt.

```

fd7: Bad file descriptor
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ gcc part2.c -o part2
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ ./part2
test1 dup() fd1=3 , fd2=4
test2 dup2() fd3=5 fd4=6 fd5=6

```

You can see the file descriptor values of test2 above. As you can see both fd4 and fd5 have the same file descriptor value which is 6. The file descriptor value of fd3 is different from theirs **but each one of them writes to the same file.**

Test of not valid file descriptor

```

/*----- Not valid file descriptor testcase -----*/
int fd6 = open("not_exist.txt",O_RDONLY);
int fd7 = dup2(fd6,fd6);

if(fd7 == -1)
{
    perror("fd7");
}
/* ----- */

```

As was mentioned in the PDF, there is a special case for dup2() function. If oldfd and newfd are equal to each other, and oldfd is not a valid file descriptor number it will return -1 and set errno to EBADF. I handled this special case in the dup2() function. **You can see how it was handled below.**

```

int dup2(int oldfd, int newfd)
{
    if(oldfd == newfd) //Special case
    {
        if(fcntl(oldfd,F_GETFL) == -1) //Check whether oldfd is valid, if it is not valid set errno to EBADF and return -1
        {
            errno = EBADF;
            return -1;
        }
        else
            return oldfd;
    }
}

```

Test results of the not valid file descriptor.

```

meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ gcc part2.c -o part2
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ ./part2
test1 dup() fd1=3 , fd2=4
test2 dup2() fd3=5 fd4=6 fd5=6
fd7: Bad file descriptor
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ _

```



```

/*----- Test case close ----- */
/* I closed fd3 but did not close fd4 which is duplicate of fd3. Lets try to write to a file */

if(write(fd4," fd4",4) == -1)
{
    perror("write");
}
close(fd4);
close(fd5);
/* All tests are done */
return 0;
}

```

In this test, I controlled the condition where fd3 is closed but the duplicates fd4 and fd5 are still open. Even though one file descriptor of the file is closed the two duplicates are still open. Therefore they can write to the file. Let's see the test results above.



- As you can see above, duplicate file descriptor fd4 can still write to the file test2.txt.

So we tested every condition for part 2, and all tests are done.

3-) PART3 (Question 3)

In this part, I wrote part3.c to show that duplicated file descriptors share a file offset and value and open file. Let's perform all the tests and see their results.

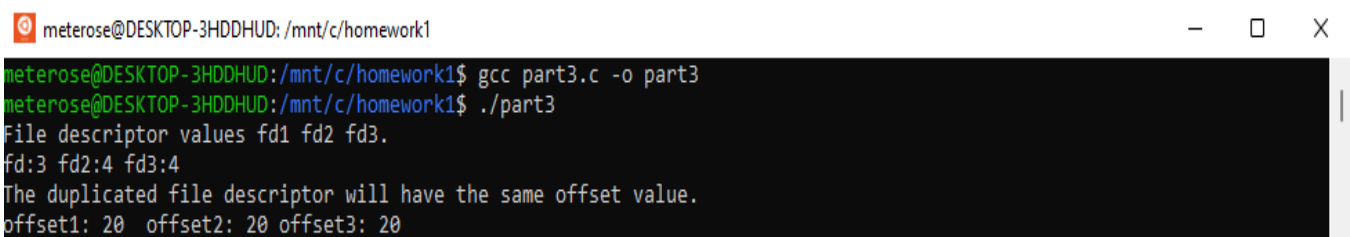
Test1

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);

int main( )
{
    mode_t modes = S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IWOTH | S_IROTH;
    int fd = open("myfile.txt", O_CREAT | O_WRONLY, modes);
    if(fd == -1)
    {
        perror("open");
        exit(EXIT_SUCCESS);
    }
    int fd2 = dup(fd);
    int fd3 = dup2(fd, fd2);
    printf("File descriptor values fd1 fd2 fd3.\n");
    printf("fd:%d fd2:%d fd3:%d\n",fd, fd2, fd3);
    /* ----- Test1 -----*/
    off_t offset1 = lseek(fd, 20, SEEK_CUR);
    off_t offset2 = lseek(fd2, 0, SEEK_CUR);
    off_t offset3 = lseek(fd3, 0, SEEK_CUR);

    printf("The duplicated file descriptor will have the same offset value.\n");
    printf("offset1: %ld offset2: %ld offset3: %ld\n",offset1, offset2, offset3);
}
```

- The first test is to see whether the duplicates of the file descriptor share a file offset value. In this example above, I created a duplicate of fd which is fd2, and after that by using the dup2() function another duplicate is created which is fd3. Right now fd, fd1, and fd2 write to the same "file.txt". Let's see whether they find the same offset or not.



```
meterose@DESKTOP-3HDDHUD: /mnt/c/homework1
meterose@DESKTOP-3HDDHUD: /mnt/c/homework1$ gcc part3.c -o part3
meterose@DESKTOP-3HDDHUD: /mnt/c/homework1$ ./part3
File descriptor values fd1 fd2 fd3.
fd:3 fd2:4 fd3:4
The duplicated file descriptor will have the same offset value.
offset1: 20 offset2: 20 offset3: 20
```

- As you see above, we set the first offset to a value by using lseek(fd, 20, SEEK_CUR), and fd2 and fd3 have been affected automatically. **Therefore, it was verified that duplicated file descriptors share a file offset value.**

Test2

Right now let's create another file descriptor and open the same file with it and see whether the new file descriptor shares the file offset with fd1, fd2 and fd3.

```
int fd4 = open("myfile.txt", O_WRONLY, modes);
if(fd4 == -1)
{
    perror("open");
}
off_t offset4 = lseek(fd4, 0, SEEK_CUR);
printf("fd4 is not a duplicated file descriptor, so that the offset4 will be different than duplicated ones.\n");
printf("offset1: %ld offset2: %ld offset3: %ld offset4: %ld\n", offset1, offset2, offset3, offset4);
/*----- Test1 END -----*/
```

- I created another file descriptor which is fd4, fd4 opens the same file "myfile.txt" as fd. However, it is not a duplicate of fd4. Let's see whether fd and fd4 share the file offset with fd and its duplicates(fd2, fd3).

```
fd4 is not a duplicated file descriptor, so that the offset4 will be different than duplicated ones.
offset1: 20 offset2: 20 offset3: 20 offset4: 0
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$
```

- Test2 is completed, as you can see above offset of fd4 is not sharing the same offset with fd and its duplicates(fd and fd3) because it is not a duplicate of fd.

All outputs

```
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ gcc part3.c -o part3
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ ./part3
File descriptor values fd1 fd2 fd3.
fd:3 fd2:4 fd3:4
The duplicated file descriptor will have the same offset value.
offset1: 20 offset2: 20 offset3: 20
fd4 is not a duplicated file descriptor, so that the offset4 will be different than duplicated ones.
offset1: 20 offset2: 20 offset3: 20 offset4: 0
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$
```

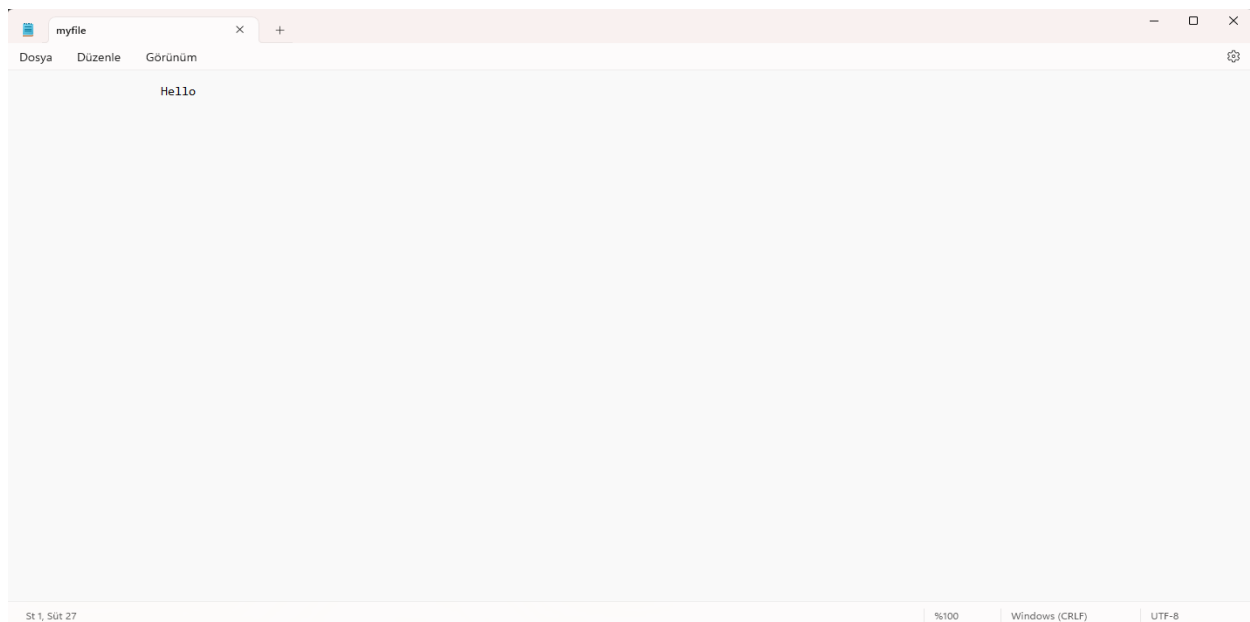
Test3

```
/* ----- Test2 ----- */
/* Try to write to the file and check whether duplicated file descriptors shares the same offset */
if(write(fd,"Hello ",6) == -1)
{
    perror("write");
}
/*
if(write(fd2,"World",5) == -1)
{
    perror("write");
}

if(write(fd4,"Test myFile",11) == -1)
{
    perror("write");
}

if(write(fd3, " Test2 is completed",19 )== -1)
{
    perror("write");
}
*/
/* ----- Test 2 END ----- */
```

- We know that fd , fd2 and fd3 share the same offset value and open file. fd4, on the other hand, has a different offset but it writes to the same file with fd, fd2, and fd3. So let's see how they write to the file. So we assigned lseek(fd, 20, SEEK_CUR) to offset1 since both offsets2 and offsets3 have also been affected. Therefore, fd, fd2, and fd3 will write to the file 20 byte after the start of the file, and fd4 on the other hand starts from the start of the file.



- As you can see above, fd write to myFile 20 byte after the start of the file.

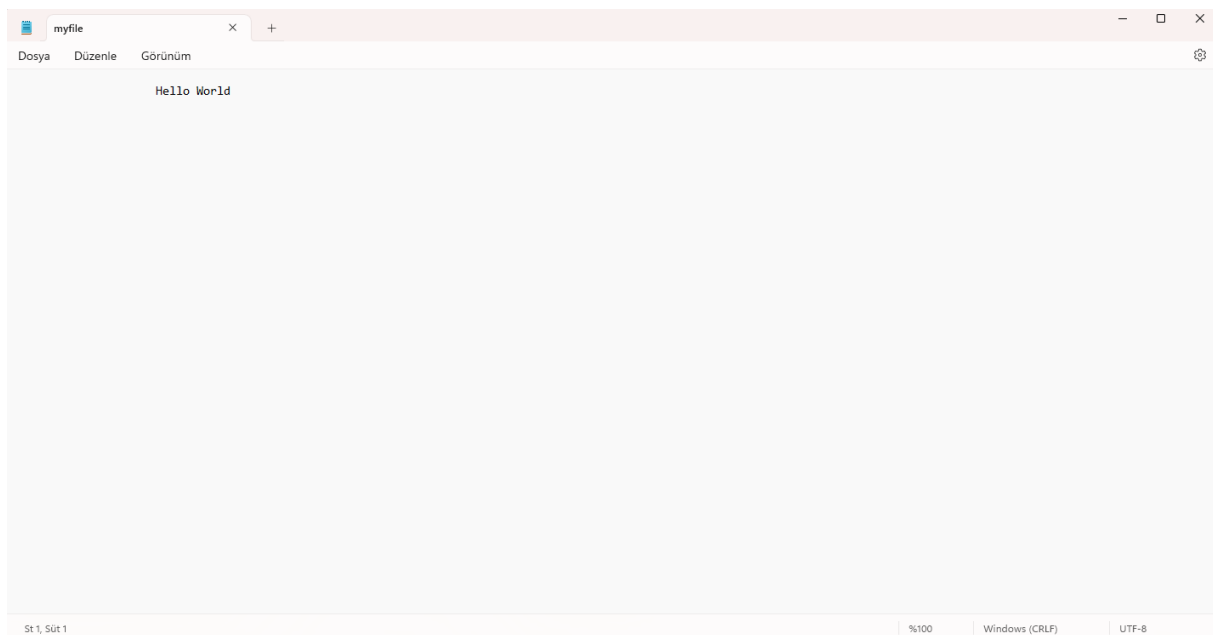
```

/* ----- Test2 ----- */
/* Try to write to the file and check whether duplicated file descriptors shares the same offset */
if(write(fd,"Hello ",6) == -1)
{
    perror("write");
}

if(write(fd2,"World",5) == -1)
{
    perror("write");
}

```

- Right now write hello to the myFile with file descriptor fd2.



- As you can see above fd2 wrote “World!” right after the position where fd left off.

```

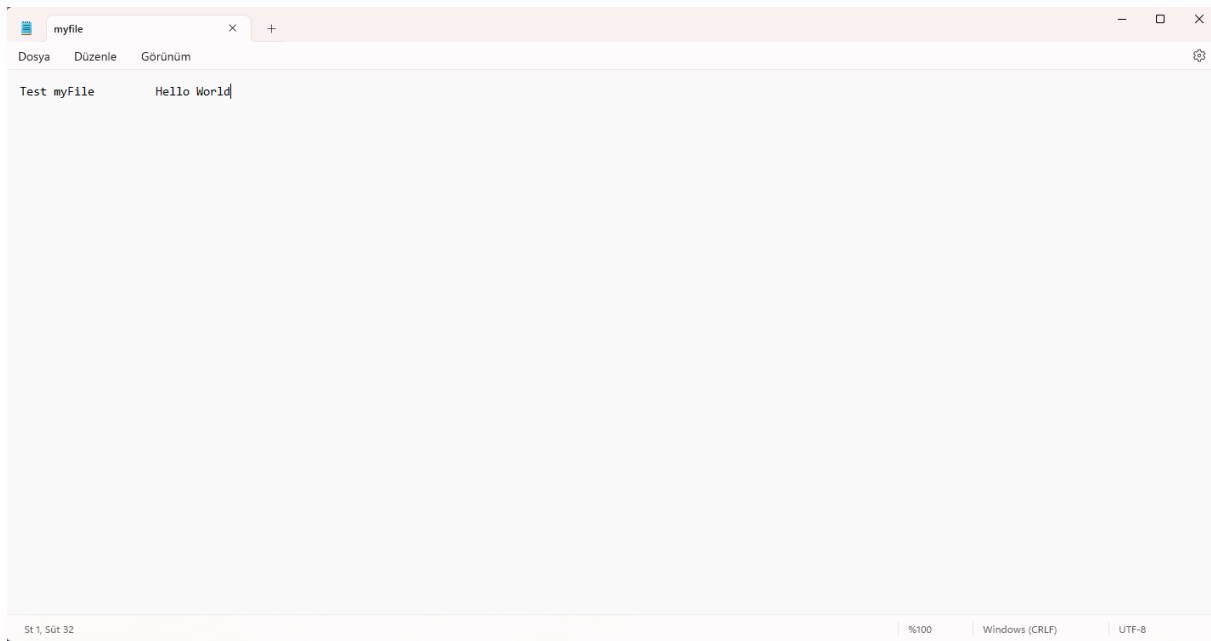
/* ----- Test2 ----- */
/* Try to write to the file and check whether duplicated file descriptors shares the same offset */
if(write(fd,"Hello ",6) == -1)
{
    perror("write");
}

if(write(fd2,"World",5) == -1)
{
    perror("write");
}

if(write(fd4,"Test myFile",11) == -1)
{
    perror("write");
}

```

- As we mentioned before **fd4 does not share the same offset and open file with fd, fd2, and fd3. So, fd4 will not write right after other file descriptors.** Let's test and see it



- As you can see above since fd4 is not a duplicate of fd it started to write starting position of the file.

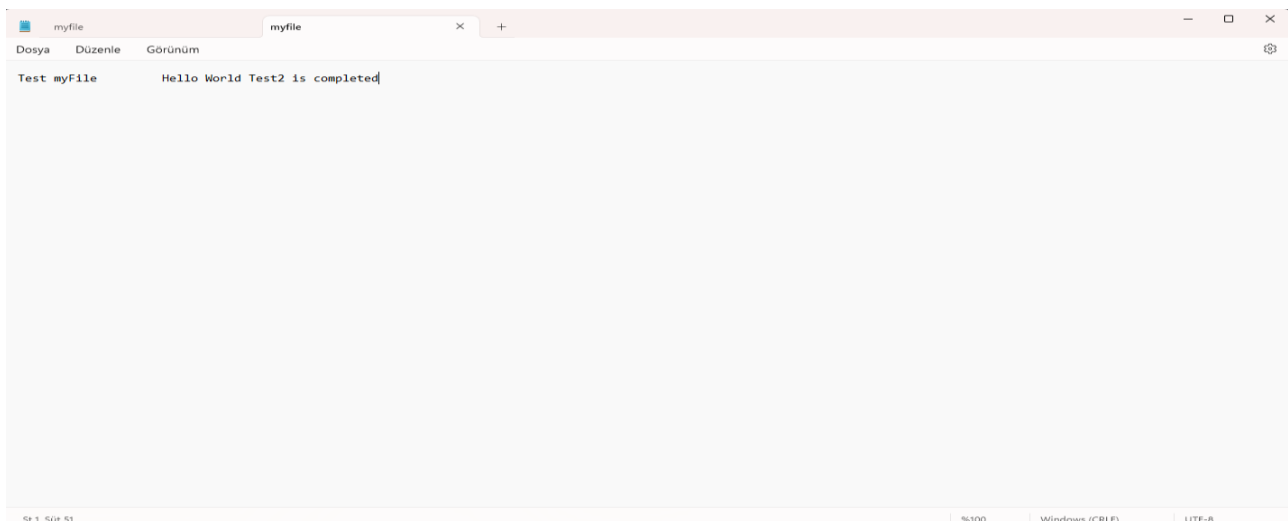
```
/* ----- Test2 ----- */
/* Try to write to the file and check whether duplicated file descriptors shares the same offset */
if(write(fd,"Hello ",6) == -1)
{
    perror("write");
}

if(write(fd2,"World",5) == -1)
{
    perror("write");
}

if(write(fd4,"Test myFile",11) == -1)
{
    perror("write");
}

if(write(fd3, " Test2 is completed",19 )== -1)
{
    perror("write");
}
```

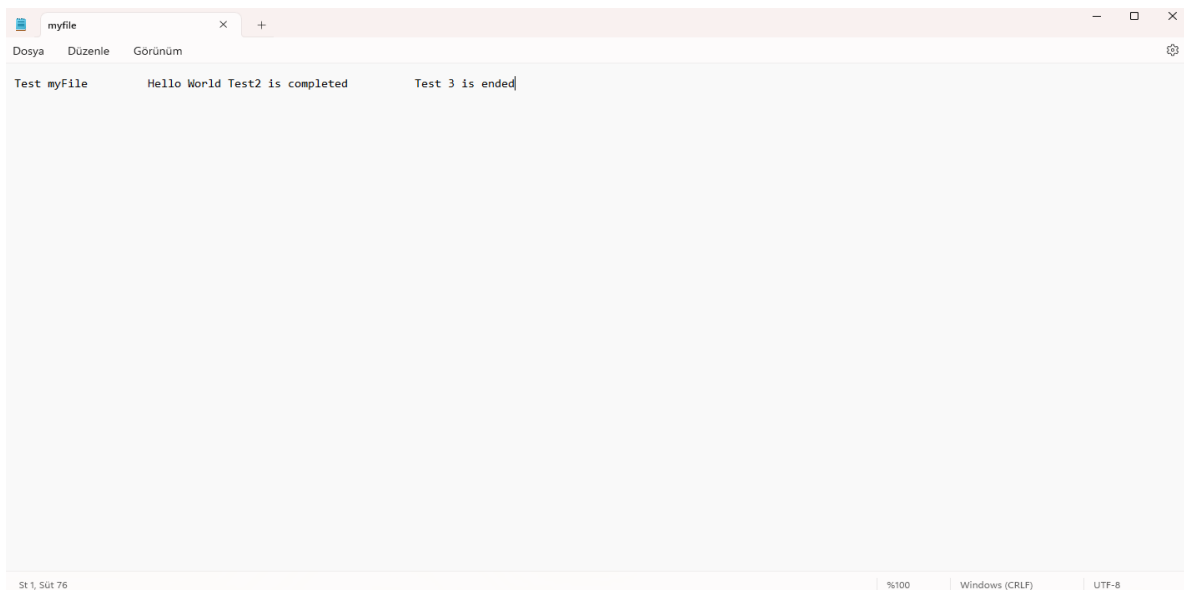
- To test it again for a very last time let's write to a file by using fd3 which is a duplicate of fd.



- As you can see above, since fd3 is duplicate of fd, fd3 wrote to a myFile right after fd and fd2.

```
/* ----- Test 3 ----- */
lseek(fd, 10, SEEK_CUR);
if(write(fd, "Test 3", 6) == -1)
{
    perror("write");
}
if(write(fd2, " is", 3) == -1)
{
    perror("write");
}
if(write(fd3, " ended", 6) == -1)
{
    perror("write");
}
/* ----- Test 3 END ----- */
/* All test are done, program shows that fd, fd2, fd3 are duplicated file descriptors and they shares the same offset value and openfile */
return 0;
```

- Set lseek 10 bytes after the current position of fd, fd2, and fd3 and write to myFile again with duplicates file descriptors.



- As you can see all other duplicates have been affected by the lseek and wrote right after fd.

Test4

In the last test, I showed that duplicated file descriptors share an open file. I got the file status of fd, fd2, and fd3 and check whether their file status are the same or not.

```
/* ----- Test 4 ----- */
/* Lets see if duplicates have the same file status ----- */
int filestatus1 = fcntl(fd, F_GETFL);
int filestatus2 = fcntl(fd2, F_GETFL);
int filestatus3 = fcntl(fd3, F_GETFL);
printf("filestatus fd1 = %d filestatus fd2 = %d filestatus fd3 = %d\n", filestatus1, filestatus2, filestatus3);
```

- Lets see the results.

```
filestatus fd1 = 32769 filestatus fd2 = 32769 filestatus fd3 = 32769
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$
```

- As you can see above since their file status is the same.

makefile and makeclean Test

```
meterose@DESKTOP-3HDDHUD:/mnt/c$ cd homework1
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ make
gcc appendMeMore.c -o appendMeMore -Wall
gcc part2.c -o part2 -Wall
gcc part3.c -o part3 -Wall
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ make makeclean
rm -f appendMeMore part2 part3
meterose@DESKTOP-3HDDHUD:/mnt/c/homework1$ _
```

All parts are explained and tested.

Mehmet Mete Şamlıoğlu

200104004093

