CSE344 HW1 Report

Ersel Celal Eren – 1901042673

Q1)

In this question, I wrote a program that takes 3 or 4 command line arguments from user and then call function to write into file.

```c
// Check if the correct number of command line arguments was provided
if(argc < 3 || argc > 4){
    printf("Error in command-line arguments. \n");
    return 1;
}

// Get the command line arguments
filename = argv[1];
num_bytes = atoi(argv[2]); // convert the second argument to an integer
third_arg = (argc == 4 && argv[3][0] == 'x'); // check if the third argument is 'x'
```

In this function, first check third argument is provided or not. According to that, O_APPEND flag is added to flags and called open(…) syscall. Next, the code checks if a third argument is present. If the third argument is present, the code sets the offset variable to the end of the file using the lseek function with the SEEK_END flag. The lseek() is used to move the file pointer associated with the file descriptor fd to the end of the file. If lseek returns -1, which means an error, prints an error with perror and returns 1.

```c
for(int i=0; i<num_bytes; i++){
    off_t offset = 0;
    // Perform lseek if the third argument is 'x'
    if(third_arg){
        offset = lseek(fd, 0, SEEK_END);
        if(offset == -1){
            perror("lseek");
            return 1;
        }
    }
    // Write a single byte to the file
    if(write(fd,&byte,1) == -1){
        perror("write");
        return 1;
    }
}
```

After setting the offset, the code writes a single byte to the file using the write function. The code then repeats the process for num_bytes iterations. So writes num_bytes copies of a single byte to the file descriptor fd. If the third argument is present, the writes will append the bytes to the end of the file. If the third argument is not present, the writes will overwrite the existing contents of the file.

The reason why f1 becomes 2 MB while f2 remains slightly more than 1 MB is that the file is opened without the O_APPEND flag, that causes the program to overwrite any existing content from the beginning of the file.

```
-rwxr-xr-x 1 ersel    ersel      16312 Mar 30 03:41 appendMeMore
-rw-r--r-- 1 ersel    ersel       1987 Mar 30 17:05 appendMeMore.c
-rw-r--r-- 1 ersel    ersel    2000000 Mar 30 17:50 f1
-rw-r--r-- 1 ersel    ersel    1071448 Mar 30 17:50 f2
```

***$ appendMeMore f1 1000000 & appendMeMore f1 1000000***

***$ appendMeMore f2 1000000 x & appendMeMore f2 1000000 x***

Calls the appendMeMore program twice with the filename f1 and argument num-bytes set to 1000000. The & at the end of the first command runs the second instance of the program in the background.

By default, the program opens the file with the O_APPEND flag, which means that all writes are appended to the end of the file without modifying the existing contents.

Because of that, when the appendMeMore program is runing without the x flag, it appends 1000000 bytes to the end of the file without modifying the existing contents. This results in two instances of the program appending their 1000000 bytes each to the end of the file f1, resulting in a final size of 2MB.

At second run, when the appendMeMore program is running with the x flag, the program omits the O_APPEND flag and performs an lseek(fd, 0, SEEK_END) call before each write() operation. The lseek() call sets the file offset to the end of the file before each write, which means that each instance of the program writes its 1000000 bytes starting from the end of the existing file. This results in a final size of 1 million bytes for the file f2.

*As a result, O_APPEND flag ensures that writes are appended to the end of the file without modifying the existing contents, while omitting the flag and using lseek() before each write allows the program to write to the end of the file starting from the existing contents.*

Q2)

In second question of homework, I implemented dup() and dup2() syscalls myself with the names of mydup() and mydup2().

```c
int mydup(int oldfd) {
    // Duplicate the file descriptor using fcntl() with F_DUPFD flag
    int newfd = fcntl(oldfd, F_DUPFD, 0);

    // Check if fcntl() failed
    if (newfd == -1) {
        // Error occurred, return -1 and set errno
        // print error message by using errno
        perror("dup");
        return -1;
    }

    // Return the new file descriptor
    return newfd;
}
```

The mydup() function takes an integer argument oldfd, which is the file descriptor will be duplicated. It first calls the fcntl syscall with the F_DUPFD flag to create a new file descriptor to the same open file description as the oldfd. If fcntl returns -1, which indicates an error, the function prints an error message using perror and sets errno to indicate the error. Otherwise, it returns the new file descriptor.

```c
int mydup2(int oldfd, int newfd) {
    int flags, res;

    // Check if newfd is valid by checking if fcntl(newfd, F_GETFD) succeeds.
    if(fcntl(newfd, F_GETFD) != -1){
        // newfd is valid, close it
        close(newfd);
    }

    // oldfd equals newfd
    if(newfd == oldfd){
        //If oldfd is not valid, F_GETFL will return -1 and set errno to EBADF.
        // Check if oldfd is valid done by checking if fcntl(oldfd, F_GETFL) succeeds.
        flags = fcntl(oldfd, F_GETFL);
        if(flags == -1){ //oldfd is not valid,
            //set errno and return -1
            errno = EBADF;
            return -1;
        }
        return newfd; // oldfd is valid, return newfd
    }

    // Duplicate the file descriptor using fcntl() with F_DUPFD flag
    res = fcntl(oldfd,F_DUPFD,newfd);
    if(res == -1){
        perror("dup2");
        return -1;
    }
    // Return the new file descriptor
    return res;
}
```

The mydup2() takes two integer arguments, oldfd and newfd. It first checks whether newfd is a valid file descriptor by calling fcntl with the F_GETFD flag. If this call

succeeds, it means that newfd is a valid file descriptor, and the function closes it using the close system call.

Next, the function checks whether oldfd is equal to newfd. If so, it checks whether oldfd is a valid file descriptor by calling fcntl with the F_GETFL flag. If this call succeeds, it means that oldfd is a valid file descriptor, and the function returns newfd without creating a new duplicate file descriptor. Otherwise, the function sets errno to EBADF to indicate an error and returns -1.

If oldfd is not equal to newfd or newfd is not a valid file descriptor, the function calls fcntl with the F_DUPFD flag to create a new file descriptor that refers to the same open file description as the oldfd, using the lowest available file descriptor greater than or equal to newfd. If fcntl returns -1, indicating an error, the function prints an error message using perror and returns -1. Otherwise, it returns the new file descriptor.

In summary, both mydup and mydup2 are used to duplicate file descriptors. mydup creates a new duplicate file descriptor using the lowest available file descriptor greater than or equal to 0. mydup2 creates a new duplicate file descriptor using the specified file descriptor number, and optionally closes the existing file descriptor if it is valid.

Test for Question 2 and Question 3 -> In file mydup.c , I implemented mydup() and mydup2() functions and wrote test cases for Q2 and Q3. To run the program user should give name of the file in command-line. Ex: ./my_dup f1

The program starts by checking the command-line arguments. It expects one or two arguments. If the number of arguments is less than 2 or greater than 3, an error message is printed and the program exits. Sets the first argument as filename.

```
//check command line arguments
if(argc<2 || argc>3){
    printf("Error in command-line arguments. \n");
    return 1;
}

//get filename from command line
char* filename = argv[1];
```

Then, the program opens the file with given filename specified by the first command-line argument using open syscall with O_RDWR and O_CREAT flags. These flags are used to open the file for both reading and writing and to create the file if it does not

exist. The file permissions are set to read and write permissions for the user (S_IRUSR | S_IWUSR).

```
// Open a file for reading and writing with read and write permissions
fd = open(filename, O_RDWR | O_CREAT , S_IRUSR | S_IWUSR);

if(fd == -1){ // Check if open() failed
    perror("open syscall");
    return 1;
}
```

If open syscall succeeds, the program duplicates the file descriptor using the mydup function. The original file descriptor is stored in fd and the new file descriptor is stored in newfd. If mydup function fails, an error message is printed with the perror function, the original file descriptor is closed with close function, and the program exits with a status code of 1.

The program then prints the original and new file descriptor values using printf function.

Next, the program duplicates the file descriptor to a specific value using the mydup2 function. The original file descriptor is stored in fd and the new file descriptor is assigned to a specific value tempfd. If mydup2 function fails, an error message is printed with the perror function, the original file descriptor is closed with close function, and the program exits with a status code of 1.

Finally, the program prints the original and new file descriptor values using printf function.

By comparing the values of the original and new file descriptors, we can verify whether the duplication has been successful or not. If the program produces the expected results, it means that the custom implementations of dup and dup2 are correct.

```
--------------------------------------------------------------------------------
---------------------------------- Q2 ----------------------------------
--------------------------------------------------------------------------------

After dup()  -> Original fd : 3 ||| new fd : 4

After dup2() -> original fd : 3 ||| tempfd : 15
```

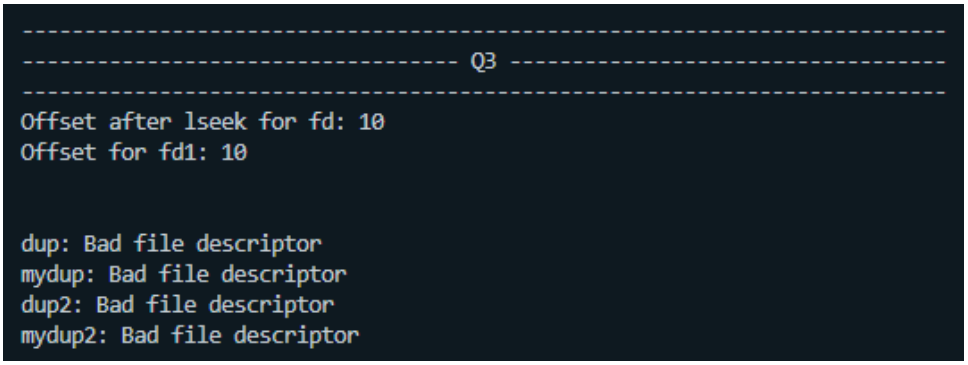
Q3)

This code tests the mydup() syscall and the file offset in a file opened with the open() syscall.

First, it duplicates the file descriptor fd to fd1. Then, it sets the file offset of fd to 10 bytes from the beginning of the file using the lseek() syscall. The lseek() system call changes the file offset of the specified file descriptor to the given offset value.

After that, it retrieves the current file offset of fd1 using another lseek() syscall. Since fd1 was duplicated from fd, it should share the same file offset as fd. Therefore, the expected output of this program is that both the file offsets of fd and fd1 are 10.

The purpose of this test is to demonstrate that duplicating a file descriptor using mydup() creates a new file descriptor that shares the same file offset as the original file descriptor. In other words, changes made to the file offset of one file descriptor are reflected in all duplicated file descriptors.




***Note: "make" command compiles appendMeMore.c and my_dup.c files. "make clean" deletes appendMeMore, my_dup, f1 and f2 files.***