Mehmet Acar

1801042095

HW1 Report

Part 1

Design

Program takes up to three command-line arguments: the filename to write to, the number of bytes to append, and an optional argument 'x' to use lseek instead of O_APPEND. It opens the file with the appropriate flags, and then writes one byte at a time to the end of the file. If the 'x' argument is present, it calls lseek before each write to ensure that the write goes to the end of the file.

Test Case 1

To run one instance of this program without the x argument to write 1 million bytes to file1, I used the following command:

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ./appendMeMore file1 1000000
```

To run one instance of this program with the x argument to write 1 million bytes to file2, I used the following command:

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ./appendMeMore file2 1000000 x
```

Test Case 1 Result

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ls -l file1 file2
-rw------- 1 mehmet mehmet 1000000 Mar 29 21:58 file1
-rw------- 1 mehmet mehmet 1000000 Mar 29 21:58 file2
```

Size of file1 and file2 is same. Because we run one process for writing number of bytes to the file1 with append mode. Also, we run one process for writing number of bytes to the file 2 with lseek. Single process does not cause to race condition and data corruption.

Test Case 2

To run two instances of this program at the same time without the x argument to write 1 million bytes to the same file, I used the following command:

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ./part1 f1 1000000 & ./part1 f1 1000000
```

To run two instances of this program at the same time, writing to a same file, but this time specifying the x argument, I used the following command:

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ./part1 f2 1000000 x & ./part1 f2 1000000 x
```

Test Case 2 Result

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ls -l f1 f2
-rw------- 1 mehmet mehmet 2000000 Mar 28 16:49 f1
-rw------- 1 mehmet mehmet 1045233 Mar 28 16:49 f2
[2]+  Done                    ./part1 f2 1000000 x
```

The difference in the sizes of f1 and f2 can be explained by the fact that when using the O_APPEND flag, the operating system guarantees that all writes to the file will be appended atomically without any other process writing to the file at the same time. In contrast, when using lseek and not using O_APPEND, two processes writing to the same file can lead to race conditions and data corruption, as they may both try to write to the same offset in the file.

Test Case 3

When we run the one instance of the program with wrong number of command line argument(less than 3 or bigger than 4), program should print an error and it gives an correct version of command line arguments as an advice to the user.

Test Case 3 Result

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ./appendMeMore f1
Command line argument number should be 3 or 4
Command line arguments should be like this and [x] is optional : ./appendMeMore <filename> <num-bytes> [x]
```

Test Case 4

When we run the two instances of the program with wrong number of command line argument(less than 3 or bigger than 4) at the same time, program should print an error it gives an correct version of command line arguments as an advice to the user.

Test Case 4 Result

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ./appendMeMore f2 & ./appendMeMore f2
[1] 10087
Command line argument number should be 3 or 4
Command line arguments should be like this and [x] is optional : ./appendMeMore <filename> <num-bytes> [x]
Command line argument number should be 3 or 4
Command line arguments should be like this and [x] is optional : ./appendMeMore <filename> <num-bytes> [x]
[1]+  Exit 255                ./appendMeMore f2
```

Part 2

Design

In this part, I create two functions in order to implement dup and dup2 system calls. Function names are mydup and mydup2.

In `my_dup(oldfd) function`, we use `fcntl()` with the `F_GETFL` command to check if `oldfd` is a valid file descriptor. If oldfd is not a valid file descriptor, we set errno to EBADF, printing error message and return -1. Otherwise, we simply call `fcntl()` with the `F_DUPFD` command and a `0` argument to get a new file descriptor that is greater than or equal to 0 and not already in use by the process. If the call succeeds, we return the new file descriptor. If the call fails, we return -1.

In `my_dup2(oldfd, newfd) function`, we call fcntl() with the F_GETFL command to check if oldfd is a valid file descriptor. If oldfd is not a valid file descriptor, we set errno to EBADF, printing error message and return -1. Then, we check if oldfd is equal to the newfd. If they are, we return newfd. Otherwise, we close the newfd. Then we use `fcntl()` with the `F_DUPFD` command and the value of `newfd` as the argument to duplicate the file descriptor `oldfd` onto `newfd`. If the call succeeds, we return the new file descriptor. Otherwise, we return -1.

Test Cases

In main part of program, I create different test cases for checking mydup(oldfd) and mydup2(oldfd,newfd) functions work correctly.

Firsly, I create a file descriptor(fd1) for testing mydup(oldfd) and mydup2(oldfd,newfd) function works correctly and this descriptor creates a input file which is part2.txt.

Then, I duplicated this descriptor with mydup(oldfd) function. So, I have one original file descriptor(fd numbered 3) and one copied file descriptor(fd numbered 4) . Later, I write a string to the part2.txt file with copied file descriptor(fd numbered 4) . If duplication operation is not applied successfully, copied descriptor can not write a string to the part2.txt. But, if you look at to the part2.txt, the string is written to the file successfully and this event proves that mydup(oldfd) function works correctly.

Later, I duplicated original file descriptor(fd numbered 3) with mydup2(oldfd,newfd) function and I get a copied another file descriptor(fd numbered 5). Later, I write a string to the part2.txt file with copied file descriptor(fd numbered 5) . If duplication operation is not applied successfully, copied descriptor can not write a string to the part2.txt. But, if you look at to the part2.txt, the string is written to the file successfully and this event proves that mydup2(oldfd,newfd) function works correctly.

Then, I send integers as a parameter to mydup(oldfd) and mydup2(oldfd, newfd) functions in order to testing error cases. For example, mydup(10) and mydup2(15,15) are called from main part of program. These integers does not belong to any file descriptors. Because of this, in body part of mydup(oldfd) and mydup2(oldfd,newfd) functions, program prints an error message. Error message is bad file descriptor.

Test Case Results

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ./part2
fd3 is an original file descriptor for part2.txt

fd4 is a copied file descriptor of fd3 after mydup(oldfd) operation
fd4 write its string to the part2.txt successfully

fd5 is a copied file descriptor of fd3 after mydup2(oldfd,newfd) operation
fd5 write its string to the part2.txt successfully

In order to test EBADF error in mydup(oldfd) function, I send an unused file descriptor number to function
Err message for mydup(oldfd) function: Bad file descriptor

In order to test EBADF error in mydup2(oldfd,newfd) function, I send an unused file descriptor numbers to function
Err message for mydup2(oldfd, newfd) function: Bad file descriptor
```

If you look at to the content of the part2.txt file with cat part2.txt command in terminal, you can see that copied file descriptors can write its strings to the part2.txt file successfully.

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ cat part2.txt
This string is written to the part2.txt file with copied file descriptor after mydup(oldfd) operation
This string is written to the part2.txt file with copied file descriptor after mydup2(oldfd,newfd) operation
```

Part 3

Design and Test Cases

In this part, mydup(oldfd) and mydup2(oldfd, newfd) functions are used for testing that original and copied file descriptors share a file offset value.

In test part, part3.txt file is created in order to original file descriptor and copied file descriptor share a file offset value.

Firstly, I create original file descriptor(fd numbered 3). This descriptor creates part3.txt file with writing mode. Then, I create copied file descriptor(fd numbered 4) of original file descriptor after mydup(oldfd) operation. Then, I write a string to part3.txt file with copied file descriptor(fd numbered 4) , then I printed current offset values of original and copied file desciptor and these two offset values are same.

Then, I create another copied file descriptor(fd numbered 5) of original file descriptor(fd numbered 3) after mydup2(oldfd,newfd) operation. . Then, I write a string to part3.txt file with copied file descriptor(fd numbered 5) , then I printed current offset values of original and copied file desciptor and these two offset values are same.Therefore, these results proves that duplicated file descriptors share a file offset value and open file.

Test Case Results

```
mehmet@mehmet-ubuntu:~/Desktop/system-prog-2023-hw/hw1$ ./part3
fd3 is an original file descriptor for part3.txt

fd4 is created with mydup(oldfd) operation and it is copied file descriptor of fd3
fd4 wrote its string to the part3.txt successfully

fd3 (original file descriptor) offset value after writing string to file: 102
fd4 (copied file descriptor of fd3) offset value after writing string to file: 102

fd5 is created with mydup2(oldfd,newfd) operation and it is copied file descriptor of fd3
fd5 wrote its string to the part3.txt successfully

fd3 (original file descriptor) offset value after writing string to file: 211
fd5 (copied file descriptor of fd3) offset value after writing string to file: 211
```

If you look at to the content of the part3.txt file with cat part3.txt command in terminal, you can see that copied file descriptors can write its strings to the part3.txt file successfully.