CSE344

HomeWork #1

Report

1901042676

Mustafa Mercan

Part 1 :

In this part of the assignment, we were asked to create a program that performs writing to a file in byte format by implementing different operations based on a specific parameter received. According to the format, if the [x] parameter exists, the lseek(fd, 0, SEEK_END) function is called before each write operation in order to perform the write operation afterwards. If the [x] parameter does not exist, the O_APPEND flag is used for writing to the file in byte format.

When we run our program in the following formats:

./appendMeMore f1 1000000 & ./appendMeMore f1 1000000 , ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x

We were asked to list the size of the created f1 and f2 files using ls -l, and explain the reason for the difference in their sizes.

```
./appendMeMore f1 1000000 & ./appendMeMore f1 1000000
[1] 22599
[1]+  Done                    ./appendMeMore f1 1000000
```

```
./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
[1] 22613
[1]+  Done                    ./appendMeMore f2 1000000 x
```

```
-rw-r--r-- 1 mustafa mustafa 2000000 Mar 30 20:11 f1
-rw-r--r-- 1 mustafa mustafa 1376427 Mar 30 20:12 f2
```

The byte difference between the created f1 and f2 files is due to the fact that when f2 is opened with the [x] parameter, before each write operation, the file position indicator is moved to the end of the file (lseek(fd, 0, SEEK_END)), and then the write operation is performed. As a result, when two different programs that operate on the same file in parallel are run, bytes are written on top of each other, and therefore the size of the file cannot reach exactly the sum of the given byte values. However, since O_APPEND flag is used in the write operations of f1 file, the file position indicator is automatically moved to the end of the file. This prevents the written data from being written on top of each other in the file.

Part 2 :

In part 2 of the assignment, we are asked to implement the dup() and dup2() functions using the fcntl() function and using the close() function where necessary. We are also asked to pay attention to the special case where oldfd equals newfd for the dup2() function. If the checks indicate that oldfd is an invalid value, it is required to return -1 with errno set to EBADF.

```
/*-------------------------------------------------------------------------*/

char file_name_1 [] = "test1.txt";
char file_name_2 [] = "test2.txt";
char file_name_3 [] = "test3.txt";
char buffer_dup_test [20];
char buffer_dup2_test [20];


int fd_1 = open(file_name_1, O_CREAT | O_RDWR | O_APPEND | O_TRUNC, 0644);
int fd_2 = open(file_name_2, O_CREAT | O_RDWR | O_APPEND | O_TRUNC, 0644);
int fd_3 = open(file_name_3, O_CREAT | O_RDWR | O_APPEND | O_TRUNC, 0644);


write(fd_1,"test1\n",6);
write(fd_2,"test2\n",6);
write(fd_3,"test3\n",6);

/*-------------------------------------------------------------------------*/
```

The dup() function creates a copy of an existing file descriptor by returning a copy of the current file descriptor. As we have seen above, I create three different variables fd_1, fd_2, fd_3 and perform an open operation with them. Then I write the values test1, test2, test3 to these files in order.

```
/*------------------------------------------------------------------------*/
/*DUP FUNCTIONS TEST*/
printf("\nDup Function\n\n");

int dup_fd = dup(fd_1);

write(dup_fd,"I'm dup fd\n",11);

printf("dup_fd -> %d\n\n",dup_fd);

lseek(dup_fd,0,SEEK_SET);

read(dup_fd,buffer_dup_test,sizeof(buffer_dup_test));

printf("test1.txt -> %s\n",buffer_dup_test);

/*------------------------------------------------------------------------*/
```

Afterwards, I create a new variable called dup_fd and assign it the value of fd_1 using the expression 'dup_fd = (fd_1);'. Then, I write the text 'I'm dup function' into the file using dup_fd as the file descriptor.

Later, I use the read() function to read from the file using dup_fd as the file descriptor, and print the result to the screen.

After finishing the tests for the dup() function, I performed a similar process for the dup2() function as follows:

```
/*------------------------------------------------------------------------*/

/*DUP2 FUNCTIONS TEST*/
printf("\nDup2 Function\n\n");

int dup2_fd = dup2(fd_2,fd_3);

write(dup2_fd,"I'm dup2 fd\n",12);

printf("dup2_fd -> %d\n",dup2_fd);

lseek(dup2_fd,0,SEEK_SET);

read(dup2_fd,buffer_dup2_test,sizeof(buffer_dup2_test));

printf("test2.txt -> %s\n",buffer_dup2_test);

/*------------------------------------------------------------------------*/
close(fd_1);
```

Finally, I closed all the files I opened and ended the process.

```
close(fd_1);
close(fd_2);
close(fd_3);
close(dup_fd);
close(dup2_fd);
```

Part2 Final Output →

```
before nothing use function fd values:
fd_1 -> 3
fd_2 -> 4
fd_3 -> 5

Dup Function

dup_fd -> 6

test1.txt -> test1
I'm dup fd


Dup2 Function

dup2_fd -> 5
test2.txt -> test2
I'm dup2 fd
```

Part 3 :

In the third part of the assignment, we were asked to use the functions we created in part 2 to check whether the offset of the duplicated file descriptors matches the offset of the original file descriptor.

First, I created the necessary variables for the test environment. I opened the files, duplicated the file with the dup() function, and wrote the message 'test1' into fd1.

```
/* TEST WITH DUP FUNCTION*/

fd1 = open("file_1.txt", O_RDWR | O_CREAT, 0644);
printf("---Test with dup function---\n\n");
if (fd1 == -1)
{
    perror("open fail");
    exit(EXIT_FAILURE);
}

write(fd1, "test1", 5);

dup_fd = dup(fd1);
if (dup_fd == -1)
{
    perror("dup error");
    exit(EXIT_FAILURE);
}
```

Afterwards, I assigned the offset values of fd1 and dup_fd to variables and compared these two values with each other. If these two values are equal, it means that the offset values are the same. After this operation, I printed these offset values to the screen using the printf() function.

```c
off_t fd1_offset = lseek(fd1,0,SEEK_CUR);
off_t dup_fd_offset = lseek(dup_fd,0,SEEK_CUR);
if (fd1_offset == dup_fd_offset)
{
    printf("fd1 and dup_fd have the same file offset.\n");
    printf("fd1 offset-> %ld\ndup_fd offset-> %ld\n",fd1_offset,dup_fd_offset);
}
else
{
    printf("File location is different.\n");
}

lseek(dup_fd, 0, SEEK_SET);
write(dup_fd, "test1", sizeof(test_buffer));

lseek(fd1, 0, SEEK_SET);
read(fd1, test_buffer, sizeof(test_buffer));
printf("fd1: %s\n", test_buffer);

lseek(dup_fd, 0, SEEK_SET);
read(dup_fd, test_buffer, sizeof(test_buffer));
printf("dup_fd: %s\n", test_buffer);
```

In order to test whether the duplicated file descriptor is pointing to the same offset as the original file descriptor, I first created the necessary variables for the test environment and opened the files. Then, I used the dup() function to duplicate the file and wrote the message 'test1' into fd1.

Next, I assigned the offset values of fd1 and dup_fd to variables and compared them with each other. If these two values are equal, it means that the offset values are equal. After this process, I printed these offset values to the screen using the printf() function.Then, using the dup_fd file descriptor, I moved to the beginning of the file and wrote the message 'test1' into it. Since the file1.txt pointed by fd_1 already had the value 'test1' written in it, we do not expect to see any change. Then, we move to the beginning of the file using the fd1 file descriptor, read the data in the file using the read() function, and store it in the test_buffer. We then repeat the same process using the dup_fd file descriptor. We see that we get the same result in both operations when we print these values.

```c
/* TEST WITH DUP2 FUNCTION*/
printf("\n---Test with dup2 function---\n\n");

fd2 = open("file_2.txt", O_RDWR | O_CREAT, 0644);
fd3 = open("file_3.txt", O_RDWR | O_CREAT, 0644);


write(fd2,"test2",5);
write(fd3,"test3",5);

dup2_fd = dup2(fd2,fd3);

off_t fd2_offset = lseek(fd2,0,SEEK_CUR);
off_t fd3_offset = lseek(fd3,0,SEEK_CUR);
off_t dup2_fd_offset = lseek(dup2_fd,0,SEEK_CUR);

if(fd2_offset == dup2_fd_offset)
{
    printf("fd2 and dup2_fd have the same file offset.\n");
    printf("fd2 offset-> %ld\nfd3 offset-> %ld\ndup2_fd offset -> %ld\n", fd2_offset,fd3_offset,dup2_fd_offset);
}
else
{
    printf("fd2 and dup2_fd have the different file offset.\n");
    printf("fd2 offset-> %ld\nfd3 offset-> %ld\ndup2_fd offset -> %ld\n", fd2_offset,fd3_offset,dup2_fd_offset);
}
```

```c
lseek(dup2_fd, 0, SEEK_SET);
write(dup2_fd, "test2", sizeof(test_buffer_dup2));


lseek(fd2, 0, SEEK_SET);
read(fd2, test_buffer_dup2, sizeof(test_buffer_dup2));
printf("fd2: %s\n", test_buffer_dup2);

lseek(fd3, 0, SEEK_SET);
read(fd3, test_buffer_dup2, sizeof(test_buffer_dup2));
printf("fd3: %s\n", test_buffer_dup2);


lseek(dup2_fd, 0, SEEK_SET);
read(dup2_fd, test_buffer_dup2, sizeof(test_buffer_dup2));
printf("dup2_fd: %s\n", test_buffer_dup2);
```

I also performed a very similar test using the dup2() function after using the dup() function.I finally close the files opened with the open() function and terminate the program.

```c
close(fd1);
close(fd2);
close(fd3);
close(dup2_fd);
close(dup_fd);
```

My Final Output For Part3