CSE344 – System Programming

Homework 1 – Report

Ömer Faruk Bitikçioğlu

161044010

1. To take command line arguments from the user, “int argc, char *argv[]” is added to the main function’s signature. The program should take 3 or 4 arguments and no other, so it is checking if the argument count is valid or not. If the argument count is invalid, the program prints the proper run command and exits with an error status.

   In a proper run of the program, argv[1] (second argument) is the file path, argv[2] (third argument) is the number of bytes to be appended, and if exists, argv[3] (fourth argument) should be x. These arguments are copied to the local variables. If the fourth argument (x) is given, omitAppendFlag is set to 1.

   If omitAppendFlag is set to 1, the file is opened without the O_APPEND flag. Before every write function calls lseek function is called to move the file descriptor to the end of the file. If omitAppendFlag is not set to 1, then the file is opened with the O_APPEND flag, and the write function is called directly.

   Also in both scenarios, the file is opened using O_WRONLY and O_CREAT flags. O_WRONLY flag indicates that the file is opened for only writing. O_CREAT flag is used to create the file if necessary.

   To test the program, firstly below command is used from the homework document:

   ./appendMeMore f1 1000000 & ./appendMeMore f1 1000000

   This command runs two appendMeMore processes simultaneously without the x argument. So each process will open the file with the O_APPEND flag and try to append 1 MB of data to the f1 file simultaneously. The file size of f1 should be 2 MB.

   Second, the below command is used from the homework document:

   ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x

   This command also runs two appendMeMore processes simultaneously but with an x argument this time. The processes will open the file without the O_APPEND flag, instead, they will use lseek to set the file descriptor to the end of the file. Then they call the write function to write a byte. The file size of f2 should also be 2 MB.

   Results of ls -l command:

   -rwxrwxrwx 1 root root 2000000 Mar 23 13:59 f1

   -rwxrwxrwx 1 root root 1992686 Mar 23 14:01 f2

   The size of the f2 is not as expected. This happened because of the way we opened the file. When opening a file with the O_APPEND flag, the system call write appends data to the end of the file atomically. This means that multiple processes can safely write to the file concurrently without any data being overwritten or intermingled. However, when using lseek to move the file position indicator to the end of the file before each write operation, it's possible for multiple processes to simultaneously execute the lseek and write functions in a way that data gets overwritten. For instance, one process may execute lseek and then be preempted by the operating system, allowing another process to execute the same lseek and start writing data at the same location, thus overwriting the data written by the first process. This is why f2 has fewer bytes than expected. Some of the bytes are overwritten by the other process.

2. In this part, my_dup() and my_dup2() are implementations of dup() and dup2() using fcntl() respectively.

   ERROR_FD and ERROR_2FD are macros defined to make the code cleaner and easy to read. They call perror with the given string and close the file descriptor(s), then return 1.

   In the implementation of my_dup(), first, it is checked if the given oldfd is a valid descriptor value by calling fcntl with the F_GETFL command. It should return the value of file status flags. But if it returns -1, then we can say that oldfd is not a valid file descriptor. If it is valid, then we duplicate the oldfd into newfd using fcntl's F_DUPFD command. The third argument is 0 because it will look for a valid file descriptor value greater than or equal to 0. It will reserve the first valid value. Then it returns the reserved newfd value. The error is not checked because if the newfd value is -1, then the error will be checked when using my_dup().

   Implementation of my_dup2 is similar to my_dup, except for some changes such as my_dup2 takes an extra argument which is newfd. The second argument indicates that it should find a valid file descriptor value equal to newfd. It is again checked if the oldfd is valid. Then, the special case when newfd is equal to oldfd is checked. If it happens, it simply returns the same value newfd without doing anything. If it is not the same, it checks if the newfd is open. If it is already open, it closes it. At last, it duplicates the oldfd into newfd using the value of the given newfd value. It should return the same value as newfd.

   The main function tests the implemented functions. If the defined macro TEST_MY_DUP is 1, the main program will test only the my_dup() function. If TEST_MY_DUP2 is set to 1, then only my_dup2 is going to be tested by the main program.

   It simply opens a file "file.txt" with proper flags. Writes "Hello from old_fd!" to the opened file. Then it duplicates the old_fd into new_fd using my_dup or my_dup2 according to the defined macros. If my_dup2 is going to be tested, the NEW_FD macro also should be determined at the start of the code. Also, old_fd can be changed for testing purposes by editing the OLD_FD macro.

   Next, it writes "Hello from new_fd!" to the same file using the duplicated file descriptor new_fd. It should append the line to the same file. Lseek is used to set the offset to the start of the file using new_fd. Then it reads the content of the file using new_fd into buf3. Finally prints the buf3 to the stdout. Both lines should be observed on stdout at the end of the process like below:

   ```
   Hello from old_fd!
   Hello from new_fd!
   ```

   This test approves that both file descriptors are accessing the same file and they share the same offset value.

3. In part 3, fcntl.h's dup function is used to demonstrate if the duplicated file descriptors share a file offset value. It is similar to what we achieved in part 2.

   BUFFER_SIZE macro is the size of the buffers used in the program.

It simply opens a file “test_offset_file.txt” with proper flags. Duplicates its file descriptor using dup(). Prints “Check this out!” to the file using the old file descriptor. Sets the file offset to the 4 before the end of the file which corresponds to the start of the word “out”. It writes “in” at the position of “out” in the file. Also, it replaces the ending characters “t!” with “!\n”. As a result, the line is changed from “Check this out!” to “Check this in!” in the file by using duplicated file descriptors. It is easy to see that both fd2 and fd1 share the same file offset. Because after using lseek with fd2 to 4 before the end of the file, the write function is used with fd1 and it wrote at the same position. After setting the offset start of the file with lseek by using fd2, the read function is used with fd1 and it read from the same position. These observations prove that both file descriptors fd1 and fd2 share the same file offset value and open file.