CSE 344 – Spring 2023

HW1 Report

GONCA EZGİ ÇAKIR – 151044054

1. Part 1

1.1. Solution Approach

According to task description implementation steps are listed below.

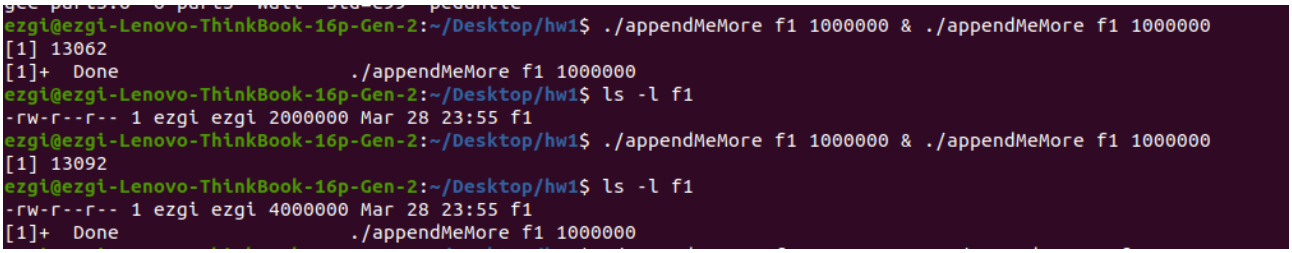
- Firstly get the console arguments by using **argv[ ]** variables to store them for further usage. If the console arguments are valid process continues otherwise returns -1 with an error message.
- **argv[3]** is checked in order to determine the user added “x” argument or not. If it is added, boolean value for appending end of the file is set to 0; otherwise it remains 1.
- A file opened with a name stored in **argv[1]**. If the file is not already exist; it is created, this is achieved by using **O_CREAT** flag in the **open** system call. If the boolean value for appending is 1, also **O_APPEND** flag is added to **open** system call.
- In a for loop given number of bytes with **argv[2]** is added to file by **write** system call. Before writing bytes into file boolean value is checked and if it is 0; **lseek** system call used in order to move to the end of the file.
- At the end file is closed.

In this implementation it is possible to receive a truncated file in some cases. Because for files which are opened without an **O_APPEND** flag, using **lseek** before **write** system call might lead to overwriting when there are more than one processes, because there a little time between processing these system calls. This case creates a race condition and might result with data corruption.

There are some test result in the next section, that explains this case.

1.2. Test Results

Command : ./appendMeMore f1 1000000 & ./appendMeMore f1 1000000



This case opens the file with the O_APPEND flag because there is no x argument at the end of the command. So writing into file with multiple processes doesn't harm any data. At the end as you can see size of the file is 1000000 + 1000000 = **2000000 bytes.**

Command : ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x

```
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
[1] 13105
[1]+  Done                    ./appendMeMore f2 1000000 x
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ls -l f2
-rw-r--r-- 1 ezgi ezgi 1631900 Mar 28 23:55 f2
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
[1] 13129
[1]+  Done                    ./appendMeMore f2 1000000 x
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ls -l f2
-rw-r--r-- 1 ezgi ezgi 3289297 Mar 28 23:55 f2
```

This case opens the file without the O_APPEND flag because there is a x argument at the end of the command. So writing into file with multiple processes possibly harms the data. At the end as you can see size of the file should be 2000000 bytes but it is **1631900 bytes** because of the race condition between write and lseek system calls.

Command : ./appendMeMore f3 1000000 & ./appendMeMore f3 1000000 x

```
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ./appendMeMore f3 1000000 x & ./appendMeMore f3 1000000
[1] 13140
[1]+  Done                    ./appendMeMore f3 1000000 x
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ls -l f3
-rw-r--r-- 1 ezgi ezgi 1916129 Mar 28 23:55 f3
```

This case is also similar to second case. It opens the file with the O_APPEND flag at the beginning but second process opens it without O_APPEND flag. So writing into file with the second process possibly harms the data. At the end as you can see size of the file should be 2000000 bytes but it is **1916129 bytes** because of the race condition between write and lseek system calls in the second process.

Also console arguments validity is checked for more or less arguments and 4th arguments correctness.

```
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ./appendMeMore f1
Usage: Console argument can't be less than 3.: Success
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ./appendMeMore f1 10000 a
Usage: Last console argument is invalid, try 'x'.: Success
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ./appendMeMore f1 10000 a b c
Usage: Console argument can't be more than 4.: Success
```

2. Part 2

2.1. Solution Approach

According to task description implementation steps are listed below.

- **dup(old_fd)** system call allocates a new file descriptor that refers to the same open file descriptor given as parameter. Then it returns the new file descriptor value, in success. So in order to do that **fcntl()** system call is used with **F_DUPFD** flag and **0** argument in order to create duplicated file descriptor with the lowest value.

```c
//creates the firstly avalible file descriptor
//returns the new_fd in successful case
//returns -1 in error case
int dup(int old_fd){
    int new_fd;

    //get new firstly avaliable file descriptor
    if((new_fd = fcntl (old_fd, F_DUPFD, 0)) == -1){
        perror("Failed while creating new_fd - dup");
        return -1;
    } else {
        return new_fd;
    }
}
```

- **dup2(old_fd, new_fd)** system call copies the old file descriptor to new file descriptor (if old file descriptor is valid), and also closes the new file descriptor beforehand if it is necessary. Then it returns the new file descriptor value, in success. **fcntl()** system call is used with **F_DUPFD** flag and **old_fd** as a argument in order to create copy file descriptor of **old_fd** onto **new_fd.**
- Before copying the file descriptor **new_fd** checked by **fcntl()** system call with **F_GETFL** flag in order to determine the file is opened. If it doesn't returns -1 this means that **new_fd** is already in use, then it is closed beforehand.
- Also special case is handled; which is occurs when the *old_fd and new_fd is equal.* If they are equal validity of the **old_fd** is checked by **fcntl()** system call with **F_GETFL** flag. If returns -1 it means this file descriptor is not valid. So in that case **errno** is set to **EBADF** and **dup2()** *returns -1*. Otherwise it returns **old_fd**.

```c
//duplicates the old_fd to new_fd
//returns the new_fd in successful case
//returns -1 in error case
int dup2(int old_fd, int new_fd){

    //special case
    if(old_fd == new_fd){

        //check that old_fd is valid
        //if it is not set the errno to EBADF
        if(fcntl (old_fd, F_GETFL) == -1){
            perror("Error case - errno is set to EBADF");
            errno = EBADF;
            return -1;
        }
        else {
            return new_fd;
        }
    }

    //check that if the new_fd is already open
    //if it is close it
    if(fcntl(new_fd, F_GETFL) != -1){
        close(new_fd);
    }

    //get a new_fd, by duplicating the old_fd
    if(fcntl (old_fd, F_DUPFD, new_fd) == -1){
        perror("Failed while duplicating old_fd - dup2");
        return -1;
    }

    return new_fd;

}
```

2.2. Test Results

Testing steps are listed below.

- A file is created with the name **testFilePart2.txt**. Then this file's descriptor fd1 is used for writing a string into file.
- To test **dup() fd1** is given as a parameter and new file descriptor **fd2** is created. Also **fd2** is used for writing a string into file in order to prove that these two file descriptors points the same file.
- To test **dup2()**, firstly **fd3** is created by using **dup()** with fd1 in order to assign the lowest possibly value to this file descriptor to *prevent any junk values.* Then **dup2()** is used for **copying fd2** value **into fd3** value. At the end **fd3** is used for writing a string into file in order to prove that these three file descriptors points the same file.

- To test error case, another f**d4** is created and set to *200 (invalid value)*. To provide this case requirements also **fd3** is set to *200*, by this we got two equal and unvalid file descriptors. In that case **dup2()** sets errno to **EBADF** and returns -1.

```
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ./part2
testFilePart2.txt is created. File descriptor is fd1:3
Written to testFilePart2.txt by fd1:3

File descriptor is created by dup(fd1). New file descriptor value fd2:4
Written to testFilePart2.txt by fd2:4

File descriptor is created by dup(fd2). New file descriptor value fd3:5
File descriptor value is changed by dup2(fd3,fd1). New file descriptor value fd3:3
Written to testFilePart2.txt by fd3:3

Error test
Error case - errno is set to EBADF: Bad file descriptor
```

3. Part 3

3.1. Solution Approach

According to task description it only includes testing.

3.2. Test Results

Testing steps are listed below.

- A file is created with the name **testFilePart3.txt**. Then this file's descriptor **fd** is used for writing "This is a test value." string (*21 characters)* into file.
- Then file descriptor **fd** is given to dup() and **dup_fd** created.
- **lseek()** system call is used with start 0 and *end SEEK_CUR* arguments to get the offset of the given file descriptor. Values are stored in **offset_fd** and **offset_dup_fd.**
- To determine these two file descriptors' open file points their inode values are checked. For this **struct stat variables** are created for both file descriptors as **inode_fd** and **inode_dup_fd.** Then those structs and file descriptors given to **fstat()** system call to be initialized.

- In the end **offset_fd** and **offset_dup_fd are** printed onto console. For inodes **inode_fd.st_ino** and **inode_dup_fd.st_ino are** printed to console as well.

We expect those two group of values to be equal in order to prove that file descriptors shows the same open file and has the same offset values. Also offset values should be equal to string length which is written into file at the beginning.

```
ezgi@ezgi-Lenovo-ThinkBook-16p-Gen-2:~/Desktop/hw1$ ./part3
testFilePart3.txt is created. Written by using file descriptor fd:3

New file descriptor is created by dup(fd). File descriptor value dup_fd:4

File descriptors offset and inode values to proof that they point to same file and offset.
offset fd -> offset:21  fd inode:3060419
offset dup_fd -> offset:21  dup_fd inode:3060419
```

4. Makefile

You can see makefile content down below. It only complies the codes with warning flags by “make” command and cleans .o files with “make clean” command.