CSE 344

Homework #1

REPORT

PART 1

```
int isXgivenToTheProgram = 0;

if(argc == 4){

    if(strcmp(argv[3], "x") == 0){
        isXgivenToTheProgram = 1;
    }
    else{
        printf("\nThe argument %s is not recognized.\n", argv[3]);
        exit(1);
    }

}
```

The 4th argument must be ‘x’ as it stated in the PDF. Therefore, firstly the validation for ‘x’ is done.

```
if(isXgivenToTheProgram == 0){
    int fd = open(argv[1], O_CREAT | O_WRONLY | O_APPEND, 0666);

    if(fd == -1){

        if(fd == -1)

        printf("Error while opening the file\n");
        exit(1);
    }

    int numberOfBytes = atoi(argv[2]);
    char* byte = (char *) malloc(1 * sizeof(char));

    for(int i = 0; i < numberOfBytes; i++){
        if(write(fd,byte,1)== -1){
            printf("Error while writing into the file\n");
            exit(1);
        }
    }


    if(close(fd) == -1){
        printf("Error while closing the file\n");
        exit(1);
    }
}
```

Then for the command without ‘x’ is done as it shown above. File is opened with O_APPEND and 0666 mode (I have added this because the second process could not reach the file due to administrative requirements.). Inside the loop, 1 byte is written into the file.

```
else{
    int fd = open(argv[1], O_CREAT | O_WRONLY, 0666);

    if(fd == -1){
        printf("Error while opening the file\n");
        exit(1);
    }
    int numberOfBytes = atoi(argv[2]);
    char* byte = (char *) malloc(1 * sizeof(char));

    for(int i = 0; i < numberOfBytes; i++){
        lseek(fd,0,SEEK_END);
        if(write(fd,byte,1)== -1){
            printf("Error while writing into the file\n");
            exit(1);
        }
    }

    if(close(fd) == -1){
        printf("Error while closing the file\n");
        exit(1);
    }
}
```

For the command with ‘x’, the file is opened without O_APPEND. Before every write operation, lseek() is performed.

```
-rwxrwxr-x 1 ardakilic ardakilic   16424 Mar 26 17:41 appendMeMore
-rw-rw-r-- 1 ardakilic ardakilic    2664 Mar 26 17:20 appendMeMore.c
-rw-rw-r-- 1 ardakilic ardakilic 2000000 Mar 26 17:41 f1
-rw-rw-r-- 1 ardakilic ardakilic 1916488 Mar 26 17:42 f2
-rwxrwxr-x 1 ardakilic ardakilic   16344 Mar 26 17:41 fileDescriptorDuplicate
-rw-rw-r-- 1 ardakilic ardakilic    2752 Mar 26 16:39 fileDescriptorDuplicate.c
-rw-rw-r-- 1 ardakilic ardakilic   34810 Mar 23 11:52 HW1.pdf
-rw-rw-r-- 1 ardakilic ardakilic     348 Mar 26 17:40 Makefile
-rwxrwxr-x 1 ardakilic ardakilic   16360 Mar 26 17:41 verifyFileDescriptors
-rw-rw-r-- 1 ardakilic ardakilic    2126 Mar 26 17:04 verifyFileDescriptors.c
```

The size of the files (f1 and f2) is shown above. The same commands run as it stated in the PDF. There are 83512 bytes difference between these 2 files. The reason for this difference is that lseek() is not safe to use for multiprocessing. If two processes use lseek() to change the file offset to the same location in the file and then perform a write operation, the data written by one process may overwrite the data written by the other process. This causes data corruption.

PART 2

```c
int myDup(int oldfd){
    return fcntl(oldfd, F_DUPFD);
}
```

The dup() method is implemented as shown above. This method just gets an old file descriptor and returns the new duplicated file descriptor.

```c
int myDup2(int oldfd, int newfd){

    if(oldfd == newfd){
        /* When oldfd is not a valid file descriptor */
        if(fcntl(oldfd,F_GETFL) == -1){
            errno = EBADF;
            return -1;
        }
        return newfd;
        //return fcntl(oldfd, F_DUPFD, newfd);
    }

    else{
        /* When oldfd is not a valid file descriptor */
        if(fcntl(oldfd,F_GETFL) == -1){
            errno = EBADF;
            return -1;
        }
        /* When newfd was previously open */
        if(fcntl(newfd,F_GETFL) != -1){
            close(newfd);
        }

        fcntl(oldfd, F_DUPFD, newfd);
        return newfd;

    }

}
```

The dup2() method is implemented as shown above. There were different cases to handle. Firstly, if oldfd is equal to the newfd, I had to check the validity of the oldfd. If oldfd is valid, then newfd will be returned. If oldfd is not valid, then errno is set to EBADF and –1 is returned.

Secondly, if oldfd and newfd are not equal, I checked the validity of oldfd. The same procedure is applied if oldfd is not valid. For the other case where newfd is previously open, the newfd is closed and by using fcntl(), newfd is set to oldfd and returned.

```c
char * buffer_fd = "This line comes from fd\n";
char * buffer_fd_dup = "This line comes from dup()\n";
char * buffer_fd_dup2 = "This line comes from dup2()\n";

/* TEST 1 - dup()  */
printf("\n\tTEST 1 - dup()\n");
int fd_test1 = open("part2_test1.txt", O_CREAT | O_WRONLY, 0666);
int test1 = myDup(fd_test1);
printf("fd is %d, the fd returned by dup() is %d.\n",fd_test1,test1);
write(fd_test1, buffer_fd, strlen(buffer_fd));
write(test1, buffer_fd_dup, strlen(buffer_fd_dup));

/* TEST 2 - dup2(), when oldfd is invalid */
printf("\n\tTEST 2 - dup2(), when oldfd is invalid\n");
int fd_test2 = open("part2_test.txt", O_CREAT | O_WRONLY, 0666);
int test2 = myDup2(500,fd_test2);
printf("fd is %d, the fd returned by dup2() is %d.\n",fd_test2,test2);

/* TEST 3 - dup2(), when oldfd and newfd are equal (oldfd is not valid) */
printf("\n\tTEST 3 - dup2(), when oldfd and newfd are equal (oldfd is not valid)\n");
int fd_test3 = open("part2_test.txt", O_CREAT | O_WRONLY, 0666);
int test3 = myDup2(500,500);
printf("fd is %d, the fd returned by dup2() is %d.\n",fd_test3,test3);

/* TEST 4 - dup2(), when oldfd and newfd are equal (oldfd is valid) */
printf("\n\tTEST 4 - dup2(), when oldfd and newfd are equal (oldfd is valid)\n");
int fd_test4 = open("part2_test4.txt", O_CREAT | O_WRONLY, 0666);
int test4 = myDup2(fd_test4,fd_test4);
printf("fd is %d, the fd returned by dup2() is %d.\n",fd_test4,test4);
write(fd_test4, buffer_fd, strlen(buffer_fd));
write(test4, buffer_fd_dup2, strlen(buffer_fd_dup2));

/* TEST 5 - dup2(), when newfd is valid */
printf("\n\tTEST 5 - dup2(), when newfd is valid\n");
int fd_test5 = open("part2_test.txt", O_CREAT | O_WRONLY, 0666);
int test5 = myDup2(fd_test5,fd_test5);
printf("fd is %d, the fd returned by dup2() is %d.\n",fd_test5,test5);
```

I have run 5 tests for both dup() and dup2(). Whether to check if the file descriptors point to the same file, I used write() operation to see the output in that file.

Part 3

```c
int verifyDuplicatedFileDescriptor(int fd1 , int fd2){
    struct stat stat1, stat2;
    if(fstat(fd1, &stat1) < 0) return -1;
    if(fstat(fd2, &stat2) < 0) return -1;
    int isSameFile = (stat1.st_dev == stat2.st_dev) && (stat1.st_ino == stat2.st_ino);
    if((ftell(fdopen(fd1, "a")) == ftell(fdopen(fd2, "a"))) && (isSameFile)){
        return 1;
    }

    return -1;

}
```

In this part, a verification function is requested for the duplicated file descriptors. There were 2 parts for the verification. Firstly, I had to check if the file descriptors point to the same file. Secondly, if the file descriptors point to the same offset of the file. To check the first, I have used fstat from <sys/stat.h>. Fstat gives the information of the file that I requested. These 2 files are checked by their st_dev and

st_ino values. If both are equal, then the file descriptors point to the same file. To check the second, I have used ftell() to get the position of the offset of that file. What ftell() does shown in the below.

```
The ftell() function shall obtain the current value of the file-
position indicator for the stream pointed to by stream.
```

ftell() is implemented in <stdio.h>. It needs FILE * as it's parameter, therefore by using fdopen(), file descriptor is converted to a FILE pointer.

If both these values are equal, then 1 is returned. If not, then –1 is returned.

```c
/* TEST 1 - Same file, different offset */
printf("\n\tTEST 1 - Same file, offset is set using lseek differently. (Second file descriptor created using dup()\n");
int test1_1 = open("part3_a.txt", O_CREAT | O_WRONLY, 0666);
int test1_2 = dup(test1_1);
lseek(test1_2,10,SEEK_SET);
lseek(test1_1,5,SEEK_CUR);
printf("Is duplicated file descriptors share = %d\n",verifyDuplicatedFileDescriptor(test1_1, test1_2));

/* TEST 2 - Different file, same offset */
printf("\n\tTEST 2 - Different file, same offset\n");
int test2_1 = open("part3_a.txt", O_CREAT | O_WRONLY, 0666);
int test2_2 = open("part3_b.txt", O_WRONLY, 0666);
lseek(test2_1,0,SEEK_SET);
lseek(test2_2,0,SEEK_SET);
printf("Is duplicated file descriptors share = %d\n",verifyDuplicatedFileDescriptor(test2_1, test2_2));

/* TEST 3 - Same file, same offset */
printf("\n\tTEST 3 - Same file, same offset\n");
int test3_1 = open("part3_a.txt", O_CREAT | O_WRONLY, 0666);
int test3_2 = dup(test3_1);
lseek(test3_1,0,SEEK_SET);
lseek(test3_2,0,SEEK_SET);
printf("Is duplicated file descriptors share = %d\n",verifyDuplicatedFileDescriptor(test3_1, test3_2));

/* TEST 4 - Different file, different offset */
printf("\n\tTEST 4 - Different file, different offset\n");
int test4_1 = open("part3_a.txt", O_CREAT | O_WRONLY, 0666);
int test4_2 = open("part3_b.txt", O_CREAT | O_WRONLY, 0666);
lseek(test4_1,5,SEEK_SET);
lseek(test4_2,10,SEEK_CUR);
printf("Is duplicated file descriptors share = %d\n",verifyDuplicatedFileDescriptor(test4_1, test4_2));
```

4 different tests are run for this part.