

GEBZE
TEKNİK ÜNİVERSİTESİ



SPRING 2023 – CSE 344 SYSTEM PROGRAMMING

HOMEWORK 1

MERT GÜRŞİMŞİR

1901042646

How to run?

First you should use make command.

Then you are going to have 2 executable files which are “appendMeMore” and “program”.

appendMeMore is the program for the first question. You can use it like follows:

```
./appendMeMore f1 1000000
```

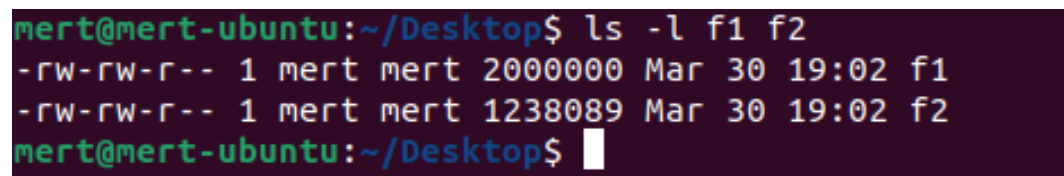
```
./appendMeMore f2 1000000 x
```

program is the test for the implementation of 2nd question. You can use it like follows:

```
./program
```

After that, you can use “make clean” to delete .o files and 2 executables.

QUESTION 1



```
mert@mert-ubuntu:~/Desktop$ ls -l f1 f2
-rw-rw-r-- 1 mert mert 2000000 Mar 30 19:02 f1
-rw-rw-r-- 1 mert mert 1238089 Mar 30 19:02 f2
mert@mert-ubuntu:~/Desktop$
```

As you can see, f1 has exactly 2 million bytes.

If we don't use x, O_APPEND flag is going to be used. It means that each write() operation will append to the end of file. After running command:

```
./appendMeMore f1 1000000 & ./appendMeMore f1 1000000
```

the file will contain 2 million bytes exactly because each command will append its own bytes to the end of the file, resulting in a larger file. These writes do not interfere because with “O_APPEND” flag, regardless of the current file position, new byte is always going to be added to the end. Therefore resulting file size is foreseeable.

BUT, if we run this command:

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

then file size will not be foreseeable since each process assumes that file pointer is at the end of the file because they use “lseek(fd, 0, SEEK_END)” which means next byte after the end of the file, but since they are working simultaneously, we cannot guarantee that pointer will be end of the file for both of them. As a result, they might overwrite each other's writes. In this case, file size will be smaller than 2 million bytes. If we do this, it is good idea to use synchronization like locks. Even if there is only 1 CPU, processes share it and operating system does the scheduling. Here, 2 processes are competing for file position. For example, if they both trying to call “lseek(fd, 0, SEEK_END)”, they may get the same file position because operating system (kernel) does not have time to change the file position between these two calls.

QUESTION 2

dup() uses next available space for the new file descriptor. Returned descriptor can be used interchangeably with the original file descriptor that is sent to dup function via parameter.

dup2() uses descriptor that it gets via parameter to duplicate the other given parameter as old file descriptor. Returned descriptor can be used interchangeably with that old file descriptor.

I have used a validation function to identify if given file descriptor is valid or not. To do this, I have used fcntl with F_GETFL flag as it is mentioned in the homework pdf:

```
int validation(int fd){
    int r = fcntl(fd, F_GETFL);
    if (r < 0){
        errno = EBADF;
        return 0;
    }
    return 1;
}
```

For dup, I have first validate the given descriptor. If it is valid, then I have used fcntl with the F_DUPFD flag which duplicates the file descriptor as we learnt during file I/O lecture:

```
int dup(int oldfd){
    int newfd;
    printf("\n-----oldfd is duplicating (dup)-----\n\n");
    if (validation(oldfd)){
        newfd = fcntl(oldfd, F_DUPFD, 0);
        return newfd;
    }
    else return -1;
}
```

For dup2, I have first validate the old file descriptor again. Then as it is mentioned in the pdf, I looked if oldfd and newfd are pointing to same place. If so, I do nothing and simply return newfd. If not, I looked if newfd already exists. If it exists, I close it. Then I used fcntl again with the F_DUPFD flag but this time I gave third parameter as newfd instead of 0 because now I want my descriptor to be lowest-numbered available file descriptor (which will be newfd because it is now available) starting from newfd:

```
int dup2(int oldfd, int newfd){
    printf("\n-----oldfd is duplicating into newfd (dup2)-----\n\n");
    if (validation(oldfd)){
        if (oldfd != newfd){
            if (fcntl(newfd, F_GETFD) >= 0) close(newfd);
            return fcntl(oldfd, F_DUPFD, newfd);
        }
        else return newfd;
    }
    else return -1;
}
```

QUESTION 3

TEST 1

For the first test, I have first open a test.txt file and duplicate it. Then I check file descriptor (fd) and duplicated new file descriptor (dupfd). As 0, 1, 2 are filled with standard input, standard output, and standard error, I expected fd to be 3 and dupfd to be 4 which is next available position. Also I have checked their file status flags:

```
fd = open("test.txt", O_RDWR | O_CREAT, 0666);
dupfd = dup(fd);
printf("fd original          --> %d\nduplicated fd          --> %d\n", fd, dupfd);
printf("original fd status flags --> %d\nduplicated fd status flags --> %d\n", fcntl(fd, F_GETFL), fcntl(dupfd, F_GETFL, 0));
```

```
mert@mert-ubuntu:~/Desktop$ ./program
TEST 1: test.txt file is going to be created (if doesn't exist) and its descriptor will be duplicated into next available place.

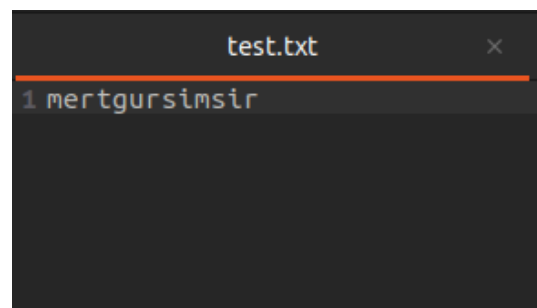
-----oldfd is duplicating (dup)-----

fd original          --> 3
duplicated fd         --> 4

original fd status flags --> 32770
duplicated fd status flags --> 32770
```

Then I do these step by step:

- write "mert" to test.txt with **fd** → 4 bytes
- write "gursimsir" to test.txt with **dupfd** → 9 bytes
- put pointer to beginning
- read 4 bytes with **dupfd**
- read 9 bytes with **fd**



```
printf("Now write mert to original fd, gursimsir to duplicated fd, seek to beginning, read 4 bytes with duplicated fd, read 9 bytes with original fd.\nREAD DATA --> ");
write(fd, "mert", 4);
write(dupfd, "gursimsir", 9);
lseek(fd, 0, SEEK_SET);
read_bytes = read(dupfd, buf1, sizeof(buf1));
for (i = 0; i < read_bytes; ++i) printf("%c", buf1[i]);
read_bytes = read(fd, buf2, sizeof(buf2));
for (i = 0; i < read_bytes; ++i) printf("%c", buf2[i]);
printf("\n\n");
```

```
Now write mert to original fd, gursimsir to duplicated fd, seek to beginning, read 4 bytes with duplicated fd, read 9 bytes with original fd.
READ DATA --> mertgursimsir
```

Lastly, I have make offset of fd to be 42. Then I assign pos2 variable to current position of dupfd and expect it to be 42:

```
printf("Now set file offset to 42 in original fd and check file offset position in duplicated fd.\n");
pos1 = lseek(fd, 42, SEEK_SET);
pos2 = lseek(dupfd, 0, SEEK_CUR);
printf("Offset of duplicated file: %d\n", pos2);
```

```
Now set file offset to 42 in original fd and check file offset position in duplicated fd.
Offset of duplicated file: 42
```

TEST 2

For this test, I have do the same things as test 1 but this time, I have duplicated the file descriptor with dup2 to position 2 next to original file descriptor. So this time, instead of 4, I have expected duplicated file descriptor to be 5 because original file descriptor is 3:

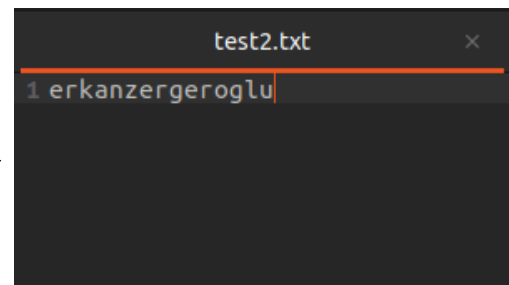
```
fd2 = open("test2.txt", O_RDWR | O_CREAT, 0666);
dupfd2 = dup2(fd2, fd2+2);
printf("fd original          --> %d\n", fd2);
printf("original fd status flags --> %d\n", fcntl(fd2, F_GETFL, fcntl(dupfd2, F_GETFL, 0)));
```

```
=====
TEST 2: test2.txt file is going to be created (if doesn't exist) and its descriptor will be duplicated into 2 next place.
-----oldfd is duplicating into newfd (dup2)-----
fd original          --> 3
duplicated fd        --> 5
original fd status flags --> 32770
duplicated fd status flags --> 32770
```

PS: After each file opening, I am controlling if open returned -1 but I didn't show it here for readability. You can check the code.

Then I do these step by step:

- write "erkan" to test.txt with **fd2** → 5 bytes
- write "zengeroglu" to test.txt with **dupfd2** → 10 bytes
- put pointer to beginning
- read 5 bytes with **dupfd2**
- read 10 bytes with **fd2**



```
printf("Now write erkan to original fd, zengeroglu to duplicated fd, seek to beginning, read 5 bytes with duplicated fd, read 10 bytes with original fd.\nREAD DATA --> ")
write(fd2, "erkan", 5);
write(dupfd2, "zengeroglu", 10);
lseek(fd2, 0, SEEK_SET);
read_bytes = read(dupfd2, buf3, sizeof(buf3));
for (i = 0; i < read_bytes; ++i) printf("%c", buf3[i]);
read_bytes = read(fd2, buf4, sizeof(buf4));
for (i = 0; i < read_bytes; ++i) printf("%c", buf4[i]);
printf("\n\n");
```

```
Now write erkan to original fd, zengeroglu to duplicated fd, seek to beginning, read 5 bytes with duplicated fd, read 10 bytes with original fd.
READ DATA --> erkanzengeroglu
```

Lastly, I have make offset of fd2 to be 42. Then I assign pos2 variable to current position of dupfd2 and expect it to be 42:

```
printf("Now set file offset to 42 in original fd and check file offset position in duplicated fd.\n");
pos1 = lseek(fd2, 42, SEEK_SET);
pos2 = lseek(dupfd2, 0, SEEK_CUR);
printf("Offset of duplicated file: %d\n", pos2);
```

```
Now set file offset to 42 in original fd and check file offset position in duplicated fd.
Offset of duplicated file: 42
```

TEST 3

For this test, I have created a file pointer as usual for test3.txt. Then I have sent it to dup2 for both of the parameters. I have expected the returned descriptor to be same as fd3's descriptor and I have expected their status flags to be same:

```
fd3 = open("test3.txt", O_RDWR | O_CREAT, 0666);
dupfd3 = dup2(fd2, fd2);
printf("fd original          --> %d\nduplicated fd          --> %d\n", fd3, dupfd3);
printf("original fd status flags --> %d\nduplicated fd status flags --> %d\n", fcntl(fd3, F_GETFL), fcntl(dupfd3, F_GETFL, 0));
```

```
=====
TEST 3: Now let's try sending the same file descriptor of test3.txt to dup2 and see their descriptor values.
----oldfd is duplicating into newfd (dup2)----
fd original          --> 3
duplicated fd        --> 3
original fd status flags --> 32770
duplicated fd status flags --> 32770
=====
```

TEST 4

For this, I have sent invalid descriptor values to dup2 and see the response of it. You can also send, for example, 10 which has no file:

```
printf("Descriptor we got from dup2 with invalid descriptor parameters: %d\n", dup2(-1, -1));
printf("errno: %s\n", strerror(errno));
```

```
=====
TEST 4: Now let's try sending the invalid file descriptor as oldfd.
----oldfd is duplicating into newfd (dup2)----
Descriptor we got from dup2 with invalid descriptor parameters: -1
errno: Bad file descriptor
=====
```

TEST 5

For this test, I have tried something different. Descriptor 1 is standard output. I have tried to duplicate it to dupfd4 with dup. Then I expected dupfd4 to be 3 which is next available place. Then I have written "HELLO!" to the dupfd4 and I can see that at the screen because dupfd4 is duplicated version of standard output:

```
dupfd4 = dup(1);
printf("Write \"HELLO!\" to descriptor we got from dup(1).\nDuplicated descriptor value: %d\n", dupfd4);
write(dupfd4, "HELLO!\n", 7);
```

```
=====
TEST 5: Duplicating the standard output --> descriptor 1.
----oldfd is duplicating (dup)----
Write "HELLO!" to descriptor we got from dup(1).
Duplicated descriptor value: 3
HELLO!
=====
```

ALL TESTS' RESULTS

```
mert@mert-ubuntu:~/Desktop$ ./program
TEST 1: test.txt file is going to be created (if doesn't exist) and its descriptor will be duplicated into next available place.

-----oldfd is duplicating (dup)-----

fd original          --> 3
duplicated fd         --> 4

original fd status flags --> 32770
duplicated fd status flags --> 32770

Now write mert to original fd, gursimsir to duplicated fd, seek to beginning, read 4 bytes with duplicated fd, read 9 bytes with original fd.
READ DATA --> mertgursimsir

Now set file offset to 42 in original fd and check file offset position in duplicated fd.
Offset of duplicated file: 42

=====
```

```
=====
TEST 2: test2.txt file is going to be created (if doesn't exist) and its descriptor will be duplicated into 2 next place.

-----oldfd is duplicating into newfd (dup2)-----

fd original          --> 3
duplicated fd         --> 5

original fd status flags --> 32770
duplicated fd status flags --> 32770

Now write erkan to original fd, zergeroglu to duplicated fd, seek to beginning, read 5 bytes with duplicated fd, read 10 bytes with original fd.
READ DATA --> erkanzergeroglu

Now set file offset to 42 in original fd and check file offset position in duplicated fd.
Offset of duplicated file: 42

=====
```

```
=====
TEST 3: Now let's try sending the same file descriptor of test3.txt to dup2 and see their descriptor values.

-----oldfd is duplicating into newfd (dup2)-----

fd original          --> 3
duplicated fd         --> 3

original fd status flags --> 32770
duplicated fd status flags --> 32770

=====
```

```
=====
TEST 4: Now let's try sending the invalid file descriptor as oldfd.

-----oldfd is duplicating into newfd (dup2)-----

Descriptor we got from dup2 with invalid descriptor parameters: -1
errno: Bad file descriptor

=====
```

```
=====
TEST 5: Duplicating the standard output --> descriptor 1.

-----oldfd is duplicating (dup)-----

Write "HELLO!" to descriptor we got from dup(1).
Duplicated descriptor value: 3
HELLO!
mert@mert-ubuntu:~/Desktop$
```