

GTU Department of Computer Engineering
CSE 344 - Spring 2022
Homework 1 Report

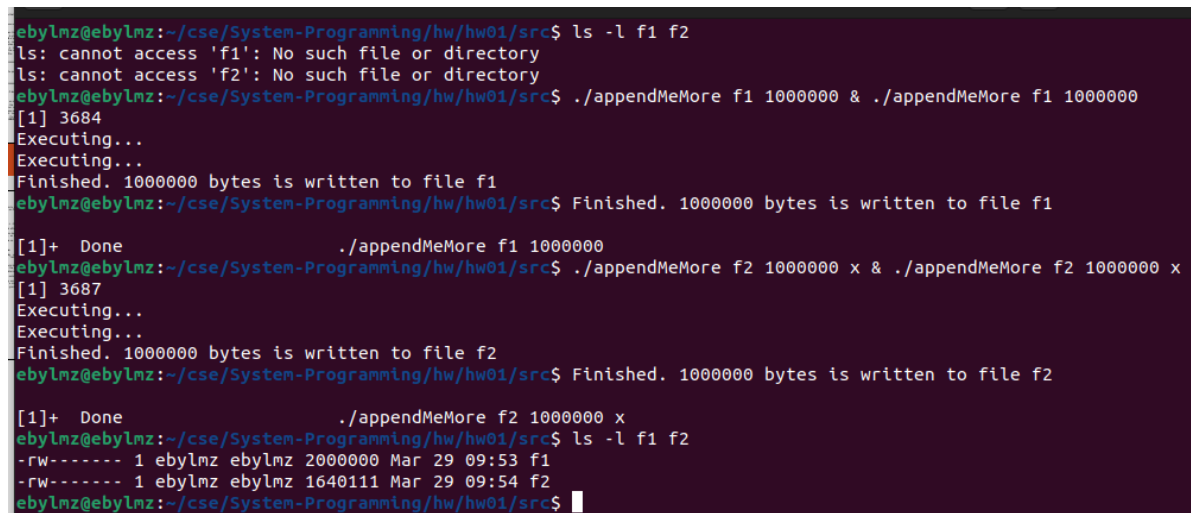
Emirkan Burak Yılmaz
1901042659

1 appendMeMore.c

1.1 Solution Approach

- First make sure that valid command line arguments are entered. In case of any usage error notify the user and terminate the execution.
- Set the file flags. If a third command-line argument (x) is not supplied, then the file should be open with O_APPEND flag.
- Open the file.
- Inside a loop write a byte at a time until specified number of bytes are written. If the third command-line argument (x) is supplied, then apply lseek(fd, 0, SEEK_END) before each write to handle multiple process/thread scenario in a way.
- Close the file.
- Use ls -l command to check the size of the file.

1.2 Why size of two file is different?

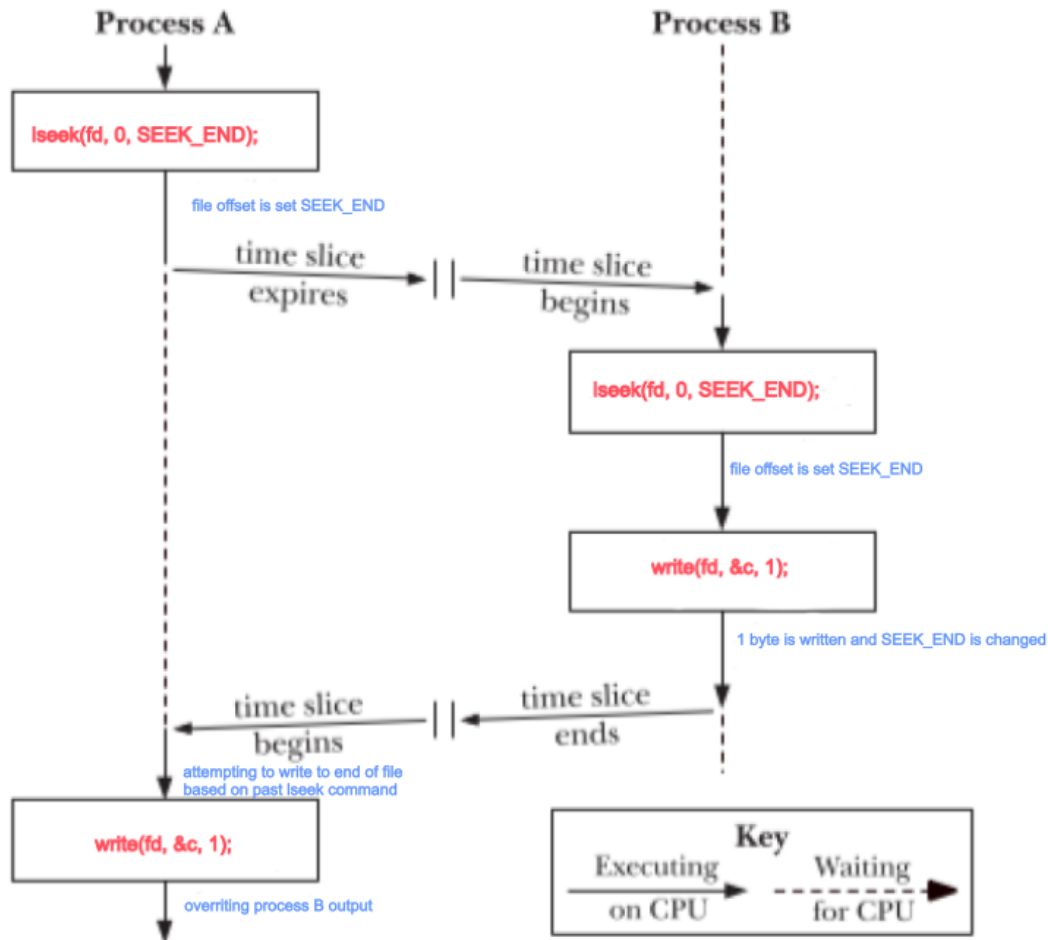


```
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l f1 f2
ls: cannot access 'f1': No such file or directory
ls: cannot access 'f2': No such file or directory
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore f1 1000000 & ./appendMeMore f1 1000000
[1] 3684
Executing...
Executing...
Finished. 1000000 bytes is written to file f1
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ Finished. 1000000 bytes is written to file f1

[1]+ Done ./appendMeMore f1 1000000
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
[1] 3687
Executing...
Executing...
Finished. 1000000 bytes is written to file f2
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ Finished. 1000000 bytes is written to file f2

[1]+ Done ./appendMeMore f2 1000000 x
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l f1 f2
-rw----- 1 ebylmz ebylmz 2000000 Mar 29 09:53 f1
-rw----- 1 ebylmz ebylmz 1640111 Mar 29 09:54 f2
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$
```

As it can be seen from the screenshot, size of the files f1 and f2 are different even if they both executing with the same intend files. The reason of difference is appendMeMore without third argument x, apply file operations atomically. Because it is opened with **O_APPEND** flag which guarantees the **atomicity** of the relevant file operations. So, the file offset is set to the end of the file prior to each write and **no intervening file modification operation** occurs between changing the file offset and the write operation. On the other hand, a file which is opened without O_APPEND flag, file operations can be interrupted, and some undesired conditions can be happen, in our case it is overriding their outputs. Because of those overrides size of f2 is less than 2 million bytes.



As seen in the diagram above process A is suspended after lseek command and then process B starts to run. It sets the offset to SEEK_END and writes 1 byte (since they executed in loop, multiple lseek and write can be executed). When process B is suspended and process A starts to run again, it overrides the process B output since it's offset is not equal the latest SEEK_END which is changed by process B. To overcome this kind of scenarios, O_APPEND flag should be used and with this way relevant file operations will be done atomically.

1.3 Test Cases & Results

- Wrong command line arguments.

```

ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore
Right usage: ./appendMeMore filename num-bytes [x]
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore test.txt
Right usage: ./appendMeMore filename num-bytes [x]
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore test.txt xyz
Right usage: ./appendMeMore filename num-bytes [x]
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l test.txt
ls: cannot access 'test.txt': No such file or directory
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore test.txt xyz 1000
Right usage: ./appendMeMore filename num-bytes [x]
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore test.txt 1000
Executing...
Finished. 1000 bytes is written to file test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l test.txt
-rw-r--r-- 1 ebylmz ebylmz 1000 Mar 29 10:32 test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$
  
```

- Append without third argument x.
- Append with third argument x.
- Run two instances of this program at the same time without the x argument to write 1 million bytes to the same file.
- Run two instances of this program at the same time with the x argument to write 1 million bytes to the same file.

```

ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l test.txt
-rw-r--r-- 1 ebylmz ebylmz 1000 Mar 29 10:38 test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore test.txt 1000
Executing...
Finished. 1000 bytes is written to file test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l test.txt
-rw-r--r-- 1 ebylmz ebylmz 2000 Mar 29 10:38 test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore test.txt 1000 x
Executing...
Finished. 1000 bytes is written to file test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l test.txt
-rw-r--r-- 1 ebylmz ebylmz 3000 Mar 29 10:38 test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore test.txt 500 & ./appendMeMore test.txt 500
[1] 4554
Executing...
Executing...
Finished. 500 bytes is written to file test.txt
Finished. 500 bytes is written to file test.txt
[1]+ Done ./appendMeMore test.txt 500
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l test.txt
-rw-r--r-- 1 ebylmz ebylmz 4000 Mar 29 10:38 test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore test.txt 500 x & ./appendMeMore test.txt 500 x
[1] 4560
Executing...
Executing...
Finished. 500 bytes is written to file test.txt
Finished. 500 bytes is written to file test.txt
[1]+ Done ./appendMeMore test.txt 500 x
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l test.txt
-rw-r--r-- 1 ebylmz ebylmz 4996 Mar 29 10:39 test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./appendMeMore test.txt 500 x & ./appendMeMore test.txt 500 x
[1] 4586
Executing...
Executing...
Finished. 500 bytes is written to file test.txt
Finished. 500 bytes is written to file test.txt
[1]+ Done ./appendMeMore test.txt 500 x
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ls -l test.txt
-rw-r--r-- 1 ebylmz ebylmz 5770 Mar 29 10:39 test.txt
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$

```

2 dup.c

2.1 Solution Approach

dup and dup2 can be implemented by using function fcntl. It takes three arguments which are file descriptor, command, and additional argument. fcntl performs certain operations based on the given command. In our case F_DUPFD command is used for duplicating given file descriptor. For learning behavior of these two functions, man pages of dup and dup2 are used.

dup(int oldfd);

- Implementation of dup is very straightforward. First make sure that oldfd is valid. To check if a file descriptor is valid or not, use fcntl with F_GETFL command to get file information's. It returns -1, when the file descriptor is not found (invalid fd) in process open file table. Otherwise, it is valid and can be duplicated.
- If oldfd is not valid, then dup sets errno to EBADF and returns -1. Otherwise, then dup calls fcntl with oldfd and F_DUPFD command and returns the return value.

dup2(int oldfd, int newfd);

- When we look at dup2, it is slightly different from dup. The difference is that dup2 duplicates the oldfd as newfd which is supplied as function argument rather than the lowest possible fd. So again, first make sure oldfd is valid for duplicate. For the case of newfd and oldfd are equal just return the newfd (before this step we are sure the validity of oldfd). If it is not the case, check if newfd is open. If so, close it to open new file descriptor. Finally, call fcntl with F_DUPFD command and return the newfd.

2.2 Test Cases & Results

dup

- Valid oldfd.
- Invalid oldfd.

```
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./dup
TEST: dup
-----

test case: oldfd oldfd is valid.
opening file test.txt...
3 was taken as file descriptor for file test.txt, keep it inside oldfd
oldfd: 3
call dup(oldfd)
duplicating...
dup returned 4

test case: oldfd is not valid.
closing file test.txt...
now oldfd is invalid
call dup(oldfd)
duplicating...
dup returned -1
```

dup2

- Valid oldfd, oldfd and newfd are different.
- Valid oldfd, oldfd and newfd are same.
- newfd is already open.
- Invalid oldfd.

```
TEST: dup2
-----

test case: oldfd and newfd are different and oldfd is valid.
opening file test1.txt...
3 was taken as file descriptor for file test1.txt, keep it inside oldfd
let newfd become 7
oldfd: 3, newfd: 7
call dup2(oldfd, newfd)
duplicating...
dup2 returned 7

test case: oldfd and newfd are same and oldfd is valid.
let newfd become same as oldfd
oldfd: 3, newfd: 3
call dup2(oldfd, newfd)
duplicating...
oldfd and newfd have the same value, dup2() does nothing and returns newfd
dup2 returned 3
```

(continues)

```
test case: newfd is already open.
opening file test2.txt...
5 was taken as file descriptor for file test2.txt, keep it inside newfd
oldfd: 3, newfd: 5
call dup2(oldfd, newfd)
duplicating...
File descriptor 5 was previously open, it is closed before being reused
Closing...
Closed
dup2 returned 5

test case: oldfd is not valid.
closing file test1.txt...
now oldfd is invalid
call dup2(oldfd, newfd)
duplicating...
oldfd is not valid, dup2() sets errno EBADF and returns -1
dup2 returned -1

ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$
```

3 verify_dup.c

3.1 Solution Approach

To verify duplicated file descriptors, share a file offset value and open file.

- Open a file (test.txt in our case) and keep the file descriptor value in int variable fd1.
- Duplicate fd1 with dup and keep the file descriptor value in variable fd2.
- Verify that the offset of the fd1 and fd2 are equal by using lseek.
- For each file descriptor make write operations.
- Verify that the offset of the fd1 and fd2 are equal by using lseek. Since their beginning offset was same and after two write operation their offset is same. We can say duplicated file descriptors share a file offset value and open file.
- We know that each file has unique inode. By using this information, we can figure out if two file descriptors refer the same file or not by just comparing their inodes. This will be more accurate verification compared to making some write read operations. For this, we can get all the information about file including its inode using function fcntl with F_GETFL command. The function call will end up with return value which is a structure type stat and inode information stored in st_ino field. When we compare these two numbers for duplicated file descriptors, it turns out that their inodes are same. So, with this way we verify that they share the same open file.

3.2 Program Output

```
ebylmz@ebylmz:~/cse/System-Programming/hw/hw01/src$ ./verify_dup
Open file test1.txt...
3 was taken as file descriptor for file test1.txt, and it became first fd
call dup(fd1)
duplicating...
dup returned 4, and it became second fd

Offset for fd 3: 0
Offset for fd 4: 0
Write operations are performing...
Offset for fd 3: 30
Offset for fd 4: 30
Two file descriptor shares the same offset value

Flags for fd 3: 33794
Flags for fd 4: 33794
Two file descriptor shares the same status flags

inode for fd 3: 918171
inode for fd 4: 918171
Two file descriptor share the same file, since both use the same inode

Detailed information about file(s) for two file descriptor:
Information for File Descriptor 3
-----
File Size:          30 bytes
Number of Links:    1
File inode:         918171
File Permissions:   -rw-----
This file is not a symbolic link

Information for File Descriptor 4
-----
File Size:          30 bytes
Number of Links:    1
File inode:         918171
File Permissions:   -rw-----
This file is not a symbolic link
```

(Additional information's like file size, file permissions are also show just for practice)

4 Missing Parts (None)

All the desired parts are implemented and tested as possible as it can. It seems there is no missing part.