Gebze Teknik Üniversitesi

Bilgisayar Mühendisliği

System Programing(CSE 344)

Homework #1

Berkan AKIN

171044073

Part #1

Problem

This program should open the specified filename (creating it if necessary) and append num-bytes bytes to the file by using write() to write a byte at a time. By default, the program should open the file with the O_APPEND flag, but if a third command-line argument (x) is supplied, then the O_APPEND flag should be omitted, and instead, the program should perform an lseek(fd, 0, SEEK_END) call before each write(). Run two instances of this program at the same time without the x argument to write 1 million bytes to the same file.

Solition

There are two different ways of writing to the file in this program:

1. By default, the file is opened with the O_APPEND flag, and each new write operation appends new data to the end of the file.
2. If a third command-line argument (x) is provided, the O_APPEND flag is omitted, and an lseek(fd, 0, SEEK_END) call is made before each write operation. This way, each write operation goes to the end of the file and adds new data to the end.

The main difference between these two methods is whether the written data is appended to the end of the file or added to a different position in the file. **When the O_APPEND flag is used, the data is always appended to the end of the file, while an lseek() call can add the data to a different position in the file.**

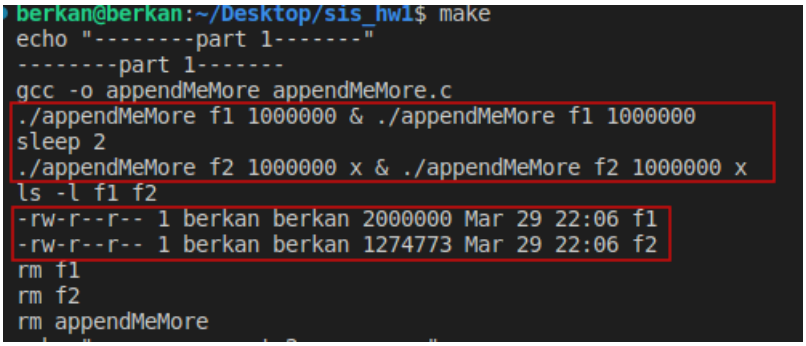
Program Test Case

The screenshots below contain the test cases for the first part. The program was run three times, and a difference was observed between the two files in all three runs. The difference is indicated within red.

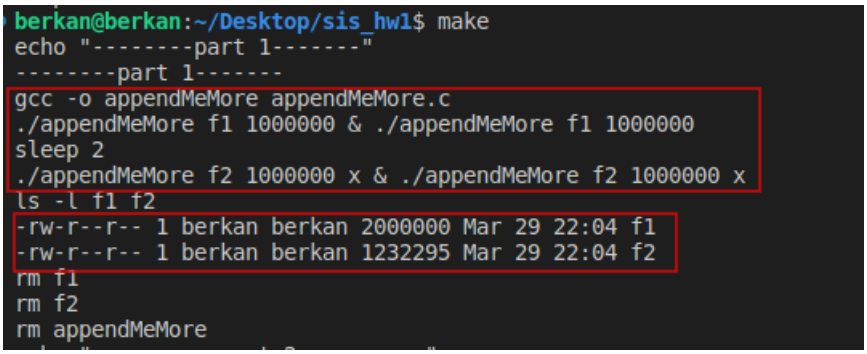
Test #1

```
berkan@berkan:~/Desktop/sis_hw1$ make
echo "--------part 1-------"
--------part 1-------
gcc -o appendMeMore appendMeMore.c
./appendMeMore f1 1000000 & ./appendMeMore f1 1000000
sleep 2
./appendMeMore f2 1000000 x & ./appendMeMore f2 1000000 x
ls -l f1 f2
-rw-r--r-- 1 berkan berkan 2000000 Mar 29 22:05 f1
-rw-r--r-- 1 berkan berkan 1296729 Mar 29 22:06 f2
rm f1
rm f2
rm appendMeMore
```

Test #2



Test #3



Results

After running the program, I observed the difference between writing to files in different ways. First, when I wrote 1 million bytes to the same file without specifying the x argument, each write operation was appended to the end of the file. However, when I repeated the same steps while writing to a different file and specifying the x argument, each write operation was written to a different position in the file instead of being appended to the end.

As a result, the program ran as expected and the difference between writing to files in different ways was clearly seen.

Part #2

Problem

In this problem, it is required to implement the dup() and dup2() functions using fcntl() and close() where necessary. For the dup2() function, there is a special case where oldfd equals newfd, and in this case, it should be checked whether oldfd is valid. This can be done by checking if fcntl(oldfd, F_GETFL) succeeds. If oldfd is not valid, then the function should return -1 with errno set to EBADF. It can be ignored that dup2() and fcntl() return different errno values for some error cases.

Solition

For the dup() function, we can call fcntl() with the F_DUPFD flag to duplicate the file descriptor. This flag returns the lowest file descriptor greater than or equal to the third argument, which is the lowest available file descriptor. If the function succeeds, it returns the duplicated file descriptor, otherwise, it returns -1 with errno set to the appropriate error code.

For the dup2() function, we can check if oldfd is equal to newfd. If so, we check whether oldfd is valid by calling fcntl(oldfd, F_GETFL) and checking whether it succeeds. If it does not succeed, we return -1 with errno set to EBADF. If oldfd is valid and equal to newfd, we simply return newfd. Otherwise, we call close(newfd) to close any open file descriptor with the same value as newfd, and then call fcntl() with the F_DUPFD flag to duplicate oldfd to newfd. If fcntl() succeeds, we return newfd, otherwise, we return -1 with errno set to the appropriate error code.

Program Test Case

Dub1

The first image shows the implementation of the dup() function. In the second image, the dup() function has been tested. The fd1 file descriptor has been copied to create fd2 using the dup() function. It has been checked whether the created file descriptors point to the same file. A string has been written into a file, and the output of the written text can be seen in the third image.

İmage 1

```c
int my_dup1(int oldfd) {
    return fcntl(oldfd, F_DUPFD, 0);
}
```

İmage 2

```
printf("----------Dub1 Function----------");
int fd2 = my_dup1(fd1); // copy of decriptor
if (fd2 == -1) {
    perror("my_dup1");
    return 1;
}
printf("\nfd1: %d, fd2: %d\n", fd1, fd2); // file decriptor address
write(fd1,"first", 5); // firstli fd1 write in same file
write(fd2,"second",6); // secondly fd2 write in same file
lseek(fd1,SEEK_SET,0);

int n= read(fd1,buffer,12);
buffer[n] ='\0';
printf("Read file:%s",buffer);
off_t offset1 = lseek(fd1, 0, SEEK_CUR);
off_t offset2 = lseek(fd2, 0, SEEK_CUR);
printf("\nFile offset1: %lld  \nFile offset2 : %lld\n", (long long) offset1, (long long) offset2);
```

İmage 3

```
----------part 2----------
gcc -o part2 part2.c
./part2
----------Dub1 Function----------
fd1: 3, fd2: 4
Read file:firstsecond
File offset1: 11
File offset2 : 11
```

Dub 2

The dup2() function has been created. The oldfd file descriptor and newfd file descriptor have been passed as parameters to the function. The validity of the oldfd parameter has been checked, and if it is valid, it is assigned to newfd. The function has been tested for proper operation and explained with screenshots of its execution.

Image 1

```
printf("\n----------Dub2 Function----------");
int fd3;
my_dup2(fd1, fd3);
if (fd3 == -1) {
    perror("my_dup2");
    return 1;
}


offset1 = lseek(fd1, 0, SEEK_CUR);
off_t offset3 = lseek(fd3, 0, SEEK_CUR);
printf("\nfd1: %d, fd3: %d\n", fd1, fd3);
printf("File offset1: %lld  \nFile offset3 : %lld\n", (long long) offset1, (long long) offset3);
```

Image 2

```c
int fd4=3;
my_dup2(fd1, fd4);
if (fd4 == -1) {
    perror("my_dup2");
    return 1;
}
offset1 = lseek(fd1, 0, SEEK_CUR);
off_t offset4 = lseek(fd4, 0, SEEK_CUR);
printf("\nfd1: %d, fd4: %d\n", fd1, fd4);
printf("File offset1: %lld  \nFile offset4 : %lld\n", (long long) offset1, (long long) offset4);
close(fd1);
close(fd2);
close(fd3);
```

Image 3

```
----------Dub2 Function----------
fd1: 3, fd3: 0
File offset1: 11
File offset3 : 11

fd1: 3, fd4: 3
File offset1: 11
File offset4 : 11
rm part2
```

Part #3

Problem

Write a program to verify that duplicated file descriptors share a fileoffset value and open file

Solition

This program will create two file descriptors that point to the same file. Firstly, we will open a file using a file opening function and obtain a file descriptor. Then, using the dup() function, we will create a second file descriptor that contains the same file descriptor as the first one.

Next, we will obtain the offset values that point to the same file for both file descriptors and print them to the screen. Then, by changing an offset value for one file descriptor and printing the offset value again for the other file descriptor, we will confirm that there is a shared offset value between the two file descriptors.

The program will use the fcntl() function to verify the offset values that the files point to and to demonstrate that there is a shared offset value between the two file descriptors.

Program Test Case

- In the first test, a file was opened and a duplicate file descriptor was created. The offset values for the created file descriptor were compared and the same value was obtained.

```c
int dup_fd = dup(fd1);
if (dup_fd == -1) {
    perror("dup");
    close(fd1);
    return 1;
}

off_t offset1 = lseek(fd1, 0, SEEK_CUR);
off_t offsetdub = lseek(dup_fd, 0, SEEK_CUR);

if (offset1 != offsetdub) {
    printf("File Offsets Diffirent\n");
    printf("File offset1: %lld  \nFile offsetdub : %lld\n", (long long) offset1, (long long) offsetdub);

}
else {
    printf("File offsets match\n");
    printf("File offset1: %lld  \nFile offsetdub : %lld\n", (long long) offset1, (long long) offsetdub);
}
```

- In the second test, text was written to the file using one file descriptor, and it was observed that both file descriptors progressed in the same way and had the same value.

```c
printf("----------- Test #2 ----------\n");
const char *text = "Hello world!";
int byte_num = write(fd1, text, strlen(text)) ;
if (byte_num  != strlen(text)) {
    perror("write");
    close(fd1);
    return 1;
}


printf("Write File Hello world!\n");
offset1 = lseek(fd1, 0, SEEK_CUR);
offsetdub = lseek(dup_fd, 0, SEEK_CUR);

if (offset1 != offsetdub) {
    printf("File Offsets Diffirent\n");
    printf("File offset1: %lld  \nFile offsetdub : %lld\n", (long long) offset1, (long long) offsetdub);

}
else {
    printf("File offsets match\n");
    printf("File offset1: %lld  \nFile offsetdub : %lld\n", (long long) offset1, (long long) offsetdub);
}
```

- In the third test, one of the progressed file descriptors was brought back to the starting offset (0 value). The two file descriptors were compared, and it was seen that they had different values.

```c
printf("---------- Test #3 ----------\n");
if (lseek(fd1, 0, SEEK_SET) == -1) {
    perror("lseek");
    close(fd1);
    close(dup_fd);
    return 1;
}
offset1 = lseek(fd1, 0, SEEK_CUR);
if (offset1 != offsetdub) {
    printf("File Offsets Diffirent\n");
    printf("File offset1: %lld  \nFile offsetdub : %lld\n", (long long) offset1, (long long) offsetdub);

}
else {
    printf("File offsets match\n");
    printf("File offset1: %lld  \nFile offsetdub : %lld\n", (long long) offset1, (long long) offsetdub);
}
```

```
---------- Test #1 ----------
File offsets match
File offset1: 0
File offsetdub : 0
---------- Test #2 ----------
Write File Hello world!
File offsets match
File offset1: 12
File offsetdub : 12
---------- Test #3 ----------
File Offsets Diffirent
File offset1: 0
File offsetdub : 12
Read from duplicated file descriptor: Hello world!
```

Results

When a file descriptor is duplicated, the values of the two file descriptors become the same. When writing to a file using one file descriptor, the values also become the same. When we change the position of one file descriptor, the position of the other one does not change. We observed this in the third test.