In this exercise we familiarize to the file handling.

Excercise 3a (Handling files, file descriptors and flags, 1,0p)

Part A

Phase 1 (Concurrent writing data to the file; consequent writes)

Use system calls in file handling (not standard i/o-library functions) in phases 1, 2 and 3.

Write two programs that both write data to the same file exlog.txt. (You can imagine, that the file is a kind of log file.)

You can make the both programs to open the file in similar way using O_CREAT flag in `open` system call. This is so because if the file already happens to exist, `open` just opens the existing file normally. Always remember, before the next test run, to delete the file exlog.txt. Both programs enter a loop after opening the file. The first program writes four characters "AAAA" to the file in the loop. The second program writes four characters "BBBB" to the file in the loop. The loop is repeated 100 000 times.

Run these programs concurrently. Start them using the command ...$ progr1.exe& progr2.exe [Enter].

Use command ...$ ls -l exlog.txt to see the size of the file. If all data were written, the size should be 800 000 bytes. What is the situation?

Remark. The starting procedure described above guarantees that programs are started almost "simultaneously" and they are run concurrently. But you never can be sure which one is started first. That's why it is recommended that you use O_CREAT flag in both programs. To get correct results from the test runs remember always to delete the file exlog.txt before the next run.

Phase 2. (Concurrent writing data to the file; consequent `lseek` and `write`)

Modify the program of phase 1 so that before each call of write function, you set the file position pointer at the end of the file (using function `lseek`).

See how many outputs have now successfully written to the file.

Phase 3. (Atomic operation using O_APPEND flag)

In phase 2 you saw the problem that was caused by the fact that context switch can sometimes happen between the `lseek` and `write` system calls.

Fix the program so that you add the flag O_APPEND in the `open` call of each program and see that there is no problem any more. It is not necessary or usefull to use `lseek` system call in this phase anymore.

Part B. (File flags)

Write first the function print_flags that takes a file descriptor as a parameter. The function reads file flags and finds out the access mode and the current settings of the file flags O_NONBLOCK and O_APPEND. The function displays the findings in the following way:

> Access mode: Read/Write (or Read Only or Write Only according the situation)
> O_NONBLOCK: OFF (or OFF)
> O_APPEND: OFF (or ON)

Write next the main function that has four sections each corresponding a certain test phase.

Test 1.
Open the terminal using the access mode Read/Write and without any other flags. This is done as follows

```
fd = open("/dev/tty", O_RDWR);
```

Use the print_flags function to see the access mode and settings for flags O_NONBLOCK and O_APPEND.

Next try to read one character from the opened terminal. Test the return value of read function. If the read was successful, display a message "Read succeeded. Character was X". If the read failed, display a message "Read failed: xyzxyzxyz", where xyzxyzxyz tells the error in plain text.

After reading, try to write the character 'A' to the terminal. Test the return value of write function. If the write was successful, display a message "Write succeeded". If the write failed, display a message "Write failed: xyzxyzxyz", where xyzxyzxyz tells the error in plain text.

Close the terminal.

Test 2.
Open the terminal using the access mode Read Only and without any other flags.

Call the print_flags function again and do all the same things as in test 1.

Do not close the terminal.

Test 3.

Do not open the terminal again! It is left open in test 2. Set file flags O_NONBLOCK and O_APPEND for the terminal. Call the print_flags function to see the effect.

Next try to read one character from the opened terminal. Test the return value of read function. If the read was successful, display a message "Read succeeded. Character was X". If the read failed, display a message "Read failed: xyzxyzxyz", where xyzxyzxyz tells the error in plain text.

Test 4.
Clear the file flag O_NONBLOCK. Call the print_flags function to see the effect. Another file flag O_NONBLOCK must be preserved. Try to read from the terminal in similar way as in Test 3 to make sure that also the way how read is functioning is changed.

Remark. O_APPEND has no effect here to the function of read operation. It is set only to see that clearing O_NONBLOCK does not modify other flags.

Extra Exercise 3b (Contents of st_mode member: file type and access rights, 0,25p)

Write a program that finds out the file type of the file that is passed as a command line parameter and displays the file type in plain text to the screen. (You have to test only three file types: Regular file, Directory file and Character special file (for example the file /dev/ttyS0)). Run the program three times, so that you can see all three possible results.

In addition to finding out the type of the file, the program also finds out the access rigths of the file as one number and displays it to the screen in the familiar octal format. For example in the format 0740, if owner has all rights, group has read access right and others have no rights to access the file. The basic idea here is that you should mask out from the st_mode the bit field of 12 bits that contains access rights and then display that bit field as an octal number.

Hint. In addtion to the access right bits the member st_mode contains only the file type bit field. There is a mask for the file type bit field (predefined constant S_IFMT). Use that mask first to construct a new mask for the access right bits field and use it to mask out the access right bits from the st_mode.