System programming

Laboratory #2: **Low-level File Handling, Processes and Pipes**

For the submission of the Lab report, please follow the rules published in Portale.
For exercises involving C program, it is also required to report the commands used for compilation and for invoking the executable.

The deadline for the second laboratory is Dec 14th 2020 at 24:00.

Exercise 1 – Dealing with folders in a program
Write a C program that is able to search for a file in a specific folder. Both folder path and filename are provided by command line interface.

Exercise 2 – Manage low level files
Write a C program that generates a simple float Matrix of random numbers and save it into a file. The size of Matrix is pre-defined by yourself in the code. The name of the file is provided by command line arguments and the program must work only if the file already exist.

Exercise 3 – Low level files
Write a C program able to read a list of integer numbers from a text file (whose name is given as a **first command line argument**) and write into another text file (whose name is given as a **second command line argument**) only the numbers that are prime numbers in the list.
E.g. given the following command line:
$> **./prime.out first.dat second.dat**
and given the content of first.dat file:
4 6
7 3
the program will create a second file second.dat:
7
3
Then write a second program preforming the same operation but using **binary file (data are stored as binary code instead of human-readable textual string)**. Eventually, report the differences of the files created with the two programs (possible advantages and disadvantages).

Exercise 4 – Low level file content handling
Write a program able to copy floating point data contained in a file into a second file, where all file names have to command line parameters. Let the user specify several options:

1. –b forces the output file to be binary.
2. –t forces the output file to be a text one.
3. –x forces the output file to be a hexadecimal representation.

Submit your program with at least 2 examples of a text file saved into a binary one and a binary one saved into a text file.

Exercise 5 – Process creation
Write a C program that creates a random number of child processes. Each child process creates another random number of processes, which will sleep for a random number of seconds (from 1 to 10) and then die. Be sure that each *relative* father is going to end only when all its children are dead.

Exercise 6 – Basic process synchronization
Write a program able to do a fork() and then to print messages both from the father and the child processes. Eventually, modify the program in order to have the messages printed from the father only after the child is finished.

Exercise 7 – Exec functionalities
Create a modified Linux shell called myshell. The new shell must have the following characteristic:

- It receives commands from user input like the standard linux shell (for the sake of simplicity, suppose that the command cannot be split on different lines).
- Every command is executed in background in a dedicated child process. (Hint: use fork and execlp)
- The output of the command is not written on the screen but is saved in a file named <PID>.log where <PID> is the process identifier of the child process executing the command.
- The shell terminates when the user enters the exit command.

Exercise 8 – Processes and Pipes
Write a program that creates a pipe, and after the fork() uses the pipe to send a large file from one process to the other during the execution. The program will accept two command line arguments indicating source filename and destination filename. The parent process will read the source file send the content to child process via Pipe, the child process will write the content to destination file.

Exercise 9 – Process and more complex task synchronization
Use the previous program skeleton to evaluate the following Taylor expansion:

$$e^x = 1 + \sum_{1}^{N} \frac{x^i}{i!}$$

The main process gets x and N value as input (x is a double number and N > 0) and performs the final sum. No boundaries are set to N value.

Each $\frac{x^i}{i!}$ is performed by the i-th process and sent to the father for the final sum.

(Hint, in child process, the values of variables are the ones when fork is invoked and will not be affected by parent process after the fork, this could be a way to separate information among child processes)

Exercise 10 – Basic consumer-producer
Write a simple producer-consumer program, in which the queue between the two processes (father and son, or two brothers) is **at most of 4 tokens**, each token being a character. Do everything using the semaphores implemented with pipes only.
Hint: POSIX semaphore or the one we introduced has a limitation of **no upbound**. So in order to achieve "at most of 4 tokens" (which is an up-bound of the semaphore values), two semaphores have to be used. Initialize one (e.g. sem_1) to 0 indicating available token, the the other (e.g. sem_2) to 4 indicating available token slot. So producer will **wait** for sem_2 and **signal** sem_1 while consumer will **wait** for sem_1 and **signal** sem_2.

Exercise 11 – Processes, synchronization and files
Write a program composed by N child processes, each of them able to generate several times a stream of characters to be written into a single file, shared among them, without storing them in any memory array. Let the user define the N number of processes, the name of the file and the number of iterations at which each process will be asked to generate a new set of characters.

Exercise 12 - Advance Processes management
Write a C program that emulates a message system. A set of children processes are given a specific ID to be identified when messages will be sent. The main process generates 100 messages of random text, assigning randomly, for each message, the ID of the child process that must manage the message. Once all 100 messages have been sent, the main process send a special message to signal the end of the computation. Each child must recognize all messages addressed to it and collect them until the end of the computation. Once the end is signaled, each child must print on screen the list of all received massages.
Let the user define the number of child processes and the maximum length of a single message by command line parameters.
Hint:For each child process a separate pipe could be created, so the parent could send the message to specific child through corresponding pipe. This is a many one-to-one scenario.

Exercise 13 [OPTIONAL]– Advanced computation in processes
This is the original exercise, with which if the list of allowed characters is large, then the number of possible password combination to be generated will explode.
To make things easy for you, you can choose to do the Exercise 14 instead of this one.

Write a C program that generates a sequence of passwords and store them into a file. Let the user indicate the filename by passing it through **command line**, together with two numbers to set the range limits for the passwords' length. Passwords have to be all the admissible (within range limits) combinations of alphanumeric characters. Let the program have three extra options:

1. –p <number>: defines how many processes generate the passwords in parallel. If not provided, your program must resort to 2 processes. Each process runs over 1/<number> of the possible passwords per item in the range. I.e., if 2 processes are provided, for a 3 characters password, each of them manages to generate $\frac{26^3}{2}$ passwords.
2. –i <string of characters without separators>: defines the list of alphanumeric characters you want your passwords generated from, i.e., `-i 0123456789abcde.,#@`, where a 3 characters password like 'fge' is not permitted. If not provided, by default the set of characters must be only alphanumeric characters (no punctuations or special chars).
3. -d <file name>: defines the dictionary file, represented by a list of words that must be excluded by the list of generated passwords.

Command line example:

```
$> ./your_program_executable_file 3 8 output.txt
```

➔ Meaning: generates all passwords ranging from 3 characters to 8 and save them into `output.txt` file.

```
$> ./your_program_executable_file –p 15 3 8 output.txt
```

➔ Meaning: generates all passwords raging from 3 characters to 8, resorting to 15 processes instead of 2, and save them into `output.txt` file.

```
$> ./your_program_executable_file –p 15 –i 0123456789 3 8 output.txt
```

➔ Meaning: generates all passwords raging from 3 characters to 8, containing only digits and resorting to 15 processes, and save them into `output.txt` file.

Hint: using system call to access the same file in multiple processes are fine (as read/write less than a I/O block size, normally 4096, is likely atomic), but will make formatted file I/O a bit tricky (you can still use sprintf to format the string first then use write system call to write to file), so another solution is to send the password to parent and parent is in charge of storing them into the file. However, if you can, try to provide two solutions.

Hint2: memory dynamic management is preferred, i.e. to use malloc and free instead of static array. However, otherwise, **assume the max length is 10**.

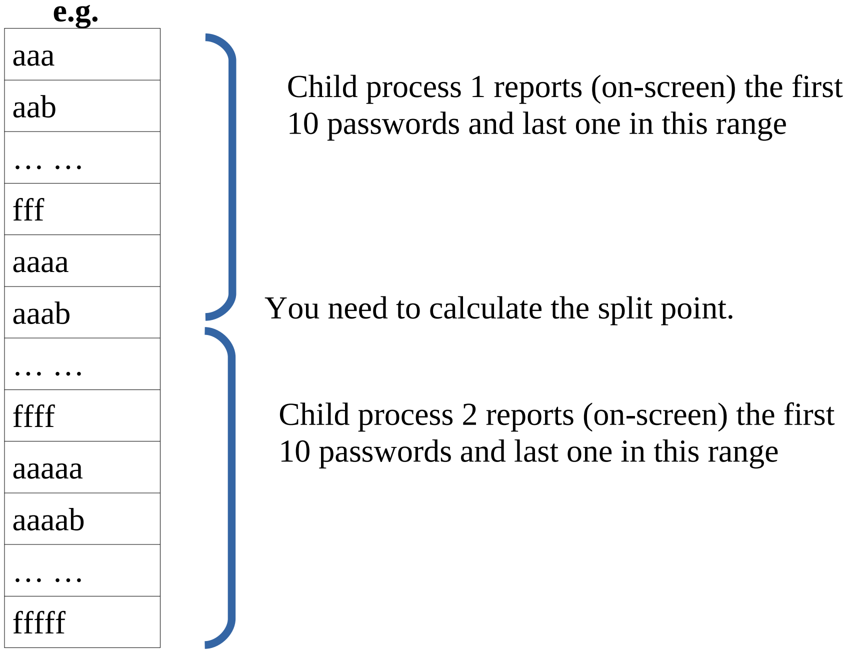
Exercise 14 – Advanced computation in processes

Write a C program that generates a sequence of passwords. The program should accept following parameters:

1. –p <number>: defines how many processes generate the passwords in parallel. If not provided, your program must resort to 2 processes. Each process runs over 1/<number> of the possible passwords per item in the range. I.e., if 2 processes are provided, for a 3 characters password, each of them manages to generate $\frac{26^3}{2}$ passwords.
2. –i <string of characters without separators>: defines the list of alphanumeric characters you want your passwords generated from, i.e., `-i 0123456789abcde.,#@`, where a 3 characters password like 'fge' is not permitted. If not provided, by default the set of characters must be only alphanumeric characters (no punctuations or special chars).
3. <min_password_length> <max_password_length>: defines the minimum and maximum password length.

For each child process, it only reports the first 10 password and the last password of range assigned to it. For example, if the list of characters is "abcdef", and the min_password_length and max_password_length are 3 and 5, then all the passwords to be generated is from "aaa" to "fffff". And if the number of processes is 2 then the range has to be divided into two ranges (different strategies could be implemented, however, try to balance the workload), then each process reports the first 10 passwords and the last password in the range assigned to the process.

e.g.



aaa
aab
… …
fff
aaaa
aaab
… …
ffff
aaaaa
aaaab
… …
fffff

Child process 1 reports (on-screen) the first 10 passwords and last one in this range

You need to calculate the split point.

Child process 2 reports (on-screen) the first 10 passwords and last one in this range