operating systems lab - week 4: exercise

Nicolo Colombo

October 19, 2020

This lab is about low-level input-output and processes. This week you will start seeing why studying C is important from a OS perspective. In particular, you will write programs that read or write on files using UNIX system calls and create processes using `fork`

Please try to complete all sections of this lab sheet before attempting this week's Moodle quiz

lab quiz - week 4

as some of the quiz questions may require you to run, comment or modify the programs you are asked to write.

Set up

Depending on your OS, use the following instructions to connect to `linux.cim.rhuk.ac.uk`:

Unix Open the terminal and run

```
ssh yyyyxxx@linux.cim.rhul.ac.uk
```

where `yyyyxxx` is your college username, and enter your password to access the teaching server.

Windows Launch the Windows SSH client `puTTY` [1], enter the following

```
linux.cim.rhul.ac.uk
```

in the empty field *Host Name (or IP address)* and click on *Open*. The client opens a new window where you are required to enter your college user name `yyyyxxx` and password.

Once logged in, you should be able to see the content of and navigate in your home directory using the standard UNIX commands, e.g. `ls`, `cd`, `cp`. Go to the `CS2850labs` directory[2] and run the command

```
$mkdir week4
```

to create a new sub-directory called `week4`. We suggest you save and compile all programs you write for this exercise in this directory.

Use a command-line text editor, e.g. emacs, nano or vim, to open, edit and save your programs. The advantage of command-line editors is that they can be used in a non-graphical SSH session. [3] You can create a new C file or open an existing C file, `file_name.c`, by running the command

```
$editorName file_name.c
```

where, e.g. `editorName` is `vim`

We suggest you save separate files for all single parts of this exercise and follow the name suggestions given in each section. [4]

Compile your C code by running

```
$clang -o file_name file_name.c
```

[1] `puTTY` should be installed on all department's machines. If you work on your own Windows machine you can download it at download puTTY and install it as explained.

[2] The parent directory you have created for the first week's lab

[3] If you do not want to open and close the editor every time you modify and save your code you can open a new SSH session and use two shell windows simultaneously.

[4] This is mainly because some of the Moodle quiz questions may refer to single pieces of code through the suggested file names.

and run the corresponding binary files `file_name` through

```
$./file_name
```

For debugging, we suggest you use the free debugging tools of `valgrind`, which is already installed on the teacing server `linux.cim.rhul.ac.uk`. To check your code, you just need to run the command

```
$valgrind ./file_name
```

and have a look at the messages printed on the terminal.

1 Input-output

Write a program, `inputOutput.c`, that copies what the user writes on the terminal into a file, `fileIn.txt`, copies a capitalized version of the text saved in `fileIn.txt` into another file, `fileOut.txt`, and prints on the terminal the content of `fileOut.txt`.

Requirements

- The executable, `inputOutput`, should accept two file names, `fileIn.txt` and `fileOut.txt`, as parameters, i.e. it will be launched by running

  ```
  $./inputOutput fileIn.txt fileOut.txt
  ```

- Once started, the program should parse the text entered by the user into the terminal until the user sends an `EOF` signal.[5]

- The program should be implemented using the following low-level I/0 functions defined in `fcntl.h`:

  - `int creat(char *fileName, int perms)`, where we suggest you set full permissions for everybody. [6] To do this, add the following line outside `main`

    ```
    #define PERMS 0666   /* RW for owner, group, others */
    ```

    and then use `PERMS` as the second parameter of `creat` everywhere.

  - `int open(char *fileName, int flags, int oPerms)`, where you can set the `oPerms` parameter to 0 everywhere and the `flags` parameter to one of the following constants [7]:

    1. `O_RDONLY`, for reading only,
    2. `O_WRONLY`, for writing only, or
    3. `O_RDWR`, for both reading and writing.

  - `int write(int fd, char *buf, int n)`
  - `int read(int fd, char *buf, int n)`
  - `int close(int fd)`

  Visit Section 13.1 of the online reference manual for more details about `open`, `creat`, and `close`, and Section 13.2 for more details about `write` and `read`.

- For reading from and writing in the terminal you should use [8]

  `int STDIN_FILENO`, `int STDOUT_FILENO`, and `int STDERR_FILENO`

  as descriptors of the files associated with the *standard input*, *standard output* and *standard error*[9]

- The headers of your program should be `<unist.h>` and `<fcntl.h>`. You should *not* include `stdio.h`.

[5] In the terminal, type `ctrl-d` to produce an `EOF` signal.

[6] On Unix, permission info is encoded by nine bits that control read, write and execute access for the owner of the file, for the owner's group, and for all others. A three-digit octal number is convenient for specifying the permissions, e.g. 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

[7] These constants are defined in `fcntl.h`

[8] These constants are defined in `unistd.h`.

[9] If you write on `STDERR_FILENO` you can see the ouptut by running

```
$./inputOutput 2> error.txt
```

that creates a new file, `error.txt`, where all error messages are printed.

Hint We suggest you read the input in chunks of 20 characters, through a buffer variable declared as

```
char buffer[20];
```

and use `buffer` as `buf` argument in the calls of `read`. You can then set the `n` argument of `read` to 20 and check how many characters you have actually saved into `buffer` by checking its return value.

For reading and copying the whole user's input into a file, call `read` and `write` within a `while`-loop that terminates when `read` returns 0.

For capitalizing a character you can use the following function

```
int upper(int c){
    if (c >= 'a' && c <= 'z')
        return c + 'A' - 'a';
    else
        return c;
}
```

Example A run of the program should produce an output analogous to

```
cim-ts-node-02$ ./inputOutput fileIn.txt fileOut.txt
enter some text here:
one
Two
three and Four
file
no f i v e
and Six!
output:
ONE
TWO
THREE AND FOUR
FILE
NO F I V E
AND SIX!
```

where the first line is printed immediately after entering the command

```
$./inputOutput
```

and

```
one
Two
three and Four
file
no f i v e
and Six!
```

is the user's input.[10] The rest is printed just before the program exits.

2 fork

Write a program, `nChildren.c`, where a parent process creates N child processes through `fork`, waits for them to complete a task and exits.

[10] The input includes a `\n` at the end.

Requirements

- The program should accept N as a command-line parameter.
- The task of the nth child consists of printing the process identifier on the terminal and sleep for $n\%(N-1)$ seconds.
- The program should print a message on the terminal when one of the children terminates.
- Before exiting, the parent prints on screen the order on which the children have terminated.
- You may use the following library functions:

  1. `int printf(const char *format, ...)` defined in `stdio.h` and described in Section 12.12 of [1]
  2. `pid_t getpid(void)` defined in `unistd.h` and described in Section 26.3 of [1]
  3. `pid_t fork(void)` defined in `unistd.h` and described in Section 26.4 of [1]
  4. `unsigned int sleep(int sec)` defined in `unistd.h` and described in Section 21.7 of [1]
  5. `pid_t wait(int *status)` defined in `sys/wait.h` and described in Section 26.6 of [1]
  6. `int WEXITSTATUS(int status)` defined in `sys/wait.h` and described in Section 26.7 of [1]

  You should include the following headers:
  `stdio.h`, `unistd.h`, `wait.h`, and `sys/types.h`

Hint To generate the child process, call `fork()` within a loop but be sure that you do not generate child-of-the-child processes. Check that you are creating the right amount of processes by printing a message at some strategic points.

Each child process can perform its task by calling a function declared as

```
void taskFunction(int sec, int n)
```

Define `taskFunction` so that, at each call,

1. the program prints the following line on the terminal

   ```
   nth child (pid=x) sleeps for sec seconds
   ```

   where `x` is the process identifier of the current process
2. the calling process is put to sleep for `sec` seconds

The parents waits for the N children to terminate by calling `wait` N times. Note that`pid_t wait(int *status)` returns the process identifier of the child that terminates and writes the return value of the child that terminates at the address passes as `status` parameter. To interpret the content of that address, you can call `WEXITSTATUS`, with the value stored at that address as an input.

Finally, to make the parent print the order of the reaped children at the very end, save the return values of `wait` and `WEXITSTATUS` into two integer arrays so that you can print their content just before the parent process terminates.

Example If $N = 3$ a run of the program should produce an output analogous to[11]

```
cim-ts-node-01$ ./nChildren 3
1th child (pid=3691411) sleeps for 0 seconds
2th child (pid=3691412) sleeps for 1 seconds
1th child exits
3th child (pid=3691413) sleeps for 0 seconds
3th child exits
```

[11] Of course, you should expect different values for the process identifiers.

```
2th child exits
1th child(pid=3691411) exited 1th
3th child(pid=3691413) exited 2th
2th child(pid=3691412) exited 3th
```

References

[1] *The GNU C Reference Manual*

[2] Randal E. Bryant and David R. O'Hallaron, 2010. *Computer Systems: A Programmer's Perspective* Addison-Wesley Publishing Company, USA.