INFO0940-1: Operating Systems (Groups of 2)
Project 1: OShell

Prof. L. Mathy - G. Gain - K. Yasukata

Submission before Wednesday, March 11, 2020

1 Introduction

A shell is a user program which provides an interface to use operating system services. It is a simple program that has a very basic logic: it loops, handles commands by reading the standard input (stdin) and then executes them. This last procedure is the interesting part of the project. It will allow you to understand what is going on with processes and how they start. Moreover, the other requirements defined in the next section will allow you to understand others important concepts of an operating system.

In this project, you must implement a custom shell called `OShell` which is designed to run multiple copies of a particular process at once. Like any other shell, `OShell` takes as input the name of the process to run (e.g., `ls`, `pwd`, ...). Nevertheless, it requires also two additional inputs:

1. The number of copies[1] of the program to execute. It must be an integer between 1 to 9;
2. The type of execution: sequential or parallel execution.

```
$ ./oshell
OShell> uname
  Copies> 5
    [S]equential (default) or [P]arallelize > S
Linux
Linux
Linux
Linux
Linux
OShell>
```

Listing 1: Simple example output

[1] If the number of copies equals one, the shell executes directly the command.

2 Requirements

The following requirements must be satisfied:

- Your shell must respect a given skeleton (see Section 4);
- Your shell must support launching programs with arguments, and return to the prompt when the program terminates;
- For any additional data structures, you **must** use `malloc()` and `free()` to manage memory. Use valgrind[2] to detect memory leak;
- Your shell must **not** handle pipes (e.g., ls -l | wc -l) and complex commands (sleep 5 && echo fail). In addition, others concepts such as redirections, quotes, auto-completion, ... must **not** be taken into account;
- Your shell must exclusively use `fork()` and `execvp()` system calls to manage process creation; Nevertheless you can use other system calls such as `waitpid`, `kill`, ... to manage process execution;
- Your shell must support the `cd`, `exit`, `showlist`, `memdump` and `loadmem` built-in commands[3]. No other built-in command is needed:
  - The command `cd` must have the same behaviour than in a classic shell.
    `cd` by itself or `cd ~` will always put you in the home directory;
    `cd ..` will move you up one directory;
    `cd dir` (without a /) will put you in a subdirectory;
    `cd /dir` (with a /) will put you in a particular directory.
  - The command `exit` is quite explicit;
  - The commands `showlist`, `memdump` and `loadmem` are discussed in Section 3.
- Your shell will exit properly upon typing "exit" or CTRL+C (SIGINT);
- For **sequential** execution (only), you need to specify a timeout of **5** seconds. If a process takes longer than 5 seconds, it is automatically killed. For this part, you need to do a bit of research...
- For binary data persistence, use exclusively `open()`, `write()`, `read()`, `lseek()` and `close()` primitives (see Section 3);
- Your code must be readable. Use common naming conventions for variable names and comments;
- Your code must be robust and must not crash. In addition, errors handling must be managed in a clean way. Only fatal errors such as failed memory allocation will stop the shell. Other errors must be displayed on `stderr`.

[2] http://www.valgrind.org

[3] Built-in commands are executed only once. In other words, the "copies" and "execution" prompts are skipped.

3 The `showlist`, `memdump` and `loadmem` commands

As more "advanced" feature, we want to save the list of processes that have been created during a session. Therefore, each time a process is created, its name, its unique identifier (pid) and its exit code are added into a list in memory.

When the `showlist` command is used, the process list is displayed in the following format $(name1; pid1; exitCode1) \rightarrow ... \rightarrow (nameN; pidN; exitCodeN)$ on the terminal output (*stdout*).

```
$ ./oshell
OShell> uname
  Copies> 2
    [S]equential (default) or [P]arallelize> S
    Linux
    Linux
OShell> showlist
(uname;34142;0)->(uname;34143;0)
OShell>
```

Listing 2: Example output of the `showlist` command

As soon as the `memdump` command is entered, the list is persisted into a binary file called `memdump.bin`. Once persisted, it is then freed from the main memory. Note that the content of this binary file is truncated each time the `memdump` command is entered.

When the `loadmem` command is used, the **previous** content of the binary file is loaded into main memory (the list is reset). If the file is empty or does not exist, the list will be empty too.

4 Skeleton

We provide a program skeleton that implements the parsing and interface logic. The skeleton simply gets user input, parses it and then exits. Your task is to add new features to this skeleton with the process logic explained above.

Given files

The following files are provided:

- `oshell.{h|c}` contain(s) all functions related to shell management;
- `main.c` is the main program. You need to call the functions from the `oshell.c` file.
- `Makefile` a first skeleton that you need to complete. Flags and executable name cannot be **modified**.

Important: You can (must) add additional functions and files to this skeleton. Make sure that all is clean (opaque struct/pointer, good project structure, ...)!

5 Report

You are asked to write a very short report (**max 1 page**) in which you briefly explain how you managed the creation of the process and files handling. You can draw diagrams to illustrate your explanations.

Finally, we would also appreciate an estimation of the time you passed on the assignment, and what were the main difficulties that you have encountered.

6 Evaluation and tests

Your program can be tested on the submission platform. A set of automatic tests will allow you to check if your program satisfies the requirements. Depending on the tests, a **temporary** mark will be attributed to your work. Note that this mark does not represent the final mark. Indeed, another criteria such as the structure of your code, the memory management, the respect of the defined skeleton, the correctness and your report will also be considered. You are however **reminded** that the platform is a **submission** platform, not a test platform.

7 Submission

Projects must be submitted before the deadline. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of 2^{N-1} marks (where N is the number of started days after the deadline).

Your submission will include your code (all the required `.c|.h` files in a directory named `src`) along with a Makefile which produces the binary file `oshell`. Please also add your report (`.pdf` or `.txt`) in the `src` directory.

Submissions will be made as a `src.tar.gz` archive, on the submission system. Failure to compile will result in an awarded mark of 0.
Bon travail...