Lab Session 2: Shell
24296 - Sistemes Operatius

1 Introduction

The objective of this lab session is to fully program a custom shell and understand how a shell works. We are going to make use of knowledge explained in class and implement an enhanced version of the seminar exercises. We are going to code, compile and execute programs that use the system calls ”open”, ”read”, ”write”, ”fork”, ”execvp”, ”wait”, ”_exit” and the supplied read_split in myutils.h. You will need to deliver a compressed folder :

- A document answering the questions of the lab session.
- The code of the file ”shell.c”. The code you submit must compile, if not you will not be evaluated.
- The document is mandatory, is the first thing that will be evaluated after checking that the code compiles. If you don’t submit a document your lab session will not be evaluated.

2 System call signatures, program main structure and includes

Here you have the system calls that we will be using for this lab session:

```
int read_split(int fd, void *buf, int maxlength, char* ch_end);
sscanf(buff, "%d", &an_int);
int fork();
int wait(int* status);
execvp(char *name, char* argv[]);       // name is searched in PATH variable
_exit(int status);
```

We are going to use the functions startTimer and endTimer of the files myutils.h and myutils.c that we also used in the previous lab session (look at it if you forgotten). Then we can compile including the code (the .c file) of the additional header file:

```
gcc main.c myutils.c -o main
```

3 Fork and Exec example with questions

Look and understand the provided file fork_exec.c. Now compile the file and create the executable ”main”. We talked about recovering the exit status of a process in theory class, you now see in the example that we need some functions to access the exact integer (POSIX Macros WIFEXITED and WEXITSTATUS). Now answer the following questions in the document:

1. Explain what the program does and each of these command shell executions in the document mentioning how many processes are created in each case: (a) ./main (b) ./main ls (c) ./main ls -la (d) ./main ps (e) ./main ps aux (f) ./main < cmds.txt (g) ./main < ls (h) ./main & (i) ./main lss (j) ./main ./main (k) ./main ./main ls (l) ./main ./main ls -la
2. Explain why we need child exit after execvp?
3. Explain why we need child _exit(1) at the end of the child if code?

4 Programming Exercise to Deliver

We are going to program a custom shell in several steps that you will need to deliver next week together with the document, explaining how you addressed each of the following steps:

1. Use read_split to read a text file with a list of shell commands without options nor arguments (the supplied cmds.txt). The text file can be passed as parameter or given in the standard input. The program will print all commands to the screen as a numbered list. We recommend that you fill an array char* cmds[] and then print it.

   Input cmds.txt:

   ```
   ls
   ls
   ps
   ```

   Output:

   ```
   1) ls
   2) ls
   3) ps
   ```

2. Now you are going to add the possibility of reading commands with and without arguments (use the supplied cmds_args.txt). Remember read_split provides the last character it read (ch_end) and you can check if it is a space or a new line. We recommend using a struct to define a command with arguments to be more clean and then declare an array of commands:

   ```
   typedef struct cmd_struct {
    char cmd[80];                  // using fixed static array size to avoid dynamic memory
    char args[10][80];
    int nargs;
   } cmd_type;

   int ncmds = 0;
   cmd_type cmds[100];
   ```

3. Now the program will print a numbered list of all commands with arguments.

   Input cmds_args.txt:

   ```
   ls -la
   ps
   sleep 5
   cat main.c
   ```

   Output:

   ```
   1) cmd: ls argv[1]: -la
   2) cmd: ps
   3) cmd: sleep argv[1]: 5
   4) cmd: cat argv[1]: main.c
   ```

4. Now we are going to add a custom shell prompt "MyShell>". We are going to wait for an integer input (using read_split) and print the corresponding command and arguments. Remember integers cannot be read directly from keyboard, you will need to do a conversion using sscanf. After the command is printed, control is returned to our custom shell until "exit" is typed or reading input is interrupted pressing keys control + D. If we type "list", the list of commands is printed again.

   ```
   MyShell>
   MyShell> 2
   25518 ttys000     0:00.06 -bash
   ...
   MyShell>
   ```

5. Now instead of printing the command we are going to try to execute it using fork and execvp and waiting for the returning child as was done in previous example fork_exec.

6. As a final step, in our custom shell we are going to allow execution in the background of commands. An integer can be introduced to indicate the command to be executed, but if the integer is followed by an & symbol, the command will be executed concurrently and control will be returned to the shell. Be careful of which process returns in the call to wait.

   ```
   MyShell>
   MyShell> 3 &
   MyShell>
   ```