Operating Systems Assignment 1: Writing a shell

Deadline: Sunday, March 27 before 23:59 hours.

1 Introduction

In the two introductory lectures of this course, we have been introduced to the shell, how it works and how to do programming with the shell. We have seen how to interface with the operating system kernel, for example through the use of system calls, in the first theory lectures.

The goal of the first assignment is to write our own, very simple, shell. The shell should print a command prompt and allow the user to enter a command. After entering this command, the shell should properly execute the command, this includes supplying the provided arguments to the program. The user can exit the shell using the `exit` command. Note that it is *not* the intention to implement advanced features like job control, redirection and shell scripting in this shell. Often the shell is responsible for changing the current working directory using the `cd` command, also this is not part of this assignment.

To become familiar with writing shell utilities as well, we will write our own small `ls` command which we will name `myls`.

2 Requirements

You may work in teams of at most 2 persons. Both your shell and your `myls` command must be written in C. Your submissions must adhere to the following requirements:

1. Submit the source code of your functioning, interactive, shell. The shell should:
   - Print a command prompt.
   - Allow the user to enter commands. For example, test whether the `man`, `cat` and `ls` commands work. Also test whether your shell can launch itself.
   - Find the program to be launched in a hard-coded array of standard locations.
   - Launch the program and pass the provided arguments to this program.
   - When the program has finished, print a command prompt again.
   - The command `exit` should exit the shell.
2. Submit the source code of your `myls` command. This command should:
   - When no command argument is provided, list the contents of the current working directory (just the file names, no other details). Suffix names of directories with a `/`.
   - Optionally a directory name may be provided as command line argument. If this is the case, the contents of this directory should be listed.
3. Your submission should contain a Makefile which will build the shell and the `myls` command and also includes a *clean* target.

When grading the submissions, we will look at whether your programs fulfill these requirements and the source code looks adequate (structure, indentation).

Make sure all files contain your names and student IDs. Put all files to deliver in a separate directory (e.g. `assignment1`), remove any object files and binaries. Finally create a gzipped tar file of this directory:

```
tar -czvf assignment1.tar.gz assignment1/
```

Mail your tar files to *krietvel (at) liacs (dot) nl* and make sure the subject of the e-mail contains *OS Assignment 1.*

Deadline: We expect your submissions by Sunday, March 27, before 23:59.

3 Programming language

Because we will start working on an operating system kernel which has been written in C in the next assignment, we would strongly encourage you to complete this assignment using the C language. Make sure to compile your code using a C and not a C++ compiler (use `.c` as extension and not `.cc` or `.cpp`). This means that you cannot use C++ features such as classes, virtual methods and `cout` and `cin` for I/O.

Some notes:

- Instead of using `cin` and `cout` for I/O, use `printf` and `scanf` functions.
- To allocate memory, use the `malloc` and `free` functions instead of `new` and `delete`.
- A man page exists about every function in the standard C library. For example, to learn more about `scanf` use `man scanf`. The manual pages about library functions are always in section 3: `man printf` will give you information about the shell command, but `man 3 printf` about the C library function. Similarly, system calls are in section 2.
- Do not include `iostream` or set a namespace. Instead, include `<stdio.h>`, `<stdlib.h>`, `<string.h>` and `<unistd.h>`.
- Ask the assistants for help if you have problems!

4 Guide to library functions

You will have to use library functions to accomplish the various tasks. Some of these library functions are wrappers around actual system calls, which are traps to the operating system kernel (e.g. `fork` and `execv`).

Reading user input. There are several ways to do this. We suggest to use `fgets`.

Parsing user input. In the command entered by the user, you will have to split the string into an argument vector. The string is usually split on the space character. Remember that the program to execute is stored in `argv[0]`. You can do this separation manually, or use a provided string manipulation function like `strsep`.

Executing a program. To execute a program, first locate the executable file. When the file name starts with `/` then the full path is already given and no search is needed. Otherwise, concatenate the name of the program to each of the paths in your hard-coded path array and use the `access` system call to test whether the file exists and is executable. If so, then create a new process using `fork` (like discussed in class) and load the executable using `execv`. Finally, in the

parent process use the `wait` system call to wait for the child process to terminate.

Reading directories. For the `myls` command, you will have to read directories. This can be done using the `opendir`, `readdir` and `closedir` functions. You can use the `struct dirent` (directory entry) structure returned by `readdir` to determine whether the file is a regular file or a directory. The `getcwd` function will return the working directory, which you need in case no arguments are provided.

5 Skeleton

You can use the following skeleton for the main routine of your shell program:

```
const char *mypath[] =
{
  "./",
  "/usr/bin/",
  "/bin/",
  NULL
};

while (...)
  {
    /* Wait for input */
    printf ("prompt> ");
    fgets (...);

    /* Parse input */
    while (( ... = strsep (...)) != NULL)
      {
        ...
      }

    /* Check if executable exists and is executable */

    /* Launch executable */
    if (fork () == 0)
      {
        ...
        execv (...);
        ...
      }
    else
      {
        wait (...);
      }
  }
```