Operating Systems 2016, Assignment 1: Writing a shell

Deadline: Friday, February 26 before 23:59 hours.

1 Introduction

In the first theory lectures of the course, we have seen how to interface with the operating system kernel through the use of system calls. We will briefly explore this interface with an operating system kernel in this first assignment, before crossing the border to actually work on a kernel itself in the three other lab assignments.

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. The shell should also have a functioning `cd` command to change the current working directory and the shell should show the current working directory in the prompt.

Commands such as `ls` and `cat` are commonly implemented as separate system programs and are not part of the shell itself. Within this assignment, we will not implement such a system program, but you can try for yourself that it is trivial to implement a simple version of the `ls` command. Using these simple system commands, bigger commands can be created where the output of one command is sent to another. This is for instance done using the pipe character (`|`). As the final part of this assignment you will implement (simplified) support for handling pipes, so that your shell is capable of executing commands such as `cat Makefile | grep gcc`.

2 Specification

The assignment is to program a shell program that conforms to the specification given in this section. The shell must be written in plain C. This is to prepare for the actual kernel coding in the next assignment, in which only plain C can be used. Make sure to use a C and not a C++ compiler to warn you for the use of C++ constructs, because points will be deducted when C++ constructs are used in the code. The shell should have the following features:

- Print a command prompt that also displays the current working directory.
- Allow the user to enter commands and execute these commands.
- The entered command string must be tokenized into an array of strings by removing the space delimiters. Also delimiters consisting of more than one space must be handled correctly.
- Implement `exit` (to exit the shell) and `cd` (to change the current working directory) as built-in commands. The `cd` should display errors if necessary.
- Ability to find program to be launched in the file system using a hard-coded array of standard locations in case the name of the executable is not preceded by an absolute or relative path. You must implement this yourself, do *not* use `execlp` or `execvp`.
- Display an appropriate error if a requested command cannot be found or is not executable.
- Execute commands and correctly pass the provided arguments to this command.
- Execute commands that contain a pipe character by starting two new processes that are interconnected with a pipe. Your shell should be able to handle data streams of arbitrary length.
  - Note 1: your shell only has to be capable of running commands that contain a *single* pipe character. Check how many pipe characters a command contains and simply display an error if more than one pipe character is found.

  - Note 2: to simplify implementation, you only have to deal with the case that the pipe character is separated by spaces, so the tokenizer you have to implement already creates a separate token for the pipe character. For instance a command of the form `cat Makefile|wc -l` does not have to be handled correctly by your program.

- A Makefile should be included to build the software. The Makefile should be of decent quality and should include a *clean* target. For a short tutorial on writing Makefiles, you can refer to `http://www.liacs.nl/~krietvel/courses/os2011/lab00/shell-utilities.pdf` (in Dutch) starting at slide 26.

3 Submission and Grading

You may work in teams of at most 2 persons. Your submission should consist of the source code of the shell implementation and a Makefile to build the software. *Make sure all files contain your names and student IDs.* Put all files to deliver in a separate directory (e.g. `assignment1`) and 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 *os2016 (at) handin (dot) liacs (dot) nl* and make sure the subject of the e-mail **equals** "OS2016 Assignment 1".

Deadline: We expect your submissions by Friday, February 26, before 23:59. No exceptions; deliveries after the deadline *will not be graded!*. Send e-mail attachments, Google Drive or DropBox links *are not accepted.*

The grade is determined based on whether the program correctly implements the functionalities listed in the specification above and whether the source code looks adequate: good structure, consistent indentation, error handling, correct memory handling and comments where these are required. Comments are usually required if the code is not immediately obvious, which often means you had to make a deliberate decision or trade-off. Document these decisions, trade-offs and why in the source code. Commenting on the obvious is superfluous and bad style. Note that we may always invite teams to elaborate on their submission in an interview in case parts of the source code need further explanation.

The maximum grade that can be obtained for this assignment is 10. The points are distributed as follows: Code Layout & Quality, Makefile (1.5 / 10), Command input handling & built-in commands (3.5 / 10), Path handling / path search (1.5 / 10), Execution of simple commands (1.5 / 10), Execution of pipe commands (2.0 / 10).

4 Programming language

Because we will start working on an operating system kernel which has been written in C in the next assignment, we require 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 the `printf` and `scanf` functions.
- To dynamically 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!

5 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 (POSIX API), which are traps to the operating system kernel (e.g. `fork` and `execv`).

Reading and parsing user input. There are several ways to obtain input from the user, we suggest to use `fgets`. The command string entered by the user will have to be split (or tokenized) into an argument vector using the space character as delimiter. You can do this tokenization manually, or use a provided string manipulation function such as `strsep`. Make sure to also properly deal with delimiters consisting of multiple spaces.

Executing a program. To start a new process and execute a program the `fork` and `execv` should be all you need (like discussed in class). In the parent process you will need to use the `wait` system call. The first item in the argument vector indicates the program to run. If this item does not contain a slash, it is not a relative or absolute path and we must find out where this program is located in the file system. To do so, concatenate the name of the program to each of the paths in the hard-coded path array and use, for example, the `stat` call to test whether the file exists.

Creating a pipe. To create a pipe you have to use the `pipe` system call. You have to pass an array of two integers as an argument, which will be filled with the file descriptors of the input and output end of the pipe. To reconnect standard input and output to the pipe, you will also need the `close` and `dup` system calls. Also useful are the defines `STDIN_FILENO` and `STDOUT_FILENO` that represent the file descriptors for `stdin` and `stdout` respectively.

6 Skeleton

You can use the skeleton below as a starting point for your shell program. As the program grows, you may want to improve the structure of the code by moving certain parts into separate functions.

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

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

    /* Parse input */
    while (( ... = strsep (...)) != NULL)
      {
        ...
      }
    /* If necessary locate executable using mypath array */

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