Operating Systems 2023
Assignment 1: Implementation of a System Program

Deadline: Friday, March 10, 2023, 18:00

1 Introduction

In the first couple of lectures of the course, we have seen how to interface with the operating system kernel through the use of system calls. We will explore this system call interface in this first assignment, to learn how programs can interact with and request services from an operating system kernel. Furthermore, we will learn how the kernel can signal events to a user-mode program.

The goal of the first assignment is to implement a system program that runs on POSIX systems and uses POSIX system calls. This year, we will be writing our own, simple text-based 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.

On POSIX systems, 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. As we have seen in the Programmeertechnieken course, composite commands (pipelines) can be created using these simple system commands and the pipe character (`|`), in which case the output of one command is sent to the input of another. As part of this assignment you will also implement support for handling pipes, so that your shell is capable of executing commands such as `cat Makefile | grep gcc`. The simple composite commands will be restricted to a single pipe. Furthermore, input/output redirections for commands will not be implemented as part of this assignment.

Finally, for achieving a grade beyond 8 you will need to implement a rudimentary form of job control that can handle suspending processes with `CTRL+Z`, bringing suspended processes to the foreground or background using the `fg` and `bg` commands, and listing currently active jobs with a `jobs` command.

2 Specification

The assignment is to implement a shell program that conforms to the specification given in this section. The program may be written in C or C++. Do note that you will have to use POSIX system calls, which are C functions that commonly expect C-style strings (!), even if you are programming your shell in C++. Your program should adhere to the following design specifications:

- 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.
- You *must* write a routine yourself to perform a search of the executable. This routine should only be used in case the name of the specified executable is not preceded by an absolute or relative path. You are *not* allowed to use `execlp` or `execvp` which automate this path search. Do not use `access`.

  Initially, you can add a hard-coded array of standard locations to your program (e.g. `./`, `/usr/bin`, `/bin` and `/usr/local/bin`), which will be searched.

  As a second step, read and parse the `PATH` environment variable (use `getenv`). This variable is a `:`-separated list of paths to search.

- Display an appropriate error if a requested command cannot be found or is not executable.
- Your program must be capable of executing commands and correctly pass the provided arguments to this command. This *must* be achieved using `fork`, `execv`, etc. system calls and not a higher level API.
- 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. In case the command contains more than one pipe character, simply log an error. Note that in order 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, we refer to the slides of the course Programmeertechnieken.

The rudimentary form of job control to be implemented consists of the following:

- Support a maximum number of jobs (say 16). Maintain the jobs using a data structure (e.g. an array). If the user attempts to start a new job when the maximum number of jobs has already been reached, simply refuse and output an error.
- Jobs are numbered. When a new job is started the smallest unused job number is assigned.
- The current job needs to be maintained using a stack. The job launched last is the current job. Do note that the job number of the current job may be *smaller* than the maximum job number in use.
- New jobs need to be started in a new progress group for signal handling to behave well (see `setpgrp` and `setpgid`).
- By suffixing a command with `&` the job is sent to the background upon startup (e.g. `xclock &`).
- Pressing `CTRL+Z` will suspend the current job on the foreground. This requires `SIGTSTP` to be handled and `SIGSTOP` to be sent to the current process.
- Pressing `CTRL+C` will terminate the current job on the foreground. But the shell must remain active! This requires `SIGINT` to be handled.
- `fg` should bring the current process to the foreground. `fg 4` will bring job number 4 to the foreground. A process (group) can be resumed by sending it the `SIGCONT` signal.
- `bg` should bring the current process to the background. `bg 3` will bring job number 3 to the background.
- `jobs` should list all active (running and suspended) jobs.
- Background jobs that are terminated should be cleaned up properly. To do so, you need to implement a `SIGCHLD` signal handler.

If in doubt, try out the features in an existing shell such as `bash` or `zsh` to get an idea how they should function.

3 Submission

You may work in teams of at most 2 persons. Your submission should consist of the source code of the implemented system program and a Makefile to build the program. *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, binaries and other unrelated files. Finally create a gzipped tar file of this directory:

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

Substitute `XXXXXXX` and `YYYYYYY` with your student IDs. Submit the tar-archive through the BrightSpace submission site. Also note your names and student IDs in the text box in the submission website. *Please, ensure only one team member submits the assignment, such that there is a single submission per team!*

Deadline: Friday, March 10, 18:00.

Notes:

- For those who need the weekend to finalize things: BrightSpace submission is open until Sunday, March 12, 23:59. After this time, submissions will be considered late. Note that no questions will be answered on the forum and mailing list after Friday 18:00.
- All source code that is submitted will be subjected to automatic plagiarism checks. Cases of plagiarism will be reported to the board of examiners.
- As with all other course work, keep assignment solutions to yourself. Do not post the code on public Git or code snippet repositories where it can be found by other students.
- In case you develop your shell on a macOS system (or Windows Subsystem for Linux), make sure to also test it on a Linux-system before handing in! Preferably test on the university Linux computers before handing in. In the case of disputes, the university Linux installation is used as reference.
- We may always invite teams to elaborate on their submission in an interview in case parts of the source code need further explanation.

4 Assessment

The maximum grade that can be obtained for this assignment is 10. The points are distributed as follows:

- **[1.0 out of 10]** Code quality
- **[2.5 out of 10]** Command input handling & built-in `cd` and `exit` commands
- **[2.0 out of 10]** Path handling / path search
- **[1.0 out of 10]** Execution of simple commands
- **[1.5 out of 10]** Execution of commands including pipes
- **[2.0 out of 10]** Job control implementation

For code quality the following is considered: good structure, consistent indentation, presence and quality of Makefile, comments where these are required.
Error handling and correct memory handling are considered for each of the different components.

5 Approach

The key to success is to implement the different features step by step and to not try to implement everything at once. So, start with the bare minimum, thoroughly test it and only after that move on to add the next feature. For example, the first steps could be:

1. Write a couple of functions to read commands to be executed from the terminal and properly tokenize these commands into argument vectors. Set up these argument vectors such that these can later be passed to `execv()`.
2. Write the functions required to find the binary of the command to be executed. So, you want to write a routine that turns `ls` into the full path `/bin/ls` (or `/usr/bin/ls` depending on your operating system).
3. Write a toy example program that starts another program using `fork()`, `execv()`, `wait()`.
4. Now, incorporate the functions you have written for command line parsing and path search into the toy example program.
5. Implement the `exit` and `cd` commands.
6. Implement pipelines.
7. and so on ...

6 Working with the POSIX API

You will have to use C 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 `execve`). In this section we give some hints and advice on dealing with C library functions in general, explain how to find more information about system calls and give some hints on which system calls to use in the assignment.

6.1 Dealing with C functions

Many of the library functions that you will have to use to complete this assignment are C functions. You can call C functions from a C++ program just fine, however, there are some things to keep in mind. Most notably, C functions do not support C++ reference parameters and cannot manipulate C++ data types such as STL containers and C++ strings. The latter is a problem that you will likely come across if you decide to implement this assignment in C++: you cannot pass `std::string` as argument to C library functions expecting strings, but must convert these to `char *` (for instance using the `.c_str()` method).

About every function in the standard C library is documented in the form of a manual (man) page. For example, to learn more about the C function `scanf` use the command `man scanf` in a terminal. 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. `man 2 fork` will display the manual page on the `fork` system call.

If you are writing a C++ program and the manual page tells you to include either `<stdio.h>`, `<stdlib.h>` or `<string.h>`, then make sure to include the C++ safe variants instead: `<cstdio>`, `<cstdlib>` or `<cstring>`.

6.2 Writing the entire program in C

You may implement this assignment in either C or C++. Because you will have to interface with POSIX functions, which are C functions that expect C-style strings, it might be wise to write the entire assignment in C. (Other than that, writing a program in pure C is also a fun exercise!).

If you choose to write the entire program in C, make sure to compile your code using a C and not a C++ compiler (use `.c` as extension and not `.cc` or `.cpp`). When using plain C, you cannot use C++ features such as classes, virtual methods and `cout` and `cin` for I/O. Instead of using `cin` and `cout`, you can use the `printf` and `scanf` functions. You need to include `<stdio.h>` for these functions.
To dynamically allocate memory, use the `malloc` and `free` functions instead of `new` and `delete`.

6.3 Overview of useful system calls

Reading and parsing user input. There are several ways to obtain input from the user. When using plain C, 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 properly deal with delimiters consisting of multiple spaces. If you are writing the program in C++ it is fine to use the string class for tokenization.

Executing a program. To start a new process and execute a program the `fork` and `execve` system calls should be all you need (like discussed in the lectures). Detailed information on these calls can also be found using man (e.g. `man 2 execve`). You may also consider using the front-end function `execv` described in man section 3.

When working with `execv` note that 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 you 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[1]. This result is used as first argument to the `execv` function, note that the first item of the argument vector must be left untouched.

Once a child process terminates it becomes a zombie. The parent must clean up such zombie processes using the `wait` or `waitpid` functions. These calls also allow you to collect the exit code of the child process.

Creating a pipe. To create a pipe you have to use the `pipe` system call. You need 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.

Working with POSIX signals. You will need to implement a number of POSIX signal handlers. Signal handlers are best installed using the `sigaction` function (see `man 2 sigaction`). Initialize a `sigaction` structure and call the `sigaction` function for the appropriate signal number (for example `SIGCHLD`). General information on POSIX signals can be found in section 7 of the manual: `man 7 signal`.

You can also send signals yourself. This is accomplished using the `kill` system call (`man 2 kill`) or `killpg` to send the signal to a process group.

A final warning concerns the possibility of race conditions. Any signal handler that you install may be invoked at any time. For example, while your program is running a function to manipulate your jobs data structure, a signal handler may be invoked interrupting the current function. In case this signal handler will update the same jobs data structure, the data structure will most likely be corrupted! This must be avoided! Using `sigaction` you can block signals from being delivered while a certain signal is being handled. Similarly, you may want to block signals while modifying your jobs data structure from outside a signal handler. You can temporarily block certain signals from being delivered using the `sigprocmask` call.

[1] It is true that the state of the file system can change between the call to `stat` and `execv`, so checking the return value of `execv` for errors remains important!