https://folk.ntnu.no/michaeng/tdt4186_21/
michael.engel@ntnu.no

Practical exercises
Spring 2021

Practical Exercises 4

Write a shell!

Please submit solutions on Blackboard by Thursday, 18.03.2021 12:00h

In this assignment, you have the opportunity to combine a number of topics you have learnt about in this course so far – such as process and memory management, I/O handling and redirection.
Your task is to implement a *simple* Unix shell called the "*woefully inadequate shell*" **wish**. wish should be an interactive shell with minimal functionality, that is nevertheless doing some useful things.
You should build wish in several steps as detailed below.

a. **Terminal I/O handling and input scanning** (2 points)

   The first thing a user sees after logging into a Unix system is the prompt of the shell. Accordingly, wish should also print a prompt – which prompt this is is left to your choice. Common examples are simple special characters such as '$' or '>'. Of course, your prompt can print more useful information, such as the current path of the shell.

   Start writing your wish implementation by writing a program that implements a loop. The loop body should display a prompt and then read a line of input from the terminal.

   After reading the input line, the shell's task is to split this line into separate *tokens*. So, for example, a command line such as

   ```
   ls -l /dev > /tmp/list
   ```

   is split into the tokens "ls", "-l", "/dev", ">" and "/tmp/list". Assume that tokens are separated from each other by whitespace (a space or tab character). The first token is always the name of the command to be executed (here: "ls"), after this come the parameters ("-l" and "/dev") and finally redirections of the form "> filename" and/or "< filename". The redirections can occur in any order.

   As the result of this first part, print a line that displays the command name, the command parameters, and possible I/O redirections that use the '<' (input redirection) and '>' (output redirection) characters.

   Hint: If you also participate in the compiler construction course, you are welcome to use lex to implement the scanner part. However, for this simple use case, you can also implement this either by hand or using the libc function `strtok(3)`. Note that strtok is quite an uncommon function, since it actually *modifies* the input string, so take care to read its man page thoroughly if you want to use it.

b. **Interpreting the command line** (2 points)

   After you have obtained a list of tokens for the entered command line, execute the command. Use `fork(2)` to create a child process and then start the entered command with one of the `exec(3)` functions. I would recommend to take a look at `execl(3)` for this shell, but feel free to use one of the alternatives if this works better for you.

   The shell itself (which becomes the parent process) should wait for the termination of the child process.

   Hint 1: Ignore I/O redirection for this part of the exercise.

   Hint 2: `exec` can fail, e.g. if you do not have execute permissions – check the return code and complain to the user if needed!

   Hint 3: If a command without a path name is given, only search for the command in the current directory (i.e., you do not need to implement the handling of a search path).

c. **Implement I/O redirection** (2 points)

Implement I/O redirection for your commands.

For output redirection, first create the destination file (if it already exists, you can choose to overwrite it if you have the permissions to or print an error in your shell) and then redirect the child process `stdout` channel using the `dup(2)` or `dup2(2)` system call.

For input redirection, first check if the input file exists and is accessible and then redirect `stdin` to this file.

You can ignore the `stderr` output channel.

Hint: Check the return value of `open(2)` (for the file to read or write, respectively) to find out if you can access it. You can simply print an error message if one of the files cannot be opened.

d. **Internal shell commands** (2 points)

The shell needs some *internal commands* to operate correctly. Try to find out for yourself why `cd` and `exit` have to be internal commands and would not work as expected when they are external programs called by `exit`.

Implement the internal command `cd /path/name` (where the parameter is either an absolute path starting with a "/" or a path relative to the current directory) to change the shell's current directory using the system call `chdir(2)` and the internal command `exit` to terminate your shell.

Hint: Internal commands have precedence before external commands. So if you detect `cd` or `exit` as command name, always use the internal version even if a file of that name exists in the current directory.

e. **Simple shell scripting** (2 points)

Simple shell scripts are lists of commands (without and control structures such as `if` or loops). Extend your shell to accept a command line parameter which indicated an optional shell script to execute *instead of* reading input from the terminal.