# Operating Systems Lab 2: Implementation of a shell and memory management

## 1 Implementing your own shell

In this exercise, you will build your own (simple) command shell. The shell must at a minimum support the following features:

- Start a program which can be found in the user's search path (`$PATH`).
- I/O redirection, for example: `a.out < in > out`
- Background processes, for example: `a.out &`
- Pipes, for example: `a.out | b.out | c.out`
- String parsing, for example `a.out "some string <> with 'special' characters"`

Any combination of the above should also be possible of course, for example `foo | bar < in > out` should put the contents of `in` as `stdin` in `foo`, while the `stdout` of `bar` should be redirected to `out`. Furthermore, `echo a & echo b & echo c` should print `a`, `b` and `c` on `stdout`, but the order is of course not defined (since the first two should run as background processes). Note the difference between `&` and `&&`, the first runs all but the last command in the background, while the second runs them in sequence. You do not have to implement `&&`. You can find an exact grammar of the syntax to accept down below.

Note that the shell should of course be able to run multiple commands successively! The shell should exit upon using the command `exit`.

Make sure to implement proper error/syntax checking. Print the following error notifications to `stdout` (not `stderr`), and of course don't forget the newlines!

- Invalid syntax, such as ending with < but not providing a filename: `Invalid syntax!`
- Input and output to the same file: `Error: input and output files cannot be equal!`
- Exiting before background processes finished: `There are still background processes running!`
- Command not found: `Error: command not found!`

As a starting point you can use the skeleton program from the Minix-book. You do not have to implement the listed `type_prompt()`.

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
      type_prompt( );                           /* display prompt on the screen */
      read_command(command, parameters);        /* read input from terminal */

      if (fork( ) != 0) {                       /* fork off child process */
          /* Parent code. */
          waitpid(-1, &status, 0);              /* wait for child to exit */
      } else {
          /* Child code. */
          execve(command, parameters, 0);       /* execute command */
      }
}
```

Note that the shell is a user level process, just like any other process (like an editor or a compiler). Therefore, you do not need to change anything in the OS its kernel (in fact, you are not allowed to). You will at least need some system calls, though. In particular, you will surely need the calls `fork`, `exec`, `open`, `close` and `dup`. Note that you need to spend extra care on termination of background process, in order to not flood the process table with zombie (defunct) processes.

You will once again submit your code to Themis. However, this time please include a makefile that produces an executable `shell` which can be run as `./shell`. All input will be on separate lines in `stdin`. Also, disable output buffering again using `setbuf(stdout, NULL);`. You are allowed to use flex as a lexer generator and/or bison as a parser generator for your shell implementation, but you are not required to do so. Any other lexer/parser generators are not installed on Themis, so you won't be able to use those.

Apart from the code, you are requested to also submit a report in which you briefly describe your design decisions, and the implementation of your shell. This does not have to be too long, and could also be a README file.

The grammar for the input lines to accept can be defined as follows. In here, a 'commandname' can be a built-in command such as `exit`, or a (path to an) executable. The 'options' part represents any set of parameters/strings that should be put into `argv` of the program.

$$
\begin{array}{lll}
\langle commandpart \rangle & ::= & \langle commandname \rangle\ \langle options \rangle \\
\\
\langle commandlist \rangle & ::= & \langle commandpart \rangle\ \texttt{|}\ \langle commandlist \rangle \\
 & | & \langle commandpart \rangle \\
\\
\langle io\_redirection \rangle & ::= & \texttt{<}\ \langle filename \rangle\ \texttt{>}\ \langle filename \rangle \\
 & | & \texttt{>}\ \langle filename \rangle\ \texttt{<}\ \langle filename \rangle \\
 & | & \texttt{>}\ \langle filename \rangle \\
 & | & \texttt{<}\ \langle filename \rangle \\
 & | & \langle empty \rangle \\
\\
\langle command \rangle & ::= & \langle commandlist \rangle\ \langle io\_redirection \rangle \\
\\
\langle inputline \rangle & ::= & \langle command \rangle\ \texttt{\&}\ \langle inputline \rangle \\
 & | & \langle command \rangle
\end{array}
$$

## Extensions

It is possible to implement some extensions in your shell, such as a prompt for each line (the `>`, or `[folder]>` you see in a normal shell), or built-in commands like `cd`, `jobs` (displaying a list of running background jobs) and `help`. You can also try to handle incoming signals, for example let `Ctrl+C` redirect to the child process and do not exit the shell itself, and make sure that all key combinations in `nano` work properly. You can also let `Ctrl+D` automatically exit your shell. Another nice addition would be support for colors.

Of course, you can get as crazy as you want. These additions can award you some bonus points. If you decide to implement some extensions, list them in your report and upload the necessary code in the separate entry on Themis.

## 2 A Simple Pipeline Program

Consider the following problem: A program is to be written to print all numbers, each on a separate line, between 1 and $n$ (inclusive) that are **not** (evenly) divisible by either 2 or 3. $n$ is given on `stdin`.

The problem is to be solved using three processes (P0, P1, P2) and two one-integer buffers (B0 and B1) as follows:

**1)** P0 is to generate the integers from 1 to $n$, and place them in B0 one at a time. After placing $n$ in the buffer, P0 places a sentinel (such as 0 or -1) in the buffer, and terminates.

**2)** P1 is to read successive integers from B0. If a value is not divisible by 2, the value is placed in B1. If the value is divisible by 2, it is ignored. If the value is the sentinel value, it is placed in B1 and P1 terminates.

**3)** P2 is to read successive integers from B1. If a value is not divisible by 3, it is printed to `stdout`. If the value is divisible by 3, it is ignored. If the value is the sentinel value, P2 terminates.

Write a program to implement P0, P1 and P2 as **separate processes** (not threads) and B0 and B1 as separate pieces of shared memory, each the size of just one integer. Use an appropriate mechanism, such as semaphores, to coordinate processing. Access to B0 should be independent of access to B1; for example, P0 could be writing to B0 while either P1 was writing into B1, or P2 was reading from it.

Once again, the code is to be submitted to Themis. Make sure to clean up all memory and to not leave any orphan processes behind! You may use `mmap` and POSIX semaphores, but you are explicitly not allowed to use busy waiting!