# Lab 5: Implementing a Command Interpreter (Shell)

You should read the introductory material from the task descriptions as part of the reading material.
Download the attached code in the task file and learn how to use it.

Thoroughly understand the following:

1. All the material from previous labs related to the C language, especially using pointers and structures in C.
2. Read the man pages and understand how to use the following system calls and library functions: getcwd, fork, execv, execvp, perror, waitpid, pipe, dup, fopen, close.

Basic introduction to the notion of fork can be found here and here.

## Attached code documentation

**LineParser:**

This package supports parsing of a given string to a structure which holds all the necessary data for a shell program.
For instance, parsing the string **"cat file.c > output.txt &"** results in a cmdLine struct, consisting of arguments = {"cat", "file.c"}, outputRedirect = "output.txt", blocking = 0 etc.

```
{
    char * const arguments[MAX_ARGUMENTS]; /* command line arguments (arg 0 is the command)*/
    int argCount;                /* number of arguments */
    char const *inputRedirect;  /* input redirection path. NULL if no input redirection */
    char const *outputRedirect; /* output redirection path. NULL if no output redirection */
    char blocking;      /* boolean indicating blocking/non-blocking */
    int idx;            /* index of current command in the chain of cmdLines (0 for the first) */
    struct cmdLine* next;       /* next cmdLine in chain */
} cmdLine;
```

---

Included functions:

1. **cmdLine* parseCmdLines(const char *strLine)**
   Returns a parsed structure cmdLine from a given strLine string, NULL when there was nothing to parse. If the strLine indicates a pipeline (e.g. "ls | grep x"), the function will return a linked list of cmdLine structures, as indicated by the **next** field.

2. **void freeCmdLines(cmdLine *pCmdLine)**
   Releases the memory that was allocated to accomodate the linked list of cmdLines, starting with the head of the list pCmdLine.

3. **int replaceCmdArg(cmdLine *pCmdLine, int num, const char *newString)**
   Replaces the argument with index *num* in the arguments field of pCmdLine with a given string.
   If successful returns 1, otherwise **-** returns 0.

---

# Lab 6: I/O Redirection and Pipes

## Standard Input/Output Redirection

Standard input and output are the "natural" feed (input) and sink (output) streams of a program: User input is read from the keyboard, and the program's output is displayed on monitor. However, in many cases one would like to read input from a file, rather than from keyboard. Similarly, one may wish to store the output in a file, rather than display it on monitor. Both requirements can be met by standard input/output redirection, without changing the code of the original program.

As a convention, input redirection is triggered by adding "<" after the invoked command. For instance, **"cat < my.txt"** tells the shell program to redirect the input of **cat** to file **my.txt**. As a result, cat displays the content of **my.txt** instead of displaying user feed from the keyboard. Output is triggered by adding ">", for instance: **"ls > out.txt"** which stores the list of files in the current directory in **out.txt**. Combining input and output redirection is also possible, e.g. **"cat < my.txt > yours.txt"**, which stores the content of **my.txt** in **yours.txt**.

How does the shell program achieve such an effect? Simply by closing the standard input stream (file decriptor 0) or the standard output stream (file decriptor 1), and opening a new file, which in turn is automatically allocated with the lowest available file-descriptor index. This effectively overrides stdin or stdout.

# Introduction to Pipelines

Pipelining is an extremely important concept in system programming. The name "pipeline" itself suggests that pipes are, somehow, involved. But what exactly is a pipeline? This is most easily explained by example, and in our case, by typing **"echo spider-pig | head -c 6"** in the Linux shell.

The output of this command, as you will notice, is the string **spider**. However, if you execute **"echo spider-pig"** and **"head -c 6"** separately, you see something quite different. The first command simply echos **spider-pig**, and the second command asks for user input, and prints the first 6 characters. One can easily deduce that when the two commands are chained using the "|" symbol, the output of **echo** is directly fed into **head**.

And this is where pipes come into play. The two processes communicate using a pipe which they both share: The standard output of **echo** is redirected to the pipe's "write-end", and the standard input of **head** is redirected to the "read-end". A pipeline, then, is simply a chain of processes where the standard output of one process feeds the standard input of the following process. The principle of pipelines easily generalizes to an unlimited number of processes. For instance, the command **"echo spider-pig | cat | head -c 6"** entails a pipeline consisting of three processes (**echo**, **cat** & **head**) and two pipes.