Labs 2: Reading Material

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:

All the material from previous labs related to the C language, especially using pointers and structures in C.

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.

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.

```
typedef struct cmdLine
{
  char * const arguments[MAX_ARGUMENTS];
  int argCount;
  char const *inputRedirect;
  char const *outputRedirect;
  char blocking;
  int idx;
  struct cmdLine* next;
} cmdLine;
```

Included functions:

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.

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.

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.

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 a 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 be from 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 descriptor 0) or the standard output stream (file descriptor 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.

Job Control

Processes belonging to a single command are referred to as a job or a process group. For example, ls|wc -l both the process that runs ls and the process that runs wc -l of this command belong to the same job/process group. Only one process group can run in the foreground and control the shell, the terminal's process group is set using tcsetpgrp. The rest of the process groups run in the background. Deciding which jobs run in the background and which run in the foreground is called job control. You can read more about job control in the following link: link1.

Signals

Signals are a used to send asynchronous events to a program such as ctrl-c, and ctrl-z. You can read more about signals here and here.

MAN: Relevant Functions and Utilities

fork(2), wait(2), waitpid(2), _exit, exit(3), close(2), open(2), read(2), write(2), getpid(2), setpgid(2), tcsetpgrp(3), tcgetpgrp(3), kill(2), signal(2), pipe(2).