Assignment 1
Internet Programming Course

Stefano Sandonà

Federico Ziliotto

19 settembre 2015

1   Answers to the Questions

Q-A: How many processes must your shell create when receiving a piped command? How many pipes? Until when must the shell wait to accept another command?

Q-B:Can you implement a shell program which only utilizes threads (instead of processes)? If your answer is yes, then write a thread-based version of `mys`, call it $mysh1_th$, and include it in your submission. If your answer is no, explain why.

Q-C: Can you use the `cd` command with shell `my`? Why?

2   Programs Descriptions

2.1   `mysh1`

This is the simple implementation of a shell that receives one command at a time (without arguments) and executes it. The source of the command is the standard input (`stdin`), which is the keyboard input stream. The `getline` function provides the entire content of one line of input from that stream (so it waits to return until the newline character is inserted). The content of the line is stored in the first argument that is passed, while the second one will contain the size of the buffer that is used to store the line.

The call to `strtok` searches for the start of the command (it skips characters like spaces and tabs) and returns a pointer to that. If no command is found (the newline is inserted

1

before any symbol that may be the start of a command) then the shell will ask for another input line.

If a command is found, first it's compared to the `exit` directive (to exit the whole program), then a new process is started with the `fork()` system call to execute the program with the name of the command. The call to `execvp()` takes the name of the command to execute and searches it in the directories listed in the **PATH** environment variable. When the child process has finished executing (or there was an error when calling `execvp`), it terminates. In the meantime, the parent process waits for the termination of any of its children with the invocation of `wait(NULL)`. In this case there is only one child to wait. After that, the parent return at the top of the cycle and waits for another command.

2.2   `mysh2`

To extend `mysh1` to accept arguments for the invoked commands, we used an array of pointers to the arguments. These pointer point towards the input string and are parsed using `strtok` to eliminate characters like spaces and tabs between each argument.

For the sake of simplicity and correctness, we allocate each time new memory to use to store the array of pointers (we use dynamic allocation because we don't know the number of arguments that may be used) and we free the memory at every new command read.

`mysh2` makes use of these two functions:
- `char **parseArguments(char *input, char *delim, int *size)`: takes the string in `input` and a character array of delimiters and returns an array of pointers to tokens that are obtained from the initial input by separating the tokens with the `delim`'s characters. The number of tokens is saved in the variable `size` and the function return the array of pointers to the tokens.
- `void deallocation(char **array, int size)`: this function takes an array of pointers to memory locations that were previously allocated with `malloc` and deallocates them with by invoking `free` for each pointer.

After the call to the function `parseArguments` has returned, in the array `args` we have:
- `args[0]`: contains the command to execute (the first token found);
- `args[1]...args[size-1]`: have pointers to the arguments of the command;
- `args[size]`: this is set to be NULL, since the call to `malloc` and `realloc` does not initialize the value of the new memory locations.

So the array `args` contains the command and the arguments for it, as required by the system call `execvp`, that is called by the child process to execute the command.

2.3 `mysh3`

2.4 `syn1`

2.5 `syn2`

2.6 `synthread1`

2.7 `synthread2`

2.8 `syn1.java`

2.9 `syn2.java`