O.S. ASSIGNMENT

Linux shell wrapper software

Ahmed Mohamed Abogabl

Table of Contents

1. Problem statement

The complete problem statement is in the assignment file attached with this report. But for a brief statement, we are required to implement a Linux shell upper layer software. This means that we will take shell commands from the user and feed them to the shell itself. But of course, the software should also add support some extra functionalities. The software should support:

- "exit" command; to exit the software.
- "&" trailing operator; to indicate execution in background.
- Some level of protection against misuse (i.e., empty commands).

NOTE: we very strongly recommend that you read the assignment file. We also assume some knowledge of Linux shell command.

2. Software design and block diagram.

2.1 Pipeline command treatment.

The software is designed as a pipeline containing different stages to allow higher modularity, separation, and ease of development and testing. Each command is feed and treated in this pipeline.

Now, we discuss each stage in detail.

2.1.1 Stage: Get Command.

This is the first and most straight forward stage. It simply gets the command text from the user. However, it does some small secondary tasks just for decoration and sorting. Like, it prints out a nice *"Shell>"* phrase at the beginning of each command just to let the user know we are ready to take the next command.

```
Shell>  ls
a.out   shell.c   test.c
Shell>
```

Next, it takes the actual command text. So, if the user enters *"ls"* and presses *Enter*, we should get some string like *"ls \n"*.

At last, we need to remove the trailing newline *"\n"* just for formatting concerns which will be elaborated later.

2.1.2 Stage: Convert Command.

We mentioned earlier that we need to feed the commands to the shell for command execution. However, the actual function that executes those commands require a specific format. For example, if the user input the following command: *"ls -l -s &"*. the formatted command should be and array like follows.

"ls"	"-l"	"-s"	NULL POINTER	&

This is exactly what it is being done in this stage, formatting the commands.

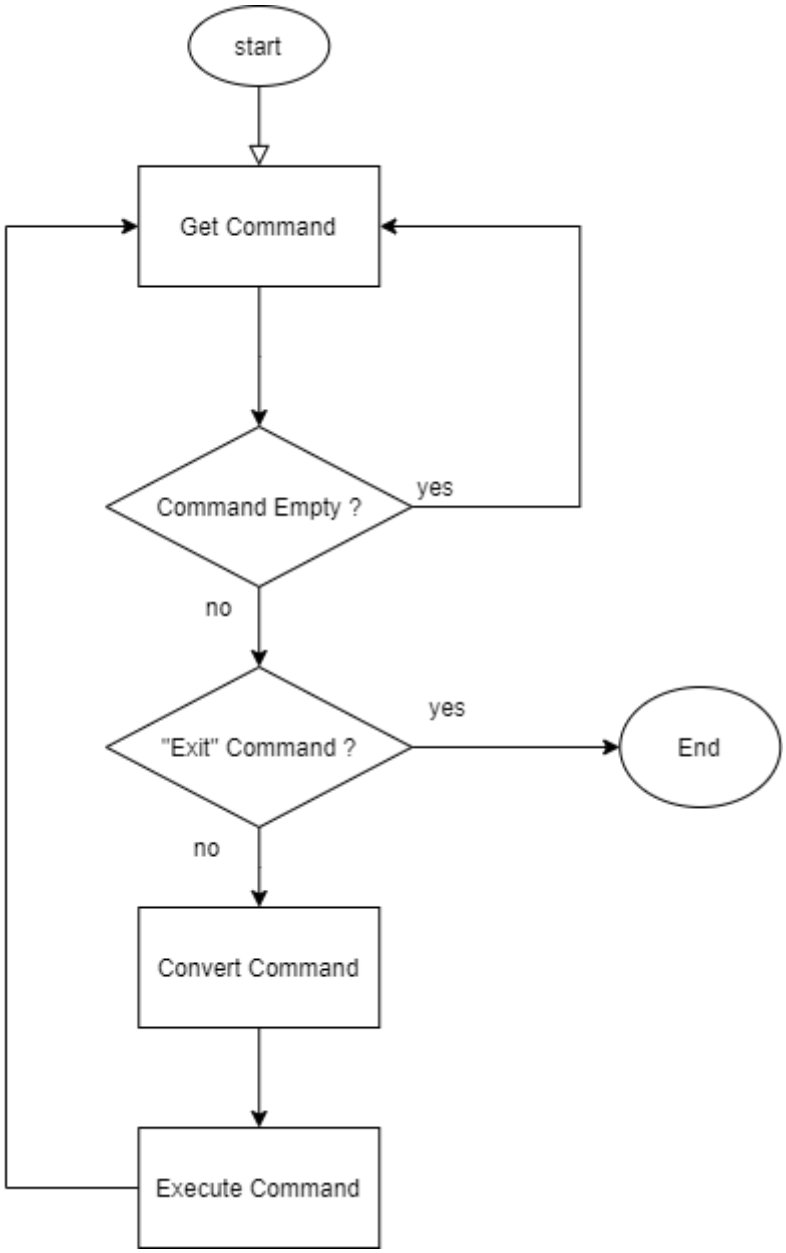
NOTE: the trailing “&” is arranged beyond the NULL pointer. That is because it is not considered an argument. Another reason is that it is easier to check for “&” in this format.

2.1.3 Stage: Execute Commands.

This the last stage in our pipeline. As the name suggests, this stage is where actual execution happens. This is where child processes are created and monitored.

2.2 Super Loop Architecture.

In previous section, we explained how a command is treated along the pipeline. However, the bigger picture involves dealing with more than one command and some condition must be continuously checked. A super loop design is used to feed successive commands into the pipeline.



Please note that this is a simplified overview. The complete flowchart shall be provided later in this report.

3. Sample Runs.

Please have a look at the video attached with this report for better software tutorial.

3.1 Minimal Foreground Command.

Foreground means that no “&” is appended to the command and therefore the software will wait for the completion of the command. In this sample we try the very simple two commands: “ls” and “pwd”.

```
Shell>  ls
a.out  shell.c  test.c
Shell>  pwd
/home/alpha/Desktop/dev/c_shell
Shell>
```

3.2 Empty and “Exit” Commands.

Note that when the user presses *Enter* on an empty line (i.e., empty command), he was presented with a new shell line just as expected. When the user enters the command “Exit”, the shell exited as intended for. Those two behaviors increase the stability and reusability of the software.

```
Shell>
Shell>
Shell>  exit
alpha@DESKTOP-O16LI2C:~/Desktop/dev/c_shell$
```

3.3 Commands with “&”.

In this sample, we elaborate the effect of appending “&” to the command. The command “*sleep*” is used to make a delay for a specified number of seconds. A command “*sleep 3*” makes a delay of three seconds.

It is better shown in the video than here. But when we do not append “&” to the command the shell waits for three seconds before prompting the user for a new command. However, when we do use “&”, the shell does not wait and immediately prompts the user for a new command.

```
Shell>  sleep 3
```

```
Shell>  sleep 3 &
Shell>
```

4. Appendix

4.1 source code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>

#define MAX_SIZE_CMD    256
#define MAX_SIZE_ARG    16

char cmd[MAX_SIZE_CMD];         // string holder for the command
char *argv[MAX_SIZE_ARG];       // an array for command and arguments
pid_t pid;                      // global variable for the child process ID
char i;                         // global for loop counter

void get_cmd();                 // get command string from the user
void convert_cmd();             // convert the command string to the required format by execvp()
void c_shell();                 // to start the shell
void log_handle(int sig);       // signal handler to add log statements

int main(){
    // tie the handler to the SGNCHLD signal
    signal(SIGCHLD, log_handle);

    // start the shell
    c_shell();

    return 0;
}

void c_shell(){
    while(1){
        // get the command from user
        get_cmd();

        // bypass empty commands
        if(!strcmp("", cmd)) continue;

        // check for "exit" command
            if(!strcmp("exit", cmd)) break;

        // fit the command into *argv[]
        convert_cmd();

        // fork and execute the command
        pid = fork();
        if(-1 == pid){
```

```
49              printf("failed to create a child\n");
50          }
51          else if(0 == pid){
52              // printf("hello from child\n");
53              // execute a command
54              execvp(argv[0], argv);
55          }
56          else{
57              // printf("hello from parent\n");
58              // wait for the command to finish if "&" is not present
59              if(NULL == argv[i]) waitpid(pid, NULL, 0);
60          }
61      }
62  }
63
64  void get_cmd(){
65      // get command from user
66      printf("Shell>\t");
67      fgets(cmd, MAX_SIZE_CMD, stdin);
68      // remove trailing newline
69      if ((strlen(cmd) > 0) && (cmd[strlen (cmd) - 1] == '\n'))
70              cmd[strlen (cmd) - 1] = '\0';
71      //printf("%s\n", cmd);
72  }
73
74  void convert_cmd(){
75      // split string into argv
76      char *ptr;
77      i = 0;
78      ptr = strtok(cmd, " ");
79      while(ptr != NULL){
80          //printf("%s\n", ptr);
81          argv[i] = ptr;
82          i++;
83          ptr = strtok(NULL, " ");
84      }
85
86      // check for "&"
87      if(!strcmp("&", argv[i-1])){
88      argv[i-1] = NULL;
89      argv[i] = "&";
90      }else{
91      argv[i] = NULL;
92      }
93      //printf("%d\n", i);
94  }
95
```

```
96  void log_handle(int sig){
97      //printf("[LOG] child proccess terminated.\n");
98      FILE *pFile;
99          pFile = fopen("log.txt", "a");
100         if(pFile==NULL) perror("Error opening file.");
101         else fprintf(pFile, "[LOG] child proccess terminated.\n");
102         fclose(pFile);
103 }
104
```

4.2 screenshot of processes