Desenvolvimento de um Mini Terminal Linux

Bárbara Zamperete Oliveira, Matheus Fellype de Moura Silva

Departamento de Ciência da Computação - Universidade Federal de Roraima (UFRR)
Boa Vista – RR – Brasil.

{Bárbara, Matheus}bahzamp25@gmail.com, matheusfellype772@gmail.com

Abstract. *This article aims to present the functionality of the mini terminal shell, such as, what it can do, it's implementation for how it can perform, among others.*

Resumo. *Este artigo tem como finalidade apresentar as funcionalidades do mini terminal shell, tais como, o que ele consegue fazer sua implementação para como ele consegue executar entre outros.*

1. O que é o Shell?

Para começarmos, falaremos brevemente sobre o que é o shell. Ele é o interpretador de comandos e que irá funcionar como a interface entre o usuário e o sistema operacional. Como falamos, essa interface intermediará o usuário e o sistema operacional através de linhas de comandos. Com isso, sua funcão é ler a linha de comando, interpretar, executá-lo e devolver o resultado pelas saídas.

Para esse trabalho, criamos um mini terminal shell chamado shellso.

2. Inicialização e Captura do comandos do usuário

Partindo para explicação das funções do shell, vamos começar pela inicialização do shell. No main, declaramos um char chamado *user_imput* de tamanho *MAX*, este definido como 100, ela irá armazenar o comando utilizado pelo usuário. Logo seguida a função *init_shell* é chamada.

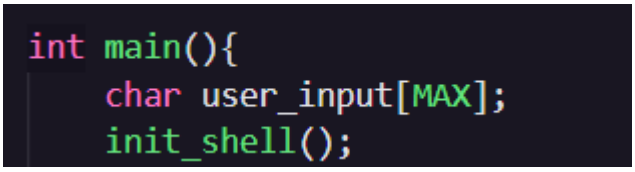

Figure 1. Main-1

A função *init_shell* é uma simples função que mostrará um cabeçalho de início do shell.

```c
void init_shell(){
    printf("\n******************\n****NOSSO SHELL****\n******************\n");
}
```

Figure 2. InitShell

Para a captura dos comandos digitados pelo usuário, utilizamos a função *get_input*, que irá receber como argumento o *user_input,* que já foi comentado. Mas antes disso, uma função chamada *print_msg* irá mostrar a inicialização do prompt com a mensagem "Sim, mestre?".

```c
print_msg();
get_input(user_input);
```

Figure 3. Chamadas

A função *print_msg* é uma função que irá mostrar uma mensagem toda vez que o usuário for digitar um comando.

```c
void print_msg(){
    printf("Sim, mestre? ");
}
```

Figure 4. PrintMsg

A função *get_input* irá capturar o comando que o usuário digitar e ele usará como argumento o char *user_input.*

```c
void get_input(char* user_input){
    scanf(" %[^\n]", user_input);
}
```

Figure 5. GetInput

3. Comandos Builtins

O mini terminal shell possui alguns comandos builtins. Aguns deles são: fim, cd, help, hello, ls. Esses comandos são tratados pela função *exe_command. E*la irá lidar com a execuções desses comandos de acordo com a opção usuário. Para isso primeiramente, criamos um ponteiro de char chamado *list_comd* de tamanho 5, onde em cada posição estará um dos comandos citados.

```c
int  exe_command(char* token, char* arg){
    char *list_cmd[5];
    char *username;

    //quais serão os comandos??
    list_cmd[0] = "fim";
    list_cmd[1] = "cd";
    list_cmd[2] = "help";
    list_cmd[3] = "hello";
    list_cmd[4] = "ls";
```

Figure 6. ExeCommand

Essa função trata o gerenciamento de qual comando irá ser executado através das condições *if/else if*.

```c
if(strcmp(token, list_cmd[0]) == 0){
    printf("\nAdeus Mestre\n");
    exit(1); //termina o programa
    return 1;

}else if(strcmp(token, list_cmd[1]) == 0){
    chdir(arg);
    return 1;

}else if(strcmp(token, list_cmd[2]) == 0){
    printHelp();
    return 1;

}else if(strcmp(token, list_cmd[3]) == 0){
    #ifdef __linux__
    username = getenv("USER");
    #elif defined _WIN32
    username = getenv("USERNAME");
    #else
    printf("ERROR");
    #endif

    printf("Hello %s, are you ok?\n", username);
    return 1;

}else if(strcmp(token, list_cmd[4]) == 0){
    DIR *dirp;
    struct dirent *dp;
    struct stat statbuf;
    char cwd[1024];
    getcwd(cwd, sizeof(cwd));
    dirp = opendir(cwd);
    while((dp = readdir(dirp)) !=NULL){

        stat(dp->d_name, &statbuf);
        printf("%s\n", dp->d_name);
    }
    return 1;

}else{
    return 0;
```

Figure 6. Comandos Builtins

4. Pipes

O pipe é uma das maneiras que o shell pode utilizar para comunicar processos, ou seja, ele faz um encadeamento de processos, e esse encadeamento de processos pode ser ativado quando o usuário usa o comando "|".

No shellso, para trabalhar com pipes, ele possui duas funções: o *findPipe* que chamamos na função main e o *exePipe.*

```
    //ver se tem pipe
    findPipe(user_input);
```

Figure 7. Main-findPipe

Primeiro ele irá chamar a função *findPipe* passando como argumento o *user_input.* Logo quando ela entrar na função, irá ser criado dois ponteiros de char chamados *pipes,* que irá pegar a segunda parte do comando, ou seja, depois do "|"; e o *command,* que irá pegar a primeira parte. Se houver pipes, ele entrará na condição, irá tratar os espaços e chamará a segunda função.

```
void findPipe(char *user_input){
    char *pipes;
    char *command;
    command = strtok(user_input, "|"); //pega a primeira pa
    pipes = strtok(NULL,""); //pega a segunda parte

    if (pipes){
        char *listpipes[MAX];
        char *listcmd[MAX];
        //retirar espaços
        listcmd[0] = strtok(command, " ");
        listpipes[0] = strtok(pipes, " ");
        for (int i=1; i<MAX;i++){
            listcmd[i] = strtok(NULL," "); //resto
            listpipes[i] = strtok(NULL, " ");
            //OBS: tem q colocar uma condição de parada par
            //ele não precisar procurar até o fim de MAX
        }
        //executa pipe
        exePipe(listcmd, listpipes);

    }else{
        //tirar espaço
        char *arg[MAX];
        int isbuiltin = 0;
        arg[0] = strtok(command, " "); //primeira parte
        for (int i=1; i<MAX;i++){
            arg[i] = strtok(NULL," "); //resto
        }
        //ve se é um comando builtin
        isbuiltin = exe_command(arg[0], arg[1]);
        if(!isbuiltin){
            //executa comando
            simplesCMD(arg);
        }
    }
}
```

Figure 8. FindPipe

Quando ele chamar a segunda função, *exePipe,* ele irá executar os pipes utilizando a função fork, para fazer as execuções dos children e parent.

```c
void exePipe(char** listcmd, char** listpipe)
{
    // 0 is read end, 1 is write end
    int pipefd[2];
    pid_t p1, p2;

    if (pipe(pipefd) < 0) {
        printf("\nPipe could not be initialized");
        return;
    }
    p1 = fork();
    if (p1 < 0) {
        printf("\nCould not fork");
        return;
    }

    if (p1 == 0) {
        // Child 1 executing..
        // It only needs to write at the write end
        close(pipefd[0]);
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);

        if (execvp(listcmd[0], listcmd) < 0) {
            printf("\nCould not execute command 1..");
            exit(0);
        }
    } else {
        // Parent executing
        p2 = fork();

        if (p2 < 0) {
            printf("\nCould not fork");
            return;
        }

        // Child 2 executing..
        // It only needs to read at the read end
        if (p2 == 0) {
            close(pipefd[1]);
            dup2(pipefd[0], STDIN_FILENO);
            close(pipefd[0]);
            if (execvp(listpipe[0], listpipe) < 0) {
                printf("\nCould not execute command 2..");
                exit(0);
            }
        } else {
            // parent executing, waiting for two children
            wait(NULL);
            wait(NULL);
        }
    }
}
```

Figure 9. ExePipe

7. References

Leonardo Xavier. (2004) “Usando Pipe”, https://www.vivaolinux.com.br/dica/Usando-o-pipe, Dezembro.

Pedro Muxfeldt. (2017) “Linux - O Shell”, https://br.ccm.net/contents/320-linux-o-shell, Dezembro.

Geeks for Geeks. (2020) “Making your own Linux Shell in C”, https://www.geeksforgeeks.org/making-linux-shell-c/?ref=lbp, Dezembro.