# Desenvolvimento de um mini terminal Linux

Josemar rocha, Acadêmico, UFRR e Pedro Aleph, Acadêmico, UFRR

***Resumo*—Este artigo Tem o objetivo de explicar o básico funcionamento de um terminal, o que ele faz, e como ele faz, com uma implementação básica.**

***Index Terms*—interpretador de comandos, pipes, shellso, prompt, background.**

## I. Introdução

INTERPRETADORES de comando são uma forma do usuário se comunicar com o kernel da máquina, de forma que ele pode executar comandos simplesmente digitando as palavras chaves, o terminal(interpretador) que é também chamado de shell, irá receber estas palavras como entradas e trata-las de forma a reconhecer o que foi lido e assim saber o que deve ser requerido para o núcleo da máquina para executar o comando e retornar o resultado para o usuário (ou retorna um erro caso não seja possivel devido a algum motivo). Pode se dizer de forma geral que eles tem a mesma essência que sistemas operacional, e pode ser usado dentro deste, mas já existiam antes, já era a forma mais básica de se interagir com a máquina, ou seja, apenas uma linha de comando na tela, sem interface gráfica.

Boa Vista/RR
27 de junho, 2019

## II. Definindo um mini terminal

O objetivo é a implementação de um interpretador de comandos na linguagem C, chamado shellso, sem usar recursos do C++, devendo criar processos e encadeados usando pipe, também usando este último em conjunto de manipulação de processos para criar um par produtor/consumidor por envios de mensagens dentro do shellso, alimentando dados disparados ou recebendo os dados de um programa. O programa (shellso) inicia sem argumentos com uma mensagem no prompt dizendo "Qual o seu comando?", e recebe a entrada dada pelo usuário e depois na saída padrão(salvo redirecionamento de pasta) o resultado do programa ou uma mensagem de erro caso não tenha sido possível executar com sucesso o comando por algum motivo. teclado). Cada linha deve conter um comando ou mais comandos para ser(em) executado(s). No caso de haver mais de um programa a ser executado na linha, eles devem obrigatoriamente ser encadeados por pipes (“|”), indicando que a saída do programa à esquerda deve ser associada à entrada do programa à direita. Um novo prompt só deve ser exibido (se necessário) e um novo comando só deve ser lido quando o comando da linha anterior terminar sua execução, exceto caso a linha de comando termine com um “”, quando o programa deve ser executado em background. O shellso aceita simbolos "=>"e «= "como comandos de redirecionamento indicando entrada e saída por pipes para processos produtores/consumidores, e só termina quando encontra o comando "fim"no inicio da linha da entrada.

## III. requisitos

Para tornar tudo possível devemos utilizar bibliotecas que possibilitam manipulação de caracteres para ler e tratar os dados da entrada, criação e uso de processos e colocando pipe, além alocação na memória. Assim temos:

*1) bibliotecas e seus recursos:*

- #include <sys/wait.h>
  - waitpid() and associated macros
- #include <unistd.h>
  - chdir()
  - fork()
  - exec()
  - pid_t
- #include <stdlib.h>
  - malloc()
  - realloc()
  - free()
  - exit()
  - execvp()
  - EXIT_SUCCESS, EXIT_FAILURE
- #include <stdio.h>
  - fprintf()
  - printf()
  - stderr
  - getchar()
  - perror()
- #include <string.h>
  - strcmp()
  - strtok()

∈ [1]

Mostraremos as funções sendo usadas na próxima sessão.

## IV. shellso - principais pontos do código

Primeiramente devemos tratar a entrada de modo a separar os argumentos que serão depois verificados.

Professor Hebert Rocha do departamento de ciência da computação, UFRR.

```c
//le o input de string de quantidade desconhecida e retorna um array char
char *inputString(FILE* fp, size_t size){
    char *str;
    int ch;
    size_t len = 0;
    str = realloc(NULL, sizeof(char)*size);
    if(!str)return str;
    while(EOF!=(ch=fgetc(fp)) && ch != '\n'){
        str[len++]=ch;
        if(len==size){
            str = realloc(str, sizeof(char)*(size+=16));
            if(!str)return str;
        }
    }
    str[len++]='\0';
    return realloc(str, sizeof(char)*len);
}
```

Aqui ele recebe todo input e retorna todo um array char do tamanho do input, contendo todo o input.

```c
if(strcmp(line, "fim")==0){
    exitv++;
}else if(strcmp(firstTwoLetters, "cd")==0){
    if(strlen(line) == 2){
        const char* s = getenv("HOME");
        chdir(s);
    }else{
        char dir[strlen(line)-3];
        memcpy(dir, &line[3], strlen(line));
        chdir(dir);
    }
}else if(strcmp(args2[0], "status")==0){
    if(lastforeground != 1){
        printf("valor de saida %d\n", exitstatus);
```

Aqui um exemplo de como ele detecta alguns comandos no input, verificando se todo o input eh igual a fim, pra finalizar a shell, ou verificando se as duas primeiras letras são iguais a cd para executar esse comando.

A chamada do sistema fork é chamada, o sistema operacional faz uma duplicação do processo e inicia ambos executando. O processo original é chamado de "pai"e o novo é chamado de "filho". fork () retorna 0 ao processo filho e retorna ao pai o número de identificação do processo (PID) de seu filho. Em essência, isso significa que o único caminho para novos processos é começar por um existente que se duplica,mas isso pode soar como um problema. Normalmente, quando você deseja executar um novo processo, você não quer apenas outra cópia do mesmo programa - você quer executar um programa diferente. Isso é o que a chamada de sistema exec () tem tudo a ver. Ele substitui o programa atual em execução por um inteiramente novo. Isso significa que, quando você chama exec, o sistema operacional interrompe seu processo, carrega o novo programa e inicia aquele em seu lugar. Um processo nunca retorna de uma chamada exec () (a menos que haja um erro).[1]

```c
//coloca o que sera executado no comando execv em um vetor chamado exec_string

        char *exec_string[size];
        i = 0;
        if(ignored == 1){
            background = 1;
        }
        for(i=0; i < size-background; i++){
            exec_string[i] = OS_string[i];
        }
        exec_string[size-background] = NULL;
        if(ignored == 1){ background = 0; }
        pid_t spawnpid = -5;
        int childExitMethod = -5;
        int ten = 10;
        result = 0;
        int backgroundpid = 99;
        spawnpid = fork();
        int spaces2 = 0;
        if(skip == 1){
            sourceFD = open("/dev/null", O_WRONLY);
        }
```

O seguinte mostra como o processo pai pode controlar a criança, esperando ela terminar de executar usando waitpid() , e depois destruindo processo com kill.

```c
default: //classe pai
    if(background == 1 ){
        printf("background pid eh %d\n", spawnpid);
    }else if(background == 0){
        waitpid(spawnpid, &childExitMethod, 0);
        kill(spawnpid, SIGKILL);
    }
    break;
```

```
//acha i_place (onde <= ocorre") e o_place (onde => ocorre)

        i = 0;
        int i_place = 0;
        int o_file = 0;
        int o_place = 0;
        int OS_string_end = 0;
        for(i=0; i<= spaces; i++){
            if(strcmp(args2[i], "<=")==0){
                i_place = i;
            }else if(strcmp(args2[i], "=>")==0){
                o_place = i;
                o_file = i+1;
            }
        }
```

Aqui são definidos "=> "e «= "como sinais de redirecionamento.

```
//faz redirecionamento de I/O se necessario
        int sourceFD = 0;
        int targetFD = 0;
        int result = 0;
        int stat = 0;
        if(i_place != 0){

            if(access(OS_string[i_place+1], F_OK)!= -1){
            }else{
                printf("nao pode abrir %s para o input\n", OS_string[i_place+1]);
                skip = 1;
            }

            stat = 1;
            sourceFD = open(OS_string[i_place+1], O_RDONLY);
            OS_string_end = i_place;
            if(o_place != 0){
                stat = 2;
                targetFD = open(OS_string[o_place+1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
            }
        }else{
            if(o_place == 0 && i_place == 0){
                stat = 3;
            }else{
            if(o_place != 0){
                    stat = 4;
                        targetFD = open(OS_string[o_place+1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
                }
            }
        }
```

Aqui são condições para o redirecionamento, caso necessário, é colocado em um source e um target.

```
//targetfd e sourcefd foram setados dependendo se a string tem input, output

        int size = 0;
        if(i_place != 0 && o_place != 0){
            size = i_place;  //input and output
        }else{
            if(i_place != 0 && o_place == 0){
                size = i_place; //input and no output
            }else if(i_place == 0 && o_place != 0){
                size = o_place; //only output
            }else if(i_place == 0 && o_place == 0){
                size = spaces;
            }
        }
        int background = 0;
```

É colocado o tipo de redirecionamento, input, output ou combinação dos dois.

E por fim o redirecionamento é dado ao processo filho nos tipos.

```
case 0:   //caso filho

if(background==0){
    spaces2 = spaces;
    if(stat == 1){ //somente redirecionamento de inputs
        result = dup2(sourceFD, 0);
        if(skip == 1){

        }
        if(result ==-1){
            perror("source dup2()");
            exitstatus = 1;
        }
    }
    if(stat == 2){ //redirecionamento de ambos input e output
        result = dup2(sourceFD, 0);
        if(result ==-1){
            perror("source dup2()");
            exitstatus = 1;
        }
        result = dup2(targetFD, 1);
        if(result ==-1){
            perror("target dup2()");
            exitstatus = 1;
        }
        OS_string[1] = NULL;
        if(sourceFD == -1){
            perror("source open()");
            exitstatus = 1;
        }
        if(targetFD == -1){
            perror("target open()");
            exitstatus = 1;
        }
```

## V. Conclusão

Até para para fazer o que seria um simples terminal com funções básicas, pode ser tornar trabalho por ter que lidar com várias funções de processos, de modo que todas as funções mesmo que individualmente sejam simples de serem usar, precisando estarem juntas em várias partes das coisas que o terminal deve realizar, Este trabalho foi importante para termos noção de que uma ideia simples pode ser complicada por baixo dos panos, apesar de ser trivial. E nem todos os requisitos da definição foram concluídos.

## Acknowledgment

Este é um projeto final final da disciplina de sistemas operacionais do curso ciência da computação da UFRR, vale deixar claro que o que fizemos aqui não é nada novo, somente mostra como é um terminal de forma técnica no seu funcionamento, não implementamos nosso próprio mini terminal, somente usamos tutorial com exemplos de códigos, e adaptamos para se encaixar no que é requerido.

## Referências

[1] Stephen Brennan, *Tutorial - Write a Shell in C*. https://brennan.io/2015/01/16/write-a-shell-in-c/ [Acessado 25-06-2019].

[2] Martin Barker, *Simple-Shell-in-C*.https://github.com/MartinBarker/Simple-Shell-in-C [Acessado 26-06-2019].