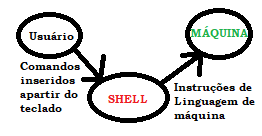
1. **Introdução**

Uma Shell é um programa que pega as entradas de comandos escritos a partir do teclado do teclado do usuário e passa estes mesmos comandos para a máquina para executar pelo Kernel. A Shell também verifica se os comandos inseridos estão correctos. Portanto, de forma geral, a Shell é uma interface para o usuário usar os serviços de um computador. Que pode ser uma interface de linha de comando – que para o nosso caso, será a interface que iremos construir – ou uma interface gráfica, tal como os softwares regulares, Windows Office, Adobe PDF, entre outros.

Fig. 1 – A Shell é a intermediária entre o Usuário e o Kernel e a Máquina



Desta feita, propôs-se criar uma Shell simplificada para o terminal do Linux em linguagem C, de modos a consolidar os conhecimentos adquiridos durante todo semestre da cadeira de Sistemas Operativos 2, desta feita foi nos parametrizada a estrutura com as funcionalidades que a Shell deve ter como é descrita abaixo:

* Execução de qualquer comando em primeiro plano (foreground) e em segundo plano (background)
* Ignorar Sinais
* Redirecionamento de Entrada e Saída
* Histórico dos Últimos Comandos Executados nesta Sessão
* Execução de Comandos Executados Anteriormente

1. **Implementação**

Para efeitos de estilizar e realizar algo com um nível de estética aceitável, decidimos implementar um output de apresentação da Shell, informando o nome do usuário que está logado, que para o efeito foi usado a função getenv(“USER”) que retorna uma String e pertence a biblioteca unistd.h, que contém todos os comandos e funções padrão do Unix, por outra foi implementado um formato do comando printf(), o printf("\033[H\033[J"), para deixar a tela limpa logo após mostrar o output e para que o usuário tenha tempo de ver o output foi usado o comando sleep.

Output:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Shell Simplificada\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*Usuário Logado: “Nome do Usuário”\*\*\*\*

\*\*\*\*\*Shell Desenvolvida pelo Grupo nº: X\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Para a implementação, o primeiro passo usado foi criar um loop infinito que sempre estará preparado para receber qualquer comando que o usuário digitar no terminal. Para tal, foi conveniente usar o comando de repetição **While** da Linguagem C, sendo o seu uso bastante simples pois apenas recebe um parâmetro que será a condição de paragem ou continuação, mas no entanto, ao em vez de colocar a condição de paragem como argumento foi bastante conveniente usá-lo como um comando qualquer que deve ser verificado e caso for inserido **exit** como comando, o loop é desfeito, foi implementado de acordo a seguinte estrutura:

While(VERDADEIRO)

{

Insira um Comando

if(Comando igual á exit)

Saia da Shell

......................................

......................................

}

Como já foi definida uma estrutura de repetição com uma condição de paragem, agora serão implementadas estruturas que satisfaçam os parâmetros pré-definidos para a correta execução do trabalho, mas primeiramente precisa-se ter em mente que como funcionam os comandos, os argumentos e os ficheiros padrão de entrada e saída de dados.

* 1. **Entrada e Saída de Dados**

Para os ficheiros padrão de entrada e saída de dados, as funções encontram-se na biblioteca **stdio.h** (Standard Input Output). É uma biblioteca que contém as funções para os processos de Input e Output do Sistema. Por exemplo, ela tem as funções para ler comandos inseridos pelo usuário e escrever na tela do **Terminal** resultados desta requisição. Na inicialização do sistema, três ficheiros de dados são abertos: **stdin** (Standard Input), **stdout** (Standard Output) e **stderr** (Standard Error) e por padrão já existem posições pré-definidas, os **File Descriptor** que na verdade são posições, essas posições indicam na verdade posições de memória, essas posições são vinculadas com algum ficheiro qualquer, de modos a serem rápida e facilmente identificadas, é como se fossem ponteiros em C, cada posição de File Descriptor aponta para um ficheiro em específico, nunca existirão duas posições iguais, apenas existirão posições diferentes que apontam para um mesmo ficheiro.

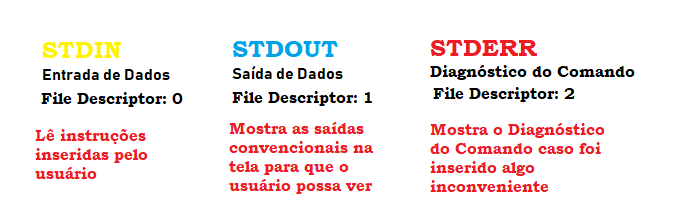


Fig. 1 – Descrição dos ficheiros padrão

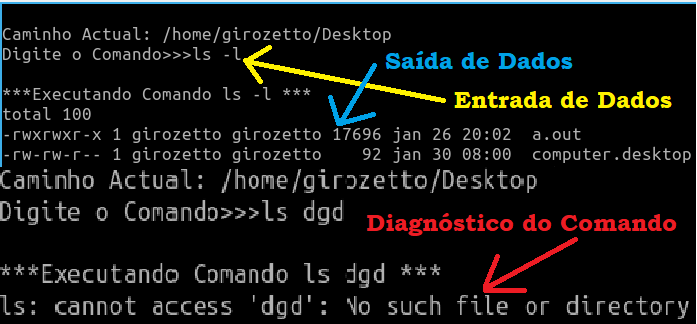


Fig. 2 – Shell executando um comando de forma correta e outro executando de forma incorrecta

* Na entrada de dados, a Shell diz: Por favor insira o que precisas.
* Na saída de dados, a Shell diz: Está aqui o que tu precisas
* No diagnóstico do comando, a Shell diz: Olá, tem algo de errado com este comando que inseriste.

Consoante estaremos avançando com a implementação, este será o tema mais usado explícita ou implicitamente pois para cada ação feita terá sempre um output ou um input ou um erro.

* 1. **Funcionamento dos Comandos**

Um comando na verdade é um programa, que realiza certo tipo de ação específica, mas em uma Shell, não basta saber apenas os nomes dos comandos, precisa-se saber também quais argumentos, parâmetros ou também chamados de modificadores podem ser inseridos para um comando em específico, por exemplo o comando “ ls ” por si só retorna um output mostrando uma grelha dos ficheiros encontrados no diretório atual, mas adicionando o parâmetro “-l ” como é mostrado na Fig. 3, o seu output é diferente, pois mostra muito mais detalhes e mostra em forma de lista organizada de todos ficheiros existentes no diretório atual, já se fosse usado um diretório além do parâmetro, neste caso, por exemplo “ ls –l /home ”, mostraria o mesmo tipo output que a situação anteriormente mencionada, a única diferença é que mostraria o que estaria dentro do diretório “/home”. Para ter informações sobre os parâmetros a usar ou informações sobre os próprios comandos, basta usar a seguinte sintaxe no Terminal: nome\_comando --help, e será mostrado a lista de todos os parâmetros que podem ser usados.

* 1. **Mostrar Diretório Actual**

Como forma de ajudar na execução do comando **cd** do terminal e na localização do diretorio em que o usuário se encontra sempre que executar um comando, decidimos usar o comando **getcwd()** da biblioteca **unistd.h**, sendo os seus parâmetros uma string onde irá ser inserido o caminho actual e o tamanho da string em bytes que pode ser obtido pela função **sizeof()**, o resultado poder ser observado na Fig. 2.

* 1. **Ignorar Sinais**

Foi pré-definido no projeto que será necessário que devemos ignorar o sinal SIGINT, que é executado sob a combinação **Ctrl+C** este sinal que serve para cancelar a execução da Shell, para tal usamos os comandos da biblioteca **signal.h**, usamos a **struct sigaction**, onde definiremos o atributo **sa\_handler**, que é na verdade um tratador de eventos, que será uma função do tipo void, o que faremos caso o usuário usar o sinal? O **handler** responde a esta questão, para o nosso caso foi uma função que imprime um output dizendo: “Eu não quero ser morto” de logo a seguir para dar o ultimato na definição, usar a função **sigaction()**, que tem como parâmetros, a flag do sinal que para o nosso caso é o SIGINT, a struct sigaction definida anteriormente e o parâmetro NULL.



Fig. 3 – Foi usada a combinação Ctrl+C e mostra a informação “Eu não quero ser morto”

* 1. **Execução de Comandos**

Sabe-se que teremos de estar constantemente comparando se um comando é equivalente a “!!” e “historia”, que representam respectivamente a execução do comando anterior e histórico dos comandos executados, para tal foi usado a função **strcmp()** que compara duas strings, se forem iguais retornar 0, se a primeira string menor que a segunda retornar -1 e no caso contrário retornar 1.

Cuja sintaxe é:

strcmp(char string1[],char string2[]);

Para a implementação das execuções dos programas, tivemos que usar as bibliotecas **stdlib.h**, **string.h**, **unistd.h** e a **sys/wait.h**, como foi nos definido que não poderíamos usar a função system, teríamos de usar unicamente uma das funções **exec**. Como já foi mencionado anteriormente no funcionamento dos comandos, podem existir n-argumentos num só comando e para tal decidi usar unicamente um vector de strings para dividir para cada posição de memória, uma string contendo um parâmetro ou comando, mas para o caso da usarmos um vector de strings é bastante conveniente a função **execvp()**, pois os seus parâmetros são: uma string que contém o nome do comando e um vector de strings que contém cada argumento que estiver separado por um espaço ou mais incluindo o próprio nome do comando, onde o nome do comando será sempre a primeira posição do vector.

Terá a seguinte sintaxe:

execvp(char comando[], char argumentos[][]);

Sabe-se que quando nós inserimos um comando, inserimos ele na íntegra, como é apenas uma string, para podermos separar cada argumento da string e coloca-la num vector de strings devemos primeiramente criar uma função usando a função **strsep()** da biblioteca **string.h** para poder separar cada elemento separado por um espaço da string e atribuir num vector de strings, a função strsep() recebe como parâmetros a string que deseja dividir e o delimitador que no nosso caso é o espaço entre os elementos e a função retorna um ponteiro de Char\* que pode ser mandada para uma posição do vector de strings á cada iteração, para tal foi usado uma estrutura de repetição for().

Terá a seguinte sintaxe:

buffer(char comando[],char separador[]);

No contexto geral, para executar funções de execução de comandos do terminal, necessita-se criar um processo filho que irá executar dita função e poder controlar o comportamento de cada comando executado no terminal. Mas para isso temos de usar o comando **fork()** que retorna um inteiro maior do que zero caso for o processo pai, retorna zero caso for filho e retorna um valor menor do que zero caso não conseguir criar o processo. Para poder esperar o processo filho terminar basta usarmos o comando **wait() ou waitpid()** no processo pai. Usando a seguinte sintaxe:

int processo = fork();

if(processo<0)

//erro

if(processo==0)

//Instruções do processo filho

else

//Instruções do processo pai

//Instruções para o pai e para o filho

Para um contexto específico constatamos que um dos comandos do terminal não é possível executar pela função **execvp(),** o comando **cd,** quando precisamos estar constantemente nos mudando de diretório em diretório, para resolver isto, decidimos usar o comando **chdir(),** que recebe um único parâmetro, uma string contendo o caminho que se deseja ir. Para tal,verificamos primeiramente o primeiro elemento do vector de strings para poder verificar se é igual ao comando **cd** usando o **strcmp()** e usamos o segundo elementos do vector como argumento do **chdir()**.

Tendo a sintaxe:

chdir(char caminho[]);

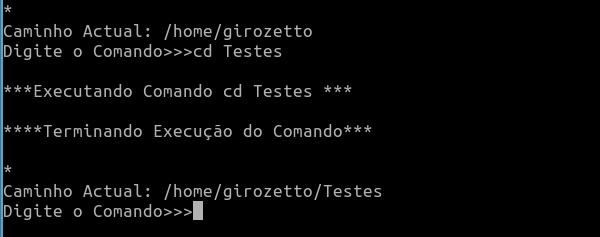


Fig. 4 – Executando o comando **cd** na Shell

* 1. **Redirecionamento de Entrada**

O redirecionamento de ficheiros é geralmente conhecido como redirecionamento I/O nos sistemas baseados no UNIX, que permite ao usuário redefinir de onde vem a entrada padrão ou para onde a saída padrão vai. O operador “ < ” é usado para mudar de onde vem a entrada padrão. Este método pode ser útil para obter inputs do usuário apartir do conteúdo de um ficheiro ou gravá-lo no buffer de um programa.

Foi nos pré-definida uma estrutura: comando < ficheiro

Para identificar algum redirecionamento usei um algoritmo que percorre toda a string inserida pelo usuário para ver se encontra o caracter “ < ” ou o caracter “ > ”.

Seguindo a estrutura pré-definida, sabe-se que iremos dividir a string inserida pelo usuário em duas partes, tendo como delimitador o “ < ”, usando métodos anteriormente usados, iremos abrir o ficheiro usando o open(), usando os parâmetros a string contendo o nome do ficheiro, uma flag **O\_RDONLY** para indicar que o ficheiro será usado unicamente para leitura e o parâmetro em octal representando permissões **0777** que retornará o File Descriptor do ficheiro um inteiro maior ou igual a zero, caso não existir irá mostrar um output de erro.

Em caso de continuação correcta, já tendo separado a parte do ficheiro da parte do comando simples e já tendo aberto o ficheiro. Podemos continuar usando o comando **dup2()** , com os parâmetros o **File Descriptor** do ficheiro que deseja duplicar e o **File Descriptor** do ficheiro que deseja substituir pelo ficheiro anterior. De acordo com a Fig. 1 por padrão cada ficheiro de entrada, saída ou erro, já é inicializado com um **File Descriptor**, tendo isto em conta basta apenas substituir a posição actual da entrada padrão que é 0 por padrão, mas também pode ser obtido usando a Flag **STDIN\_FILENO,** pelo ficheiro que foi inserido, executar o comando e depois fechar o ficheiro duplicado de modos que a entrada padrão se torne novamente a entrada padrão.

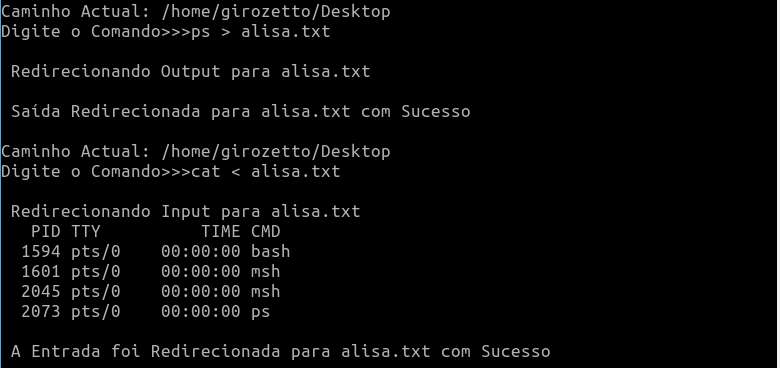


Fig. 5 – Executa um redirecionamento de saída e um redirecionamento de entrada usando o output como sendo o ficheiro alisa.txt e logo a seguir colocando-o como input

* 1. **Redirecionamento de Saída**

Como já foi explicado no ponto anterior, o processo é o mesmo, a única diferença é que em vez de procurar por “ < ”, procuramos por “ > ” e em vez de usar o 0(zero) ou STDIN\_FILENO, usar o 1 ou o STDOUT\_FILENO e também que como pretendemos escrever num ficheiro o output de um comando como é mostrado na Fig. 5, ao abrir o ficheiro com a função **open()**, usar as flags **O\_WRONLY** | **O\_CREAT**, abrir o ficheiro apenas para escrita pois não iremos ler e caso não existir, criar um novo com o mesmo nome, o contrário do redirecionamento de entrada.

* 1. **Histórico dos Últimos Comandos Executados**

Nos foi pedido que o comando para mostrar o histórico fosse o “historia”, para poder ver o histórico de todos últimos comandos executados na corrente sessão, tal como diz o enunciado, necessita-se usar 4 comandos de manipulação de ficheiros pertencentes a biblioteca padrão do **Unix**, **unistd.h**, também chamados de **Chamadas do Sistema** que são eles:

* Open()
* Read()
* Write()
* Lseek()

Para qualquer operação com ficheiros precisamos sempre abrir ele primeiro, como foi nos pedido que todo o histórico fosse criado num ficheiro de nome “**msh.hist**”, precisávamos usar o comando **open()** para abrir o ficheiro logo no início da execução da Shell, sendo o seu primeiro parâmetro uma cadeia de caracteres contendo o nome do ficheiro, o segundo parâmetro serão as **Flags** de abertura de ficheiros, como para o nosso caso precisamos que seja possível ser lido e ser escrito usamos a Flag **O\_RDWR**, como queríamos que fosse criado o ficheiro caso o ficheiro não existisse usamos a Flag **O\_CREAT**, para combinar com a Flag anterior apenas separamos as duas Flags com um “ | ” e também adicionamos mais uma Flag ainda, sendo que o enunciado nos pedia que o histórico fosse reinicializado sempre que a Shell fosse executada, para tal adicionamos mais uma Flag , a **O\_TRUNC**, sendo que esta última permitiu-nos limpar todo o histórico a cada execução da Shell. E para o último parâmetro que representa as permissões de abertura colocamos o Octal **0777** para nos dar acesso total ao ficheiro.

E assim ficou tendo inserido todos os parâmetros:

int ficheiro=open("msh.hist",O\_RDWR | O\_CREAT | O\_TRUNC,0777);

OBS: O open() retorna sempre um File Descriptor em inteiro, onde retorna um número maior ou igual que zero caso for criado corretamente e menor que zero no caso contrário.

No momento, precisávamos escrever no ficheiro já aberto, sendo que já temos a permissão para tal, usamos o comando **write()** sendo uma chamada do sistema que nos permite escrever nos ficheiros, definimos que para cada execução de um comando na nossa Shell fosse escrito o comando no ficheiro, sendo o primeiro parâmetro do comando, o File Descriptor do ficheiro aberto anteriormente, o segundo parâmetro será o que deseja escrever no ficheiro que no caso é um comando inserido pelo usuário e por fim, o terceiro parâmetro que é a quantidade em bytes do comando, como forma de ler correctamente os comandos decidi usar o mesmo tamanho que a String comando tem que no caso é o **MAXLETRAS**.

E assim ficou tendo inserido todos os parâmetros:

write(ficheiro,comando,MAXLETRAS);

Já para poder ler e mostrar no terminal o histórico de comandos, decidimos usar os comandos **lseek()** e o **read()**, primeiro é que o lseek() é um espécie de ponteiro onde estiver posicionado será o princípio de leitura do comando read() ou o princípio de escrita do comando write(), para o nosso caso será o comando read(), o comando lseek() tem com parâmetros, o File Descriptor do ficheiro, a posição do ponteiro no ficheiro e uma Flag modificadora, já o comando read() tem como parâmetros o File Descriptor do ficheiro que deseja ler, o buffer, uma string onde será enviada cada texto lido temporariamente e imprimido no mesmo instante e o tamanho de leitura, quanto do ficheiro deseja ler, para termos o mesmo tamanho de escrita, também usamos o MAXLETRAS. No lseek() se começarmos do zero implica o início do ficheiro e o read() irá ler todos caracteres do zero até MAXLETRAS, neste caso, se não mudarmos a posição do lseek(), continuaremos lendo sempre os mesmos textos até consumir toda memória do Computador para tal decidimos incrementar a posição do lseek(), MAXLETRAS mais MAXLETRAS, sendo que chegará ao fim caso read() for menor que zero. Esta foi a lógica aplicada.

E assim ficou tendo inserido todos os parâmetros:

lseek(ficheiro,posição,SEEK\_SET);

read(ficheiro,buffer,MAXLETRAS);



Fig. 6 - Output do comando historia mostrando o histórico de comandos executados

* 1. **Execução do Comando Anterior**

Para poder executar um comando anterior, usamos apenas o mesmo método usado no histórico de comandos, ao em vez de mostrar cada comando move o ponteiro lseek() até o fim do ficheiro e usa o último comando encontrado como sendo o comando que deve ser executado usando o método read() e uma string buffer, pode ser verificado na Fig. 7.

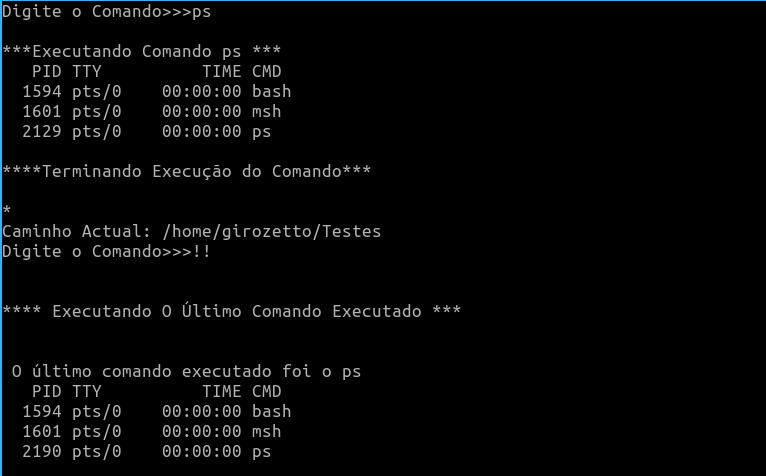


Fig. 7 – Executando um comando anterior

1. **Considerações Finais**

Como grupo, realizamos e implementamos as funcionalidades para a Shell que nos foi pedida no enunciado como foi observado durante a leitura deste relatório, não foi algo perfeito e nem algo fácil, mas o mais importante foi a aprendizagem, com este projeto tivemos a oportunidade de poder aprender como funciona uma Shell simples, desde o ponto de vista de um programador, que não é tão complexo entender os processos e o gerenciamento de processos, foi de mútuo interesse o ponto de redirecionamento de entrada e saídapois estamos habituados apenas a escrever e á ver os resultados não sabemos como realmente funcionam por trás, graças a este projeto obtivemos tal conhecimento, de certo modo foi satisfatório o trabalho.

1. **Referências Bibliográficas**

* Silberschatz A.,Galvin P.B.,Gagne G. **– Operating System Concepts –** 9th Edition **–** John Wiley & Sons 2013
* Tanenbaum A.S. – **Sistemas Operacionais Modernos –** 3ª Edição – Pearson – 2010
* Robbins K.A.,Robbins S. – **Unix Systems Programming: Communication, Concurrency and Threads** – Prentice Hall PTR – 2003
* Hincaple R. – **Medium – Tutorial to code a simple shell in C –** 15 de Abril de 2020
* GeeksForGeeks – **Making your own Linux Shell in C –** 7 de Junho de 2020