



Shell Linux

Documentação do Projeto

Hericles Felipe Ferraz - 11811EMT022
Luiz da Silva Moura - 11611EMT028
Marcus Vinicius Miata - 11811BCC017
Pedro Henrique Rabis Diniz - 11811BCC024

1.Introdução	2
1.1. Proposta	2
1.2. Metodologia	2
2. Representação da Arquitetura	2
2.1. O que é o Shell	3
2.2. Objetivo	5
3. Funções Do Sistema	6
3.1. Ls	6
3.2. Mkdir	6
3.3. Rmdir	6
3.4. Cd	6
3.5. Cp	7
3.6. Pwd	7
3.7. Pipe	7
3.8. Operadores de Redireção	8
3.9. Curingas	9
4. Desenvolvimentos	11
4.1. Códigos de Execução	11
4.1.0. Shell	12
4.1.1. Cd	16
4.1.2. Cp	18
4.1.3. Execução de Programas	20
4.1.4. Execução de Multiprogramas	22
4.1.5. Ls	23
4.1.6. Mkdir	28
4.1.7. Pwd	30
4.1.8. Redirecionamento de entrada/saída	32
4.1.9. Rmdir	35
4.2. Testes de Execução	36
4.3. Comparações	39
5. Possíveis Melhorias	43
5.1. Ideias Complementares	43
6. Resultados e Conclusões	44
6.2. Observações Adicionais	44
7. Referências Bibliográficas	45

1.Introdução

1.1. Proposta

Projetar e programar um interpretador de linha de comando para o sistema da família Unix. Algumas dos comandos que o código deve interpretar são os:

- mkdir que cria diretórios e rmdir que exclui diretórios;
- cd responsável pela navegação de diretórios (pastas) e pwd que informa o diretório corrente (atual);
- ls que lista os arquivos e diretórios, implementar também suas variante (-l e -R);
- cp para a cópia de arquivos que deve conter as opções: -r e uso de “*”;
- Execução de programas;
- Execução de múltiplos programas, encadeados, canalizando suas entradas e saídas via pipe.
- Execução de um programa com redirecionamento de entrada/saída padrão.

1.2. Metodologia

O projeto seguiu uma abordagem colaborativa em que se agilizou o progresso do projeto, com alguns pontos a se destacar, como:

- Divisão de cada conjunto de comandos para cada integrante do grupo, reunindo e revisando o progresso gradual individual e geral do projeto;
- Discussões semanais sobre possíveis abordagens e compartilhamento de ideias na resolução dos problemas;
- Utilização de técnicas simples até avançadas da linguagem de programação C, como ponteiros duplos, arquivos, entrada/saída padrão;
- Utilização de técnicas concebidas e padronizadas pelo projeto GNU para sistemas POSIX, como forks, threads, diretórios e arquivos.
- Acompanhamento da produção individual de desenvolvimento utilizando-se o git para facilitar a troca de colaboração entre o desenvolvimento de cada membro.

2. Representação da Arquitetura

2.1. O que é o Shell

Na computação, um Shell é um programa de computador que expõe os serviços de um Sistema Operacional a um usuário humano ou outros programas. Em geral, Shells de Sistemas Operacionais usam o CLI ou GUI, dependendo do objetivo do computador e da operação em particular. É nomeado Shell pois é uma camada externa em volta do Sistema Operacional, como uma Concha.

Em outras palavras, o Shell é um programa que recebe, interpreta, executa os comandos de usuário e acessa os serviços do kernel no Sistema Operacional. É o interpretador de comandos inserido por outros aplicativos ou inserido diretamente na chamada do sistema.

Além disso, os recursos do Shell também são essenciais para processar vários arquivos ao mesmo tempo, repetindo tarefas ou organizando operações para ocasiões específicas.

O Shell em si não é padrão para todos os sistemas, sendo assim, cada organização projeta de sua própria maneira atendendo às próprias exigências. Dessa forma, Sistemas Operacionais diferentes podem ter Shells com funcionalidades diferentes, ajustando-se ao ambiente em que são criados.

Shells em CLI requerem que o usuário seja familiar com comandos sua sintaxe de chamada, e que entendam conceitos sobre a linguagem específica do shell, como, por exemplo, o Bash.

Shells em GUI são comumente utilizados para usuários iniciantes de computadores, por se caracterizar como de fácil uso. A maioria dos Shells em GUI também apresentam Shells em CLI.

Uma Interface de Linha de Comando (CLI) processa comandos para um programa de computador no formato de linhas de texto. O programa que cuida da interface é chamado Interpretador/Processador de Linha de Comando. Sistemas Operacionais implementam CLI no Shell interativo para acesso às funções e serviços de si mesmo. Esse acesso foi primeiramente provido aos usuários pelos terminais de computador, no meio da década de 1960, e continuou a ser usado até as décadas de 70 e 80 em VAX/VMS, sistemas Unix e sistemas de computadores pessoais (PC), tais como DOS, CP/M e Apple DOS.

Ultimamente, a maioria dos usuários de PCs preferem interagir com a Interface Gráfica de Usuário (GUI) e com menus. Contudo, algumas programações e tarefas de manutenção podem não apresentar uma GUI e ainda serem usadas como uma CLI.

Alternativas ao CLI incluem: Interface de Menus Baseada em Texto, como o IBM AIX SMIT; Atalhos de Teclado e várias interfaces centradas no uso de um ponteiro (o mais comumente usado sendo o mouse). Exemplos disso incluem o Microsoft Windows, DOS Shell, e Mouse Systems PowerPanel. CLI são usualmente implementadas em dispositivos com terminais que também são capazes de

Interface de Menus Baseada em Texto que usam o endereçamento do cursor para colocar símbolos na tela.

Programas com CLI geralmente são mais fáceis de automatizar via scripts.

Muitos sistemas de softwares implementam CLI para controle e operação, incluindo ambientes de programação e programas de utilitários. Um exemplo é mostrado conforme mostra a Fig 1.

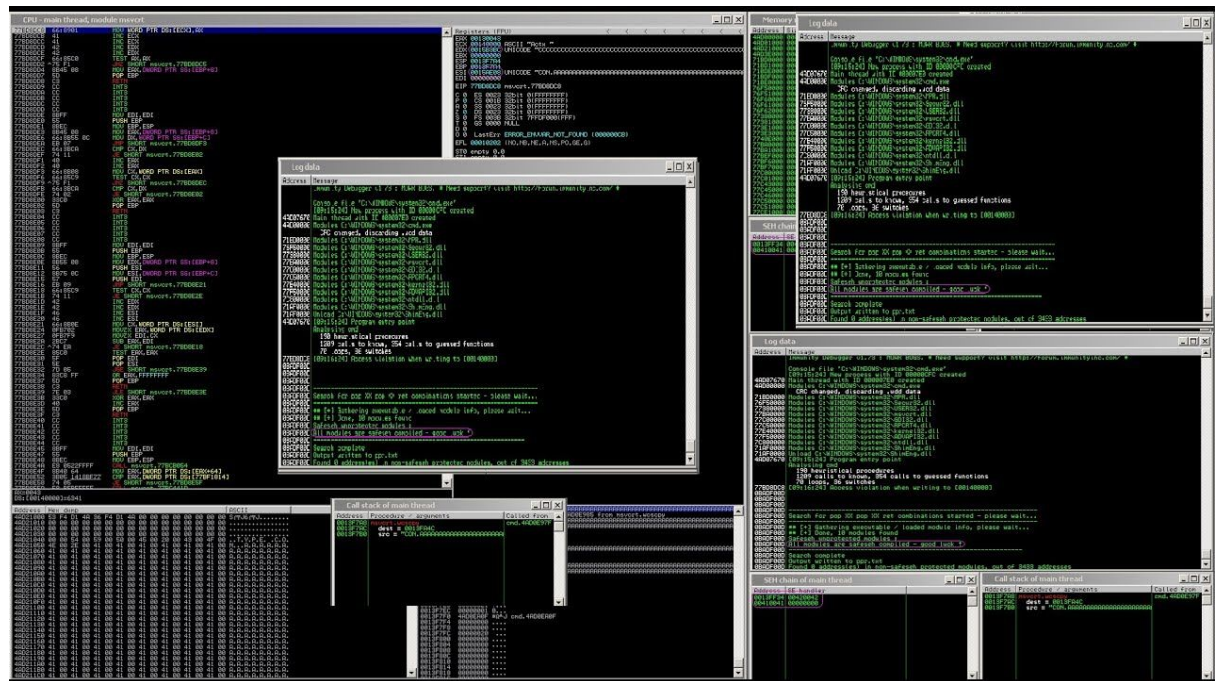


FIG 1 - Exemplo do CMD (Shell CLI dos Sistemas Operacionais Windows da Microsoft)

O Shell de um sistema é uma ferramenta poderosa, uma vez que dá acesso a comandos e funcionalidades avançadas do sistema e é de suma importância para profissionais da área de informática. Portanto, a familiaridade com o Shell é desejado em qualquer profissional de qualidade, aumentando a qualidade do software que este produz.

Além disso, há também o Shell GUI (Graphical User Interface) que permite a visualização e interação mais acessíveis das ferramentas do sistema, como a visualização de arquivos, mouse e navegadores de internet.

A Interface Gráfica de Usuário (GUI) é uma forma de Interface de Usuário que permite que o usuário interaja com o dispositivo eletrônico através de ícones gráficos e indicadores sonoros como uma primeira abordagem, ao invés de Interface de Usuário Baseada em Texto, rótulos comandos tipados ou navegação de texto. GUI foram introduzidas com a reação a percepção da curva de aprendizagem dos CLI, que requerem comandos a serem escritos pelo teclado do computador. Uma GUI é restrita ao escopo de telas bidimensionais que são capazes de descrever informações genéricas.

A ação que uma GUI faz é usualmente performed através de manipulação direta dos elementos gráficos. Além dos computadores, GUIs são utilizadas em muitos dispositivos móveis como os celulares, em videogames e controles de escritórios.

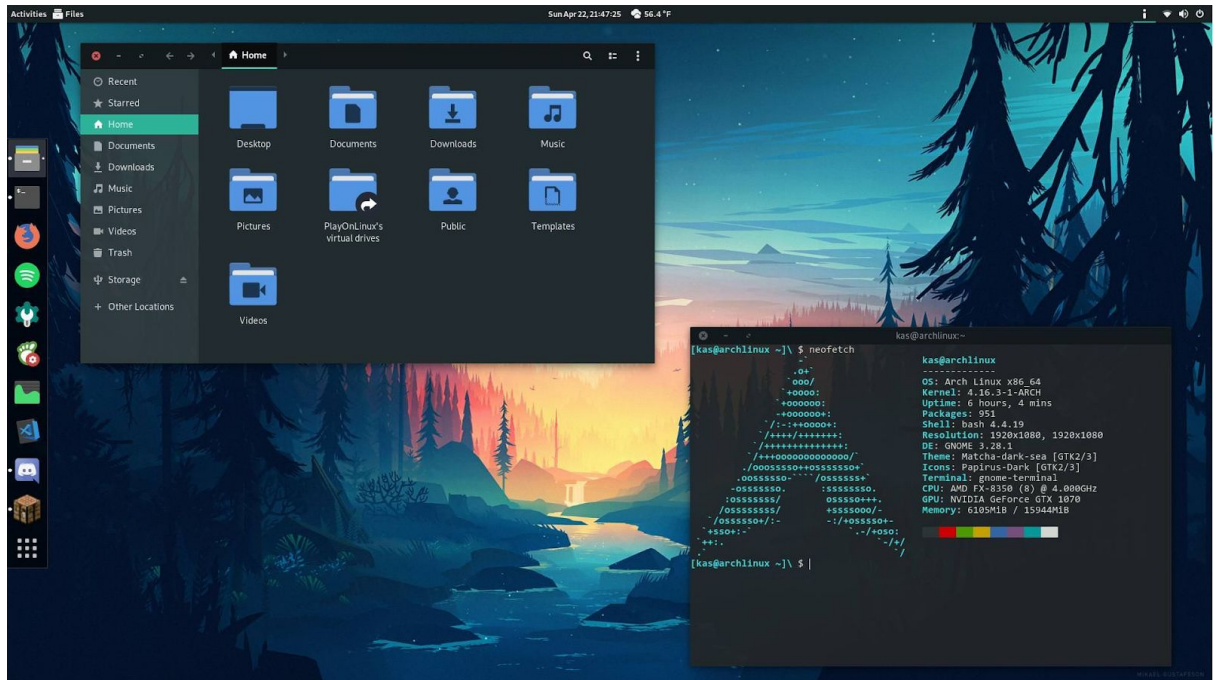


FIG 2 - Exemplo do GNOME3 (Shell GUI projetado para Sistemas Operacionais UNIX-like)

2.2. Objetivo

Utilizar os conhecimentos obtidos na matéria de sistemas operacionais para projetar e programar um interpretador de linha de comando (shell) para o Linux de forma eficaz e que se adeque a todos os requisitos estipulados.

Com isso utilizar e entender de forma prática o funcionamento do shell expandindo assim entendimento dos envolvidos sobre algumas execuções recorrentes do sistema operacional.

3. Funções Do Sistema

Abaixo, seguem os comandos supracitados que o trabalho se propõe a realizar. Assim, os detalhes individuais da teoria das funcionalidades serão apresentadas com exemplos que não necessariamente estão no escopo do projeto, a fim exclusivamente de ilustração e de facilitar o entendimento.

3.1. Ls

O comando “ls” (list directory content) tem como finalidade listar arquivos e pastas do diretório atual no qual o usuário se encontra quando utilizado, sendo possível utilizar-se também a passagem do diretório como parâmetro. Algumas opções adicionais são utilizar-se o comando “ls” com os parâmetros “-l” e “-R”.

O parâmetro “-R” faz com que o comando de listagem seja efetuado de forma recursiva, listando os sub arquivos e subpastas que estão contidos em cada diretório.

Já a opção “-l” traz uma opção mais detalhada dos arquivos listados, oferecendo detalhes como, data e hora da última modificação, tamanho e os níveis de permissões que aquele arquivo possui em relação àquele determinado usuário.

3.2. Mkdir

O comando “mkdir” (make directory) é usado para criar diretórios. Quando o usuário cria um diretório novo com o comando 'mkdir', esse diretório recebe automaticamente dois elementos ocultos. Esses elementos são: 'o.' (ponto) e 'o..' (dois pontos), representando ligações entre o novo diretório e o diretório pai.

3.3. Rmdir

O comando “rmdir” (remove directory) é usado para remover diretórios. Esse comando remove todo o conteúdo de um diretório, incluindo subdiretórios dentro do mesmo.

3.4. Cd

O comando “cd” que proporciona a navegação pelo sistema e altera o diretório corrente na linha de comandos (terminal) do Linux.

3.5. Cp

O comando responsável por copiar um arquivo pelo terminal. Usado geralmente o comando *cp* seguindo do arquivo de origem e o destino para ele, que pode ser tanto uma nova pasta quando um novo arquivo, com nome diferente. Exemplo: *cp arquivo1.txt arquivo2.txt* ou, então, *cp arquivo1.txt pastanova/*.

Para copiar um diretório todo, não se esqueça de inserir o parâmetro *-r*. Se quiser clonar uma pasta, use *cp -r pasta1 pasta2*, por exemplo.

3.6. Pwd

O “pwd” (print working directory) é um comando que imprime o nome do diretório de trabalho atual (caminho completo) em uma interface de linha de comando.

3.7. Pipe

O caractere de barra vertical “|” é chamado de operador *pipe* ou *canal* e é usado para enviar a saída de um comando para outro na linha de comandos. Use a redireção de entrada e saída com os canais para rapidamente construir comandos personalizados.

Os canais (ou pipes) podem poupar tempo e esforço e trabalham com muitos comandos encontrados no Linux. Para que você possa usar um comando em um canal, o programa deve ser capaz de ler a entrada padrão e gravar na saída padrão. Tais programas também são chamados de *filtros*.

Os canais variam de simples a complexos e são usados para muitas finalidades. A seguir, alguns exemplos de como os canais podem ser usados:

```
$ cat report.txt | wc -l > number_of_lines.txt
```

O exemplo anterior conta o número de linhas em um arquivo e gera um relatório.

```
$ find / | wc -l > number_of_lines.txt
```

Essa linha de comando conta o número de arquivos no seu sistema, iniciando no diretório “/” ou raiz.

```
$ find / | sort | uniq -d > duplicate_filenames.txt
```

Esse exemplo gera um relatório dos arquivos duplicados no seu sistema (arquivos com o mesmo nome).


```
$ string /usr/lib/ispell/american.hash | sort | tee uppercase.dict | \ tr A-Z a-z > lowercase.dict
```

Esse exemplo extrai uma lista de palavras do dicionário do sistema usando o comando *strings*. A saída é então classificada. O comando *tee* é usado depois para salvar uma cópia do dicionário, ainda na sua forma original em maiúsculas, enquanto a saída é convertida para minúsculas pelo comando *tr* e, depois, salva em minúsculas. Isso cria dois dicionários diferentes: um contendo as palavras em maiúsculas e o outro contendo as palavras em minúsculas.

3.8. Operadores de Redireção

Os operadores de redireção do shell são usados para direcionar a entrada ou saída de um programa. O shell pode ser usado para alimentar a entrada de um programa a partir de outro, da linha de comandos ou até mesmo de outro arquivo. Esses operadores são usados para copiar, criar ou sobrepor arquivos, para construir relatórios ou criar bancos de dados

O operador de redireção “<” (entrada padrão) alimenta um programa com informações da seguinte forma:

```
$ cat < myfile.txt
```

Aqui, o comando lê o conteúdo do arquivo *myfile.txt* e envia o seu conteúdo para o monitor.

O operador de redireção “>” (saída padrão) alimenta um arquivo com informações de um programa da seguinte forma:

```
$ cat myfile.txt > newfile.txt
```

Se o arquivo *newfile.txt* não existe, ele é criado. Se o arquivo existe, o conteúdo do arquivo *myfile.txt* é sobrescrito no arquivo *newfile.txt*.

O operador de inclusão “>>” é utilizado a fim acrescentar saída a arquivos existentes, da seguinte forma:

```
$ cat < myfile.txt >> newfile.txt
```

Se o arquivo *newfile.txt* não existe, ele é criado. Se o arquivo existe, o conteúdo do arquivo *myfile.txt* é incluído no fim do arquivo *newfile.txt*.

O operador aqui “<<” é utilizado a fim de informar o shell quando deve parar de ler a entrada:

```
$ cat << end
```

```
> this is
> the
> end
this is
the
%
```

Quando o shell lê a palavra especificada que segue o operador "<<", ele para de ler a saída.

3.9. Curingas

Os *curingas* são usados para especificar arquivos por nome ou extensão. Os shells para o Linux oferecem formas sofisticadas de curingas para executar correspondência de padrões complexos. O método de criar curingas é conhecido como construção de expressões regulares.

As *expressões regulares* são padrões de curingas, construídos usando uma linha de comando especial, nessa sintaxe se usa vários caracteres especiais, como:

- " * ", corresponde a todos os caracteres;
- " ? ", corresponde a um único caracter;
- [a-z], corresponde a uma faixa de caracteres;
- [0123], corresponde a uma faixa de caracteres;
- "\?", corresponde ao caractere ?
- "\"), corresponde ao caractere)
- "^abc", corresponde ao padrão abc no início das linhas;
- "\$abc", corresponde ao padrão abc no fim das linhas.

Use o asterisco (*) para localizar todas as correspondências de padrões e posteriores:

```
$ ls *.txt
```

A linha de comando anterior lista todos os arquivos ".txt". Para listar todos os arquivos no seu diretório home, você deve implementar o seguinte:

```
$ ls *.*
```

Esse comando lista somente os arquivos com um ponto dentro do nome do arquivo. Para tentar listar todos os arquivos ocultos que iniciam com a letra x, deve-se usar o seguinte:

```
$ ls .x*
```

A ordem dos caracteres em expressões é importante. Para localizar caracteres normalmente interpretados pelo shell como caracteres de significados especiais, use a barra invertida.

```
$ ls *\?*
```

Esse comando lista todos os arquivos com o carácter ? no nome do arquivo.

4. Desenvolvimentos

Abaixo estarão breves resumos de como foi desenvolvida a implementação das funções planejadas. Utiliza-se trechos de códigos e fluxogramas objetivando o melhor entendimento do modo de operação dos programas.

4.1. Códigos de Execução

A seguinte hierarquia de projeto foi definida: A pasta **include** serve para serem colocados os arquivos.h, os denominados cabeçalhos. Na pasta **src** existem os códigos desenvolvidos pelo grupo.

```
include
  -arquivos.h
src
  -codigos.c
```

Onde cada código representa no geral um dos comandos requeridos, por exemplo o ls, pwd, cd, cp possui seu respectivo código, ls.c, pwd.c, cd.c, cp.c, dentre outros, e cada codigo.c possui o seu análogo .h. Existem comandos adicionais, como o clear e o exit, que foram criados para facilitar a construção de outros comandos.

Para compilar o código principal do projeto, entre no diretório do projeto “shell-linux-so/src” e execute o comando :

```
$ gcc -o shell -pthread shell.c
```

Este comando irá compilar todos os códigos que foram utilizados. Para executá-lo, faça o seguinte comando:

```
$ ./shell
```

O código desenvolvido deverá ter sido inicializado, e estará esperando a entrada do usuário.

4.1.0. Shell

O código principal do projeto é o código shell.c, e é aqui que todos os demais códigos desenvolvidos se reúnem, é como se fosse um behaviour para a tomada de decisão de qual comando será executado.

Inicialmente são definidos recursos para se compartilhar a memória, este uso de memória compartilhada é feito com o pwd.c, devido a necessidade de saber-se o diretório atual do usuário a cada vez que forem fornecidas entradas de dados pelo usuário.

É mostrado de forma mais detalhada esta etapa no código a seguir na Fig 3.

```
int i;
key_t key=1234; // Criação da //chave de identificação
struct PWD_area *PWD_area_ptr;
void *PWD_memory = (void *)0;
int shmid;

//Criação do segmento de memória
shmid = shmget(key, MEM_SZ, 0666|IPC_CREAT);
// Verificando status da criação do segmento de memória
if ( shmid == -1 )
{
    printf("shmget falhou\n");
    exit(-1);
}

//Recebendo endereço de memória alocado
PWD_memory = shmat(shmid, (void*)0, 0);
// Verificando status da criação
if (PWD_memory == (void *) -1 )
{
    printf("shmat falhou\n");
    exit(-1);
}

//Área de memória compartilhada
PWD_area_ptr = (struct PWD_area *) PWD_memory;
PWD_area_ptr->PIDexec=0;
signal(SIGINT, StopProcess);
readDirectory(0);
```

FIG 3 - Memória compartilhada entre o shell.c e o pwd.c

Em seguida é feita uma segurança para caso o usuário não digite nada, apertando apenas a tecla enter. Esta verificação é necessária, uma vez que as funções que vêm a seguir necessitam de valores a serem lidos. A demonstração desta parte é feita na Fig 4.

```
bool flagCommand = 0;
while(!flagCommand)
{
    /*Parte não tão importante
    .
    */
    if((strlen(userCommand)==0) || (strcmp(userCommand,"0")==0))
    {
        flagCommand = false;
    }
    else
    {
        break;
    }
}
pipeCommand(userCommand);

if(inputOutput(userCommand) == 0)
{
    chooseCommand(userCommand);
}
```

FIG 4 - Condição de segurança

Conforme mostrado acima, existem as seguintes funções sendo dispostas das funcionalidades:

- pipeCommand(char *buffer): É a primeira situação a ser analisada, necessária para verificar-se se no comando inserido pelo usuário existe o caractere "|".
- inputOutput(char *buffer): É a segunda verificação, é a função correspondente ao redirecionamento de entrada e saída. Caso existam os caracteres "<",">",">>" ela retorna um valor diferente de 0, efetuando o caso correspondente a cada um dos campos.

- chooseCommand(char *buffer): É a função que analisa os primeiros campos e comandos mais básicos, como ls, mkdir, pwd, cd, clear, etc. A Fig 5 a seguir exemplifica de melhor forma o funcionamento geral da chooseCommand.

```
char *separeCommand = (char *)malloc(sizeof(char) * space);

separeCommand = strtok(buff, " ");
char *tempCommand[i];
strcpy(tempCommand, auxCommand);

/*Parte não tão importante
.
.
*/
if (strcmp (separeCommand, "mkdir") == 0)
{
    separateParams (auxCommand);
}

else if (strcmp (separeCommand, "pwd") == 0)
{
    readDirectory(1);
}
else if (strcmp (separeCommand, "clear") == 0)
{
    header();
}

else if (strcmp (separeCommand, "cd") == 0)
{
    mountPath(auxCommand, "cd");
    readDirectory(0);
}

/*
./
else
{
    execDefault (auxCommand);
}
}
```

FIG 5 - Exemplo de organização da chooseCommand

Conforme mostrado no código acima, a variável `separeCommand` é utilizada para comparar a primeira palavra digitada pelo usuário com um dos possíveis comandos. No instante em que algum dos comandos for encontrado, uma função correspondente àquele comando será iniciada. É importante salientar que foi inserido no `else` uma função bastante importante, que é a `execDefault(char *name)`, responsável por efetuar os comandos que não foram programados e requisitados conforme a atividade proposta, executando por exemplos comandos como o `cat`, `wc`, dentre outros.

4.1.1. Cd

Inicialmente o programa verifica se possui a palavra “cd” na entrada do usuário, caso tenha, é contado o que está escrito a partir deste ponto de forma análoga ao que foi mostrado no caso do shell onde se analisava a primeira palavra.

Em seguida é construído o diretório utilizando-se o `getcwd` para se considerar o diretório atual do usuário, em seguida é chamada a função `cd(char *pth)`. Nesta função é feita a construção de fato do diretório para onde o usuário deseja entrar, adicionando a “/”. Uma parte do código é descrita abaixo para exemplificar na Fig 6.

```
int cd(char *pth) {
    char path[BUFFERSIZE];
    strcpy(path, pth);

    //Caso seja apenas "cd "
    if((strlen(pth)==0) || (strcmp(pth, "0")==0) || (
strcmp(pth, "\0")== 0))
    {
        chdir(getenv("HOME"));
    }

    char cwd[BUFFERSIZE];
    if(pth[0] != '/')
    {
        getcwd(cwd, sizeof(cwd));
        strcat(cwd, "/");
        strcat(cwd, path);
        if( opendir(cwd) == 0)
        {
            printf("cd: no such file or directory: %s \n", pth);
        }

        else
        {
            chdir(cwd);
        }
    }
    return 0;
}
```

FIG 6 - Código do comando cd

Com a finalidade de deixar-se mais intuitivo, o comando é mostrado conforme a seguir no Fluxograma da Fig 7.

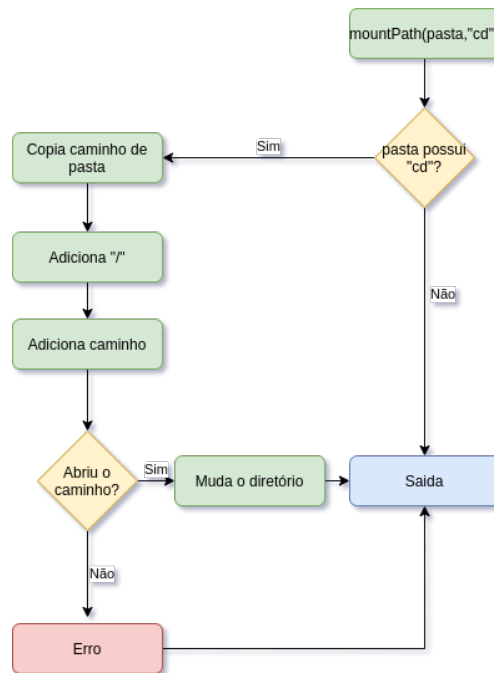


FIG 7 - Fluxograma da funcionalidade do comando "cp"

4.1.2. Cp

Neste código foi de vital importância utilizar funções que acessassem o sistema de arquivos, para isso houve a necessidade de utilizar principalmente variáveis do tipo `FILE` que possibilita a leitura e escrita de arquivos e `DIR` que possibilita a leitura e escrita de pastas.

Para um melhor entendimento o código criado pode ser descrito da seguinte forma, conforme mostrado na Fig 8:

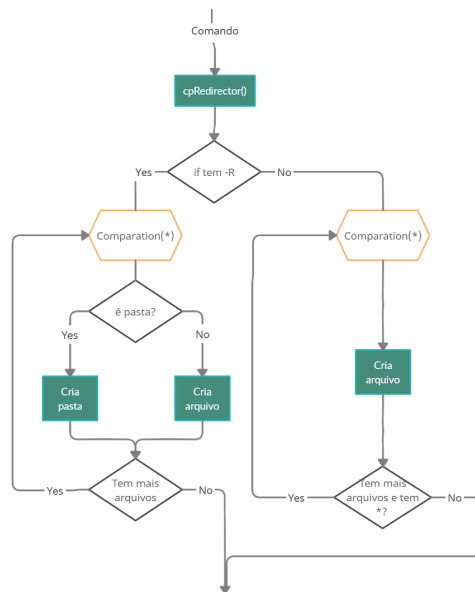


FIG 8 - fluxograma simplificado do código de cp

A função `comparison()` é responsável por interpretar o código inserido e chamar a função responsável por compilar, assim abre-se o diretório que tem o arquivo a ser copiado.

Com o diretório aberto é efetuado a leitura de cada arquivo através da função `readdir()` verificando se é para copiar ou não, se for para copiar faz uma confirmação se é pasta ou não.

```
while ((nextDir = readdir(dir)) != NULL) {
    ...
}
```

FIG 9 - Trecho do código que lê todos arquivos de determinada pasta

Caso encontrado uma pasta a ser copiada e se o comando inserido for do tipo `-R` é criada a pasta através da função `mkdir()` entrando novamente na função responsável de copiar de forma recursiva.

```
if( (check = mkdir(nextpaste, 0777) ) != 0)
    perror("\ncp: cannot create directory" );
if(type=='R')
    in = cpComplete('R',nextcopy,R,nextpaste)+in;
```

FIG 10 - Trecho do código que cria pasta e caso necessário chama a si mesmo de forma recursiva

Se o arquivo não for uma pasta é copiado através de dois comandos `FILE` um do tipo `r` que lê o arquivo é o outro do tipo `w` que cria o arquivo com anterior leu.

```
int copyPaste(char *oldArq, char *newArq) {
    FILE *CtrlC, *CtrlV;
    CtrlC = fopen(oldArq , "r");
    CtrlV = fopen(newArq , "w");
    int ch;
    while ((ch = getc (CtrlC )) != EOF)
        putc (ch, CtrlV);
}
```

FIG 11 - Função do código responsável por copiar arquivos

4.1.3. Execução de Programas

Para a criação deste código foi utilizado principalmente o conhecimento adquirido sobre controles de processos onde houve a necessidade de um processo filho e a utilização da família de funções primitivas `exec()`;

Antes de executar o comando de execução de um programa é preciso interpretar o comando inserido pelo usuário, podendo ser separado da seguinte forma:

Caminho/Prog Arg₁ Arg₂ .. Arg_n

Onde o *Caminho* é a localização do diretório a ser executado, caso seja no diretório corrente o *Caminho* corresponde ao caractere ponto('.');

Prog é o nome do programa a ser executado e

Arg_{1-n} é uma lista de argumentos que o usuário está passando para a execução, sendo uma possibilidade a ausência deste parâmetro.

O funcionamento do programa pode ser descrito pelo fluxograma a seguir:

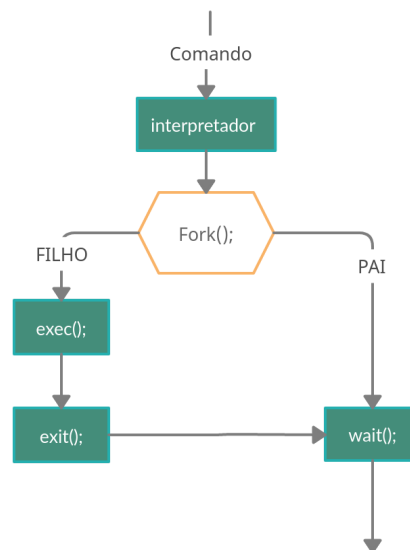


FIG 12 - Fluxograma simplificado do código de execução de programas externos

Onde o interpretador é a parte do código que manipula o comando inserido no shell de uma forma aceitável para o `exec`.

Para a implementação de um comando da família `exec` é necessário a criação de um processo independente do shell devido que na execução deste comando à substituição do programa requerido executando a partir deste momento independente do código que o chamou.

Como neste caso o processo filho executará uma tarefa diferente do pai, a criação do filho é através do comando `fork`, neste processo é salvo o PID(process identifier) do filho em uma memória compartilhada. Onde o filho chamará a função `exec` enquanto o pai espera o término do processo.

```

if ( pid == 0 )
{
    sem_wait((sem_t*)&PWD_area_ptr->mutex);
    PWD_area_ptr->PIDexec= getpid();
    sem_post((sem_t*)&PWD_area_ptr->mutex);
    //execl(prog,name,0);
    execv (prog, args);
    printf("shell: command not found:%s.", copy);
    exit(0);
}

```

FIG 13 - Trecho do código do filho

O comando escolhido da família `exec` foi o `execv` devido a sua arquitetura, demonstrada a seguir, que tem como entrada argumentos de entradas do programa de execução caso necessário.

*int execv(const char * path, char * const argv[]);*

Como implementação adicional foi criado o código para matar o processo filho caso necessite, isso foi feito por meio da função que executará caso receba o sinal `SIGINT`(comando que pode ser enviado através do CTRL+C), que aciona a função `KILL` para PID do processo salvo na memória compartilhada.

4.1.4. Execução de Multiprogramas

O comando pipe inicialmente é feito utilizando-se um parsing dos comandos inseridos pelo usuário, em seguida é feito um for que procura quantos “|” se encontram para utilizar-se o pipe. O pipe feito executa chamadas para o próprio sistema por meio do `execvp(arg[0], arg)`. Tentou-se inicialmente efetuar-se as chamadas desenvolvidas pelo grupo, no entanto erros ocorreram e este é um dos problemas não resolvidos a tempo. Este código foi baseado no modelo `onlineshell.c` do usuário `parthnan`, referenciado nas referências bibliográficas.

```
pid = fork();
if(pid == 0) //Processo Filho
{
    if( commandc!=0 )
    {
        if(dup2(pipefds[(commandc-1)*2], 0) < 0)
        {
            perror("Erro de entrada");
            exit(1);
        }
    }
    if(commandc!=pipeCount)
    {
        if(dup2(pipefds[commandc*2+1], 1) < 0 )
        {
            perror("Erro de saida");
            exit(1);
        }
    }
    for( i = 0; i < 2*pipeCount; i++ ){close(pipefds[i]);}
    args=splitline(token);
    //Linha com comandos que deveria ser lida
    //chooseCommand(line);
    execvp(args[0],args);
    exit(0);
}
```

FIG 14 - Trecho do código do pipe

É feito um `fork()` para o processo filho efetuar o comando correspondente, conforme mostrado no código acima. Outro problema encontrado nesta parte de desenvolvimento é que o processo filho quando termina não retorna à tela inicial esperando novamente a entrada no usuário.

4.1.5. Ls

Para a execução do comando LS (List Directory) verifica-se primeiramente o comando ("-l", "-R" ou qualquer outro) e redireciona-se para o conjunto de comandos adequados. Após isso, é aberto o diretório que foi passado como parâmetro e também os arquivos que estejam dentro deste, caso existam, percorrendo o diretório e listando.

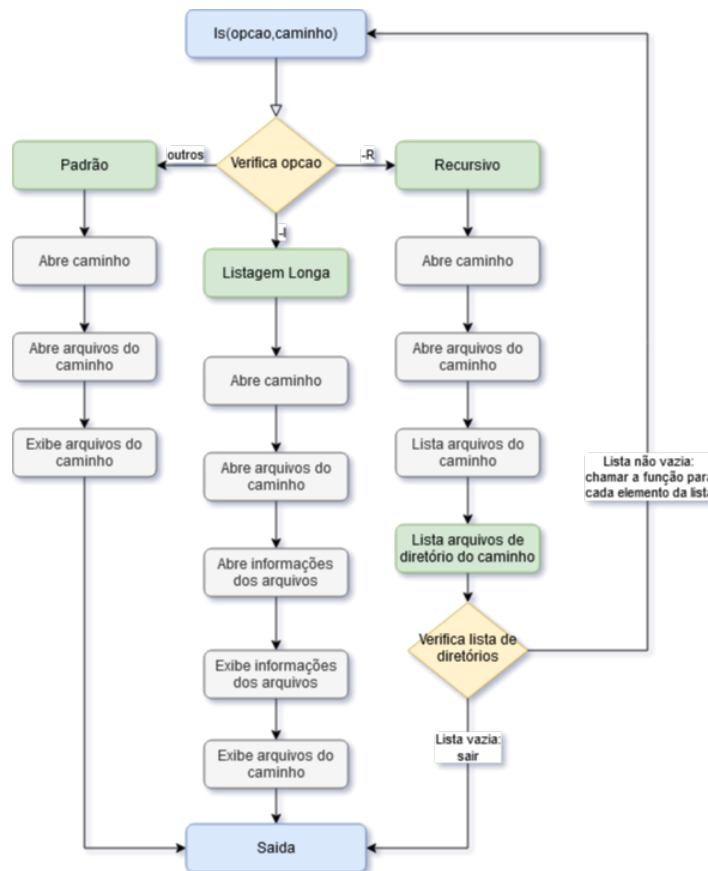


FIG 15 - Fluxograma da funcionalidade do comando "ls"

No caso da abordagem Padrão ("ls"), apenas exibe os arquivos existentes dentro do diretório utilizado, podendo ser estes quaisquer tipos de arquivos que não sejam ocultos.

```
DIR *dir;
struct dirent *entry;
dir = opendir(path);
...
while((entry = readdir(dir))) {
    if(entry->d_name[0] == '.') {
        continue;
    } else {
        printf("%s\t", entry->d_name);
    }
}
```



```

    }
}
closedir(dir);
}

```

FIG 16 - Trecho do código referente a função “ls” com parâmetros neutros.

Já no caso da Recursiva (“ls -R”), por se tratar de uma recursão, o planejamento para este problema é diferenciado: listas são utilizadas. Dessa forma ajustando a saída e mantendo coesão na chamada recursiva.

```

int qtdNames=0, qtdFolders=0;
char names[100][100];
char folders[100][100];

for(int i=0; i<100; i++){
    for(int j=0; j<100; j++){
        names[i][j] = '\0';
        folders[i][j] = '\0';
    }
}

```

FIG 17 - Trecho do código de criação e inicialização das listas de diretórios e arquivos gerais

Assim, a execução trata-se de: inicializar a lista de diretórios com o nome do diretório em que já se encontram (para o devido acesso aos subdiretórios), abrir o diretório, listar todos os arquivos deste em uma lista (“names”), listar, também, todas as pastas desse diretório em outra lista (“folders”), caso ambas existam, exibe a lista de arquivos e, em seguida, chama a mesma função para cada pasta listada anteriormente.

```

int lenPath = strlen(path);
int lenFolders[100];
for(int i=0; i<100; i++){
    for(int j=0; j<lenPath; j++){
        folders[i][j] = path[j];
    }
    folders[i][lenPath] = '/';
    lenFolders[i] = strlen(folders[i]);
}

```

FIG 18 - Trecho do código em que são atribuídos valores iniciais aos diretórios que poderão ser utilizados.

É importante ressaltar que todos os diretórios que serão acessados precisam, necessariamente, de manter o diretório pai anotado. Por isso, essa abordagem acima foi adotada: descrever corretamente o(s) caminho(s) que será(ão) percorrido(s) pela(s) recursão(ões) subsequentes, caso ocorram.

A exibição por lista adequa a saída de dados ao padrão esperado, acabando com problemas de sobreposição de dados, como por exemplo a chamada de uma recursão enquanto está exibindo-se os arquivos existentes dentro de outro diretório.

```
for(int i=0; i<qtdNames; i++){
    printf("%s\t", names[i]);
}

printf("\n\n");
for(int i=0; i<qtdFolders; i++){
    listDirectory(command, folders[i]);
}

closedir(dir);
```

FIG 19 - Trecho do código em que exibe-se o conteúdo do diretório atual e chama a recursão pros subdiretórios, caso existam.

Além disso, na Listagem Longa ("ls -l"), além de abrir o diretório e os arquivos, abre-se também todas as propriedades deste último e exibe-as.

Dado a sua natureza extensa, essa Listagem requer um tratamento especial, especificando algumas características: nome; permissões de leitura/escrita/execução (read/write/execution); quantidade de hard links; usuário dono; grupo que pertence; data da última modificação; quantidade de blocos utilizados no diretório (não contabilizando os subdiretórios).

```
if(attribute.st_mode & S_IFDIR) {
    ...
    // rwd Owner
    // S_IRWXU = mask for file owner permissions
    if(attribute.st_mode & S_IRUSR) {
        ...
    if(attribute.st_mode & S_IWUSR) {
        ...
    if(attribute.st_mode & S_IXUSR) {
        ...
    // rwd Group
    // S_IRWXG = mask for group permissions
```

```

if(attribute.st_mode & S_IRGRP){
    ...
if(attribute.st_mode & S_IWGRP){
    ...
if(attribute.st_mode & S_IXGRP){
    ...
//  rwd Others
//  S_IRWXO = mask for permissions for others (not in
group)

if(attribute.st_mode & S_IROTH){
    ...
if(attribute.st_mode & S_IWOTH){
    ...
if(attribute.st_mode & S_IXOTH){
    ...
//  LINK COUNT
printf("%ld\t", (long)attribute.st_nlink);

struct group *groups;
struct passwd *users;
//  UID
users = getpwuid((long)attribute.st_uid);
printf("%s\t", users->pw_name);
//  GID
groups = getgrgid((long)attribute.st_gid);
printf("%s\t", groups->gr_name);

//  FILE SIZE
printf("%ld\t", (long)attribute.st_size);
//  LAST FILE MODIFICATION
printf("%s", ctime(&attribute.st_mtime));

//  Blocks allocated
blockFiles += (attribute.st_blocks/2.00);

```

FIG 20 - Trecho do código em que se acessa e exibe os atributos de cada arquivo.

Para isto, é preciso utilizar estruturas específicas para se ter acesso e comparar estes dados, por meio da “struct stat”. Dessa forma, todos os campos de estado de um arquivo são acessados, caso a função “lstat()” tenha um retorno positivo.

4.1.6. Mkdir

Essa função recebe todo o buffer que foi digitado inicialmente pelo usuário, por exemplo “mkdir nova_pasta1 nova_pasta2”. O primeiro passo do programa mkdir.c é verificar se o que foi digitado inicialmente é um *mkdir*, caso não seja, a função é finalizada informando um erro.

Em seguida, é feito um parsing para quebrar os argumentos escritos após a palavra “mkdir”, direcionando cada novo argumento a um possível nome de pasta. Caso exista naquele diretório algum arquivo com este mesmo nome, é retornado um erro, do contrário, é criada uma pasta com este nome. Uma descrição mais detalhada é dada a seguir, mostrando um trecho do código.

```
separator = strtok(buff, " ");
if (strcmp (separator, "mkdir") == 0)
{
    while(separator != NULL )
    {
        if (separator != NULL)
        {
            separator = strtok( NULL, " " );
        }
        if(separator == NULL) break;
        if( (separator[strlen(separator)]-1) == '\\n')
        {
            separator[strlen(separator)-1] = '\\0';
        }
        int check;
        if( (check = mkdir(separator, 0777) ) != 0)
        {
            printf("\\nmkdir: cannot create directory '%s':
File exists \\n", separator);
        }
    }
}
```

FIG 21 - Trecho de código do mkdir.c

Conforme mostrado no código acima, como o parsing é feito dentro de um while são consideradas todas as palavras que o usuário digitou, criando múltiplas pastas no mesmo comando. A fim de se mostrar de forma mais detalhada, o mkdir é descrito na Fig 22.

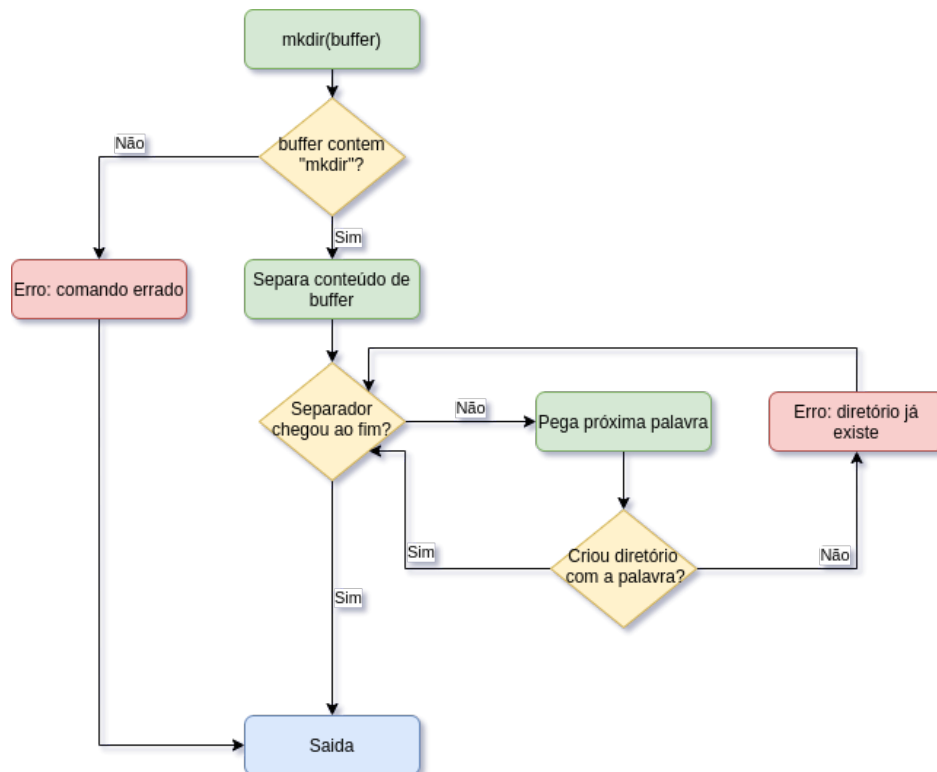


FIG 22 - Fluxograma da funcionalidade do comando `mkdir,c`

A medida que o parsing é feito novas pastas vão sendo criadas, conforme vai se analisando cada um dos nomes, o comando principal que cria estes novos diretórios é dado por: `mkdir("separator", 0777)`, onde o separator é dado pelo parsing que faz esta separação, valendo o nome daquela respectiva pasta a cada iteração, conforme mostrado no trecho de código e na Figura acima.

4.1.7. Pwd

O pwd não é uma função que o usuário chama, esta função serve para mostrar na tela o caminho completo do diretório que o usuário está manipulando, no entanto, caso o usuário digite o comando “pwd” o diretório atual é mostrado da mesma forma.

Na construção do código pwd.c inicialmente foi construído acesso a memória compartilhada, conforme mostrado anteriormente no caso do shell. Os procedimentos para a criação da chave de acesso e uso de memória compartilhada seguem o mesmo padrão. Os detalhes são mostrados no código abaixo na Fig 23.

```
int readDirectory(bool outputCase) {  
    /*  
    .  
    */  
    if (getcwd(bufferDir, nameDir) == NULL)  
    {  
        perror("getcwd");  
        exit(EXIT_FAILURE);  
    }  
  
    else  
    {  
        strcpy(PWD_area_ptr->PWD,bufferDir);  
        if(outputCase)  
            printf("%s \n", bufferDir);  
    }  
  
    free(bufferDir);  
    return EXIT_SUCCESS;  
}
```

FIG 23 - Trecho de código do pwd.c

Conforme mostrado acima, a função readDirectory é um booleano para escolher-se quando o diretório será mostrado na tela ou não, pois, existem os casos onde o usuário requisita o pwd, que é quando o booleano é verdadeiro, mostrando um resultado de saída na tela, e nos outros casos quando o booleano é falso o diretório é apenas carregado na tela, de forma análoga ao shell padrão.

A parte responsável de fato por mostrar o diretório atual é dada pela função `getcwd`. O diagrama abaixo da Fig 24 explica com mais detalhes a maneira como isso é feito.

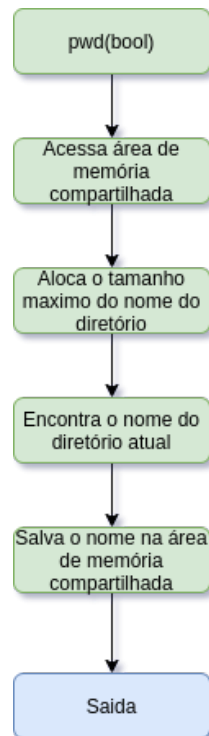


FIG 24 - Fluxograma da funcionalidade do comando “cp

4.1.8. Redirecionamento de entrada/saída

O redirecionamento é tratado pelo código `re_direct.c`, e nele também é feito o parsing para se quebrar os caracteres. No instante em que um dos comandos de redirecionamento é encontrado “<”, “>”, “>>”, é construído um responde equivalente a aquela situação, conforme mostra o código abaixo na Fig 25.

```
if(!firstLoop)
{
    if (strcmp (args[count], ">") == 0)
    {
        //printf("tem o > : ");
        response = 1;
        firstLoop = true;
    }
    else if (strcmp (args[count], ">>") == 0)
    {
        //printf("tem o >> :");
        response = 2;
        firstLoop = true;
    }
    else if (strcmp (args[count], "<") == 0)
    {
        //printf("tem o < :");
        response = -1;
        firstLoop = true;
    }
    else
    {
        response = 0;
    }
}
strcpy(nameOutput, args[count]);
```

FIG 25 - Casos de entrada e saída

Em seguida, com base nesse response é considerado em que condição será executada. É interessante salientar que esta parte é feita pelo processo filho, conforme mostra o código abaixo.

```
if(response != 0)
{
    int response_pid = 0;
    response_pid = fork();
    if (response_pid < 0)
    { //Processo filho
        printf("failed\n");
        exit(1);
    }
    else if (response_pid == 0)
    {
        int redirect_fd = 0;

        if(response == 1) // Caso >
        {
            redirect_fd = open(nameOutput, O_CREAT | O_TRUNC |
O_WRONLY, 0777);
            i =0;
            if(redirect_fd == -1)    printf("error /n");
            dup2(redirect_fd, STDOUT_FILENO);
            close(redirect_fd);
            outputCases(args[0],nameOutput);
            signal(SIGINT, SIG_DFL);
            exit(0);
        }

        else if(response == 2) // Caso >>
        {
```

FIG 26 - Casos de redirecionamento

A partir deste ponto, o processo filho executa a condição conforme a situação do response em que ele se encontra. O comportamento do redirect.c é melhor mostrado no fluxograma abaixo na Fig 27.

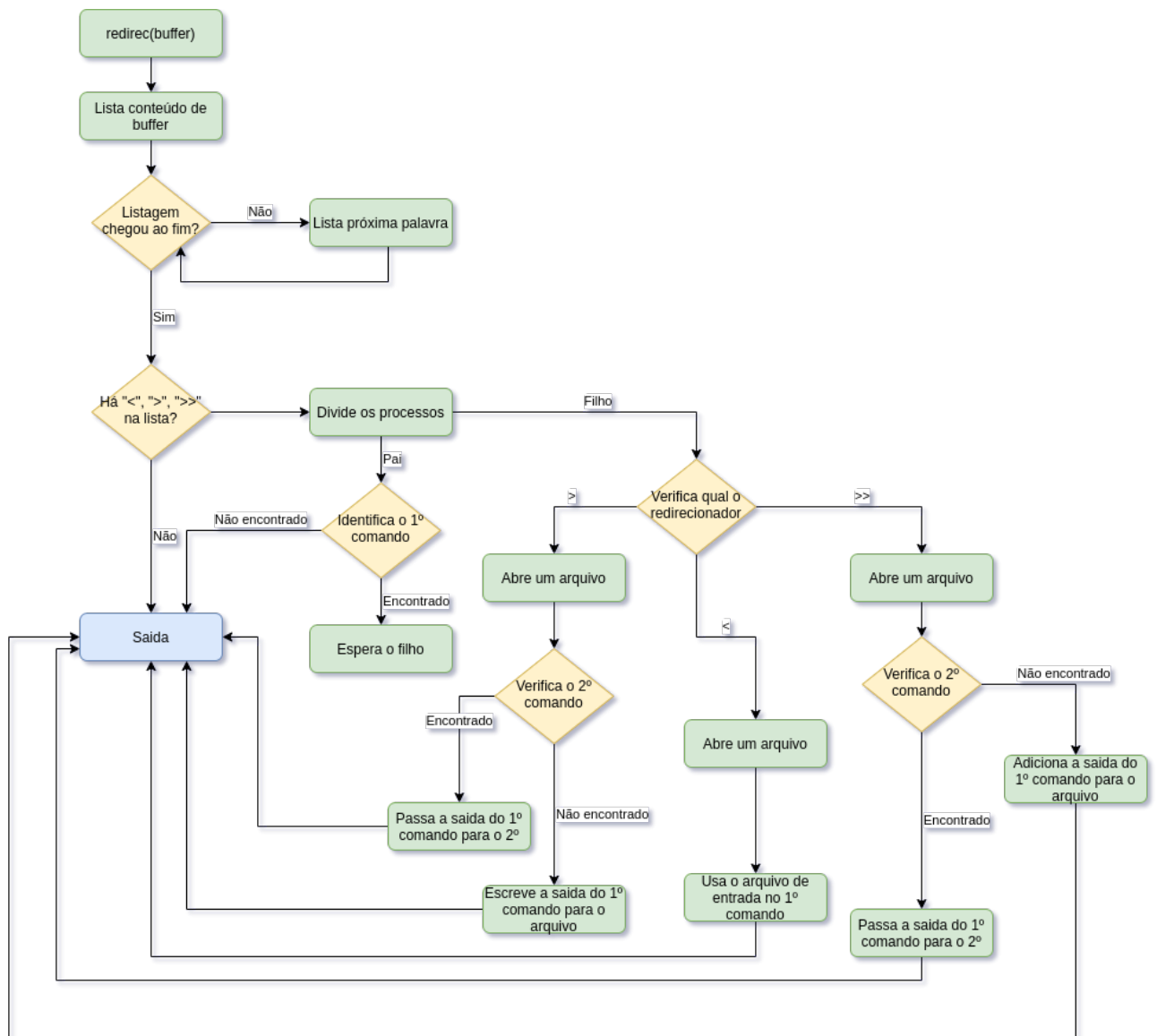


FIG 27 - fluxograma do redirecionamento

4.1.9. Rmdir

Na execução do comando *rmdir*, o programa verifica se o diretório recebido está vazio (arquivos ocultos, não são considerados nessa verificação), caso esteja ele é removido. Caso ele não esteja vazio é verificado o que está armazenado dentro deste diretório, se for apenas arquivos comuns, eles são removidos e posteriormente o diretório inicial é removido também. Caso haja outro diretório armazenado, é utilizado uma recursão para que este diretório filho também seja esvaziado e removido. Pode haver vários diretórios ou arquivos dentro do diretório principal que todos serão removidos, antes do principal ser removido.

O diagrama a seguir explica de maneira simples o funcionamento do programa:

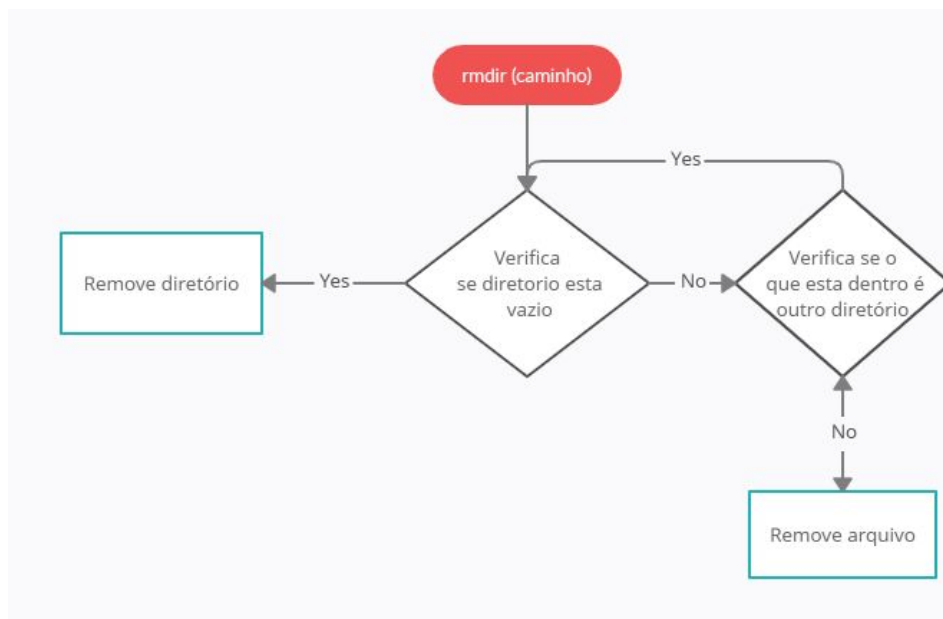


FIG 28 - fluxograma do rmdir

4.2. Testes de Execução

A seguir serão exibidos os testes de execução dos comandos dentro do shell projetado:

```
rabispedro@pop-os: ~/shell-linux-so/src $ ./shell
rabispedro:/home/rabispedro/shell-linux-so/src
$ ls
List Directory: .
shell.c re_direct.c stop.c ls.c rmdir.c mkdir.c apenas_um_teste.txt asd.txt exec.c cp.c shell cd.c pwd.c

rabispedro:/home/rabispedro/shell-linux-so/src
$ cd ..
rabispedro:/home/rabispedro/shell-linux-so
$ ls
List Directory: .
texto.txt ls.c ls out.txt src include

rabispedro:/home/rabispedro/shell-linux-so
$ cp out.txt src
rabispedro:/home/rabispedro/shell-linux-so
$ ls
List Directory: .
texto.txt ls.c ls out.txt src include

rabispedro:/home/rabispedro/shell-linux-so
$ cd src
rabispedro:/home/rabispedro/shell-linux-so/src
$ ls
List Directory: .
shell.c re_direct.c stop.c ls.c rmdir.c out.txt mkdir.c apenas_um_teste.txt asd.txt exec.c cp.c shell cd.c pwd.c

rabispedro:/home/rabispedro/shell-linux-so/src
$ |
```

Fig 29 - Teste de execução dos comandos “cd”, “ls” e “cp”.

```
List Directory: .
texto.txt ls.c ls out.txt src include

rabispedro:/home/rabispedro/shell-linux-so
$ cp src srcTeste
cp: -R não especificado; omitindo o diretório 'src'

rabispedro:/home/rabispedro/shell-linux-so
$ ls
List Directory: .
texto.txt ls.c ls out.txt src include

rabispedro:/home/rabispedro/shell-linux-so
$ krita
Name krita

(process:56872): Gtk-WARNING **: 17:05:38.966: Locale not supported by C library.
Using the fallback 'C' locale.
Invalid profile : "/usr/share/color/icc/colord/Crayons.icc" "Crayon Colors"
Invalid profile : "/usr/share/color/icc/colord/x11-colors.icc" "X11 Colors"
libpng warning: icCP: profile 'icc': 'RGB ': RGB color space not permitted on grayscale
libpng warning: icCP: profile 'icc': 'RGB ': RGB color space not permitted on grayscale
libpng warning: icCP: profile 'icc': 'RGB ': RGB color space not permitted on grayscale
qt.gui.icc: fromIccProfile: failed general sanity check
QPngHandler: Failed to parse ICC profile
qt.gui.icc: Unsupported ICC input color space 47524159
qt.gui.icc: fromIccProfile: failed general sanity check
QPngHandler: Failed to parse ICC profile
qt.gui.icc: Unsupported ICC input color space 47524159
qt.gui.icc: fromIccProfile: failed general sanity check
QPngHandler: Failed to parse ICC profile
qt.gui.icc: Unsupported ICC input color space 47524159
qt.gui.icc: fromIccProfile: failed general sanity check
QPngHandler: Failed to parse ICC profile
qt.gui.icc: Unsupported ICC input color space 47524159
qt.gui.icc: fromIccProfile: failed general sanity check
QPngHandler: Failed to parse ICC profile
QIODevice::startTimer: Timers cannot have negative intervals
/usr/lib/x86_64-linux-gnu/krita-python-libs/krita added to PYTHONPATH
QLayout: Attempting to add QLayout "" to QWidget "", which already has a layout
```

Fig 30 - Teste de execução de “execução de programas abrindo Krita”

```
rabispedro:/home/rabispedro/shell-linux-so
$ ls -l
List Directory: . -l
ls.c -rw-rw-r-- 1 rabispedro rabispedro 4240 Thu Dec 17 19:29:57 2020
teste -rw-rw-r-- 1 rabispedro rabispedro 17256 Wed Dec 16 16:21:45 2020
teste.c -rw-rw-r-- 1 rabispedro rabispedro 1886 Wed Dec 16 16:21:38 2020
ls -rw-rw-r-- 1 rabispedro rabispedro 17488 Wed Dec 16 19:20:00 2020
out.txt -rw-rw-r-- 1 rabispedro rabispedro 0 Wed Dec 16 10:08:36 2020
src drwxrwxr-x 2 rabispedro rabispedro 4096 Fri Dec 18 15:28:58 2020
include drwxrwxr-x 2 rabispedro rabispedro 4096 Fri Dec 18 15:29:24 2020
Total: 60

rabispedro:/home/rabispedro/shell-linux-so
$ ls -R
List Directory: . -R
ls.c teste teste.c ls out.txt src include

List Directory: ./src -R
shell.c re_direct.c stop.c ls.c rmdir.c mkdir.c exec.c cp.c shell cd.c pwd.c

List Directory: ./include -R
cp.h stop.h pwd.h exec.h kernel.h mkdir.h shell.h general.h rmdir.h ls.h cd.h

rabispedro:/home/rabispedro/shell-linux-so
$ mkdir teste1 teste2

rabispedro:/home/rabispedro/shell-linux-so
$ ls
List Directory: .
teste1 ls.c teste teste.c ls out.txt teste2 src include

rabispedro:/home/rabispedro/shell-linux-so
$ rmdir teste1 teste2

rabispedro:/home/rabispedro/shell-linux-so
$ ls
List Directory: .
ls.c teste teste.c ls out.txt src include

rabispedro:/home/rabispedro/shell-linux-so
$ |
```

Fig 31 - Teste de execução dos comandos “ls”, “mkdir” e “rmdir”.

```
rabispedro:/home/rabispedro/shell-linux-so/src
$ pwd
/home/rabispedro/shell-linux-so/src

rabispedro:/home/rabispedro/shell-linux-so/src
$ ls
List Directory: .
shell.c re_direct.c stop.c ls.c rmdir.c mkdir.c apenas_um_teste.txt asd.txt

rabispedro:/home/rabispedro/shell-linux-so/src
$ cd ..

rabispedro:/home/rabispedro/shell-linux-so
$ ls
List Directory: .
ls.c ls out.txt src include

rabispedro:/home/rabispedro/shell-linux-so
$ pwd
/home/rabispedro/shell-linux-so

rabispedro:/home/rabispedro/shell-linux-so
$ gedit texto.txt
Name gedit texto.txt

rabispedro:/home/rabispedro/shell-linux-so
$ |
```

Open [icon] *texto.txt ~/shell-linux-so Save [icon] [icon]

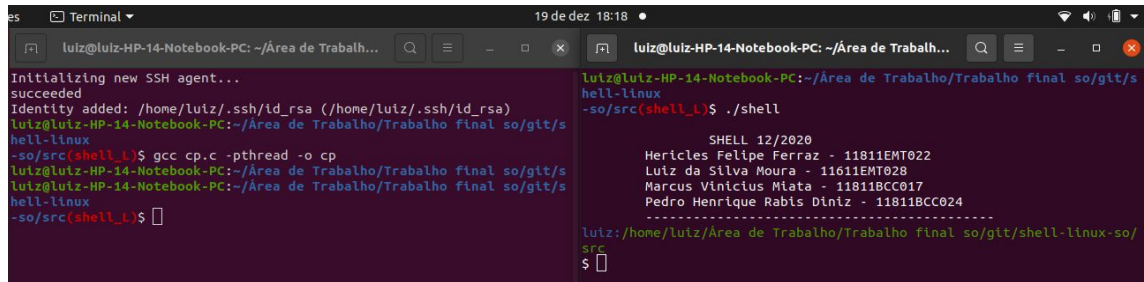
1 Um teste de teste na fase de testes, testando a capacidade dos comandos desenvolvidos pro shell cli.

Plain Text Tab Width: 2 Ln 1, Col 101 INS

Fig 32 - Teste de execução dos comandos “cd”, “pwd” e “execução de programas abrindo Gedit”.

4.3. Comparações

Mesmo com base no código shell do linux houve algumas diferenças, mas contendo a mesma função, entre essa diferenças é possível ver ao abrir o nosso shell que apresenta um cabeçalho para diferenciação entre os interpretadores de comando, além das cores.

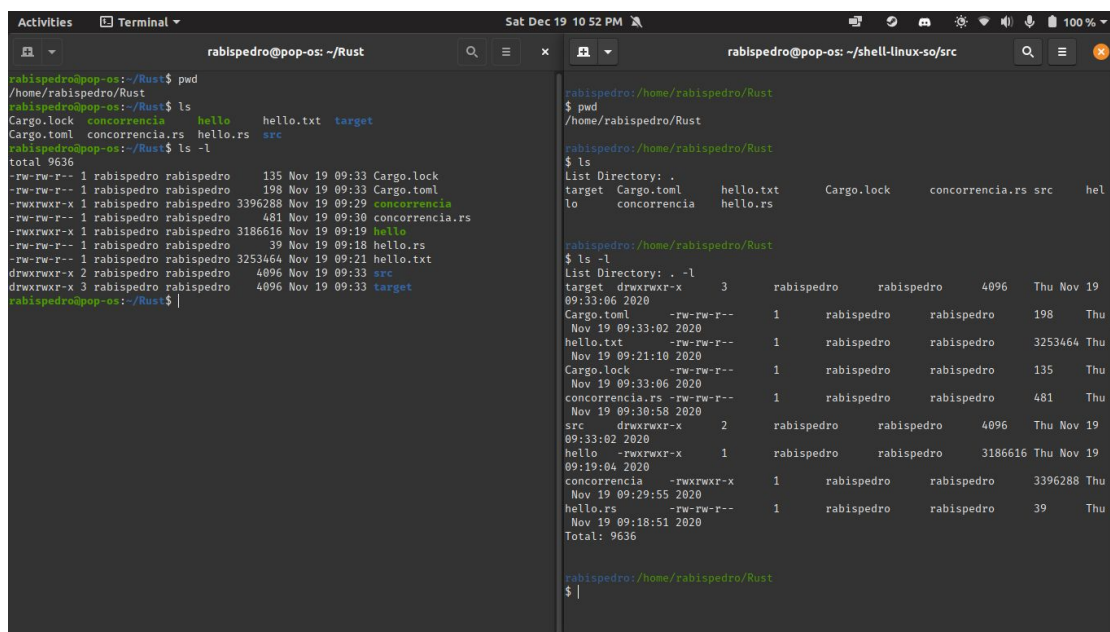


```
Initializing new SSH agent...
succeeded
Identity added: /home/luiz/.ssh/id_rsa (/home/luiz/.ssh/id_rsa)
luiz@luiz-HP-14-Notebook-PC: ~/Área de Trabalho/Trabalho final so/git/s
hell-linux
- so/src(shell_L) $ gcc cp.c -pthread -o cp
luiz@luiz-HP-14-Notebook-PC: ~/Área de Trabalho/Trabalho final so/git/s
hell-linux
- so/src(shell_L) $
```

```
luiz@luiz-HP-14-Notebook-PC: ~/Área de Trabalho/Trabalho final so/git/s
hell-linux
- so/src(shell_L) $ ./shell

SHELL 12/2020
Hericles Felipe Ferraz - 11811EMT022
Luiz da Silva Moura - 11611EMT028
Marcus Vinicius Miata - 11811BCC017
Pedro Henrique Rabis Diniz - 11811BCC024
-----
luiz:/home/luiz/Área de Trabalho/Trabalho final so/git/shell-linux-so/
src
$
```

Fig 34 - Comparação entre os dois interpretadores de comando, onde o da esquerda é do linux padrao e o da direita é o criado



```
rabispedro@pop-os: ~/Rust
$ pwd
/home/rabispedro/Rust
$ ls
Cargo.lock  concorrencia  hello      hello.txt  target
Cargo.toml  concorrencia.rs  hello.rs  src
rabispedro@pop-os: ~/Rust $ ls -l
total 9636
-rw-rw-r-- 1 rabispedro rabispedro 135 Nov 19 09:33 Cargo.lock
-rw-rw-r-- 1 rabispedro rabispedro 198 Nov 19 09:33 Cargo.toml
-rwxrwxr-x 1 rabispedro rabispedro 3396288 Nov 19 09:29 concorrencia
-rw-rw-r-- 1 rabispedro rabispedro 481 Nov 19 09:30 concorrencia.rs
-rwxrwxr-x 1 rabispedro rabispedro 3186616 Nov 19 09:19 hello
-rw-rw-r-- 1 rabispedro rabispedro 39 Nov 19 09:18 hello.rs
-rw-rw-r-- 1 rabispedro rabispedro 3253464 Nov 19 09:21 hello.txt
drwxrwxr-x 2 rabispedro rabispedro 4096 Nov 19 09:33 src
drwxrwxr-x 3 rabispedro rabispedro 4096 Nov 19 09:33 target
rabispedro@pop-os: ~/Rust $
```

```
rabispedro:/home/rabispedro/Rust
$ pwd
/home/rabispedro/Rust
$ ls
List Directory: .
target Cargo.toml hello.txt Cargo.lock concorrencia.rs src hel
lo concorrencia hello.rs

rabispedro:/home/rabispedro/Rust
$ ls -l
List Directory: . -l
target drwxrwxr-x 3 1 rabispedro rabispedro 4096 Thu Nov 19
09:33:06 2020
Cargo.toml -rw-rw-r-- 1 rabispedro rabispedro 198 Thu
Nov 19 09:33:02 2020
hello.txt -rw-rw-r-- 1 rabispedro rabispedro 3253464 Thu
Nov 19 09:21:10 2020
Cargo.lock -rw-rw-r-- 1 rabispedro rabispedro 135 Thu
Nov 19 09:33:06 2020
concorrencia.rs -rw-rw-r-- 1 rabispedro rabispedro 481 Thu
Nov 19 09:30:58 2020
src drwxrwxr-x 2 rabispedro rabispedro 4096 Thu Nov 19
09:33:02 2020
hello -rwxrwxr-x 1 rabispedro rabispedro 3186616 Thu Nov 19
09:19:04 2020
concorrencia -rwxrwxr-x 1 rabispedro rabispedro 3396288 Thu
Nov 19 09:29:55 2020
hello.rs -rw-rw-r-- 1 rabispedro rabispedro 39 Thu
Nov 19 09:18:51 2020
Total: 9636

rabispedro:/home/rabispedro/Rust
$
```

FIG 35 - Comparação entre os comandos “pwd”, “ls” e “ls -l” entre o Bash (à esquerda) e o shell desenvolvido no projeto (à direita).

The screenshot shows two terminal windows side-by-side. The left window is a standard Bash shell with the prompt 'rabispedro@pop-os: ~/Rust'. It shows the output of 'ls -R' for a directory structure containing Cargo.lock, Cargo.toml, and various source files. The right window is a custom shell with the prompt 'rabispedro@pop-os: ~/shell-linux-so/src'. It shows the output of '\$ ls -R', which replicates the same directory listing as the Bash shell, demonstrating that the custom shell's 'ls' command works identically to the system's.

FIG 36 - Comparação entre o comando “ls -R” entre o Bash (à esquerda) e o shell desenvolvido no projeto (à direita).

The screenshot shows two terminal windows side-by-side. The left window is a Bash shell with the prompt 'rabispedro@pop-os: ~/Rust'. It shows the output of 'screenfetch' (displaying system information like OS, kernel, and hardware), 'mkdir teste1', and 'ls' (showing the contents of the current directory). The right window is a custom shell with the prompt 'rabispedro@pop-os: ~/shell-linux-so/src'. It shows the output of '\$ screenfetch', '\$ mkdir teste2', and '\$ ls', which all produce the same results as the corresponding commands in the Bash shell, confirming the custom shell's ability to execute system utilities correctly.

FIG 37 - Comparação entre os comandos “execução de programas (screenfetch)”, “mkdir” e “ls” entre o Bash (à esquerda) e o shell desenvolvido no projeto (à direita).

```
Activities Terminal Sat Dec 19 11 03 PM
rabispedro@pop-os: ~/Rust/src
rabispedro@pop-os:~/Rust$ rmdir teste2
rabispedro@pop-os:~/Rust$ ls
Cargo.lock  concorrencia  hello  hello.txt  target
Cargo.toml  concorrencia.rs  hello.rs  src
rabispedro@pop-os:~/Rust$ vim teste1.txt
rabispedro@pop-os:~/Rust$ cp teste2.txt teste1.txt
rabispedro@pop-os:~/Rust$ ls
Cargo.lock  concorrencia  hello  hello.txt  target  teste2.txt
rabispedro@pop-os:~/Rust$ cd src
rabispedro@pop-os:~/Rust/src$ ls
main.rs  teste2.txt
rabispedro@pop-os:~/Rust/src$ |

rabispedro:/home/rabispedro/Rust
$ rmdir teste1
rabispedro:/home/rabispedro/Rust
$ ls
List Directory: .
target  Cargo.toml  hello.txt  Cargo.lock  concorrencia.rs  src  hel
lo  concorrencia  hello.rs
rabispedro:/home/rabispedro/Rust
$ vim teste2.txt
Name vim teste2.txt
rabispedro:/home/rabispedro/Rust
$ cp teste2.txt src
rabispedro:/home/rabispedro/Rust
$ ls
List Directory: .
teste1.txt  target  Cargo.toml  hello.txt  Cargo.lock  concorrenci
a.rs  src  hello  concorrencia  hello.rs  teste2.txt
rabispedro:/home/rabispedro/Rust
$ cd src
rabispedro:/home/rabispedro/Rust/src
$ ls
List Directory: .
main.rs  teste2.txt
rabispedro:/home/rabispedro/Rust/src
$ |
```

FIG 38 - Comparação entre o comandos “rmdir”, “cp”, “ls” e “cd” entre o Bash (à esquerda) e o shell desenvolvido no projeto (à direita).

```
hericles:/home/hericles/shell-linux-so/src
$ pwd | wc
/home/hericles/shell-linux-so/src
hericles:/home/hericles/shell-linux-so/src
$
1      1      34

hericles@Dolores: ~/shell-linux-so/src 142x18
$ cd ../../../../
hericles@Dolores: ~/shell-linux-so/src 130 ↵
$
hericles@Dolores: ~/shell-linux-so/src 130 ↵
$ pwd | wc
1      1      34
hericles@Dolores: ~/shell-linux-so/src
$
```

FIG 39 - Comparação entre o pipe (shell desenvolvido a cima) e o zsh abaixo

5. Possíveis Melhorias

5.1. Ideias Complementares

- Cores no terminal: utilização de cores para o output dos comandos a fim de facilitar o entendimento dos dados apresentados pelos mesmos.
- Armazenador de comandos utilizados: memorizar comandos do usuário para se ter a possibilidade de reutilizá-los posteriormente, facilitando a utilização do programa.
- Desenvolver-se uma própria função de leitura para receber os comandos do usuário, permitindo-se editar caracteres voltando no buffer. Acrescentar um histórico de comandos para acesso rápido também seria algo bastante válido para deixar o shell com melhor performance.

6. Resultados e Conclusões

Com a finalização deste projeto, conclui-se que um Shell é uma ferramenta extremamente poderosa, agilizando comandos através de scripts, provendo feedback textual sobre o funcionamento interno de determinados processos, dando acesso a partes do Sistema Operacional (respeitando os devidos privilégios de usuário). Assim, seu funcionamento é igualmente complexo, exigindo padrões e estruturas específicas da organização mantenedora do projeto, no caso POSIX (com padrões especificados no projeto GNU).

6.2. Observações Adicionais

É válido ressaltar que este projeto não contempla toda a funcionalidade e riqueza de comandos que um Shell normalmente dispõe, atendo-se apenas ao escopo de ser um estudo sobre o mesmo. Há limitações tanto na quantidade de comandos quanto na funcionalidades dos mesmos.

7. Referências Bibliográficas

<https://app.creately.com/diagram>
<https://app.diagrams.net/>
<https://www.ime.usp.br/~pf/algoritmos/aulas/solucoes/io1.html>
<https://www.inf.unioeste.br/~marcio/SO/Aula9SistemadeArquivos.pdf>
<https://www.geeksforgeeks.org/operating-systems/>
<https://sites.uclouvain.be/SystInfo/usr/include/dirent.h.html>
<https://man7.org/linux/man-pages/man2/>
<https://stackoverflow.com/questions/48970420/>
<https://www.cplusplus.com/reference/cstdlib/>
<https://stackoverflow.com/questions/7430248/>
<http://www.br-c.org/doku.php/>
[https://pt.wikipedia.org/wiki/Shell_\(computa%C3%A7%C3%A3o\)](https://pt.wikipedia.org/wiki/Shell_(computa%C3%A7%C3%A3o))
https://en.wikipedia.org/wiki/Command-line_interface
https://en.wikipedia.org/wiki/Graphical_user_interface
[https://en.wikipedia.org/wiki/Shell_\(computing\)](https://en.wikipedia.org/wiki/Shell_(computing))
<https://github.com/parthnan/Shell-With-n-Pipe-in-C/>