

RELATÓRIO - LISTA DE IMPLEMENTAÇÃO 01

Para iniciarmos nossa implementação do shell, pesquisamos quais bibliotecas iríamos precisar para fazer nosso código na linguagem C. Como resultado deste estudo, decidimos incluir as bibliotecas na imagem abaixo:

```
// SPDX-License-Identifier: GPL-3.0
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <stdlib.h>
```

Destaque para algumas bibliotecas importantes e suas principais funções:

- **sys/wait.h** - Inclui a função `waitpid`, utilizada para manipular a ordem da execução dos processos.
- **unistd.h** - Utilizado pois inclui as funções `chdir`, `dup2`, `execvp`, e `fork` usadas para execução dos processos unix.
- **fcntl.h** - Controle e manipulação de arquivos, abrir arquivos etc.

A primeira função que fizemos foi o “cd”, que é conhecido como `chdir` e é um comando que significa “change directory” ou seja, mudar diretório.

```
void cd(char **args)
{
    if (args[1] != NULL)
        chdir(args[1]);
    else
        printf("argumento esperado\n");
}
```

Quando iniciamos nosso shell no prompt de comando, é possível executar o comando “help” que faz a apresentação das funcionalidades do shell implementado para o usuário. Já o comando “exit”, finaliza o programa e printa uma mensagem de encerramento.

```
$ ./"shell"
ll seq> help
```



```
-----
|                               |
| PROMPT DE LETICIA E LUCAS   |
| Digite nomes de programas e argumentos, e aperte enter          |
| as seguintes funcoes estao implementadas:                       |
|   cd                                                                |
|   help                                                            |
|   exit                                                            |
| Use '|' para criar pipes, '<', '>' e '>>' para                    |
| redirecionar entradas e saidas                                   |
|                               |
|-----|
```

```
ll seq> exit
```



```
BYE!
```

Logo de início, depois do tratamento da entrada e separação dos argumentos, fazemos uso da função `execute` para interpretar os comandos solicitados do usuário, e assim executar o comando desejado.

Na função identificamos que o argumento corresponde a alguma das nossas funções implementadas e a executamos, ou executamos o argumento através da `execvp`, também vemos se o `style` do shell foi alterado e executamos o `waitpid` para o sequencial.

```
void execute(char *args[])
{
    pid_t pid;

    if (strcmp(args[0], "cd") == 0) {
        cd(args);
    } else if (strcmp(args[0], "help") == 0) {
        help(args);
    } else if (strcmp(args[0], "exit") == 0) {

        printf("
        printf("
        printf("
        printf("
        executar = 0;
    } else {

        pid = fork();

        if (pid == 0) {

            execvp(args[0], args);
        } else {
            if (sequential)
                waitpid(pid, NULL, 0);
            else
                sequential = 0;
        }
        redirecionar_entrada("/dev/tty");
        redirecionar_saida("/dev/tty", 'w');
    }
}
```

```
void redirecionar_entrada(char *nome_arquivo)
{
    int in = open(nome_arquivo, O_RDONLY);

    dup2(in, 0);
    close(in);
}
```

A função `direcionar entrada` é responsável por “puxar” o conteúdo do arquivo como dados a serem manipulados pela próxima função, ela é chamada quando se utiliza “<” entre argumentos.

```

void redirecionar_saida(char *nome_arquivo, char tipo)
{
    if (tipo == 'w') {
        int out = open(nome_arquivo, O_WRONLY | O_TRUNC | O_CREAT, 0600);

        dup2(out, 1);
        close(out);
    } else {
        int out = open(nome_arquivo, 'a', 0600);

        dup2(out, 1);
        close(out);
    }
}

```

A redirecionar saída é usada para guardar as informações que seriam impressas no prompt, em um arquivo de destino, usada quando o usuário solicita através do “>”.

```

void createPipe(char *args[])
{
    int fd[2];

    pipe(fd);

    dup2(fd[1], 1);
    close(fd[1]);

    execute(args);

    dup2(fd[0], 0);
    close(fd[0]);
}

```

Na função CreatePipe, fizemos o uso do pipe e do dup2, que são chamadas de sistema bastante utilizadas. O Pipe (|) é uma ferramenta indispensável. Ele permite direcionar a saída de um comando para outro, permitindo criar vários tipos de filtros e executar operações complicadas de uma forma muito mais concisa. Sem o Pipe, muitas funções simples seriam muito maiores e complicadas. Por outro lado, a chamada de sistema duplo apaga o antigo descritor de arquivo e insere o novo descritor. Isso é do nosso interesse pois, por exemplo, para fazer dois processos se comunicarem o pai e o filho usando a chamada de sistema pipe(), é preciso apagar o antigo descritor de saída padrão do filho e redirecionar para a saída do pipe. Assim, ao terminar a operação você volta ao processo pai, restaura o descritor de arquivo de saída padrão e redireciona a entrada padrão para o pai, usando a dup2() novamente.

```

char *formatar_entrada(char *entrada, int tamanho)
{
    int j = 0;
    char *formatado = (char *)malloc((1024) * sizeof(char));

    for (int i = 0; i < tamanho; i++) {
        if (entrada[i] != '>' && entrada[i] != '<' && entrada[i] != '|' &&
            entrada[i] != ';') {
            formatado[j++] = entrada[i];
        } else {
            formatado[j++] = ' ';
            formatado[j++] = entrada[i];
            if (entrada[i + 1] == '>') {
                formatado[j++] = entrada[i + 1];
                i++;
            }
            formatado[j++] = ' ';
        }
    }
    formatado[j++] = '\\0';
    formatado[j++] = '\\0';

    return formatado;
}

```

O `formatar_entrada`, como o nome já diz, é responsável pelo tratamento de entrada e evita erros entre os comandos por conta de espaçamentos extras, por exemplo.

O shell rodando no terminal:

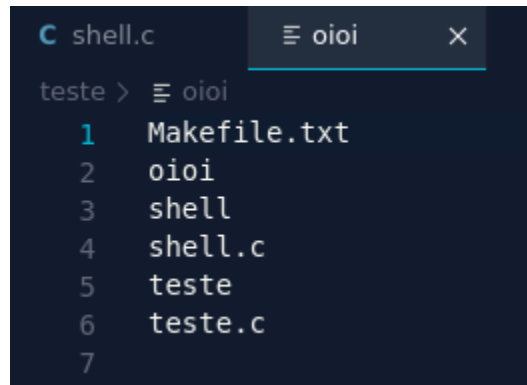
```

$ ./"shell"
ll seq>    ls |      sort
Makefile.txt
shell
shell.c
teste
teste.c
ll seq> ls  >  oioi
ll seq> ls
Makefile.txt oioi shell shell.c teste teste.c
ll seq>

```

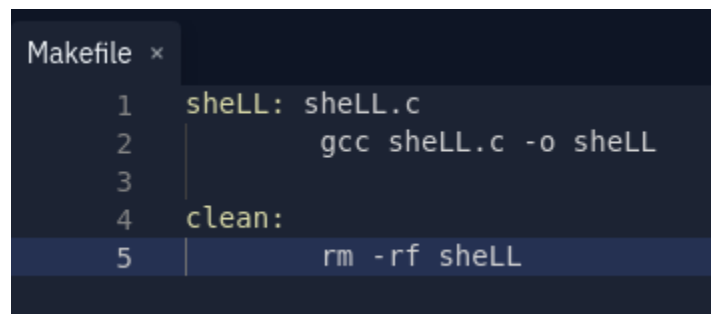
No exemplo acima, visualizamos na prática que o comando `ls` com o pipe e o `sort`, mesmo com vários espaços aleatórios entre os comandos, funciona mesmo assim (mostrou os arquivos na vertical). Do mesmo modo, o tratamento de erro com espaços entre os redirecionamentos

também funciona, e no segundo “ls” dá para ver que o arquivo “oioi” foi criado e dentro dele tem os arquivos do diretório atual.



```
C shell.c  oioi x
teste > ls
1  Makefile.txt
2  oioi
3  shell
4  shell.c
5  teste
6  teste.c
7
```

Inclusive, vamos agora discorrer sobre o Makefile implementado.



```
Makefile x
1  sheLL: sheLL.c
2      gcc sheLL.c -o sheLL
3
4  clean:
5      rm -rf sheLL
```

Nele, fizemos duas simples macros, o “sheLL” simplesmente cria o executável do sheLL.c e o “-o” renomeia para apenas “sheLL”. Assim, na hora de compilar, precisamos apenas digitar no terminal o “./sheLL”. Além disso, quando finalizar o programa, é possível usar o comando “make clean” que tem o papel de apagar o arquivo executável que criamos logo quando rodamos o Makefile. Na imagem abaixo fica melhor para visualizar como o nosso Makefile funciona. Primeiro, apenas o “make” já cria o arquivo sheLL (e na tela printa o comando que foi executado - gcc sheLL.c -o sheLL). Depois disso, com o “ls” listamos todos os arquivos do diretório e certificamos que o sheLL executável foi criado. E por fim, depois de sair do programa, podemos digitar “make clean” e é realizado o rm -rf sheLL que apaga o arquivo. O funcionamento pode ser comprovado quando digitamos “ls” novamente e checamos que o arquivo foi apagado com sucesso.


```
int main(int argc, char *argv[])
{
    char *args[512];
    char *comandos;
    char *style = "ll seq> ";
    int batch = 0;
    char *nome_arquivo;
    FILE *batch_file;

    if (argc == 2) {
        batch = 1;
        nome_arquivo = argv[1];

        batch_file = fopen(argv[1], "r");

        if (batch_file == NULL){
            printf("arquivo nao encontrado\n");
            return 0;
        }
    } else if (argc > 2) {
        printf("entrada invalida\n");
        return 0;
    }

    while (executar) {
        char entrada[512];

        if (!batch){
            printf("%s", style);
            if(fgets(entrada, 512, stdin) == NULL){
                args[0] = "exit";
                execute(args);
            }
        } else {

            if (fgets(entrada, 512, batch_file) == NULL){
                break;
            }
            printf("\n%s\n", entrada);
        }
    }
}
```



```

while (arg) {
    if (*arg == '<') {
        redirecionar_entrada(strtok(NULL, " "));
    } else if (strcmp(arg, ">>") == 0) {
        redirecionar_saida(strtok(NULL, " ") , 'a');
    } else if (*arg == '>') {
        redirecionar_saida(strtok(NULL, " "), 'w');
    } else if (*arg == ';') {
        args[i] = NULL;
        execute(args);
        i = 0;
    } else if (*arg == '|') {
        args[i] = NULL;
        criar_pipe(args);
        i = 0;
    } else {
        args[i] = arg;
        i++;
    }
    arg = strtok(NULL, " ");
}

args[i] = NULL;

execute(args);

```

Posteriormente a esse tratamento da execução por batch ou não, temos o loop que percorre os argumentos para identificar os operadores que indicam o redirecionamento ou o pipe, e também o “;” que é utilizado para executar vários comandos separados indicados em uma mesma linha.