# Shellso - um interpretador de comandos

**Giovanna M. Garbácio, Marcos Vinícius T. Coêlho**
Departamento de Ciência da Computação
Universidade Federal de Roraima (UFRR) – Boa Vista, RR – Brazil

gio.garbacio@gmail.com, marcosvinicius.bv@hotmail.com

***Abstract.*** *A terminal is an interface that acts as an intermediary between the user and the operating system. This work presents a theoretical basis about terminals and, for learning purposes, it aims to develop a command interpreter, called Shellso, using C programming language. It accepts most conventional Linux terminal commands, in addition to creating and treating processes, and despite attempts, it does not support wildcards and character encoding in the terminal.*

***Resumo.*** *Um terminal é uma interface que realiza um intermediário entre o usuário e o sistema operacional. Esse trabalho apresenta uma base teórica sobre um terminal e, para efeitos de aprendizado tem objetivo de desenvolver um interpretador de comandos, chamado Shellso, utilizando a linguagem de programação C. Ele aceita a maioria dos comandos convencionais do terminal Linux, além da criação de processos e tratamento dos mesmos, no entanto, ele possui suporte a wildcards e a codificação de caracteres no terminal.*

## 1. Introdução

Um interpretador de comandos é uma interface que permite que o sistema operacional e o usuário interajam por meio de inputs de linhas de comando ou arquivos, que são analisados e interpretados permitindo que o Kernel execute as ações correspondentes ao que foi inserido pelo usuário (KIDWAI et. al, 2021). Ele também pode ser chamado de CLI, (Command Line Interface), interface de usuário do console, terminal ou processador de comando.

Existem diferentes interpretadores de comandos que podem ter características e recursos adicionais ou diferenciados, dependendo do sistema operacional e da sua implementação específica. Na maioria dos sistemas operacionais Windows, por exemplo, os principais interpretadores de linhas de comando são o Prompt de Comando (Command Prompt) e o PowerShell. O Shell padrão mais comumente utilizado em sistemas operacionais Linux é o Bash (Bourne Again Shell) mas existem muitos outros como C Shell, Korn Shell, The Z shell e ash shell.

Entre as principais funções de um terminal são: (a) receber e analisar comandos inseridos pelo usuário. Isso envolve separar os elementos do comando, como o nome do comando e os argumentos, e verificar sua sintaxe para garantir que estejam corretos; (b) executar comandos internos que são implementados diretamente no interpretador e não exigem a execução de programas externos; (c) localizar e executar programas externos; (d) gerenciar processos no sistema operacional, ele pode iniciar programas em execução em primeiro ou segundo plano e permitir que o usuário controle esses processos; (e) redirecionar entrada e saída dos comandos; (f) suporte a wildcards (caracteres curinga como *, ~, ?, elas representam padrões ou conjuntos de caracteres que correspondem a nomes de arquivos ou

diretórios) para corresponder a múltiplos arquivos ou diretórios; e (g) execução de scripts (sequências de comandos armazenadas em um arquivo).

A implementação do shell é composta por três componentes distintos: o parser, o executor e os subsistemas do terminal. O parser é responsável por analisar os comandos do usuário com base na Tabela de Comandos e interpretar a sua sintaxe e guardá-los em uma lista de comandos a serem executados, em seguida, o executor cria processos para executar os comandos analisados pelo parser e garante que eles sejam realizados, ainda, se necessário, o executor do shell pode estabelecer pipes para permitir a comunicação entre a saída de um processo e a entrada de outro, por último, os subsistemas do terminal são componentes adicionais que fornecem funcionalidades e recurso ao shell, como manipulação de variáveis, histórico, wildcards, autocompletar, etc.

Para efeitos de aprendizado, esse presente trabalho tem objetivo de descrever o desenvolvimento de um mini terminal Linux, ou seja, um interpretador de comandos, na linguagem de programação C. O terminal, apelidado de Shellso, aceita a maioria dos comandos convencionais do terminal Linux, além da criação de processos e tratamento dos mesmos. Foi tentado também a implementação no shell de suporte a wildcards e codificação de caracteres no terminal, onde não foi obtido sucesso.

Esta seção apresentou uma resumida base teórica para a aplicação da prática de desenvolver um shell, a Seção 2 apresenta a metodologia de como foi realizado o projeto, a Seção 3 descreve a implementação para criação de processos, a implementação de pipe e redirecionamento no terminal, a lista de comandos disponíveis e as tentativas em relação ao suporte a wildcards e codificação de caracteres no terminal.

## 2. Método proposto

Durante o processo de desenvolvimento, foram pesquisados diversos recursos disponíveis na web, como documentações, tutoriais passo a passo e exemplos de códigos relacionados à criação de um shell. Esses recursos foram explorados para compreender os conceitos e técnicas necessárias para a implementação. Os sites e fóruns consultados abordavam tópicos relacionados à programação em C, sistemas operacionais, gerenciamento de processos e interação com o usuário através do terminal. Através dessa pesquisa, foram adquiridos conhecimentos práticos e teóricos que serviram como base para o desenvolvimento do Shellso.

## 3. Avaliação experimental e resultados

Para realizar a criação de processos em um terminal, primeiramente é criada uma variável do tipo pid_t. Em seguida, fazemos uma verificação para determinar se o processo será executado em segundo plano. A criação do processo é feita através da função fork(), e seu retorno é atribuído à variável pid. Em seguida, é realizada uma verificação para determinar o valor atual de pid. Se pid for menor que 0, significa que o processo falhou em ser criado. Se for igual a 0, significa que o processo foi criado e o processo filho está sendo executado no momento. Se pid for maior que 0, significa que o Shellso já retornou ao processo pai. Caso seja um processo que vá rodar em segundo plano, ele retorna o PID do processo e continua rodando. Se não for um processo que deva rodar em segundo plano, o Shellso espera que ele seja concluído para dar continuidade.

```c
int executar_processos(char** parseArgs){
  pid_t pid;
  int back = background(parseArgs);
  int argc = argCount(parseArgs);
  char* inputFile = NULL;
  char* outputFile = NULL;
  if (argc >= 3) {
    if (strcmp(parseArgs[argc - 2], "<") == 0) {
      inputFile = parseArgs[argc - 1];
      parseArgs[argc - 2] = NULL;
    } else if (strcmp(parseArgs[argc - 2], ">") == 0) {
      outputFile = parseArgs[argc - 1];
      parseArgs[argc - 2] = NULL;
    }
  }
  signal(SIGCHLD, handle_sigchld);
  pid = fork();
  if (pid == -1) {
    printf("\nFalha no forking");
    return 1;
  } else if (pid == 0) {
    // Redirecionando a entrada
    if (inputFile != NULL) {
      int inputFd = open(inputFile, O_RDONLY);
      if (inputFd < 0) {
        printf("\nErro ao abrir o arquivo de entrada");
        exit(1);
      }
      dup2(inputFd, STDIN_FILENO);
      close(inputFd);
    }
```

FIGURA 1 - Função de executar processos únicos parte 1

```c
    // Redirecionando a saída
    if (outputFile != NULL) {
      int outputFd = open(outputFile, O_WRONLY | O_CREAT | O_TRUNC, 0600);
      if (outputFd < 0) {
        printf("\nErro ao abrir o arquivo de saída");
        exit(1);
      }
      dup2(outputFd, STDOUT_FILENO);
      close(outputFd);
    }
    close(STDIN_FILENO);
    if (execvp(parseArgs[0], parseArgs) < 0) {
        printf("\nNão foi possível executar o comando..");
    }
    exit(0);
  } else {
      if(back == 0){
        wait(NULL);
      } else{
        printf("\nPID Filho 1 [%d]",pid);
      }
      return 1;
  }
}
```

FIGURA 2 - Função de executar processos únicos parte 2

Para a implementação de pipes, durante o processamento das strings é feita uma verificação para determinar se a linha de comando possui um pipe. Em seguida, a linha de comando é dividida em partes separadas pelos pipes. É aplicada a função parseSpace em cada uma das linhas de comandos para separar os argumentos em um vetor. Então, o pipe é executado usando uma função diferente daquela que executa processos individuais. Essa função é bastante similar àquela que executa processos individuais, mas ela cria um pipe e executa dois processos simultaneamente, garantindo que o segundo processo sempre receba a saída do primeiro através do dup2.

```c
int parsePipe(char* str, char** strpiped){
  //Checa se existe pipe
  int i;
  for (i = 0; i < 2; i++) {
    strpiped[i] = strsep(&str, "|");
    if (strpiped[i] == NULL)
        break;
  }
  if (strpiped[1] == NULL)
        return 0; // Se não tiver Pipe
    else {
        return 1;
    }
}
```

FIGURA 3 - Identificador e separador de pipe

```c
int inputProcess(char* str, char** parsed, char** parsedPipe){
  char *strpiped[2];
  int piped = 0;
  removerN(str);
  piped = parsePipe(str, strpiped);

  if(piped == 1){
    parseSpace(strpiped[0],parsed);
    parseSpace(strpiped[1],parsedPipe);
  } else{
    parseSpace(str,parsed);
  }
  if(piped ==1){
    return 1;
  }else{
    return 0;
  }

}
```

FIGURA 4 - Processador de Strings

```c
if (p1 == 0) {

  close(pipefd[0]);
  dup2(pipefd[1], STDOUT_FILENO);
  close(pipefd[1]);

  if (inputFile != NULL) {
    int inputfd = open(inputFile, O_RDONLY);
    if (inputfd < 0) {
        perror("Erro ao abrir arquivo de entrada");
        exit(1);
    }
    dup2(inputfd, STDIN_FILENO);
    close(inputfd);
  }
  close(STDIN_FILENO);
  if (execvp(parseArgs[0], parseArgs) < 0) {
      printf("\nNão foi possível executar o comando 1..");
      exit(0);
    }
} else {
```

FIGURA 5 - Executor do primeiro comando de um pipe

```c
if (p2 == 0) {

  close(pipefd[1]);
  dup2(pipefd[0], STDIN_FILENO);
  close(pipefd[0]);

  if (outputFile != NULL) {
  int outputFd = open(outputFile, O_WRONLY | O_CREAT | O_TRUNC, 0600);
  if (outputFd < 0) {
    printf("\nErro ao abrir o arquivo de saída");
    exit(1);
  }
  dup2(outputFd, STDOUT_FILENO);
  close(outputFd);
  }
  close(STDIN_FILENO);
  if (execvp(parseArgsPipe[0], parseArgsPipe) < 0) {
      printf("\nNão foi possível executar o comando 2..");
      exit(0);
  }
} else {
```

FIGURA 6 - Executor do segundo comando de um pipe

Na implementação do redirecionamento de entrada/saída, é realizada uma verificação para determinar se existem as chaves de redirecionamento "<=" ou "=>". Após essa análise, é feita uma contagem dos argumentos da linha de comando para identificar o nome do arquivo e salvá-lo. Em seguida, a chave é substituída por NULL para que não seja lida pelo execvp. Dentro do processo filho, o arquivo é aberto usando argumentos diferentes, dependendo se é para entrada ou saída, e o STDIN ou STDOUT do arquivo é redirecionado para a execução usando o dup2. Após isso, o arquivo é fechado e o código executa o comando normalmente, utilizando o arquivo como entrada ou saída de dados.

```c
int argc = argCount(parseArgs);
char* inputFile = NULL;
char* outputFile = NULL;
if (argc >= 3) {
  if (strcmp(parseArgs[argc - 2], "<=") == 0) {
    inputFile = parseArgs[argc - 1];
    parseArgs[argc - 2] = NULL;
  } else if (strcmp(parseArgs[argc - 2], "=>") == 0) {
    outputFile = parseArgs[argc - 1];
    parseArgs[argc - 2] = NULL;
  }
}
```

FIGURA 7 - Identificador de redirecionadores

```c
// Redirecionando a entrada
if (inputFile != NULL) {
  int inputFd = open(inputFile, O_RDONLY);
  if (inputFd < 0) {
    printf("\nErro ao abrir o arquivo de entrada");
    exit(1);
  }
  dup2(inputFd, STDIN_FILENO);
  close(inputFd);
}

// Redirecionando a saída
if (outputFile != NULL) {
  int outputFd = open(outputFile, O_WRONLY | O_CREAT | O_TRUNC, 0600);
  if (outputFd < 0) {
    printf("\nErro ao abrir o arquivo de saída");
    exit(1);
  }
  dup2(outputFd, STDOUT_FILENO);
  close(outputFd);
}
```

FIGURA 8 - Redirecionando dados (Linkando os arquivos para entrada e saída)

A implementação dos wildcards não foi realizada.

A codificação de caracteres do terminal foi configurada usando a função setlocale da biblioteca locale.h. Por meio dessa função, a linguagem do Shellso foi definida como UTF-8.

```c
#include <locale.h>
```

FIGURA 9 - Importando biblioteca “locale.h”

```c
int main(void) {
  setlocale(LC_ALL, "en_US.UTF-8");
```

FIGURA 10 - Ativando codificador de caracteres

Alguns dos comandos disponíveis entre outros comandos padrões do Linux, são:

TABELA 1 - Alguns dos comandos disponíveis no Shellso

| |
|---|
| cd – Entrar em um diretório |
| fim – Finalizar o Shellso |
| ls - Lista todos os arquivos do diretório |
| df - Mostra a quantidade de espaço usada no disco rígido |
| top - Mostra o uso da memória |
| mkdir - Cria um diretório |
| rm - Remove um arquivo/diretório |
| cat - Abre um arquivo |
| grep - Buscador de correspondências |
| echo - Exibe uma linha de texto |

## 4. Conclusão e trabalhos futuros

Foi possível implementar com sucesso o Shellso, que atendeu aos objetivos propostos de fornecer uma forma prática de aprendizado e exploração dos conteúdos de interpretadores de comandos, criação e manipulação de processos, linguagem C e o gerenciamento de programas e arquivos.

Durante o processo de implementação, foi realizada uma extensa pesquisa na web, explorando sites especializados em computação e fóruns de perguntas. Essa pesquisa foi fundamental para adquirir conhecimento sobre os conceitos e técnicas necessárias para o desenvolvimento do shell, além de nos ajudar a resolver alguns problemas, como o problema do New Line (\n) que ficava salvo junto ao último argumento e estava causando erro na execução dos comandos. Os exemplos e tutoriais encontrados na internet foram adaptados e aplicados de forma apropriada no projeto.

No entanto, mesmo com o sucesso geral da implementação, não foi possível adicionar suporte a wildcards. Essa funcionalidade requer um nível de complexidade adicional e demanda um conhecimento mais avançado de programação e manipulação de strings. Dessa forma, a ausência desses recursos no shell é reconhecida como uma limitação.

Dessa forma, é indicado para trabalhos futuros a aprimoração desse mesmo shell, incluindo o suporte a wildcards. Isso pode envolver o estudo mais aprofundado desse conceito, a pesquisa de técnicas e abordagens para implementá-las e a aplicação prática no desenvolvimento do shell. Além disso, outras melhorias e expansões podem ser consideradas, como a adição de recursos avançados, a implementação de comandos adicionais e a melhoria da interface do usuário.

## Referências


KIDWAI, Abdullah; ARYA, Chandrakala; SINGH Prabhishek; DIWAKER, Manoj; SINGH, Shilpi; SHARMA, Kanika; KUMAR, Neeraj. **A comparative study on shells in Linux: A review.** Materialstoday: Proceedings. Vol. 37, parte 2, p. 2612-2616, 2021.
DOI: https://doi.org/10.1016/j.matpr.2020.08.508

DEY, Suprotik. **FAZENDO SEU PRÓPRIO LINUX SHELL EM C**. Acervo Lima. Disponível em: <https://acervolima.com/fazendo-seu-proprio-linux-shell-em-c/>. Último acesso em 27 de junho de 2023.

**Redirecting I/O in a custom shell program written in C.** StackOverflow. Disponível em: <https://stackoverflow.com/questions/52939356/redirecting-i-o-in-a-custom-shell-program-written-in-c>. Último acesso em 27 de junho de 2023.