Universidade Federal
de São João del-Rei

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
SISTEMAS OPERACIONAIS

TRABALHO PRÁTICO 1

Diogo Augusto Martins Honorato

Guilherme Garcia de Oliveira

Leonardo da Silva Vieira

São João del-Rei

2025

Sumário

1 Introdução

Este é um trabalho prático da disciplina de Sistemas Operacionais no curso de Ciência da Computação na UFSJ, tendo como docente o professor Rafael Sachetto Oliveira.

1.1 Objetivo

Neste trabalho prático, temos como objetivo aprofundar o conhecimento sobre criação de processos, uso de pipes, leitura de entrada e execução de programas no Linux por meio da construção de um shell de comandos.

2 Abordagem do Problema

A construção de um Shell de comandos em linguagem C teve como objetivo principal a criação de uma interface capaz de interpretar e executar comandos inseridos pelo usuário, simulando o comportamento básico de terminais. A abordagem adotada seguiu uma estrutura modular, separando responsabilidades em componentes distintos para facilitar manutenção, compreensão e futuras expansões.

Inicialmente, o shell foi projetado para ler entradas do usuário, identificar comandos internos (como cd e exit) e executar comandos externos através da criação de processos filhos com fork() e execvp(). Em seguida, foram adicionadas funcionalidades como execução em background (com &) e, posteriormente, suporte à execução encadeada de comandos com pipes (|), utilizando chamadas de sistema como pipe(), dup2() e wait() para estabelecer a comunicação entre processos.

Essa abordagem incremental permitiu o desenvolvimento controlado de cada funcionalidade, garantindo que o shell permanecesse funcional em todas as etapas e facilitando testes modulares.

3 Introdução aos Algoritmos

Para compreender melhor as análises deste trabalho, iremos, de maneira breve, explicar um pouco do funcionamento dos algoritmos implementados, sendo eles

3.1 Função `ShellCd`

A função shellCd implementa o comando interno cd, que é usado para mudar o diretório de trabalho atual do shell. Funcionalidade:

• Recebe o vetor de strings argv (argumentos do comando). argv[0] contém "cd"e argv[1] (se existir) contém o diretório para o qual mudar.

• Se argv[1] for NULL, tenta mudar para o diretório home do usuário (obtido através da variável de ambiente HOME).

• Se argv[1] existir, tenta mudar para o diretório especificado.

• Em caso de erro ao mudar o diretório (por exemplo, diretório não existe), imprime uma mensagem de erro usando perror.

3.2 Função `help`

A função shellHelp implementa o comando interno help, que exibe uma mensagem de ajuda com os comandos internos disponíveis e algumas instruções de uso do shell.

3.3 Função `executeCommand`

A função executeCommand é responsável por interpretar e executar os comandos inseridos pelo usuário no shell. Ele faz a análise do comando, verifica se é um comando interno (builtin) ou externo, e executa a ação apropriada.

Funções Principais:

• Recebe a string do comando e a raiz da árvore Trie (que armazena os comandos internos).

• Utiliza a função tokenString() (do arquivo ReadCommand.c) para dividir o comando em tokens (palavras individuais).

• Verifica se o comando é vazio e, se for, libera a memória e retorna.

• Identifica se o comando deve ser executado em background (verificando o símbolo "&").

• Conta a quantidade de pipes ("|") no comando.

• Se houver pipes (pipeCount > 0): Chama a função execute_with_pipes() (do arquivo Pipe.c) para lidar com a execução dos comandos encadeados por pipes. Esta função se encarrega de criar os pipes necessários, bifurcar processos para cada comando, redirecionar as entradas e saídas adequadamente e aguardar a finalização dos processos filhos.

• Se não houver pipes: Procura o comando na árvore Trie (Search(root, tokens[0])). Se encontrar, chama a função interna correspondente. Se não for um comando interno, assume que é um comando externo e chama a função createProcess() (do arquivo CreateProcess.c) para criar um novo processo e executá-lo.
• Libera a memória alocada para os tokens

3.4 Função `Pipe`

A funcionalidade de pipes (|) foi adicionada ao shell com o objetivo de permitir a execução de comandos encadeados, onde a saída padrão de um processo serve como entrada padrão para o próximo.

A implementação se baseia na criação de múltiplos processos filhos, conectados por canais de comunicação (pipes), utilizando as chamadas de sistema pipe(), fork(), dup2() e execvp().

3.5 Função `createProcess`

A função `createProcess` é responsável por criar um novo processo com a chamada `fork()` e executar um comando utilizando `execvp()`. Caso o comando deva ser executado em background, o processo pai não espera sua finalização. Caso contrário, o processo pai aguarda o término do processo filho com `wait()`.

3.6 Função `readCommand`

A função `readCommand` realiza a leitura do comando digitado pelo usuário por meio da biblioteca `readline`. Também trata o sinal `SIGINT` para garantir que o shell permaneça estável após interrupções (Ctrl+C) e adiciona o comando ao histórico de execução.

3.7 Estrutura `Trie`

A estrutura `Trie` é utilizada no shell para armazenar e gerenciar comandos internos (builtins) de forma eficiente. Ela permite associar uma sequência de caracteres (nome do comando) a uma função específica, facilitando a execução rápida de comandos implementados diretamente no shell, como `cd`, `exit` e `help`.

Etapas da Implementação:

1. Divisão do comando

• O comando inserido pelo usuário é dividido em subcomandos usando o delimitador |.

• Cada subcomando representa uma etapa da pipeline.

2. Criação de Pipes

•Para conectar cada par de comandos, é criado um pipe com pipe(pipefd).

•Um pipe fornece dois descritores de arquivo: um para leitura (pipefd[0]) e outro para escrita (pipefd[1]).

3. Criação de Processos (fork)

•Para cada subcomando, é criado um processo filho usando fork().

4. Redirecionamento de Entrada e Saídas

•Se o processo não é o primeiro, seu stdin é redirecionado para a leitura do pipe anterior.

•Se o processo não é o último, seu stdout é redirecionado para a escrita no pipe atual.

5. Execução do Comando

•Cada processo executa seu subcomando com execvp(), que substitui o processo atual pela execução do comando desejado.

6. Fechamento de Descritores

•Os processos pais fecham os descritores que não utilizam, evitando vazamentos de recursos.

7. Sincronização com wait()

•O processo pai espera a finalização de todos os filhos para manter a sincronização e evitar zumbis.

4 Conclusão

A implementação de um Shell de comandos em linguagem C proporcionou uma compreensão prática e aprofundada dos principais mecanismos de controle de processos, manipulação de entrada e saída, e comunicação entre processos no ambiente Unix/Linux.Durante o projeto, foi possível aplicar na prática conceitos como criação de processos com fork(), execução de programas com execvp() e comunicação entre processos com pipe().

Foram adicionadas funcionalidades importantes, como comandos internos, execução de comandos externos, suporte à execução em background e uso de pipes para encadear comandos.

No geral, esse projeto contribuiu bastante para o entendimento de como o terminal opera por trás das interfaces gráficas, reforçando conhecimentos sobre processos, chamadas de sistema e controle de fluxo na linguagem C, de compreender o funcionamento do sistema operacional para resolver problemas mais complexos.

5 Referências

TANENBAUM, A. S. Sistemas operacionais modernos. Rio De Janeiro (Rj): Prentice-Hall Do Brasil, 2010.