Trabalho de Explorador de Arquivos

Integrantes;

- Gabriel Lima Scheffler
- Lucas Scheffer Hundsdorfer
- Gustavo Romagnoli Mazur

Resumo

Este documento apresenta uma análise técnica aprofundada de um simulador de terminal de linha de comando implementado em linguagem C. O escopo da análise abrange desde as bibliotecas e estruturas de dados fundamentais até a lógica de implementação de cada comando da interface do usuário e suas respectivas funções auxiliares. Detalha-se a arquitetura de dados baseada em uma árvore n-ária, o ciclo de vida da aplicação, as rotinas de manipulação de dados, o gerenciamento de memória e as estratégias de persistência de dados. O objetivo é fornecer uma visão completa e formal da engenharia de software empregada no projeto.

1. Introdução

O software em análise constitui um simulador de terminal que emula o comportamento de uma interface de linha de comando (CLI) para navegação e manipulação de um sistema de arquivos virtual. Este sistema é inteiramente mantido em memória durante a execução do programa, sendo sua estrutura carregada de um arquivo de texto no início da sessão e salva de volta ao final. O presente documento visa dissecar os componentes de software do projeto, oferecendo uma análise funcional de seus módulos.

A implementação utiliza um conjunto de bibliotecas padrão da linguagem C, cuja inclusão é fundamental para a sua operacionalidade. A biblioteca stdio.h é empregada para todas as operações de entrada e saída (I/O), como a leitura de comandos do usuário e a manipulação de arquivos de texto. A gestão dinâmica de memória, incluindo alocação e liberação, é provida pela biblioteca stdlib.h. Por fim, a biblioteca string.h oferece o ferramental necessário para a manipulação de strings, como cópias, comparações e tokenização, operações essenciais no contexto de uma CLI.

2. Arquitetura de Dados

A representação do sistema de arquivos virtual é o pilar central da arquitetura do software. A escolha e a implementação da estrutura de dados definem a eficiência e a flexibilidade de todas as operações subsequentes.

2.1. Definição das Estruturas de Dados

A base da representação de dados consiste em uma união enumerada (enum) e uma estrutura (struct). A enumeração tipoNo categoriza cada entidade do sistema de arquivos:

C
```
typedef enum {
   PASTA,
   ARQUIVO
} tipoNo;
```

A estrutura no define a composição de cada elemento (seja pasta ou arquivo) na árvore hierárquica:

```
C
typedef struct no {
   char* nome;
   tipoNo tipo;
   struct no* pai;
   struct no* primeiroFilho;
   struct no* proxIrmao;
} no;
```

Cada campo possui uma finalidade específica:

- char* nome: Um ponteiro que armazena o nome do arquivo ou diretório. A memória para este nome é alocada dinamicamente, frequentemente através de strdup, para garantir que cada nó possua sua própria cópia do nome.
- tipoNo tipo: Armazena o tipo do nó, sendo PASTA ou ARQUIVO, conforme definido pela enumeração.
- struct no* pai: Um ponteiro para o nó ascendente direto, permitindo a navegação "para cima" na hierarquia, funcionalidade essencial para comandos como cd .. e para a construção de caminhos absolutos.
- struct no* primeiroFilho: Ponteiro para o primeiro descendente direto de um nó do tipo PASTA.
- struct no* proxIrmao: Ponteiro para o próximo nó no mesmo nível hierárquico, formando uma lista encadeada de "irmãos".

2.2. Modelo de Representação "Primeiro Filho, Próximo Irmão"

A combinação dos ponteiros primeiroFilho e proxIrmao implementa o padrão de projeto "primeiro filho, próximo irmão" (first-child, next-sibling). Esta técnica é particularmente vantajosa para representar árvores n-árias (árvores com um número variável de filhos por nó), como um sistema de arquivos. Em vez de utilizar um array de ponteiros para os filhos, que seria ineficiente em termos de memória ou limitante em capacidade, este modelo utiliza apenas dois ponteiros por nó para construir e atravessar toda a sub-árvore, oferecendo flexibilidade e otimização de memória.

3. Análise Funcional e Implementação

A funcionalidade do terminal é dividida em módulos lógicos que tratam do carregamento de dados, do processamento de comandos e da execução de operações.

3.1. Carregamento e Construção da Estrutura de Dados

O estado inicial do sistema de arquivos é construído a partir de um arquivo externo. A função ler_arquivo é a primeira a ser acionada neste processo. Sua lógica consiste em abrir o arquivo especificado, calcular seu tamanho total utilizando as funções fseek e ftell, alocar um buffer de memória de tamanho correspondente e ler todo o conteúdo do arquivo para este buffer de uma só vez com fread. Esta abordagem de leitura em bloco é eficiente para arquivos de tamanho moderado.

O conteúdo lido, uma longa string contendo múltiplos caminhos, é então processado pela função construir_arvore_do_arquivo. Esta, por sua vez, itera sobre cada linha da string e delega a lógica de inserção na árvore para a função processar_caminho, que constrói a hierarquia de nós de forma incremental.

3.2. Ciclo de Vida da Interface e Despacho de Comandos

A operacionalização da interface é orquestrada pela função iniciar_terminal, que estabelece o ciclo de vida da interação com o usuário através de um laço de repetição. A cada iteração, a entrada do usuário é capturada e delegada à função processar_comando. Esta atua como um despachante (dispatcher), segmentando a entrada em comando e argumentos e invocando a rotina de implementação correspondente.

3.3. Catálogo de Comandos e Análise de Implementação

- **ls**: Implementado por comando_ls, efetua a listagem do diretório corrente ao percorrer a lista encadeada de nós filhos a partir de primeiroFilho.
- **cd**: Implementado por comando_cd, altera o diretório de trabalho. Para tal, utiliza a função auxiliar encontrar_diretorio para buscar o destino, que por sua vez emprega strcasecmp_custom para a comparação de nomes. Em caso de falha, mostrar_alternativas é acionada.
- **pwd e tree**: O comando pwd, implementado em comando_pwd, utiliza obter_caminho_completo para gerar o caminho absoluto do diretório atual. O comando tree, em comando_tree, invoca a função recursiva imprimir_arvore para exibir a hierarquia.
- **search**: A função comando_search inicia uma busca global que é executada pela rotina recursiva buscar_recursivo. Esta última utiliza stristr_custom para a busca textual e obter_caminho_completo para exibir os resultados.
- **mkdir e touch**: Os comandos mkdir e touch, implementados por comando_mkdir e comando_touch respectivamente, são responsáveis pela criação de novas entidades. Ambos validam o nome proposto, verificam duplicatas com strcasecmp_custom e, se bem-sucedidos, invocam a função criar_no para alocar e inicializar o novo nó na árvore.
- **rm e rmdir**: A remoção de arquivos é feita por comando_rm, que localiza o nó e o remove da lista encadeada, liberando sua memória. A remoção de diretórios, implementada em comando_rmdir, é mais complexa: após desvincular o nó do diretório da estrutura principal, invoca a função crítica liberar_no_recursivo para desmantelar de forma segura toda a sub-árvore e garantir a integridade da memória.
- history para exibir o historico de comandos executados anteriormente, sendo ele apenas history para os ultimos 5 comandos ou com um sufixo para mostrar a quantidade pre determinada como history x

3.4. Funções Auxiliares Críticas

Certas funções auxiliares são fundamentais para a modularidade e robustez do sistema.

- **Gerenciamento de Nós (criar_no, liberar_no_recursivo)**: A função criar_no encapsula a alocação de memória (malloc) e a inicialização de um novo nó, incluindo a duplicação segura de seu nome com strdup. Em contrapartida, liberar_no_recursivo implementa um algoritmo de travessia em pós-ordem para liberar recursivamente todos os nós de uma sub-árvore, prevenindo vazamentos de memória (memory leaks).

- **Manipulação de Caminhos (obter_caminho_completo)**: Esta função constrói o caminho absoluto de um nó através de um elegante processo recursivo que ascende na árvore pelo ponteiro pai até a raiz, montando a string do caminho na ordem correta durante o retorno das chamadas recursivas.
- **Funções de String Customizadas (strcasecmp_custom, stristr_custom)**: Para garantir a portabilidade do código e evitar dependências de extensões não-padrão da biblioteca C, foram implementadas versões customizadas de funções de comparação de string insensíveis a maiúsculas/minúsculas. Elas operam realizando uma conversão manual de cada caractere para minúsculo antes da comparação, caractere por caractere.

4. Persistência de Dados

A persistência do estado do sistema de arquivos entre sessões é garantida pela função salvar_arvore_no_arquivo. Esta função abre o arquivo de destino em modo de escrita e invoca a rotina auxiliar escrever_caminhos_recursivo. A função recursiva realiza uma travessia em pré-ordem na árvore: ela primeiro processa (escreve) o caminho do nó atual e depois visita recursivamente todos os seus filhos. Este método serializa a estrutura da árvore em memória de volta para o formato de lista de caminhos no arquivo de texto.

5. Considerações de Segurança e Robustez

Para além da funcionalidade principal, a implementação incorpora mecanismos de verificação e validação com o objetivo de aumentar a segurança operacional e a robustez do sistema contra entradas maliciosas ou errôneas.

5.1. Validação de Entrada na Criação de Nós (`mkdir` e `touch`)

As funções `comando_mkdir` e `comando_touch` implementam um rigoroso processo de validação de entrada antes de proceder com a criação de um novo nó. Primeiramente, verificam se o nome fornecido não é nulo ou vazio. Em seguida, uma verificação explícita é realizada para proibir o uso de caracteres com significado especial em sistemas de arquivos, tais como `/`, `\`, `:`, `*`, `?`, `"`, `<`, `>` e `|`. Esta é uma medida de segurança fundamental para prevenir vulnerabilidades de injeção de caminho (path injection). Adicionalmente, há uma proibição explícita da criação de nós com os nomes `.` e `..`, que são reservados para a navegação relativa, salvaguardando assim a integridade estrutural do sistema de arquivos virtual.

5.2. Mecanismos de Proteção na Remoção de Nós (`rm` e `rmdir`)

De forma análoga, as operações de remoção contêm salvaguardas críticas. As funções `comando_rm` e `comando_rmdir` proíbem categoricamente a remoção dos diretórios `.` e `..`. Esta é uma proteção essencial que impede o usuário de desestabilizar o estado da sessão ao tentar remover o diretório atual ou seu ascendente direto. Outro mecanismo de segurança relevante é a verificação de tipo: a função `comando_rm` apenas atua sobre nós do tipo `ARQUIVO`, enquanto `comando_rmdir` atua exclusivamente sobre nós do tipo `PASTA`. Essa distinção semântica, espelhada em sistemas operacionais padrão, previne a execução de operações destrutivas por engano, como a tentativa de remover um diretório e todo o seu conteúdo com o comando destinado a arquivos. Coletivamente, estas verificações mitigam o risco de erros do usuário e protegem a integridade da estrutura de dados em memória.

6. Desafios e Dificuldades na Implementação

O desenvolvimento de um simulador de terminal em linguagem C, como o apresentado, envolve uma série de desafios técnicos e conceituais. A natureza de baixo nível da linguagem e a complexidade do problema exigem um planejamento cuidadoso e uma implementação rigorosa para garantir a funcionalidade, estabilidade e segurança do sistema.

O desafio mais significativo do projeto reside no gerenciamento manual de memória, imposto pela ausência de um coletor de lixo automático em C. Cada alocação realizada com `malloc` ou `strdup` exige uma correspondente liberação com `free`, e a garantia de que todos os caminhos de execução possíveis liberem a memória corretamente é uma tarefa complexa, especialmente em casos de erro ou ao remover nós da árvore. A implementação da desalocação recursiva, através da função `liberar_no_recursivo`, é particularmente crítica; uma falha em sua lógica poderia resultar em vazamentos de memória significativos ao usar o comando `rmdir`. Adicionalmente, a complexidade das interconexões na árvore (pai, filho, irmão) aumenta o risco de criação de ponteiros suspensos (dangling pointers) após a liberação de um nó.

Adicionalmente, a própria representação do sistema de arquivos como uma árvore n-ária utilizando ponteiros, embora poderosa, é intrinsecamente complexa e propensa a erros. A manipulação dos múltiplos ponteiros de cada nó (`pai`, `primeiroFilho`, `proxIrmao`) para inserir ou remover elementos exige rigor para manter a consistência da estrutura. Funções que modificam o estado global do terminal, como `comando_cd`, apresentam um desafio adicional ao exigirem o uso de ponteiros para ponteiro (`no**`), um conceito que demanda um entendimento aprofundado da indireção em C para alterar o ponteiro do diretório atual na função chamadora.

A elegância da recursão, utilizada em funções como `imprimir_arvore`, `buscar_recursivo` e `obter_caminho_completo`, contrasta com a dificuldade de sua implementação e depuração. A correta definição de casos base é crucial para evitar estouros de pilha, e a lógica de construção de resultados no retorno das chamadas, como em `obter_caminho_completo`, pode ser pouco intuitiva. Essa complexidade algorítmica é complementada pela necessidade de garantir a robustez do sistema contra entradas do usuário. Um terminal funcional deve ser resiliente a erros e entradas maliciosas, o que exigiu a implementação de múltiplas camadas de verificação. Dentre elas, destacam-se a proibição de caracteres especiais e nomes reservados (`/`, `..`) na criação de nós através de `comando_mkdir` e `comando_touch`. Da mesma forma, foi necessário implementar salvaguardas explícitas para impedir operações destrutivas, como `rmdir .`, e para garantir a consistência semântica dos comandos, como assegurar que `rm` opere apenas em arquivos e `rmdir` apenas em pastas. Pensar em todos os casos de uso indevido e criar barreiras para eles foi uma tarefa complexa que impactou diretamente a usabilidade e a segurança do programa.

7. Conclusão

A análise do código revela uma arquitetura de software bem definida e modular para um simulador de terminal. A escolha da estrutura de dados "primeiro filho, próximo irmão" mostra-se adequada e eficiente para o problema. A clara separação de responsabilidades entre as funções de interface, de processamento de comandos e as rotinas auxiliares, somada às camadas de validação e segurança, contribui para a legibilidade, manutenibilidade e robustez do código. A implementação de funções customizadas e a gestão cuidadosa da memória demonstram uma preocupação com a portabilidade e a estabilidade do sistema. O projeto, como um todo, constitui um exemplo prático e completo de aplicação de conceitos fundamentais de estrutura de dados e algoritmos em engenharia de software.