

Sistemas Operacionais

Documentação TP1

Aluno 1: Daniel Oliveira Barbosa

Matrícula: 2020006450

Aluno 2: Marco Antônio de Alcântara Machado

Matrícula: 2022036101

Parte 1:

- **Introdução:**

Este projeto tem como objetivo a criação de um shell básico que suporta a execução de comandos simples, redirecionamento de entrada e saída e uso de pipes. A implementação permite que o shell interprete e execute comandos tradicionais do Unix, como ls, cat, sort, uniq, e wc, usando chamadas de sistema para criar processos e manipular fluxos de entrada e saída. A meta é possibilitar a execução de comandos encadeados e com redirecionamento, sem o uso da função system, focando nas chamadas exec, fork, pipe, open e close.

- **Decisões de Projeto:**

Tarefa 1: Implementação de Comandos Simples

- i. **Descrição:** O objetivo desta tarefa é permitir que o shell execute comandos básicos como ls, cat, e echo, sem argumentos adicionais de redirecionamento ou pipes. O código já oferece uma estrutura `execcmd`, que armazena o nome do comando e seus argumentos. Para executar o comando, usamos `execvp`, que busca o comando especificado no PATH e o executa no mesmo processo.
- ii. **Syscall Utilizada:** `execvp`
 - **`execvp`** é uma função da família `exec` que substitui o processo atual pelo comando especificado, permitindo a execução de programas externos. O `execvp` em particular usa o PATH para localizar o comando, tornando-o ideal para interpretar comandos de shell comuns como ls e cat.
- iii. **Tratamento de Erros:** Como o `execvp` substitui o processo atual, ele não retorna caso o comando seja executado com sucesso. No entanto, se ocorrer um erro (por exemplo, o comando não existir), `execvp` retorna um valor negativo. O código inclui uma verificação para capturar essa falha e exibir uma mensagem de erro, informando que o "comando não foi encontrado", ajudando o usuário a entender o motivo da falha.

Tarefa 2: Implementação do Comando `cd`

- **Descrição:** Diferentemente de outros comandos, `cd` (mudar de diretório) precisa ser executado no contexto do próprio shell, pois alterar o diretório de trabalho em um processo filho (por exemplo, usando `fork`) não afetaria o shell principal. O comando `cd` é, portanto, tratado diretamente na função principal antes de qualquer `fork`.
- **Syscall Utilizada:**

- **chdir**: `chdir` altera o diretório de trabalho do processo atual para o especificado. Ao usar `chdir` diretamente no processo pai (o shell), a mudança de diretório permanece no shell principal, facilitando a navegação entre diretórios sem afetar comandos subsequentes.
- **Tratamento de Erros**: Se o diretório especificado em `cd` não existir ou não puder ser acessado, `chdir` retorna um valor negativo. O código inclui uma verificação para capturar esse erro e exibir uma mensagem informando "não foi possível mudar para o diretório especificado". Esse feedback ajuda a identificar diretórios inválidos ou problemas de permissão.

Tarefa 3: Redirecionamento de Entrada/Saída (< e >)

- **Descrição**: O redirecionamento permite que o shell redirecione a entrada ou saída de comandos para arquivos específicos, utilizando > para saída e < para entrada. O parser de linha de comando já identifica esses operadores e gera uma estrutura `redircmd` para cada caso, que inclui o comando a ser executado, o arquivo para redirecionamento e o modo de abertura do arquivo (leitura ou escrita). Na função `runcmd`, o código manipula `dup2` e `open` para associar o fluxo de entrada ou saída padrão ao arquivo especificado.
- **Syscalls Utilizadas**:
 - **open**: Abre o arquivo no modo apropriado (somente leitura para < e escrita para >). O modo de abertura e as permissões (`S_IRWXU`) são definidos conforme o tipo de redirecionamento.
 - **dup2**: Redireciona o descritor de arquivo padrão (`STDIN_FILENO` para < e `STDOUT_FILENO` para >) para o descritor do arquivo aberto. Isso permite que o comando subsequente leia a partir do arquivo especificado ou escreva nele.
 - **close**: Fecha o descritor de arquivo após o redirecionamento, liberando o recurso.
- **Tratamento de Erros**: Se `open` falhar (por exemplo, se o arquivo não puder ser criado ou acessado), o código exibe uma mensagem de erro e encerra o processo com `exit(1)`. O código também verifica o sucesso de `dup2` para garantir que o redirecionamento ocorra sem problemas. Caso `dup2` falhe, o shell exibe uma mensagem e encerra o processo. Esse tratamento de erro protege o shell contra problemas comuns de manipulação de arquivos, como permissões inadequadas.

Tarefa 4: Implementação de Pipes (|)

- **Descrição**: Pipes permitem a passagem de dados da saída de um comando para a entrada de outro. Por exemplo, `ls | sort | uniq` cria um fluxo contínuo de dados entre `ls`, `sort`, e `uniq`. O código do shell utiliza a estrutura `pipecmd` para representar comandos encadeados com pipes e a função `pipe` para criar uma conexão entre dois processos. Cada lado do pipe é manipulado em um processo filho criado por `fork`, com o descritor de saída de um comando redirecionado para o descritor de entrada do próximo.

- **Syscalls Utilizadas:**

- **pipe:** Cria um par de descritores de arquivo (leitura e escrita) que formam o canal de comunicação entre os dois processos.
- **fork:** Cria um processo filho para cada lado do pipe, permitindo que o shell continue em execução enquanto cada comando do pipe é executado em paralelo.
- **dup2:** Redireciona o descritor de saída do primeiro comando para o lado de escrita do pipe e o descritor de entrada do segundo comando para o lado de leitura do pipe, permitindo a comunicação entre os dois comandos.
- **close:** Fecha os descritores de arquivo não mais necessários após o redirecionamento, evitando vazamentos de recursos.

- **Tratamento de Erros:** O código verifica se pipe foi criado corretamente e exibe uma mensagem de erro em caso de falha. Da mesma forma, dup2 e fork também são monitorados para erros, com mensagens de erro exibidas e o processo encerrado se houver falhas. Essas verificações são cruciais para garantir que o encadeamento de processos funcione conforme esperado, especialmente porque pipes e forks podem gerar problemas difíceis de diagnosticar quando ocorrem falhas não tratadas.

- **Testes:**

O código foi submetido a uma série de testes automatizados para garantir a execução correta das funcionalidades implementadas.

- **Testes de Comandos Simples:** Foram testados comandos como ls e cat, verificando se a saída gerada pelo shell corresponde à saída esperada no terminal padrão.
- **Testes de Redirecionamento:** Testes com comandos que envolvem redirecionamento de entrada (<) e saída (>) foram executados. Por exemplo, cat < arquivo e echo "teste" > arquivo garantiram que o shell lida corretamente com a manipulação de arquivos.
- **Testes de Pipes:** Comandos encadeados com pipes, como ls | sort | uniq | wc, foram testados para validar que a saída de um comando alimenta a entrada do próximo, conforme esperado.
- **Execução de Sequências e Scripts:** Foi realizado um teste automatizado com scripts contendo múltiplas execuções e redirecionamentos. Os resultados foram comparados com a execução equivalente em um shell Unix para validar a precisão do shell implementado.

- **Como Executar o Shell:**

- 1) Vá para o diretório shell
- 2) Execute:

```
$ gcc sh.c -o myshell
```

```
$ ./myshell
```

- 3) Isso abrirá o shell no terminal. Execute os comando comandos padrão do shell unix tais como `ls`, `grep`, `cat`. Você também pode utilizar redirecionamentos (> e <) e pipes (|) juntamente com esses comandos padrão.
- 4) Para sair do shell aperte ctrl+D

OBS: Caso a permissão para executar seja negada, execute o comando:
chmod +x sh

- **Como Executar Testes:**

Para executar os testes, basta ir para diretório shell e executar o seguinte comando:

\$./grade.sh

Parte 2:

- **Introdução:**

A segunda parte do projeto visa a criação de uma ferramenta semelhante ao comando *top* do Linux, chamada *meutop*, que exibe informações sobre processos em execução no sistema em uma tabela continuamente atualizada. A ferramenta utiliza dados dos arquivos especiais em */proc*, que fornecem informações sobre os processos. O programa exibe o PID, o usuário que está executando o processo, e o estado do processo em uma tabela formatada, atualizada a cada segundo.

Além disso, o programa permite o envio de sinais para processos em execução. O usuário pode enviar sinais para um processo específico digitando o número do PID seguido do sinal desejado.

- **Decisões de Projeto:**

Função `get_user()`

Descrição: recebe o PID de um processo e retorna o nome do usuário que iniciou o processo, armazenando-o na variável `user`. Ela constrói o caminho para o diretório `/proc/[PID]`, onde cada processo possui uma entrada própria. Usando a função `stat`, a função obtém o ID do usuário (UID) responsável pelo processo e, em seguida, utiliza `getpwuid` para converter o UID em um nome de usuário legível. Caso ocorra algum erro ao acessar essas informações, a função preenche o nome de usuário com "Unknown".

Função `get_process_info()`

Descrição: recebe o PID e extrai o nome e o estado do processo. O caminho para o arquivo `/proc/[PID]/stat` é montado, e o arquivo é lido para extrair dados essenciais sobre o processo. Ela busca o nome do processo dentro dos parênteses e o estado, que está logo após, permitindo que essas informações sejam apresentadas de forma clara na tabela. Em caso de erro, `process_name` é definido como "Unknown", e o estado é indicado como "?".

Função `print_table()`

Descrição: exibe o cabeçalho da tabela com os campos "PID", "User", "PROCNAME" e "Estado". Ela define a formatação da tabela, para que cada coluna tenha um espaço bem delimitado, facilitando a leitura da lista de processos na tela.

Função `add_line()`

Descrição: recebe as informações de um processo (PID, usuário, nome do processo e estado) e as imprime em uma linha formatada que corresponde ao cabeçalho da tabela. Cada chamada a `add_line` exibe uma nova linha na tabela, contendo os dados de um processo específico.

Função `main()`

Descrição: coordena o funcionamento do programa, atualizando a exibição de processos e gerenciando entradas do usuário para envio de sinais. Ela entra em um loop infinito onde a tela é limpa e o cabeçalho da tabela é impresso. A função então abre o diretório `/proc` e percorre os processos, chamando `get_user()` e `get_process_info()` para obter informações e `add_line()` para exibi-las. A cada iteração, `select()` é usado para verificar a entrada do usuário, permitindo que um sinal seja enviado a um processo específico caso o usuário insira um comando no formato PID SINAL. Também implementa tratamento de erros e validações para garantir que apenas dados válidos sejam exibidos e que sinais sejam enviados corretamente.

- **Testes:**

O código *tester* foi disponibilizado para testar a funcionalidade de envio de sinais no *meutop*. Ele configura manipuladores para os sinais `SIGHUP` e `SIGINT`, imprimindo o número do sinal recebido e encerrando o processo ao capturá-lo. Em um loop infinito com pausas de um segundo, o *tester* mantém o processo ativo, permitindo que o *meutop* liste seu PID e envie sinais para ele. Esse teste ajuda a verificar se o *meutop* consegue enviar sinais corretamente, observando se o *tester* responde com a mensagem esperada ao receber cada sinal.

- **Como Executar o *meutop*:**

1) Abra um terminal de comando

2) Compile o programa:

```
$ gcc meutop.c -o meutop
```

3) Execute o programa:

```
$ ./meutop
```

4) Para o envio de sinais: digite o PID do processo e em seguida o sinal numérico que deseja enviar. Por exemplo:

```
> 516315 2
```

- **Como Executar o Teste:**

Dentro do diretório *top*:

1) Abra um terminal de comando (terminal 1), compile e execute *meutop*:

```
$ gcc meutop.c -o meutop
```

```
$ ./meutop
```

2) Em outro terminal (que chamaremos de terminal 2), compile e execute o *tester*:

```
$ gcc tester.c -o tester
```

```
$ ./tester
```

3) Abra o último terminal (que chamaremos de terminal 3) e execute:

```
$ ps aux | grep tester
```

Isso retornará o PID do tester, logo ao lado do nome do usuário. Copie esse PID.

4) Abra o último terminal (que chamaremos de terminal 3) e execute:

```
$ ps aux | grep tester
```

5) Vá para o terminal 1 (que está executando o meutop) e digite (ou cole) o PID e o sinal que você deseja enviar para o *tester*. Exemplo:

```
> 516315 1
```

OBS: Caso a permissão para executar seja negada, execute o comando:

```
chmod +x meutop
```