TP1 - SISTEMAS OPERACIONAIS - Shell + TOP + Sinais

RELATÓRIO

1. Termo de compromisso

Os membros do grupo afirmam que todo o código desenvolvido para este trabalho é de autoria própria. Exceto pelo material listado no item 3 deste relatório, os membros do grupo afirmam não ter copiado material da Internet nem obtiveram código de terceiros.

2. Membros do grupo e alocação de esforço

Alícia Marzola Chaves <aliciamarchaves@gmail.com> 50%
José Gabriel <josegabrielvds2004@gmail.com> 50%

3. Referências bibliográficas

FLAVIOVDF. sistemas-operacionais/exemplos/02-Processos/forkpipe at master · flaviovdf/sistemas-operacionais. Disponível em: <https://github.com/flaviovdf/sistemas-operacionais/tree/master/exemplos/02-Processos/forkpipe>. Acesso em: 6 nov. 2024.

proc(5) - Linux manual page. Disponível em: <https://man7.org/linux/man-pages/man5/proc.5.html>.

THLORENZ. procps/deps/procps/proc/readproc.c at master · thlorenz/procps. Disponível em: <https://github.com/thlorenz/procps/blob/master/deps/procps/proc/readproc.c>. Acesso em: 6 nov. 2024.

kill(2): send signal to process - Linux man page. Disponível em: <https://linux.die.net/man/2/kill>. Acesso em: 6 nov. 2024.

Standard streams. Disponível em: <https://en.wikipedia.org/wiki/Standard_streams>.

COMBEAU, M. Sending and Intercepting a Signal in C - codequoi. Disponível em: <https://www.codequoi.com/en/sending-and-intercepting-a-signal-in-c/>. Acesso em: 6 nov. 2024.

RAZIKA BENGANA. Understanding Signals in the C Language: Harness the Power of Asynchronous Event Handling. Disponível em: <https://medium.com/@razika28/signals-ad83f38f80b6>. Acesso em: 6 nov. 2024.

cunha-dcc605 / shell-assignment · GitLab. Disponível em: <https://gitlab.dcc.ufmg.br/cunha-dcc605/shell-assignment>. Acesso em: 6 nov. 2024.

4. Estruturas de dados

Parte 1 - Desenvolvendo um Shell Básico

-

Sh.c = Arquivo ***.c** que descreve um terminal com operações de chamadas de sistema, compilado com compilador ***gcc*** e que com o comando ***gcc sh.c -o myshell*** gera um arquivo executável ***myshell,*** que será o terminal shell.

- (buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' ') // task 1

Lê o comando 'cd' e acessa o diretório filho

- execvp: // task2

Substitui o processo atual pelo ecmd->argv[0]. Ex: ls ou cd
Se execvp executa com sucesso, o processo atual é substituído pelo novo programa, e a execução da função chamadora termina. No entanto, se execvp falha, ela retorna -1, e errno é ajustado para refletir o tipo de erro.

- int fd = open(rcmd->file, rcmd->mode, S_IRWXU); // task 3

Open abre um arquivo especificado por rcmd->file.

rcmd->mode é o modo de abertura (por exemplo, O_RDONLY para leitura, O_WRONLY para escrita).

S_IRWXU define permissões do arquivo (leitura, escrita, e execução para o usuário). Isso é relevante se o arquivo for criado no processo. Se open falhar, retorna -1.

- dup2(fd, rcmd->fd);

dup2 duplica o descritor fd para rcmd->fd.

Após essa linha, rcmd->fd (como 0 para stdin ou 1 para stdout) apontará para o arquivo aberto

- if (pipe(p) == -1): // task 4

pipe(p) cria um pipe (canal de comunicação) entre processos.

p é um array de dois inteiros onde p[0] é o extremo de leitura do pipe e p[1] é o extremo de escrita.

- if (fork1() == 0):

fork1 cria um processo filho. Se o retorno for 0, significa que está no processo filho.

No primeiro fork1, o objetivo é configurar o processo para enviar dados pelo pipe.

close(p[0]); fecha o extremo de leitura do pipe, pois esse processo só vai escrever.

dup2(p[1], STDOUT_FILENO); redireciona a saída padrão (STDOUT_FILENO) para o extremo de escrita do pipe (p[1]), de modo que qualquer saída do processo seja enviada ao pipe.

close(p[1]); fecha o descritor original de escrita, já que dup2 já redirecionou a saída padrão.

runcmd(pcmd->left); executa o comando à esquerda do pipe (pcmd->left), enviando sua saída pelo pipe.

- Segundo if (fork1() == 0):

No segundo fork1, cria-se um segundo processo filho, configurado para receber dados do pipe.

close(p[1]); fecha o extremo de escrita do pipe, pois esse processo só vai ler.

dup2(p[0], STDIN_FILENO); redireciona a entrada padrão (STDIN_FILENO) para o extremo de leitura do pipe (p[0]), de modo que qualquer entrada esperada pelo processo venha do pipe.

close(p[0]); fecha o descritor original de leitura, pois dup2 já redirecionou a entrada.

runcmd(pcmd->right); executa o comando à direita do pipe (pcmd->right), consumindo dados enviados pelo primeiro processo.

- close(p[0]); close(p[1]);:

No processo pai (que criou os filhos), ambos os extremos do pipe são fechados, pois o pai não precisa mais deles.

- wait(0); wait(0);:

O processo pai aguarda a finalização dos dois processos filhos, garantindo que eles completem a execução antes de o pai continuar.

grade.sh
Arquivo de avaliação com testes da implementação do shell. Executa um total de 9 testes, mostrando quais foram executados e o motivo de possíveis erros.

Parte 2 - Utilizando /proc/ para fazer um TOP

Para essa parte do trabalho, foi projetado um programa para monitorar processos ativos em um sistema Linux, exibindo informações relevantes como o ID do processo (PID), o usuário que o possui, o nome do processo e o estado atual. Além disso, o programa permite ao usuário enviar sinais para processos específicos. Suas principais estruturas de dados são:

- ProcessInfo:

A estrutura `ProcessInfo` armazena informações de processos em execução no sistema.

Ela possui quatro membros:
- pid: um número inteiro que representa o identificador do processo (PID).
- user: uma string que armazena o nome do usuário proprietário do processo.
- procname: uma string para o nome do processo.
- state: um caractere que indica o estado atual do processo (ex.: 'R' para execução, 'S' para dormindo, etc.).

Essa estrutura é usada para organizar as informações relevantes de cada processo de forma centralizada, facilitando o acesso e manipulação desses dados ao longo do programa.

- displayProcesses:

A função `displayProcesses` recebe um array de `ProcessInfo` e o número de processos a serem exibidos, e imprime uma tabela formatada com informações sobre cada processo. Ela usa o loop `for` para iterar sobre o array de processos e, para cada processo, imprime seu PID, nome do usuário, nome do processo e estado.

- getProcesses:

`getProcesses` é responsável por ler o diretório `/proc` para obter informações sobre os processos em execução. Ela usa a estrutura `DIR` e `struct dirent` para navegar pelo sistema de arquivos `/proc`, identificando entradas de processos válidas (diretórios que contêm apenas números). Para cada processo, a função armazena o PID, obtém o nome do usuário dono do processo, usando a função `getpwuid`, e lê o arquivo `stat` para obter o nome do processo e seu estado.

Os dados são armazenados no array `procs`, usando um contador `count` que garante que o número máximo de processos (`MAX_PROCS`) não seja ultrapassado. Essa função é essencial para capturar o status atualizado dos processos a cada ciclo do programa.

- signalInputThread

`signalInputThread` é executada em uma thread separada e é responsável por receber entrada do usuário (PID e número do sinal) para enviar sinais aos processos. Ela usa a função `scanf` para capturar o PID e o sinal a ser enviado e, em seguida, usa a função `kill` para enviar o sinal ao processo especificado.

Essa funcionalidade permite que o programa interaja dinamicamente com os processos, recebendo comandos para manipular os processos diretamente pela entrada padrão.

- input_thread:

Em `main`, a thread `input_thread` é criada para executar a função `signalInputThread` de forma assíncrona, permitindo que o programa principal continue sua execução (atualizando a lista de processos) enquanto a thread escuta por comandos de sinal do usuário.

Essa estrutura permite o funcionamento simultâneo da atualização da lista de processos e do recebimento de comandos do usuário, tornando o programa mais interativo e responsivo.

- Variáveis Importantes:

MAX_PROCS: Define o número máximo de processos que o programa pode armazenar e exibir simultaneamente. Facilita a gestão da memória, limitando o tamanho do array `procs`.

PROC_PATH e STAT_FILE: Constantes que armazenam o caminho base para acessar informações dos processos (`/proc`) e o arquivo de estatísticas (`stat`). Essas definições evitam que strings repetitivas sejam usadas no código, facilitando sua manutenção.