Relatório do Trabalho Prático 01

Sistemas Operacionais

Thiago Martin Poppe, Giovanna Louzi Bellonia

25/08/2020

1 Uma breve descrição

1.1 Problema proposto

Este trabalho prático foi dividido em duas partes. A primeira consiste em completar um código para a execução de comandos tais como: **ls**, **cat**, operadores de **redirecionamento** e **pipe**, em um shell simplificado. Já a segunda, consiste em implementar o comando **top**, que seria capaz de mostrar os processos que estão sendo executados na máquina e enviar sinais para os mesmos.

1.2 Observações

- O desenvolvimento de ambas as partes foi feito utilizando o WSL. Posteriormente, os mesmos códigos foram testados em uma máquina virtual Linux com distribuição Ubuntu 20.04 LTS.
- Foi utilizado o manual do Linux para obter mais conhecimento sobre as funções necessárias para este trabalho prático.

2 Parte 1 - Shell básico

2.1 Executando Comandos Simples

Estudando o manual das funções da família **exec**, utilizamos a função **execvp** que aceita como primeiro argumento uma string com o comando a ser executado e como segundo um array de parâmetros terminado em NULL. Por exemplo, a chamada **execvp("ls", {"ls", "-la", NULL})** seria equivalente ao comando **ls -la**.

Com a estrutura **ecmd** disponibilizada no código, conseguimos obter o nome do comando digitado e seus parâmetros utilizando o campo **argv**. Com isso, criamos um array auxiliar que é igual ao **argv**, porém com um espaço a mais preenchido por NULL para que a função execute corretamente.

2.2 Redirecionamento de Entrada e Saída

Utilizando o manual das funções **open**, **close** e **dup2**, fomos capaz de redirecionar a entrada e saída do programa para um arquivo específico através de *file descriptors*, onde, por padrão, os valores 0, 1 e 2 referem-se, respectivamente, ao *stdin, stdout* e *stderr*.

Através da estrutura **rcmd** disponibilizada no código, conseguimos obter o nome do arquivo, o modo de abertura do mesmo (leitura ou escrita) e em qual *file descriptor* iriamos realizar a substituição. Com isso, abrimos o arquivo utilizando **open**, passando o nome do arquivo, modo de abertura e 0777 como terceiro parâmetro apenas para criamos arquivos com todas as permissões. Uma vez que a abertura do arquivo foi bem-sucedida, substituímos o *file descriptor* correto utilizando **dup2** e executamos o comando desejado.

2.3 Sequenciamento de Comandos

Utilizando o manual das funções **pipe** e **fork**, fomos capaz de implementar o sequenciamento de comandos com o uso de pipes através da estrutura **pcmd** disponibilizada no código, que retorna o comando da esquerda e da direita com relação ao pipe.

Primeiramente, utilizamos um array de duas posições para representar o sequenciamento, onde a posição 0 será a entrada do pipe e 1 a saída. Após sua inicialização com a função **pipe**, utilizamos a função **fork** para criar um processo filho. Tal processo irá fechar a posição 0 do pipe para evitar possíveis problemas, redirecionando a saída padrão para a posição 1 do pipe e executando o comando da esquerda. Uma vez terminado, voltamos para o processo pai que irá, por sua vez, fechar a posição 1 do pipe, redirecionando a entrada padrão para a posição 0 do pipe e executando o comando da direita. Note que, no processo pai, chamamos a função **wait** como uma forma de previnir que o processo filho não se torne um processo *zombie* ao terminar sua execução.

2.4 Histórico de Comandos

A implementação do histórico de comandos foi totalmente feita do zero. Como estrutura de dados, optamos por utilizar um buffer circular de tamanho igual a 50, como definido na documentação do trabalho prático. Visto que o comando **history** deverá limpar os comandos mais antigos quando a capacidade máxima for atingida, a escolha dessa estrutura de dados foi muito bem encaixada, garantindo uma complexidade temporal de escrita e leitura da ordem de $O(n)$ e uma complexidade espacial também da ordem de $O(n)$.

A inserção de comandos foi feita antes da chamada da função **runcmd**, verificando sempre se o comando inserido não foi algo como um \n, apenas para não inserirmos linhas em branco no nosso histórico; já a verificação e execução do comando foi feito dentro da função para manter o código consistente.

3 Parte 2 - Comando TOP

3.1 Implementação

Utilizando o manual do Linux para aprender mais sobre a pasta **/proc**, fomos capazes de implementar um comando **top** simples onde exibimos o pid do processo, usuário dono, comando que está sendo executado e estado do mesmo. Limitamos a exibição em até 20 processos, onde essas informações serão atualizadas de 1 em 1 segundo até que o usuário deseje sair pressionando Control D. Como adicional, exibimos também a data e horário atual, apenas para ficar mais semelhante com o comando **top** original.

A leitura da pasta será feita chamando a função **opendir** da biblioteca **dirent.h** cujo retorno iremos utilizar como parâmetro para a função **readdir**, tendo assim acesso à todas as subpastas de **/proc**. Com isso, filtramos apenas as pastas que começam com um número representando o pid do processo. Em outras palavras, pastas que representam links simbólicos para os processos que estão sendo executados.

Para cada pasta resultante dessa filtragem, iremos acessar o arquivo **/proc/[pid]/stat**, que contém todos os dados necessários sobre o processo, onde os três primeiros correspondem ao pid, comando e estado, respectivamente.

Finalizando, através das bibliotecas **pwd.h** e **sys/stat.h** conseguimos descobrir o nome do usuário dono do processo. Para tal, iremos utilizar a função **stat** juntamente da função **getpwuid** que, de um modo geral, irá nos fornecer o nome do usuário apenas conhecendo o pid do processo.

3.2 Envio de sinais

A implementação de envio de sinais foi feita utilizando threads através da biblioteca **pthread.h**. Durante a primeira versão, tanto a interface quanto a entrada do usuário apenas atualizava a cada um segundo, tornando assim a escrita do sinal um pouco ruim de se fazer devido ao delay experimentado pelo usuário. Para resolver tal problema, decidimos deixar a interface congelada enquanto o usuário escreve o sinal a ser enviado. Para isso, o mesmo deve pressionar a tecla enter uma única vez e digitar o pid e sinal que deseja enviar. Com isso, chamamos a função **kill**, através da biblioteca **signal.h**, que por sua vez irá encerrar um processo caso bem-sucedida. Caso contrário, iremos exibir um erro para o usuário, retormando a exibição da interface normalmente.

Para a sincronização do input e output, utilizamos variáveis globais (flags) que são setadas nas horas certas, permitindo assim que apenas uma das threads estejam em execução. Temos ciência da existência de mutexes e conditional variables como primitivas de sincronização em pthreads, mas não conseguimos usar de forma eficiente e prática em nosso código.