Cesar Nascimento & Maximiliam Araujo

| DCC605 | Flavio Figueiredo

 May 6, 2017

TP 1

Shell + ps(top) + sinais + modulos

# Especificação do Projeto

O trabalho proposto foi adaptado do material do Professor Italo Cunha. Conceitos como Pipe, Estrutura de Processos do Kernel e Sinais, dentro da disciplina de Sistemas Operacionais foram abordados.

O trabalho foi dividido em 4 partes. A seguir um descritivo de cada parte:

1. Desenvolvendo um Shell Básico: Baixar o esqueleto do shell e implementar o código para execução dos comandos como Redirecionamento de entrada/saída; Sequenciamento de Comandos
2. Lendo o /proc/ para fazer um OS-Tree: Imprimir a árvore de processos usando tab para separa-los.
3. Uma TOP Simples (topzera): Modificar o comando PS para imprimir os processos em sequência (remover os tabs). Além disto, alterar o mesmo para identificar o PID do programa, o usuário que está executando o mesmo e o estado do processo. Com isto, imprimir os programas em execução em uma tabela.
4. Sinais: Permitir que o comando TOP envie sinais, ou seja, criar uma função no TOP que envia sinais para um PID. Tal função pode ser apenas um “PID SINAL”.

A seguir, para cada parte será apresentado uma breve descrição de como foi feita sua implementação.

# Parte 1: Desenvolvendo um Shell Básico

Case ‘ ‘:

execvp(ecmd->argv[0], ecmd->argv);

Os comandos da família exec substituem a imagem do processo corrente com a imagem do novo processo. O primeiro argumento da função exec é o nome do arquivo que será executado. A utilização do execvp ocorreu por uma questão de facilidade no entendimento de seu funcionamento. Apenas passamos o arquivo a ser executado por parâmetro e a função se encarrega de criar o processo.

case '>':

case '<':

int file = open(rcmd->file, rcmd->mode, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH);

dup2(file, rcmd->fd);

close(file);

Esta primitive (dup2) cria uma cópia de um descritor de arquivo existente (oldfd) e fornece um novo descritor (newfd) tendo exatamente as mesmas características que aquele passado como argumento na chamada. Dup2 determina que newfd será a cópia de oldfd, fechando antes newfd se ele já estiver aberto.

case '|':

int pipefd[2];

if (pipe(pipefd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

int pid = fork1();

if(pid == -1){

perror("fork");

exit(EXIT\_FAILURE);

}

if(pid == 0){

close(pipefd[0]);

close(1);

dup(pipefd[1]);

close(pipefd[1]);

runcmd(pcmd->left);

}

else if(pid != 0){

close(pipefd[1]);

close(0);

dup(pipefd[0]);

close(pipefd[0]);

wait(NULL);

runcmd(pcmd->right);

}

O comando acima descrito cria um pipe, em seguida, fork1() cria um processo filho. Após o fork1(), cada processo fecha os descritores que não são necessários para o pipe. O pai então a utiliza o dup(pipefd[1]) para se comunicar com o filho. Já o filho utiliza o dup(pipefd[0]) para se comunicar com pai.

# Parte 2: Lendo o /proc/ para fazer um OS-Tree

# Parte 3: Uma TOP Simples (topzera)

# Parte 4: Sinais

# Decisões de implementação

Preferimos não fazer a parte 5, que no caso era ponto extra. Além disso, na parte 4, não implementamos a parte de atualização a cada 1 segundo por não encontrar uma forma simples de realiza-la.

# Testes

Para testar a

# Conclusão

O trabalho foi bastante

# Referencias bibliográficas

<https://linux.die.net/man/3/execvp>

<http://www.dca.ufrn.br/~adelardo/cursos/DCA409/node22.html>

https://linux.die.net/man/2/pipe