Sistemas Operacionais
Implementação de um protótipo de shell Unix-like

Gabriel de Mello Cambuy Ferreira - RA: 142641
Gabriela Carregari Verza - RA: 148756
Melissa Frigi Mendes - RA: 133693
Raissa Barbosa dos Santos - RA: 148551

Universidade Federal de São Paulo
São José dos Campos

Abril de 2024

1. Introdução

Neste relatório, será apresentada a implementação de um protótipo de shell Unix-like utilizando linguagem C. Este shell é capaz de executar: comandos unitários com múltiplos parâmetros; comandos encadeados com o operador pipe (|), para combinar entrada e saída entre n comandos; comandos condicionados com operadores OR (||) e AND (&&); e comandos em background (segundo plano) identificados com o operador '&', liberando o shell para receber novos comandos do usuário.

2. Funções do código

'remover_aspas': Esta função é responsável por remover as aspas duplas ou simples de uma string. Ela percorre a string enviada por parâmetro, copiando todos os caracteres e sobrescrevendo o conteúdo original dessa string, até encontrar aspas simples ou duplas, que não serão copiadas. Ela é importante para argumentos que possuem aspas, como nos comandos grep e echo, para que sejam interpretados corretamente.

'divide_comandos_pipe': Recebe como parâmetros uma string e um array de strings. A string de entrada é percorrida, concatenando os caracteres em uma posição do array, e quando encontra o operador '|', passa para a próxima posição do array. Se identificar dois operadores '|' em seguida, possui uma tratativa para não realizar a divisão, a fim de não descaracterizar comandos com operador lógico OR (||). A função retorna a quantidade de tokens encontrados.

'divide_comandos': Os parâmetros enviados para essa função são uma string de entrada e um array de strings para saída. A string de entrada é dividida com base no delimitador " ", por meio da função 'strtok', e cada parte é armazenada em uma posição do array de strings. Se na primeira posição do token for encontrada uma aspa, continua concatenando até encontrar as aspas finais, para que argumentos entre aspas que possuem espaços não sejam divididos e fiquem como um único token. Essa função é importante para dividir um comando em argumentos, e também retorna quantos foram encontrados.

'executa_comando': Esta função executa um comando específico a partir da string enviada como parâmetro. Ela também recebe um indicador que determina se esse comando deve ser executado em segundo plano. A string com o comando é dividida em argumentos, utilizando a função 'divide_comandos', e para cada argumento, a função 'remover_aspas' é chamada. A função 'fork' é utilizada para criar um novo processo. A seguir, com o pid, identifica se é o processo filho, no qual cria uma nova sessão pela função 'setsid' caso o comando deva ser executado em segundo plano, e então usa 'execvp' para executá-lo; ou se é o processo pai, que aguarda o término do filho com 'waitpid'.

‘trata_operadores’: Esta função identifica operadores condicionais OR (||) e AND (&&), e o operador ‘&’, que indica se o comando deve ser executado em segundo plano. Ela procura pelos operadores '&&' e '||' na string enviada por parâmetro. Se ambos os operadores forem encontrados, ela verifica qual aparece primeiro. Se ‘&&’ aparecer antes de ‘||’, o comando é dividido em dois com base na posição do operador ‘&&’ e chama ‘executa-comando’ para o comando antes do operador. Se esse comando tiver status SUCESSO, realiza chamada recursiva para executar o comando após o operador, se não, executa o comando após o operador ‘||’. Se ‘||’ aparecer antes de ‘&&’, o comando é dividido em dois com base na posição do operador ‘||’ e ‘executa_comando’ é chamada para executar o comando antes do operador. Se essa execução falhar, realiza chamada recursiva para o comando após o operador. Se apenas um operador for encontrado, a função divide a string em dois comandos distintos, chama ‘executa_comando’ para o comandos antes do operador, e faz ou não chamada recursiva para tratar o comando após o operador de acordo com a lógica estabelecida pelo operador. Se nenhum operador for encontrado, a função executa o comando diretamente por meio de ‘executa_comando’. Além disso, se encontrar o operador ‘&’, usa o indicador enviado por parâmetro a ‘executa_comando’ para avisar que o comando deve ser executado em background. Por fim, essa função retorna um status de execução, para tratar a recursividade.

‘executa_comandos_pipe’: Esta função é responsável por tratar a execução de uma sequência de comandos encadeados por pipe. De forma semelhante à ‘executa_comandos’, trata processo pai e filho. Para cada comando, um pipe é criado por meio da função ‘pipe’. O processo filho manipula os arquivos descritores do pipe, repassa a saída de um comando para a entrada de outro utilizando a função ‘dup2’, e chama a função ‘trata_operadores’.

‘main’: É o ponto de entrada do programa, pois recebe comandos do usuário através da entrada padrão e os processa. Ela inicia chamando ‘divide_comandos_pipe’ para verificar se os comandos são encadeados com pipe, e depois chama ‘executa_comandos_pipe’, de forma que o fluxo segue para todas as funções mencionadas anteriormente.

3. Contribuições dos Integrantes

Para este trabalho, o grupo decidiu por realizar o desenvolvimento sempre em conjunto, com todos os membros presentes, em horários vagos na faculdade ou em calls. Mas, para que fosse justo, cada membro ficou responsável por fazer o código suportar um tipo de comando dentre os quatro requisitos do protótipo, recebendo auxílio e opinião dos demais quando necessário. Raissa ficou responsável pelos comandos unitários, Melissa pelos encadeados com pipe, Gabriela pelos condicionados com operadores lógicos e Gabriel pelos executados em background.

4. Descrição do caminho de execução dos comandos

4.1 Comandos unitários com múltiplos parâmetros

Exemplo 1: ls -la

Para iniciar a realização desse comando, ele é recebido pela função ‘fgets’ e vai servir como base para os demais.

A função principal que determina a tratativa dos comandos unitários é a função ‘executa_comando’, que recebe como parâmetro o comando dado no terminal e uma variável indicativa de trabalho em background. A princípio, é iniciado o vetor ‘argumentos[]’ que vai armazenar todos os comandos que vieram como entrada, depois uma variável que armazena a quantidade de comandos que está contida no vetor é criada e iniciada com 0. A partir dessa variável, a função ‘divide_comandos’ é chamada para realizar esse cálculo.

No cálculo, “ls -la” é passado como parâmetro para ‘divide_comandos’, juntamente com um “ ” (espaço) que é o nosso separador de tokens e o vetor ‘argumentos[]’ para armazenar os tokens. Aqui, o primeiro token, que é o “ls”, é definido, então a execução entra no *while* e vai direto para o *else* por conta de que a variável ‘entre_aspas’ está zerada’. No else, “ls” é inserido no vetor na posição 1 e é feita uma verificação quanto a existência de aspas como primeiro caractere do token ‘ls’, mas como não é o caso, a iteração sai do else e a função strtok é chamada novamente para retomar a tokenização a partir de onde tinha parado, recebendo “-la”. Novamente esse fluxo é percorrido, porém, agora, com o novo token. Por fim, a última posição do vetor é preenchida com NULL para indicar que o vetor de tokens já está completo para o exemplo em questão e é retornada a quantidade de tokens, 2.

Agora, retornando à função principal, tem uma iteração em um laço *for* que remove as aspas do comando, caso haja, mas que não é o caso, então esse for não vai ocasionar em nenhuma diferença. Depois disso acontece um *fork()* que cria um processo filho e logo após, vem o *if* tradicional que faz com que o novo processo tenha andamento, dividindo em pai e filho. O processo pai aguarda o filho e ele, por sua vez, ganha sua identidade com o *execvp()* e finaliza com sucesso sua execução, listando todos os arquivos e diretórios que estão dentro do diretório onde o comando está sendo executado, juntamente com algumas informações adicionais como relativo a segurança, data de edição e nome.

Exemplo 2: grep “UNIFESP” README.md

Após o comando ser digitado no teclado, a partir da função ‘executa_comando’ a quantidade de argumentos é contabilizada na ‘divide_comandos’. La, o primeiro token identificado é o “grep”, que entra no while, passa pelo else onde ele é inserido no vetor de tokens e depois retoma a tokenização. O próximo token é o “”UNIFESP””, que vai fazer um caminho diferente. Primeiro ele passa pelo while e depois entra no else, como no

anterior, mas além de ter o token inserido no vetor, a variável ‘entre_aspas’ recebe 1 e logo após recebe 0 novamente, pois esse comando inicia e termina com aspas. Novamente a tokenização é continuada e o último token, “README.md”, é identificado e passa pelo mesmo caminho que “grep” percorreu.

Como retorno, tem-se o valor 3, e a função ‘executa_comando’ pode ter continuação. Após isso, cada um dos argumentos do comando passa pela função ‘remover_aspas’ e, então, o *fork()* é dado e o novo processo filho é criado e segue seu caminho até concluir sua execução, como explicado para o exemplo anterior, mostrando a linha que contém a palavra “UNIFESP” dentro do arquivo “README.md”.

4.2 Comandos encadeados com pipe

Exemplo 1: ls -la | grep “teste”:

O código inicializa lendo os comandos com ‘fgets’, e é direcionado para a função ‘divide_comandos_pipe’, que é utilizada para separar dois comandos ao se utilizar o pipe ‘|’, inicializando uma variável para contabilizar a quantidade de comandos e fazendo uso de alocações dinâmicas para cada um dos comandos. Após entrar no loop e passar por algumas condicionais que verificam se há uso de pipes ou se são operadores lógicos, ou então se são caracteres do comando a serem concatenados à lista de string do comando, a lista é finalizada e a função retorna a quantidade de comandos somado a 1, pois o contador de começa em 0. Portanto, os comandos serão divididos em ‘ls -la’ e ‘grep “teste”’.

Em seguida, a próxima função chamada pela main é ‘executa_comandos_pipe’, que inicializa com um vetor ‘fd[]’ para armazenar arquivos descritores do pipe (stdin e stdout) e uma variável ‘fd_in’ que armazena o descritor de entrada do pipe, ou seja, representa a extremidade de leitura do pipe que vem do processo anterior, e em seguida inicializando um loop com a quantidade de comandos. Dentro do laço, é feita uma chamada de sistema ‘pipe(fd)” para criar o pipe para cada comando, sendo seguido por uma chamada fork() para criar um processo novo e uma condicional para o processo filho, onde o comando ‘dup2(fd_in,0)’ é utilizado para redirecionar a entrada do filho como sendo a entrada do descritor de arquivo do pipe, portanto a entrada do programa filho será obtida a partir do descritor de arquivo especificado em fd_in, fazendo a comunicação através dos pipes. Após a condicional, é chamada a função ‘trata_operadores’, porém como não há nenhum operador lógico, a execução pula para o último ‘else’ e chama a função ‘executa_comando’.

Na função ‘executa_comando’, primeiramente será feita qualquer remoção de aspas com a chamada da função “remove_aspas”, em seguida, os argumentos serão separados e tratados com base no espaço entre os caracteres (tokens), ou seja, ‘ls’, ‘-la’, ‘grep’ e ‘texto’, e para cada um desses argumentos separados serão removidas quaisquer aspas, como no caso de “teste”. O comando fork é chamado, e o processo filho irá executar o comando com a verificação de execução em background, utilizando o comando ‘execvp’ para executar os comandos enquanto o processo pai aguarda o processo filho terminar de executar com ‘waitpid’, retornando com sucesso a execução, imprimindo na tela as

informações referentes ao comando ‘ls -la’ sendo este passado como entrada para o comando ‘ grep”teste” ’, logo todas as linhas do comando ‘ls -la’ que contém o nome “teste”.

Exemplo 2: cat /proc/cpuinfo | grep "model name" | wc -l :

Neste exemplo, novamente é inicializado na main com o comando ‘fgets’ e redirecionado para a função ‘divide_comandos_pipe’ que novamente faz a divisão dos comando a partir de pipes “|” seguindo as tratativas já citadas dentro das condicionais, separando os comandos em ‘cat /proc/cpuinfo’, ‘grep “model name” e ‘wc -l’. Após a função ter sido completada, é chamado a próxima função ‘executa_comandos_pipe’ pela main que irá fazer a execução dos três comandos encadeados pelo pipe através de processos pai e filho e da criação de pipes pelo comando ‘pipe(fd)’, seguido pelo comando ‘dup2’ que irá fazer o redirecionamento de descritores de entrada e saída do processo filho para o pipe permitindo que o processo filho consiga ler os dados de saída do processo pai e o mesmo segue para os próximos processos que serão criados.

Dentro da condicional do processo filho na função ‘executa_comandos_pipe’, é chamada a função ‘trata operadores’, e como não há nenhum no comando com operadores o último ‘else’ chama a função ‘executa_comando’ que, por sua vez, vai realizar a execução dos comando separados por espaço “ “ após a remoção de aspas de qualquer string, portanto os comando serão tratados como ‘cat’, ‘proc/cpuinfo’, ‘grep’, ‘model_name’, “wc” e “-l”. Com os comandos encadeaos pelo pipe “|”, o que irá ser retornado é o número de linhas do arquivo ‘proc/cpuinfo’ que contém as palavras ‘model_name’.

4.3 Comandos condicionados com operadores lógicos AND e OR

Exemplo 1: cat 12345 || echo "arquivo inexistente"

Na função ‘main’, o comando é lido do usuário usando ‘fgets’. A entrada é dividida em comandos separados pelo operador “|” com ‘divide_comandos_pipe’. A função ‘executa_comandos_pipe’ é chamada, enviando todo o comando da entrada como parâmetro, pois não foi encontrado o operador para realizar a divisão. Em ‘executa_comandos_pipe’, no processo filho, ‘trata_operadores’ é chamada, recebendo como parâmetro toda a entrada do usuário.

Na função ‘trata_operadores’, apenas o operador ‘||’ será identificado, e separará a entrada em dois comandos “cat 12345” e ‘echo “arquivo inexistente”’. A seguir, ‘executa_comando’ é chamado para executar o comando “cat 12345”.

Na função ‘executa_comando’, o comando “cat 12345” tenta exibir o conteúdo do arquivo 12345. De volta em ‘trata_operadores’, se o status da execução anterior for

diferente de SUCESSO, ‘executa_comando’ é chamada para executar o comando ‘echo “arquivo inexistente”’. Na função ‘executa_comando’, o comando ‘echo “arquivo inexistente”’ é executado, exibindo a mensagem "arquivo inexistente". Mas, se “cat 12345” for executado com sucesso, o conteúdo do arquivo é exibido, e o comando ‘echo “arquivo inexistente”’ não é executado.

Exemplo 2: ping -c1 www.unifesp.br.br && echo "SERVIDOR DISPONIVEL" || echo "SERVIDOR INDISPONIVEL"

Na função ‘main’, o comando é lido do usuário usando ‘fgets’. A entrada é dividida em comandos separados pelo operador “|” com ‘divide_comandos_pipe’. A função ‘executa_comandos_pipe’ é chamada, enviando todo o comando da entrada como parâmetro, pois não foi encontrado o operador para realizar a divisão. Em ‘executa_comandos_pipe’, no processo filho, ‘trata_operadores’ é chamada, recebendo como parâmetro toda a entrada do usuário.

Na função ‘trata_operadores’, ambos os operadores serão identificados. Como o operador ‘&&’ aparece primeiro, a função separará a entrada com base na posição dele, obtendo dois comandos: ‘ping -c1 www.unifesp.br.br’ e ‘echo "SERVIDOR DISPONIVEL" || echo "SERVIDOR INDISPONIVEL"’. O comando ‘ping -c1 www.unifesp.br.br’ é executado via ‘executa_comando’.

De volta a ‘trata_operadores’, se o comando ‘ping’ for bem-sucedido, será realizada chamada recursiva para o comando ‘echo "SERVIDOR DISPONIVEL" || echo "SERVIDOR INDISPONIVEL"’. A função encontrará apenas o operador ‘||’ e dividirá o comando em dois com base na posição dele. ‘echo "SERVIDOR DISPONIVEL"’ será executado via ‘executa_comando’, exibindo a mensagem "SERVIDOR DISPONIVEL" e portanto finaliza com sucesso, de forma que ‘echo "SERVIDOR INDISPONIVEL"’ não será executado. Mas, se o comando ‘ping’ falhar, o comando ‘echo "SERVIDOR DISPONIVEL" || echo "SERVIDOR INDISPONIVEL"’ será separado com base na posição do operador ‘||’, de forma que apenas ‘echo "SERVIDOR INDISPONIVEL"’ é executado via ‘executa_comando’, e a mensagem “SERVIDOR INDISPONIVEL” é exibida.

4.4 Comandos em background

Exemplo 1: ping -c5 www.unifesp.br & e ls -la

A execução de comandos em background é quando o shell lê um comando do usuário, começa a executar e já retorna imediatamente para o prompt de comando, possibilitando que um segundo comando seja lido e executado enquanto o primeiro é executado. Para a execução em background foram utilizados os comandos: ping -c5 www.unifesp.br & e ls -la.

Na função ‘trata_operadores’ é identificado se o último caractere do comando inserido é &, o que identifica se é uma execução em background. Se o último caractere é & então a variável background recebe o valor 1.

Então o código vai para a função ‘executa_comando’, como a variável background terá o valor 1 então será chamada a função setsid(). A função setsid() é responsável por criar uma nova sessão de processo e deixar o terminal livre para receber um novo comando. Assim que o processo for completado, os resultados serão exibidos no terminal.