UNIFESP – Universidade Federal de São Paulo

São José dos Campos – SP

Parque Tecnológico

Alunos:

Abner Yahalom Daflon Cicarino Canellas 150738

Henrique Dedini Carvalho Silveira 93930

Implementação Shell (Linux)

São José dos Campos

2020

Implementação Shell (Linux)

Relatório no modelo cientifico sobre a implementação de um Shell para o SO Linux utilizando linguagem C, como medida avaliativa para a UC Sistemas Operacionais, ministrada pelo Prof.º Bruno Kimura, no primeiro semestre de 2020, na Universidade Federal de São Paulo, campus São José dos Campus, Parque Tecnológico.

Sumário

1. Introdução

O trabalho realizado consiste em desenvolver um Shell para o sistema computacional Linux. Esse Shell deve executar comandos básicos e compostos, recebendo uma linha de entrada do usuário que será a cadeia de comandos a ser executada.

2. Resumo

O projeto, desenvolvido em dupla para a disciplina, visa criar um interpretador de comandos, conhecido como Shell, recebidos como entrada pelo usuário. Desenvolvido em linguagem C, o programa usa de chamadas de sistema para separar a entrada de comando e transforma-la em uma matriz de comandos separados, que possam ser executados individualmente através das bibliotecas da linguagem e de binários executáveis do próprio sistema operacional Linux.

O objetivo principal do trabalho era desenvolver a habilidade dos participantes de utilizar as chamadas “fork” (chamada de sistema que inicializa um outro processo) e utilização de “pipes” (ferramenta de linguagem para comunicação entre processos). Além de como objetivo final, desenvolver um programa que simule o interpretador de comandos presente no sistema operacional.

3. Realização e Objetivos

Realizado em dupla, o projeto foi desenvolvido através de reuniões e discussões online, onde discutíamos sobre possíveis caminhos a serem tomados para a implementação do projeto e compartilhamos ideias e soluções viáveis para correção de erros e bugs de programação.

Para o projeto, era necessário a realização de alguns parâmetros impostos pelo professor. O protótipo final deveria ser capaz de executar comandos simples, como “ls -la”. Deveria também ser capaz de executar comandos compostos, através da utilização de pipes “|”, onde deveria saber que o resultado de um comando anterior ao indicador do Pipe iria enviar sua saída para o comando posterior ao indicador, que receberia como entrada.

Era necessário que o protótipo fosse capaz de executar redirecionamento de entrada, denominado por “<”, indicando de onde o comando deveria pegar os dados de entrada, e redirecionamento de saída, denominado por “>” e “>>”, no caso desse, o programa deveria criar (caso necessário) o arquivo indicado e truncar (sobrescrever) ou concatenar, respectivamente, a saída do comando.

Para verificação de que o programa estava saindo como desejado, usamos o Terminal do próprio sistema para identificar qual era a saída do comando executado e verificar se código obtinha o mesmo resultado.

4. Implementação

A implementação do projeto tem inicio com a função main, que irá receber a linha de comando do usuário até que receba o comando “exit” ou encontre um erro. Incluímos as bibliotecas sys/stat.h, fcntl.h, unistd.h, errno.h, wait.h e string.h, para podermos usarmos as funções necessárias, além das bibliotecas stdio.h e stdlib.h, bibliotecas padrão de linguagem. Para isso, a função se inicia abrindo um comando While, com condição de para 1, isso significa que, enquanto 1 o programa executa novamente. Dentro desse comando, inicializamos uma variável do tipo string (vetor de variáveis char), imprimimos na tela um caractere “#” indicando que o programa está rodando e recebemos essa variável do teclado do usuário, utilizando uma máscara para recebermos corretamente até que o usuário insira um enter, indicando o final do comando.

A partir disso, fazemos uma comparação para sabermos se o usuário digitou um comando de saída do programa, caso tenha digitado, o programa finaliza. Caso contrário, um novo processo é criado, que por sua vez irá interpretar e executar o comando recebido, chamando o procedimento treatCommand, passando a string recebida como parâmetro.

Segue pseudocódigo:

```
int main()
        while 1
                char *bruteCommand
                printf "#"
                bruteCommand <- input
                if bruteCommand = "exit" ou bruteCommand = "exi"
                        break
                pid <- fork()
                if not pid
                        treatCommand(bruteCommand)
                        break
                else
                        wait(NULL)
        return 0
```

No procedimento treatCommand, começamos inicializando as variáveis necessárias onde iremos guardar o tamanho da string de comando, quantos pipes e quantos redirecionamentos de entrada devem ser interpretados.

Após isso, faremos a contagem do redirecionamento de entrada, caso haja algum, removemos o indicador mantendo o nome do arquivo de entrada, uma vez que os comandos do sistema podem receber um arquivo como parâmetro, quando localizamos um indicador de redirecionamento de entrada precisamos garantir que o comando certo receba tal arquivo como

parâmetro. Para isso, criamos um vetor de string (ou uma matriz de caracteres), sliceflux, com tamanho igual a um a mais que o número de redirecionamentos que irá guardar a separação da string de entrada nas substrings divididas pelo indicador de redirecionamento. Após criado essa matriz, chamamos o procedimento split, passando como parâmetro a string de entrada, a matriz sliceflux e o indicador de redirecionamento.

O procedimento split recebe como parâmetro a string de entrada do usuário, uma matriz de caracteres e um caractere a ser usado como comparação. Essa função faz uso do comando strtok, da biblioteca string.h para separar a string recebida como parâmetro e salvar a separação na matriz recebida, sendo que o comando separa a string nas ocorrências do caractere desejado, assim como chama a função removeSpaces, que retira espaços indesejados que possam ocorrer no início das strings referentes as divisões e faz o tratamento de aspas duplas e simples, no caso de direcionamento de arquivos.

Depois, juntamos a string de comando novamente, utilizando da função strcat da biblioteca string.h.

Terminado o tratamento dos redirecionamentos de entrada (caso houvesse algum), faremos a contagem dos pipes existentes. Cria-se uma matriz de caracteres chamada slicedCommand, com tamanho um maior que o número de pipes e chamamos novamente o procedimento split, passando como parâmetro a string de comando do usuário (cmd), a matriz slicedCommand e o indicador de pipe. Feito isso, chamamos o procedimento execCommands, que fara a interpretação da string cmd, realizando seus comandos individualmente.

Segue pseudocódigo das funções removeSpaces, spilt e treatCommand respectivamente:

```
char *removeSpaces(char fileName[])
        i <- strlen(fileName) - 1

        for j=0 to i
                if fileName[j] = 34 ou fileName[j] = 39
                        k <- j
                        while k<j
                                fileName[k] <- fileName[k+1]
                                k <- k +1

        while fileName[i] = ' '
                fileName <- '\0'
                i <- i +1

        k <- 0
        while fileName[k] = ' '
                for i = 0 to strlen(fileName)
                        fileName[i] <- fileName[i+1]
                k <- k+1

        return fileName

split(char cmd[], char *sliceCmd[], char c[])
        char *aux
        int numSlices <- 0, sizeCommand <-0
        aux <- strtok(cmd, c)

        while aux != NULL
                slicedCmd[numSlices] <- aux
                aux <- strtok(NULL, c)
                numSlices <- numSlices + 1
        sliceCmd[numSlices] <- NULL

        for j = 0 to numSlices
                sliceCmd[j] <- removeSpaces(sliceCmd[j])
```

```
treatCommad(char cmd[])
        cmdsize <- strlen(cmd)
        numpipes <- 0
        numantiflux <- 0
        k <- 0

        for i = 0 to cmdsize
                if cmd[i] = '<'
                        numantiflux <- numantiflux + 1

        if numantiflux > 0
                char *sliceflux[numantiflux + 1]
                split(cmd, sliceflux, "<")

                while(numantiflux > 0)
                        strcat(sliceflux[0], " ")
                        strcat(sliceflux[0], sliceflux[k+1])
                        numantiflux <- numantiflux - 1
                        k <- k + 1

                cmd <- sliceflux[0]
                for i=0 to cmdsize
                        if cmd[i] = '|'
                                numpipes <- numpipes + 1

                char *slicedCommand[numpipes + 1]
                split(cmd, slicedCommand, "|")

                execCommands(slicedCommand, numpipes)
```

O procedimento execCommands é o mais complexo do projeto, ele deve fazer a interpretação dos comandos, gerenciar quantos processos serão criados, identificar entrada e saída entre esses processos e chamar os demais procedimentos e funções do código no momento certo. Recebemos como parâmetro a matriz de caracteres, que guarda os comandos a serem executados, e o número de pipes a serem realizados.

Começamos criando uma variável para guardar o id do processo a ser realizado e criamos o número de pipes necessários, que será duas vezes o valor recebido como parâmetro, uma vez que cada processo necessita de um pipe, que tem uma posição de leitura e uma de escrita, fazemos a inicialização dos pipes e o tratamento de erro dos mesmos.

Antes de entrarmos no laço que era realizar o trabalho de fato, criamos e inicializamos uma variável de controle, que irá ditar o caminhar das interações do laço.

Entramos no laço e inicializamos um novo processo, com isso criamos uma condicional para saber se estamos no processo filho ou se tivemos um erro na criação do processo, esse laço anda duas posições a cada iteração.

Caso o processo tenha sido criado sem erros, começamos contando o número de redirecionamentos de saída, feito isso entramos numa sequencia de condicionais para tratar erros nos pipes dos filhos. Seguimos com um laço de repetição para fechar os pipes já utilizados e criamos uma variável chamada currentCommand, um vetor de strings com tamanho igual ao número de comandos na divisão que queremos executar, para isso chamamos a função countCommands.

Nessa função, separamos o comando de seus parâmetros, para que seja devidamente interpretado pela chamada execvp.

Após termos inicializado essa variável, entramos em outra sequencia de condicionais para sabermos como tratar a string de comandos. Verificamos se temos algum número de redirecionamento de saída, caso haja, chamamos o procedimento split, passando como parâmetro a fatia de string de comandos que queremos dividir, a matriz currentCommand e o caractere indicador de redirecionamento de saída. Feito isso, iremos chama o procedimento executeOutputFileCommand. Esse procedimento recebe como parâmetro a primeira e a segunda fatia do trecho a ser executado, isto é, a primeira e a segunda strings da matriz currentCommand, recebe também o número de ocorrências do indicador de redirecionamento. Iremos discutir essa função posteriormente.

Terminada essa condicional, chamaremos o procedimento split novamente, dessa vez passamos como parâmetro a fatia desejada que não contem indicadores de redirecionamento, a matriz currentCommand, e uma indicação de espaço (“ “). Assim podemos chamar a chamada de sistema execvp, passando como parâmetro a primeira posição da matriz currentCommand e a própria matriz.

Saindo do laço principal da função, iremos adicionar mais dois laços que tem como função fechar os pipes do processo pai (para cada filho criado) e esperar os filhos terminarem a execução, respectivamente. Assim terminamos o procedimento execCommands.

Segue pseudocódigo da função countCommands e dos procedimentos execCommands, executeOutputFileCommand, respectivamente:

```
int countCommands(char cmd[])
        spaces <- 0
        sizeCommand <- size of cmd
        for i=0 to sizeCommand
                if cmd[i] == ' ' and cmd[i+1] == ' '
                        spaces <- spaces + 1
        return spaces
```

```
void execCommands(char *sliced[], int numpipes)
        pid_t pid;
        int pipefd[2*numpipes]
        for i = 0 to numpipes
                if pipe(pipefd+i*2) < 0
                        perror("erro ao criar pipe")
                        exit(EXIT_FAILURE)
        count <- 0
        while count < (numpipes+1)*2
                pid <- fork()
                if pid = 0
                        numflux <- 0
                        for i=0 to strlen(sliced[count/2])
                                if sliced[count/2][i] = '>'
                                        numflux = numflux + 1
                        if count != 0
                                if dup2(pipefd[count -2], STDIN_FILENO) < 0
                                        perror("error dup2 input")
                                        exit(EXIT_FAILURE)
                        if numpipes*2 != count
                                if dup2(pipefd[count+1], STDOUT_FILENO) < 0
                                        perror("error dup2 output")
                                        exit(EXIT_FAILURE)
                        for i = 0 to 2*numpipes
                                close(pipefd[i])
                        char *currentCommand[countCommands(sliced[count/2])]
                        if numflux > 0
                                split(sliced[count/2], currentCommand, ">")
                                executeOutputFileCommand(currentCommand[0], currentCommand[1], numflux)
                                continue
                        split(sliced[count/2], currentCommand, " ")
                        if execvp(currentCommand[0], currentCommand) < 0
                                fprintf(stderr, "error while executing command %s\n", currentCommand[0])
                                exit(EXIT_FAILURE)
                else if pid < 0
                        perror("Fork error")
                        exit(EXIT_FAILURE)
                count += 2
        for i = 0 to numpipes*2
                close(pipefd[i])
        for i = 0 to numpipes+1
                wait(NULL)
```

```
void executeOutputFileCommand(char args[], char fileName[], int x)
        int fd0
        chose x
                case 1
                        fd0 <- open(fileName) for overwrite
                case 2
                        fd0 <- open(fileName) for append
                default
                        perror("Use one 1 per command")
                        exit(EXIT_FAILURE)
        dup2(fd0, STDOUT_FILENO)

        char *command[countCommands(args)]
        split(args, command, " ")
        if execvp(command[0], command) < 0
                perror("error execvp executeOutput")
                exit(EXIT_FAILURE)

        close(STDOUT_FILENO)
        close(STDIN_FILENO)
        close(fd0)
```

Com isso concluímos o código e obtemos um Shell funcional que cumpri objetivos propostos pelo professor.

5. Conclusão

Nesse trabalho vimos como funciona o tratamento de uma linha de comando no Terminal Linux. Bem como desenvolvemos habilidades com a manipulação de processos e comunicação entre processos.

Através de chamadas de sistema com execvp, desenvolvemos um interpretador de comandos funcional, que realiza tarefas como execução de comandos simples, comandos compostos, enviar o resultado de uma cadeia de comandos para um arquivo designado pelo usuário, assim como executar um comando a partir de um arquivo preexistente.

Durante a execução do trabalho, tivemos vários erros de lógica e sintaxe, tal como mudanças de como executarmos o projeto proposto, mas o grupo se mostrou resiliente em achar uma solução em conjunto, através de reuniões online, e recomeçar o projeto de vária maneiras até encontrar uma solução viável, aproveitando os erros e acertos de outras iterações. Foi necessário ajuda externa, tanto do professor quando de outros alunos mais avançados no curso, para sanar eventuais duvidas que se apresentaram em diferentes momentos. A maior dificuldade do trabalho foi conseguir organizar corretamente a comunicação entre os processos criados, uma vez que é impossível saber quantos comandos o usuário vai desejar realizar.