Sistemas Operativos

Projeto - *Shell Simples para o Linux*

Ano: 2019/2020 / 1º semestre / 2º Ano

O Docente:
Luís B Garcia

Discentes:
Jorge Colaço/16524
Hugo Alexandre Silva/18544

Beja, aos 30 de Janeiro de 2020

Índice

1. Introdução

Este projeto trabalha com desenvolvimento de funções para a matéria de Sistemas Operacionais onde o professor responsável Luiz Garcia nos deu como tarefa criar um conjunto de shell simples denominada ss (simple shell). Além disso as funções devem fornecer informações ao utilizador durente o processo. Utilizamos um terminal virtual com Linux Ubuntu. O ponto a cumprir, deve aceitar um comando introduzido pelo utilizador e devolvê-lo, através de parâmetros command; estes comandos devem ser armazenados em um vetor; construindo assim um vetor que contenha em cada posição um dos argumentos do comando e na posição seguinte ao último argumento deve ser colocada a constante null como indicação do fim dos argumentos estabelecidos; tendo em conta uma shell filho onde construímos um vetor de argumentos para executar o comando através da função execvp e deve-se esperar a conclusão da shell filho para então ficar disponível para a introdução de um outro comando.

Para o desenvolvimento do projeto, utilizamos a linguagem de programação C, uma linguagem de alto nível onde podemos trabalhar próximo da máquina, tendo disponível diversas características de portabilidade, compilação, eficiência e regularidade.

Para trabalhos com os códigos utilizamos o editor “gedit”, onde temos uma facilidade de percepção do código, uma vez que o mesmo identifica a linguagem, além das variáveis e funções, separando as mesmas por cores e sequência, o que ajuda nas identificações e facilidade de uso para o devido trabalho.

Para testar o funcionamento da shell ao longo do desenvolvimento, é necessário utilizar um compilador para traduzir linguagem de alto nivel em linguagem de máquina, no nosso caso utilzamos o compilador Gnu Compiler Collection, abreviado por G.C.C.

Utilizamos juntamente com esse desenvolvimento a ferramenta GitHub para compartilhar o desenvolvimento e partes de documentação encontradas por ambos componentes do grupo, ao que poderá ser encontrado no seguindo url: “https://github.com/MistahJorge/SO-TG2”, a qual o docente já se encontra com o convite enviado para se juntar ao compartilhamento e com isso ter acesso aos commits, comentários e participação do desenvolvimento.

Concluímos que se tudo ocorrer devidamente bem teremos sempre em ecrã as informações e ações executadas durante todo o processo, juntamente com os processos de excução da shell pai e filho.

2. void GetCommand(char command[])

Nesta primeira etapa pretende-se criar um desenvolvimento onde essa função deverá escrever na ecrâ o comando inserido pelo utilizador dentro de um ciclo, até que o utilizador digite ‘quit’ para sair do programa.

No código utilizamos o strcmp para comparar a string comando, com “quit”, enquanto não for igual, dando resultado diferente de 0, continua o ciclo. “fgets” para receber a entrada de string a qual recebe os caracteres e armazena no buffer apontado para o “s”, até a interrupção após a EOF ou uma nova linha. Completamos a ação com o comando “strlen”, que calcula o comprimento da string, excluindo o byte de terminação nula.

Em relação as bibliotecas que estamos usando, são as suas características e ações conforme suas rotinas carregadas: “stdio”, funções de biblioteca para entrada e saída padrão, “string”, executa operações em cadeia terminado em nulos, “stdlib”, é um nível superior ao stdio, a qual faz parte da stdlib, controle de rotinas automáticas, “unistd.h”, composta por funções wrapper de chamadas do sistema, apoia ações do fork, pipe e primitivas de entrada e saída, “errno”, fornece macros para identificar e relatar erros de execução através de códigos de erro, “sys/wait”, emite chamadas de espera sistema, que suspende a execução de determinados processos, até o status de saída para o sistema operacional e liberação para outro processo, utilizado para controlo de processos entre pai e filho.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>

void GetCommand(char command[]){
        printf("Para confirmar, o seu commando é: %s \n", command);
}

```

Imagem1 – código getcommand

```c
int main(){
        char comando[256];
        char *ponteiro = comando;
        char *argVetor[512];
        char *cmdVetor[512];
        while (strcmp(comando,"quit") != 0){
                printf("Introduza um commando para eu executar: ");
                fgets(comando, 255, stdin);
                ponteiro[strlen(ponteiro)-1] = 0;
                GetCommand(comando);
                makeCmdVector(comando, cmdVetor);
        }
        return 0;
}
```

Imagem2 – código main

Optamos por criar um main para que durante o processo de execução do código e desenvolvimento efetuássemos o chamamento das demais funções aqui no "MAIN", sendo assim o ciclo do GetCommand é executado no Main com o seu chamamento, assim como as demais funções conforme será demonstrado nas sequências abaixo.

3. int PrintArguments(char command[])

Neste desenvolvimento os comando digitados deverão ser divididos nos argumentos e demonstrados em tela sua divisão, utilizando a função "strtok", str - conteúdo é modificado e dividido em strings menores (tokens), "delim"- esta é a string que contém os delimitadores.

```
12  int PrintArgs(char command[]){
13          const char delim[2] = " ";
14          char *token;
15          char str[512];
16          strcpy(str, command);
17          token = strtok(str, delim);
18          while(token != NULL) {
19                  printf("%s\n", token);
20                  token = strtok(NULL, delim);
21          }
22          return 0;
23  }
```

Imagem3 – código printarguments

4. makeArgVector(char command[], char *argVector[])

Para o desenvolvimento destas função foi solicitado para que a mesma execute a construção de um vetor que contenha os argumentos em suas posições, ao preencher todos as posições o ultimo deve conter a constante NULL para indicar o fim, deve ser usado a função “malloc”, ou seja, alocação dinâmica de memória, aloca espaço para um bloco de bytes consecutivos na memoria RAM e devolve o endereço desse determinado bloco. Por isso a utilização do ponteiro e void, pois é uma função do tipo genérico e armazenamos o endereço num ponteiro do tipo apropriado, no nosso caso “char”.

```
40  void makeArgVector(char command[], char *argVector[]){
41          int i = 0;
42          const char delim[2] = " ";
43          char *token;
44          char str[256];
45
46          argVector = malloc(sizeof(str));
47          strcpy(str, command);
48          token = strtok(str, delim);
49          while( token != NULL ){
50                  argVector[i] = malloc(sizeof(token));
51                  argVector[i] = token;
52                  i = i + 1;
53                  token = strtok(NULL, delim);
54          }
55          argVector[i+1] = malloc(sizeof(char*));
56          argVector[i+1] = NULL;
57          for(int j = 0; j <= i; j++){
58                  printf("argVector[%i] = %s\n", j, argVector[j]);
59          }
60          exeCommand(argVector);
61          return;
62  }
```

Imagem4 – código makeargvector

5. execvp, ss - exeCommand

Nesta etapa cinco concluímos a primeira parte do trabalho com o caso para desenvolver a versão da “ss”, para que o comando recebido pela shell deva lançar uma shell filha e constuir um vetor de argumentos com suas devidas execuções através da função “execvp”, que é da família execução, cuja ação é substituir o processo atual por uma nova imagem do processo, onde no caso do “vp”, fornece uma matriz ou vetor com ponteiros para sequenciar as ações e terminar em nulo, que representa o fim da lista de argumento para disposição de um novo programa.

Para isso criamos uma função “exeCommand()” que recebe os valores de argVector e temos o chamamento do pai e filho e as diretrizes. “Fork”, cria um processo através da duplicação do processo de chamada, referido como o filho, “pid_t”, retorna um valor positivo, o ID do processo filho, para o pai. O ID do processo retornado é do tipo pid_t definido em sys/types.h, “getpid”, para recuperar o ID do processo atribuído e demonstrar na tela conforme printf.

```
25  void exeCommand(char *argVector[])
26  {
27          pid_t pid, childPid;
28          pid = fork();
29          if(pid == 0){
30                  execvp(argVector[0], argVector);
31                  printf("\nProcesso pai. PID:|%d|\n",getppid());
32                  printf("\nProcesso filho. PID:|%d|\n",getpid());
33          }else{
34                  wait (&childPid);
35                  printf("\nConluido!\n");
36                  printf("\nProcesso pai. PID:|%d|\n",getppid());
37                  printf("\nProcesso filho. PID:|%d|\n",getpid());
38          }
39  }
```

Imagem5 – código execommand

6. funcionalidades avançadas na shell

Nesta etapa iniciamos a parte dois do projeto, onde devemos implementar usos de possibilidades de execução com determinadas funcionalidades dispostas em projeto e solicitada pelo professor.

Exemplos: implementar a possibilidade de execução de listas de comandos, por exemplo: sleep 5; date; sleep5; date

implementar a possibilidade de utilização da redireção, por exemplo: ls -l > file_list.txt

implementar a possibilidade de utilização de pipelines, por exemplo: ls -l | grep ‘^d’ | wc -l

Aqui temos disponível o código para a primeira tarefa, sequencial ao “;”, onde será o nosso ponto de separação, para execução dos comandos, armazenamento dos argumentos. Usamos como base de processo o código da primeira parte onde, usamos o “strtok” para a dividir as string, mas antes com o “cpy” copiamos a string original para efetuarmos os trabalhos numa string nova, o “delim” nesse caso continua recebendo o valor [2], devido a receber a identificação da string “;”, que corresponde a 1(um) e “/0”, que corresponde a 2(dois), que significa o fim da string e deve efetuar a separação, conduzindo para as demais operações de argumentos, comandos e execução separadas.

Com a criação da nova função makecmdvector, buscamos aperfeiçoar na função a utilização de quase todas as funções solicitadas durante o trabalho, para assim garantirmos resultados perfeitos, usando além das funções acima já citadas, também fazendo recurso do “malloc” e utilização de um contador para as posições no vetor, e mais dois “for”, um para impressão dos resultados e acompanhamento em tela dos argumentos atribuídos e o segundo para chamar as demais funções para o ciclo de funcionamento conforme ainda existe processos ou aguardo de processos para serem executados.

A saber a execução visada para o projeto é em cascata, onde uma função recebe da outra função os comandos, efetua as suas atribuições de argumento separadas e efetuando suas comunicações e chamamento pelo main, até que o utilizador solicite a sua interrupção ou saída do programa.

```
63  void makeCmdVector(char command[], char *cmdVector[]){
64          int i = 0;
65          const char delim[2] = ";";
66          char *argVetor[512];
67          char *token;
68          char str[256];
69          cmdVector = malloc(sizeof(str));
70          strcpy(str, command);
71          token = strtok(str, delim);
72          while( token != NULL ){
73                  cmdVector[i] = malloc(sizeof(token));
74                  cmdVector[i] = token;
75                  i = i + 1;
76                  token = strtok(NULL, delim);
77          }
78          cmdVector[i+1] = malloc(sizeof(char*));
79          cmdVector[i+1] = NULL;
80          for(int j = 0; j <= i; j++){
81                  printf("cmdVector[%i] = %s\n", j, cmdVector[j]);
82          }
83          for(int j = 0; j <= i; j++){
84                  PrintArgs(cmdVector[j]);
85                  makeArgVector(cmdVector[j], argVetor);
86          }
87          return;
88  }
```

Imagem6 – código execução de listas de comandos

7. Conclusões Finais

O trabalho foi apresentado e elaborado dentro da proposta de vários funçõess, para ação e execução das tarefas baseadas a partir de um desenvolvimento para shell do linux.

A elaboração de criação do programa demanda principalmente da performance do sistema e utilização das funções solicitadas em cada tarefa, tendo em consideração os comandos digitado pelo utilizador e permissões de acesso e escrita, adaptando e eliminando falhas para não prejudicar a execução das tarefas.

Baseado no proposto, podemos dizer que este trabalho foi um bom começo para o aprendizado sobre como trabalhar com shell, comandos, funçõess, funções, segurança no fornecimento do processo, o que nos incentivou este desenvolvimento sequencial foi a possibilidade de ser aplicado em algo real e que pode ser utilizado para outros modelos e parâmetros.

Concluindo este projeto podemos dizer que a necessidade de um processo com respostas de confirmação em cada etapa das funções foi necessária para garantir a integridade e os utilizadores poderem ver cada etapa e as funções executadas, conforme código.

Estamos satisfeitos com os resultados e conhecimento adquirido e esperamos ter alcançado os objetivos do professor!

8. Referências bibliográficas

Moodle

https://cms.ipbeja.pt/pluginfile.php/227883/mod_resource/content/1/tarefa4.pdf

https://cms.ipbeja.pt/pluginfile.php/228485/mod_resource/content/1/Introdu%C3%A7%C3%A3o%20%C3%A0%20Gest%C3%A3o%20de%20Processos.pdf

https://cms.ipbeja.pt/pluginfile.php/229019/mod_resource/content/1/Escalonamento%20Processos.pdf

https://cms.ipbeja.pt/pluginfile.php/229766/mod_resource/content/1/Comunica%C3%A7%C3%A3o%20e%20Sincroniza%C3%A7%C3%A3o%20Processos.pdf

Youtube

https://www.youtube.com/watch?v=tF0Qau7zcsw&list=PLS1QulWo1RIYmaxcEqw5JhK3b-6rgdWO_&index=27

https://www.youtube.com/watch?v=EOLPUc6oo-w&list=PLucm8g_ezqNrYgjXC8_CgbvHbvI7dDfhs

Tutoriais

https://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html

http://www.br-c.org/doku.php?id=fork

https://linux.die.net

https://terminalroot.com.br/2015/07/30-exemplos-do-comando-sed-com-regex.html

https://www.livrosdelinux.com.br/respostas-livro-shell-linux/