UNIVERSIDADE FEDERAL DE SÃO PAULO
INSTITUTO DE CIÊNCIA E TECNOLOGIA

ARTHUR LOSANO DE ARAUJO SILVA - RA 163564
EVANDRO KEIJI KAYANO - RA 163707
LUCAS APOLINÁRIO DE OLIVEIRA - RA 163913
THIAGO CORSO CAPUANO - RA 163996

IMPLEMENTAÇÃO DE UM SHELL NA LINGUAGEM C

RELATÓRIO SOBRE O DESENVOLVIMENTO E FUNCIONAMENTO

SÃO JOSÉ DOS CAMPOS
2024

Introdução

Neste relatório, exploraremos o desenvolvimento de um shell básico em linguagem C. É uma das ferramentas mais fundamentais e poderosas disponíveis em sistemas operacionais baseados em UNIX, como Linux e macOS, bem como em sistemas semelhantes.

Um shell é essencialmente um interpretador de comandos que aceita entradas de texto do usuário e as converte em ações específicas do sistema operacional. Ele atua como uma camada intermediária entre o usuário e o kernel do sistema operacional, facilitando a execução de tarefas como gerenciamento de arquivos, execução de programas, configuração do sistema e muito mais.

Os shells oferecem uma maneira consistente e padronizada de interagir com o sistema operacional, independentemente do ambiente de trabalho ou interface gráfica disponível. Isso os torna uma ferramenta indispensável para administradores de sistemas, desenvolvedores de software, cientistas de dados e qualquer pessoa que trabalhe extensivamente com sistemas baseados em UNIX.

Estrutura do Programa

O programa em questão é uma implementação de um shell básico em linguagem C, projetado para executar em sistemas operacionais Linux. Sua estrutura geral é organizada de forma modular, com funções distintas desempenhando papéis específicos na execução e no gerenciamento de comandos do usuário. Vamos examinar mais de perto as principais componentes do programa:

- **Loop principal**

A função atua como o ponto de entrada do programa. Ela inicia o shell, exibe uma mensagem de boas-vindas e invoca a função “printDir()” para imprimir o prompt inicial do shell. Além disso, a função contém o loop principal do shell, onde aguarda o input do usuário, processa os comandos inseridos e executa as ações correspondentes.

O loop principal do programa continua executando até que o comando exit seja inserido pelo usuário. Ele solicita o input do usuário, processa os comandos inseridos e executa as ações correspondentes. Durante esse processo, pode identificar e lidar com operadores especiais, como “||”, “&&”, “|” e “&”.

- **Função “printDir”**

A função pega o nome do diretório atual, nome do usuário e nome da máquina e imprime elas de forma que simule um terminal linux. Caso o diretório atual exceda 3 diretórios, é apenas impresso o atual diretório e o anterior, separados por uma “/” e os diretórios anteriores não são impressos e em seus lugares é impresso reticências e uma barra “.../” .

```
void printDir(){
  system("sleep 1");
  char dir[100];
  getcwd(dir,sizeof(dir));
  char *user = getenv("USER");
  char host[100];
  gethostname(host,sizeof(host));

  char* actual = strtok(dir, "/");
  char** address = malloc(sizeof(char*) * 10);
  int i = 0, count;

  while(actual != NULL){
    if(strcmp(actual, user) == 0)
      count = i;

    address[i] = actual;
    actual = strtok(NULL, "/");
    i++;
  }
```

```
  printf(GREEN"%s@%s"WHITE":"BLUE"~/",user,host);

  if(i>4)
    printf(".../%s/%s",address[i-2],address[i-1]);
  else{
    for(int j=count+1; j<i-1; j++)
      printf(BLUE"%s/",address[j]);
    printf(BLUE"%s",address[i-1]);
  }
  printf(WHITE"$ ");
}
```

- **Processamento de input de usuário (processamento da string de input)**

Uma função evidentemente central em um shell é, evidentemente, a possibilidade de enviar comandos para que a máquina possa processar. Desta forma, o processamento de input (entrada) de usuário é crucial para o programa. Tal processamento deve ser feito da maneira correta para que o computador consiga entender os comandos e seus argumentos sem erros.

Com isto, realizamos a obtenção do input de usuário através do comando fgets e adicionamos um operador \0 em seu final, para indicar para o computador onde acaba nossa string (comando necessário por conta de particularidades do fgets). Após isso, verificamos onde existem aspas duplas (“) no input, já que estas não devem ser passadas para o execvp ao executar o comando. Porém, como em nosso próximo passo estaremos separando os comandos e argumentos em um vetor usando a tecla espaço como delimitador, removemos as aspas porém substituímos quaisquer espaços que podem haver dentro do argumento que está sendo agrupado por elas, pelas aspas duplas (neste caso, elas servirão apenas de placeholder para que este argumento não seja separado, o que resultaria em bugs no execvp, e serão retiradas logo mais). Como estaremos (possivelmente) removendo caracteres da string, se faz necessário mudar a posição dos outros caracteres.

```
fgets(input, sizeof(input), stdin);
input[strlen(input)-1] = '\0';

int aux = 0, len = strlen(input), quot = 0;
for (int z = 0; z+aux < len; z++) {
  if (input[z] == 34) {
    aux++;
    quot++;
    if (input[z+1] == 34) aux++;
  }
  if (aux != 0) input[z] = input[z+aux];
  if (input[z] == 32 && quot % 2 == 1) input[z] = 34;
}
if (aux != 0) input[len-1-aux] = '\0';
```

Agora, estaremos utilizando a função strtok para separar todos os comandos/argumentos do nosso input que estão, obviamente, separados por espaços, e salvaremos cada comando/argumento em um novo vetor.

```
char *palavra = strtok(input, " ");
int i = 0;
while(palavra != NULL){
  argumentos[i] = palavra;
  palavra = strtok(NULL, " ");
  i++;
}
argumentos[i] = NULL;
```

Por fim, iremos verificar cada comando/argumento em nosso novo vetor à procura de quaisquer aspas duplas (caso encontrada, recebemos, através de strchr, sua posição na string do argumento) e as substituímos por espaço (lembrando que

esta tarefa está sendo feita ao mesmo tempo que a de detecção do operador de background, ampersand (&)).

```
int amp = -1;
char ampers[2] = "&";
for (int x = 0; x < i; x++) {
  char *auxchar = strchr(argumentos[x], 34);
  while (auxchar != NULL) {
    auxchar[0] = 32;
    auxchar = strchr(argumentos[x], 34);
  }
  if (strchr(argumentos[x], 38) && strlen(argumentos[x]) == strlen(ampers))
    amp = x;
}
```

- **Função “identificaOperador”**

A função desempenha um papel crucial na análise dos argumentos inseridos pelo usuário. Ela percorre a lista de argumentos fornecida como parâmetro e compara cada elemento com os operadores especiais, como “||”, ”&&”, “|” e “&”. Se um operador é encontrado, a função retorna imediatamente com informações sobre o tipo de operador identificado e sua posição na lista de argumentos. Isso permite que o programa identifique operadores especiais e tome decisões adequadas com base neles durante a execução dos comandos.

- **Função “fazCd”**

A função permite que o usuário navegue pelo sistema de arquivos do sistema operacional. O comando “cd” (change directory) é uma operação comum em sistemas baseados em UNIX, e essa função implementa sua funcionalidade dentro do shell.

Ao fazer a chamada de sistema “cd”, devemos considerar três situações:

- Somente é passada a função “cd”. Neste caso o shell muda o diretório atual para o diretório home do usuário;
- É passado “cd ..”. Para este caso, o shell “retorna um diretório”, ele volta para o diretório pai;
- E o caso padrão em que é passado “cd xxx” onde “xxx” é o nome de um subdiretório do diretório atual. O shell muda do diretório atual, para um de seus subdiretórios.

```c
int fazCd(char** argumentos,int i) {
  if(i == 1){
    char *home = getenv("HOME");
    chdir(home);
    return 0;
  }

  if(strcmp(argumentos[1],"..") == 0){
    chdir("..");
    return 0;
  }

  if(chdir(argumentos[1]) != 0){
    perror("cd error");
    return -1;
  }
return 0;
}
```

- **Processamento de comandos em background (ampersand &)**

O processamento de comandos em background é muito importante para que seja possível realizar tarefas que demoram muito tempo, sem que sejamos impedidos de realizar outras neste meio tempo. O sinal ampersand (&) é usado para demonstrar que queremos executar o comando que está escrito antes dele em background.

Com isto, primeiramente analisamos o input do usuário para descobrir se há algum sinal ampersand (&) e, caso haja, salvamos sua posição no vetor de argumentos em uma variável (esta tarefa é feita juntamente do processamento de aspas duplas (“) no input, no mesmo looping for).

```c
int amp = -1;
char ampers[2] = "&";
for (int x = 0; x < i; x++) {
  char *auxchar = strchr(argumentos[x], 34);
  while (auxchar != NULL) {
    auxchar[0] = 32;
    auxchar = strchr(argumentos[x], 34);
  }
  if (strchr(argumentos[x], 38) && strlen(argumentos[x]) == strlen(ampers))
    amp = x;
}
```

Sabendo a posição (caso haja) do operador indicador de processo em background, transformamos ele em NULL no nosso vetor (para auxiliar na execução do código através do execvp, que roda os argumentos até encontrar NULL) e realizamos um fork(). Com isso, limpamos a segunda parte (tudo após o operador &) do vetor de argumentos do filho (atribuímos NULL à todas as posições correspondentes) e, no caso do pai, simplesmente atualizamos seu ponteiro do array de argumentos para que ele comece na posição seguinte ao operador &. Desta forma, o filho irá realizar o processamento dos comandos em background enquanto o pai irá realizar (caso existam) os comandos que devem ser realizados no processo principal (em primeiro plano). Ambos processos irão seguir normalmente o resto do código, com a exceção de que nos locais em que o processo filho poderia chegar ao fim do loop principal e seguir para a próxima iteração, há um comando condicional que encerra o processo.

```c
int pidamp;
if (amp != -1) {
  argumentos[amp] = NULL;
  pidamp = fork();
  if (pidamp == 0) {
    for (int g = amp; g < i; g++) argumentos[g] = NULL;
  }
  else if (pidamp > 0) {
    argumentos = &argumentos[amp+1];
    if (argumentos[0] == NULL) goto inicio;
  }
}
```

Caso não haja operador ampersand, isto é, o comando deve ser realizado em primeiro plano, realizamos um fork() em que a única função do pai é esperar o filho terminar, enquanto o filho irá realizar a execução do comando pelo código até encontrar a condicional que encerrará seu processo. Quando o processo filho encerrar seu processo, o processo pai, que estava o aguardando, agora realiza um goto para retornar ao início do looping principal.

```c
else if (amp == -1) {
  amp = 0;
  pidamp = fork();
  if (pidamp > 0) {
    waitpid(0, NULL, 0);
    goto inicio;
  }
}
```

- **Processamento de pipes**

Os pipes são de suma importância em nosso shell, já que são eles que permitem que a saída de um comando se torne a entrada de um próximo comando na sequência sucessivamente até a chegada do último comando, que terá sua saída normalmente no terminal.

O processamento de pipes no programa começa na própria main(), quando o programa checa para ver se existe um operador pipe “|” (barra vertical única) no comando recebido do usuário. Caso de fato haja, sabemos que precisamos processar este comando separadamente (a fim de processar seu pipe, já que precisaremos redirecionar saída de um comando para a entrada de um próximo através de pipes e dos File Descriptors STDIN_FILENO e STDOUT_FILENO). Com isso, fazemos um fork() para fazer com que o processo filho chame a função executaPipe(). Temos uma condicional para checar se pidamp possui valor 0, pois se trataria do processo filho caso haja operador ampersand “&” no comando (neste caso, ao invés de entrar no looping do shell, o processo filho apenas deve fazer return 0 para encerrar). Caso não se trate de tal processo filho, temos um “goto” para retornar ao início do looping e receber outro comando (“goto” foi necessário pois estamos dentro de 2 nestes loops, o “for” de checagem de pipe e o “while” que continua a execução do shell, então seria necessário realizar “break” seguido de

continue, o que não seria tão trivial, situação em que o “goto” realiza perfeitamente a tarefa).

```
char pip[2] = "|";
int tamanho = i;
for (int j = 0; j < tamanho - 1 - amp; j++) {
  if (strcmp(argumentos[j], pip) == 0 && strlen(argumentos[j]) == strlen(pip)) {
    if (fork() == 0) executaPipe(argumentos, tamanho - amp - 1);
    waitpid(0, NULL, 0);
    if (pidamp == 0) return 0;
    goto inicio;
  }
}
```

A função “executaPipe”, então, recebe os comandos encadeados por pipes e o número de comandos. Com isso, usamos um ponteiro para apontar para o começo da lista de comandos (será útil para passarmos para o “execvp” a lista começando no comando da vez, seja o primeiro, segundo, até o último comando) e então contamos quantos operadores pipe “|” existem na sequência (útil pois devemos rodar um looping n (número de pipes) vezes, que seria o mesmo que m (número de comandos) - 1, pois o último comando deve ser executado fora do looping, já que este não terá seu “STDOUT_FILENO” redirecionado) e os substituímos por “NULL” (pois o execvp executa o comando e seus argumentos até encontrar “NULL”, e o pipe é justamente o que está delimitando neste caso).

```
void executaPipe(char** arg, int tamanho) {
  int count = 0, fd[2], i, j = 0;
  char **comando = arg;

  char pip[2] = "|";
  for (i = 0; i < tamanho; i++) {
    if(strcmp(arg[i], pip) == 0) {
      count++;
      arg[i] = NULL;
    }
  }
```

Criamos a variável que carregará o resultado do pipe anterior com valor atual “STDIN_FILENO” (ou seja, 0) e iniciamos a nossa iteração que executará n (número de pipes) vezes. Dentro dessa iteração, abrimos o pipe com “pipe(fd)” e realizamos um “fork”. Com isso, o processo filho irá checar se há algum input do comando anterior e, caso haja, ele o receberá (ao fazer o redirecionamento do resultado do comando anterior para “STDIN_FILENO”). Com o input do comando anterior (caso haja), agora redirecionamos (usando “dup2”) a saída do comando (STDOUT_FILENO) que será executado para o write-end (ponto de escrita) do pipe (“fd[1]”) e executamos o comando. Com isso, o processo pai (que estava aguardando o processo filho terminar), salvará o read-end (ponto de leitura) do pipe (“fd[0]”) em sua variável. Agora, antes de encerrar a iteração, precisamos fazer com que nosso ponteiro aponte para o próximo comando a ser executado, então fazemos um for que procura NULL em nosso vetor de comandos (lembrando que, anteriormente, este era o operador pipe “|”) e, ao encontrá-lo, soma 1 à variável que estava atualizando o índice dessa procura (para estarmos no índice do comando após o NULL/operador pipe) e atualizamos o ponteiro com esta nova localização:

```c
int result_anterior = STDIN_FILENO;

for (i = 0; i < count; i++) {
  if (pipe(fd) < 0) {
    printf("Broken Pipe\n");
    return;
  };
  pid_t pid = fork();
  if(pid == 0) {
    if (result_anterior != STDIN_FILENO) {
      dup2(result_anterior, STDIN_FILENO);
      close(result_anterior);
    }
    dup2(fd[1], STDOUT_FILENO);
    close(fd[1]);
    execvp(comando[0], comando);
    exit(1);
  }
  else if (pid > 0) {
    waitpid(0, NULL, 0);
    close(result_anterior);
    close(fd[1]);
    result_anterior = fd[0];
    for(j = 0; comando[j] != NULL; j++);
    j++;
    comando = &comando[j];
  }
}
```

Por fim, agora fora do loop for principal da função, vamos fazer algo semelhante: vamos checar se há input do comando anterior (caso haja, vamos redirecioná-lo para o “STDIN_FILENO”) e vamos executar o comando (note que, desta vez, não redirecionamos “STDOUT_FILENO” pois queremos receber o resultado deste último comando através do terminal). Com isso, temos o fim do processamento de pipes.

```c
  if (result_anterior != STDIN_FILENO) {
    dup2(result_anterior, STDIN_FILENO);
    close(result_anterior);
  }
  execvp(comando[0], comando);
  exit(1);
}
```

- **Processamento de operadores lógicos**

Assim que a string input é manipulada, dividida em substrings e verificada se possui pipe (|) ou ampersand (&), o shell entra em um do-while, que se mantém no loop até que o input seja um comando simples sem operadores. Em seguida, é chamada a função “identificaOperador”, que retorna um vetor de duas posições ops[2].

Quando a função “identificaOperador” for concluída, ela retorna um vetor, onde sua primeira posição (ops[0]) nos mostra qual foi o primeiro operador detectado e em sua segunda posição (ops[1]), nos mostra sua posição na lista de argumentos do input. Se o valor de ops[0] for igual a -1, o input não possui operadores lógicos. Se possuir um operador, este é substituído por NULL na lista de argumentos e um ponteiro “backup” aponta para os argumentos após o operador, em seguida o comando é executado, conforme o valor de ops[0], utilizando a função execvp() que recebe um nome de arquivo ou caminho de um novo arquivo de programa, como primeiro parâmetro e um array de argumentos do programa, como segundo parâmetro. A função execvp executa até um NULL e ao finalizar, automaticamente ele encerra o processo atual.

Quando ops[0] for igual a 0, “identificaOperador” detectou o operador lógico “||” (OR), o shell então tenta executar o primeiro comando antes do operador lógico, caso ele falhe, retorna ao loop do-while para executar o comando após “||”, senão ele executa o primeiro comando e encerra o loop do-while.

Quando ops[0] for igual a 1, “identificaOperador” detectou o operador lógico “&&” (AND), neste caso o shell executa o primeiro comando e se houver sucesso, ele também executa o segundo comando após o operador, caso contrário, falhe em executar o primeiro comando, o shell sai do loop do-while.

Se ops[0] for -1, ou seja, não há operadores lógicos, o processo filho executa o comando, por meio da função execvp(), que após concluir o comando, automaticamente termina o processo, enquanto o processo pai aguarda a conclusão do processo filho. Quando o filho termina e o pai terminou de esperar, ao invés de retornar ao loop, o pai sai do do-while e retorna ao looping principal do shell, onde ele aguarda outro input.

Para que seja mantido o loop do-while, a lista de argumentos do comando é atualizada, assim ela executa os comandos após o operador, fazendo o ponteiro de comando apontar para backup que está após o operador ainda agora processado, assim fazendo a detecção de novos operadores.

As funções exibidas e desenvolvidas neste trabalho foram realizadas pelos respectivos estudantes abaixo:

- Laço principal / PrintDir = Arthur ;
- Identifica Operado / Uso de Operadores / Tratamento do vetor = Evandro;
- Operações em Background / FazCD = Lucas;
- Demonstrações / Processamento de Pipe = Thiago.