Relatório Trabalho 1

Implementação de um shell
Sistemas Operacionais

Alunos
Mariana Furtado dos Santos - RA 150209
Mateus Vespasiano de Castro - RA 159505
Vicente Paulo Perpetuo Santos - RA 140278

São José dos Campos
2024

Sumário

1 Introdução

Este relatório apresenta a implementação de um interpretador de comandos, também conhecido como shell, como parte das atividades desenvolvidas na disciplina de Sistemas Operacionais, ministrada pela Profa. Thaína Aparecida Azevedo Tost. O objetivo deste trabalho é atender a uma série de requisitos específicos, proporcionando aos alunos uma compreensão prática dos conceitos fundamentais de sistemas operacionais.

2 Desenvolvimento

2.1 Main

A implementação do shell inicia com a função main em um loop infinito, garantindo que o programa esteja sempre pronto para receber comandos do usuário. Dentro desse loop, é alocado espaço para armazenar o comando fornecido pelo usuário e, em seguida, é chamada a função read_command. Esta função verifica se o comando digitado não é "exit". Caso seja, o programa é encerrado e uma mensagem de encerramento é exibida na tela.

Na função read_command, o array argv, que armazena os argumentos do comando, é inicializado com NULL. Em seguida, a entrada do teclado é lida e armazenada na variável command, que é finalizada com o caractere nulo.

Após isso, a função find_operand é chamada para verificar se há operadores no comando ou se trata-se de um comando unitário. Se a função find_operand retornar qualquer valor diferente de -1, indica que há operadores no comando. Nesse caso, a execução é redirecionada para a função init_operands, que chama a função específica do operador presente no comando.

Por outro lado, se a função find_operand retornar -1, significa que não foram encontrados operadores no comando. Nesse caso, é chamada a função exec_command para executar o comando unitário.

2.2 Comandos unitários, com múltiplos parâmetros.

Para a execução de comandos unitários foi criado a função exec_command(). Primeiramente, a função recebe dois parâmetros, uma string command, que representa o comando a ser executado, e uma matriz de strings argv, que armazena os argumentos do comando.

Em seguida, a string command é dividida em tokens usando a função strtok(), onde o delimitador é o espaço em branco. Cada token obtido é então armazenado na matriz argv, e o número total de argumentos é contado para determinar o tamanho da matriz.

Após a preparação dos argumentos do comando, a função cria um novo processo usando a função fork(). Se o valor retornado por fork() for menor que 0, indica um erro na criação do processo, e uma mensagem de erro é exibida. Caso contrário, se o valor for diferente de 0, significa que o código está sendo executado no processo pai. Nesse caso, o processo pai espera pelo término do processo filho usando waitpid.

Se o valor retornado por fork() for 0, isso indica que o código está sendo executado no processo filho. No processo filho, a função execvp() é usada para substituir o programa atual pelo comando fornecido pelo usuário, juntamente

com seus argumentos. Abaixo está a parte do código que é responsável por essa parte.

```
// Cria um processo filho
pid =fork();
if (pid < 0) { //erro no fork
        perror("fork");
        exit(1);
}
if (pid != 0){
    // Código do processo pai, suspenso esperando o filho terminar
    waitpid(pid, &status, 0);
} else {
        /* Código do processo filho */
        execvp(argv[0], argv);
        perror("execvp");
        exit(1);
}
return status;
```

Por fim, se ocorrer algum erro durante a execução do comando, uma mensagem de erro é exibida, e o processo filho é encerrado. Assim, após a execução do comando, o status de saída do processo filho é retornado para o chamador da função.

2.3 Comandos encadeados, utilizando o operador pipe (|) para combinar saída e entrada entre n comandos.

Ao ser direcionado à função pipe_, o código passa por várias etapas. Primeiro, a linha de comando (command) fornecida pelo usuário é dividida em tokens, cada um representando um comando separado pelo caractere pipe (|). Esses tokens são armazenados em um array de ponteiros (commands) para posterior uso. Durante essa divisão, também é contado o número de tokens na linha de comando (num_commands). Abaixo está a parte do código que é responsável por essa parte.

```
char *commands[100];
int num_commands = 0;
char *token = strtok(command, "|");

while (token != NULL) {
    commands[num_commands++] = token;
    token = strtok(NULL, "|");
}
```

Após o término do loop while, onde num_command armazena o número de comandos na linha de entrada, é criada uma matriz bidimensional de inteiros chamada pipes, que servirá como uma matriz de pipes. A matriz tem um número de linhas igual ao número de comandos menos um, pois o último comando não requer um pipe. Cada linha da matriz contém dois elementos, representando os

descritores de leitura e escrita para cada pipe. Em seguida, a função pipe() do C é usada para inicializar cada pipe na matriz.

```
int pipes[num_commands - 1][2];

for (int i = 0; i < num_commands - 1; i++) {
   if (pipe(pipes[i]) == -1) {
     perror("pipe");
     exit(EXIT_FAILURE);
   }
}
```

Em seguida, iteramos sobre cada comando na sequência e criamos um processo filho para cada um usando fork(). No processo filho, configura os redirecionamentos de entrada e saída para que a saída de um comando seja enviada como entrada para o próximo. Isso é feito usando dup2 para redirecionar os descritores de arquivo dos pipes adequadamente.

O código dessa implementação pode ser visto abaixo:

```
if (pid == 0) { // Código do filho
  // Configurar entrada e saída
  if (i == 0) // Se for o primeiro comando
      dup2(pipes[i][1], STDOUT_FILENO); // Redirecionar saída para o pipe
  else if (i == num_commands - 1) // Se for o último comando
      dup2(pipes[i - 1][0], STDIN_FILENO); // Redirecionar entrada para o pipe
  else {
      dup2(pipes[i][1], STDOUT_FILENO); // Redirecionar saída para o pipe
      dup2(pipes[i - 1][0], STDIN_FILENO); // Redirecionar entrada para o pipe
                                           //anterior
  }
```

No código acima, no primeiro comando, o redirecionamento é feito apenas para a saída padrão (STDOUT_FILENO). O resultado do primeiro comando será enviado para a entrada do próximo comando, pois o primeiro comando não tem entrada anterior para se conectar.

No caso do último comando, o redirecionamento é feito apenas para a entrada padrão (STDIN_FILENO). Aqui, a entrada do último comando é conectada ao último pipe criado anteriormente, para receber a saída do penúltimo comando.

Para os comandos intermediários, tanto a entrada quanto a saída são redirecionadas. O redirecionamento da saída é para o pipe associado a este comando, enquanto o redirecionamento da entrada é do pipe anterior, conectando-se à entrada padrão. Isso garante que a saída do comando anterior seja a entrada deste comando e que sua saída seja direcionada para o próximo comando na cadeia.

Ainda dentro do processo filho, os argumentos do comando são processados, incluindo a manipulação de aspas duplas, se presente. No caso de aspas, elas são removidas e verifica-se se as aspas estão no mesmo token. Se sim, o token é

adicionado aos argumentos. Se não, a flag in_quotes é ativada para sinalizar que estamos dentro de aspas e o token é adicionado ao array de argumentos.

Se estamos dentro de aspas, os tokens são concatenados com o token anterior. Caso contrário, os tokens são adicionados ao array de argumentos. E finalmente, o comando é executado usando execvp. A implementação disso segue abaixo:

```
while (token != NULL) {
  // Verifica se o token começa com aspas duplas
  if (token[0] == '"') {
      // Remove as aspas duplas do início
      memmove(token, token + 1, strlen(token));

      // Verifica se as aspas duplas estão no mesmo token
      if (token[strlen(token) - 1] == '"') {
          token[strlen(token) - 1] = '\0'; // Remove as aspas duplas do final

          command_args[arg_index++] = token; // Adiciona o token sem as aspas ao
      } else {
          // Se as aspas duplas não estiverem no mesmo token, marcamos que estamo
          in_quotes = 1;
          command_args[arg_index++] = token;

          token = strtok(NULL, " ");
      }
  } else if (in_quotes) {
      // Se estamos dentro de aspas, concatenamos o token com o anterior
      printf("token em quote: %s\n", token);
      strcat(command_args[arg_index - 1], " "); // Adiciona um espaço
      strcat(command_args[arg_index - 1], token); // Concatena o token
      // Verifica se este token termina com aspas duplas
      if (token[strlen(token) - 1] == '"') {
          in_quotes = 0; // Marcamos que saímos de aspas
      }
  } else {
      // Se não estamos dentro de aspas, basta adicionar o token
      command_args[arg_index++] = token;
  }
  token = strtok(NULL, " ");
}

command_args[arg_index] = NULL;

execvp(command_args[0], command_args);
```

Enquanto isso, o processo pai fecha todos os pipes que não são mais necessários para evitar vazamento de recursos. Após a execução de todos os comandos, o processo pai espera que todos os processos filhos terminem usando wait.

2.4 Comandos condicionados com operadores OR (||) e AND (&&).

No terminal Linux, os operadores || e && são utilizados para gerenciar o fluxo de execução dos comandos. O operador || (OU lógico) executa o comando à direita somente se o comando à esquerda falhar. Por outro lado, o operador && (E lógico) executa o comando à direita apenas se o comando à esquerda for bem-sucedido.

Na implementação do interpretador de comando, ao encontrar os operadores || ou &&, a execução é redirecionada para as funções or_() e and_(), respectivamente. Ambas as funções recebem como parâmetros uma string command, que representa o comando a ser executado, e uma matriz de strings argv, que armazena os argumentos do comando.

A primeira ação realizada por ambas as funções é separar os dois comandos usando a função strtok(). Para extrair o primeiro comando, utilizamos o delimitador apropriado, sendo |"para or_() e "&&"para and_(). Já para o segundo comando, o delimitador "&|"é empregado, garantindo a correta extração independentemente do próximo operador ou até mesmo se não houver outro.

```
// Divide o comando em tokens usando "&&"
char *token1 = strtok(command, "&&"); // é utilizado "||" na função or_()
char *token2 = strtok(NULL, "&|");
```

Após a separação dos comandos, uma variável chamada "resultado"é criada para armazenar o resultado da primeira operação. Na função and_(), o primeiro comando é executado usando exec_command(). Se executado com sucesso, o segundo comando é executado. Caso contrário, "false"é atribuído à variável "resultado". Se o segundo comando for bem-sucedido, "true"é atribuído a "resultado"; caso contrário, "false"é atribuído.

```
// Declara uma variável para armazenar o resultado dos comandos
char *resultado;

// Verifica se o primeiro comando é bem-sucedido
if (exec_command(token1, argv) == 0) {
    // Verifica se o segundo comando é bem-sucedido
    if (exec_command(token2, argv) == 0) {
        resultado = "true ";
    } else {
        resultado = "false ";
    }
} else {
    // Se o primeiro comando falhou
    resultado = "false ";
}
```

Na função or_(), se o primeiro comando falhar, o segundo é executado. Se o primeiro for bem-sucedido, "true"é atribuído a "resultado"; caso contrário, "false"é atribuído. Essa variável "resultado"é crucial para casos em que o comando original contenha múltiplos operadores condicionais || ou &&, pois determina o resultado geral dos primeiros dois comandos, impactando as ações subsequentes.

```
// Declara uma variável para armazenar o resultado dos comandos
char *resultado;

// Verifica se o primeiro comando falha
if (exec_command(token1, argv) != 0) {
    // Verifica se o segundo comando falha
    if (exec_command(token2, argv) != 0){
        resultado = "false ";
    }else{
        resultado = "true ";
    }
}else{
    resultado = "true ";
}
```

O resultado da operação lógica é então combinado com o restante do comando original para formar o novo comando, que é atualizado com o resultado e enviado como parâmetro para init_operands() para análise e execução. Por fim, as memórias alocadas dinamicamente são liberadas após o uso.

```
// Encontra a posição do segundo token na cópia do comando original
char *pos = strstr(copia_command, token2);

// Aloca memória para o novo comando e o resto do comando
char *novoComando = (char *)malloc(strlen(resultado) + strlen(pos + tamanho_token
// Verifica se a alocação de memória foi bem-sucedida
if (novoComando == NULL) {
    printf("Erro ao alocar memória.\n");
    // Libera a memória alocada para a cópia do comando original
    free(copia_command);
    return;
}

// Constrói o novo comando
strcpy(novoComando, resultado);
strcat(novoComando, pos + tamanho_token);

// Atualiza o comando original com o novo comando
command = novoComando;
// Inicializa os operandos do novo comando
init_operands(command, argv);

// Libera a memória alocada para a cópia do comando original e o novo comando
free(copia_command);
free(novoComando);
```

Em suma, as funções or_() e and_() desempenham um papel fundamental no interpretador de comandos, permitindo a execução de operações lógicas de OU e E entre dois comandos no terminal Linux. Ao separar os comandos, executá-los conforme a lógica determinada pelos operadores || e &&, e atualizar o comando original com o resultado da operação, essas funções garantem o correto fluxo de

execução e a interpretação adequada dos comandos condicionais. Assim, contribuem significativamente para a funcionalidade e eficiência do interpretador de comandos.

2.5 Comandos em background, liberando o shell para receber novos comandos do usuário

Ao encontrar o operador '&' no final do comando, a execução é redirecionada para a função execute_background. Esta função inicia declarando as variáveis necessárias, incluindo um array de ponteiros de caracteres para armazenar os parâmetros do comando. Em seguida, remove-se o '&' do final do comando e preenche-se o array de argumentos com os argumentos do comando usando a função strtok para dividir o comando em tokens.

```
void execute_background(char *comando) {
        char *arg[100];
        char *token;
        int num_arg = 0;
       //Remove o '&' do final do comando
        comando[strlen(comando) - 1] = '\0';

      //Divide o comando em argumentos
        token = strtok(comando, " ");
        while (token != NULL) {
            arg[num_arg++] = token;
            token = strtok(NULL, " ");
          }
      arg[num_arg] = NULL;
```

Após o fork, três condições são verificadas. Se o processo for o filho, ele utiliza a função execvp para trocar sua imagem e executar o comando digitado pelo usuário. Se o valor de pid for negativo, indica um erro ao criar o processo filho e uma mensagem de erro é exibida. No caso do processo pai, ele não espera pela finalização do processo filho, permitindo que o shell continue em execução e o filho rode em background.

```
        pid_t pid;
        pid = fork();

        if (pid == 0) {   // Processo filho
            execvp(arg[0], arg);
            perror("execvp");
            exit(EXIT_FAILURE);
        } else if (pid < 0) { // Erro ao criar processo filho
            perror("fork");
        } else {    //processo pai
            waitpid(-1, NULL, WNOHANG);
            printf("Processo %d esta executando em background\n", pid);
        }
        return;
```

```
}
```

No caso do processo pai, o uso do parâmetro WNOHANG garante que ele não fique bloqueado esperando o filho terminar. Assim, o filho não se torna um zumbi, mantendo a conexão entre pai e filho.

3 Conclusão

A implementação deste interpretador de comandos reflete a aplicação prática dos conceitos aprendidos na disciplina de Sistemas Operacionais. Ao atender aos requisitos de execução de comandos unitários, encadeados, condicionais e em background, exploramos aspectos cruciais como chamadas de sistemas e comunicação entre processos. A colaboração entre os membros do grupo foi fundamental para o sucesso do projeto, proporcionando uma experiência prática enriquecedora.