UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

Primeiro Trabalho Prático da disciplina Sistemas Operacionais

Documentação referente ao primeiro trabalho prático da disciplina Sistemas Operacionais 2018/1, desenvolvido pelos alunos Júlio César Mendes de Resende, Felipe Henrique Faria e Gustavo Detomi Rodrigues do curso de Ciência da Computação da UFSJ.

São João Del Rei

Maio/2018

1 Introdução

Este trabalho tem como objetivo descrever a implementação de um interpretador de comandos (chamado no Unix/Linux de shell), que foi realizada na linguagem C com o auxílio de chamadas de sistema como *execvp(), fork()* e *waipid()*. Para os casos em que foi necessário fazer redirecionamento da entrada e/ou saída padrão para arquivos foi utilizado a função *freopen()*, e para os casos em que foi necessário comunicar dois processos, foi implementado um pipe.

2 Entrada

A entrada do programa pode acontecer por texto digitado no terminal, ou por um arquivo de entrada. Caso ao executar o programa o nome do arquivo de entrada seja passado por parâmetro, este (caso exista), será definido como a entrada padrão do programa. Se nenhum arquivo for passado como parâmetro, será solicitado ao usuário que digite um novo comando a cada passo.

Em todos os casos é necessário realizar um tratamento do texto recebido. Tal tratamento consiste em transformar uma linha de comando em um vetor de palavras (matriz de caracteres), considerando a definição de palavra como qualquer conjunto de caractere que não esteja separado por espaço, tabulação ou quebra de linha. Nesse processo, a única função pronta da linguagem C utilizada foi o *fgets()*, utilizado para recuperar uma linha de comando. O restante do processo foi implementado em uma função que recebeu o nome de *parse()*.

Esse processo de manipulação dos caracteres é útil para a execução da função *execvp()* posteriormente, e também facilita a identificação de alguns comandos como <=, =>, |, e &. Outra vantagem é contar a quantidade de "palavras"contidas no texto, o que permite identificar se o usuário digitou uma linha em branco, por exemplo.

2.1 Formato da entrada

Após a formatação do texto recebido é hora de realizar o processamento do vetor identificando os comandos especiais. Tais comandos são:

- => - Esse comando indica redirecionamento da saída padrão. Sempre que identificado, a saída padrão dos comandos a esquerda deverá ser o arquivo a direita. Isso faz com que o algoritmo verifique sempre se existe um comando a esquerda e um nome de arquivo

a direita. Caso o arquivo não exista um novo será escrito, e caso exista, o mesmo será sobrescrito.

- <= - Esse comando indica redirecionamento da entrada padrão. Sempre que identificado, a entrada padrão dos comandos a esquerda deverá ser o arquivo a direita. Nesse caso, se o nome do arquivo a direita não representar um arquivo válido, será impossível realizar a execução do comando, e então uma mensagem de erro será exibida.

- | - Esse comando indica que a saída do programa a esquerda, deve ser associada a entrada do programa a direita, sendo necessário a comunicação de ambos por um pipe.

- & - Por último, esse comando indica execução em background. O mesmo só pode aparecer ao final de uma linha de comando.

Dessa forma, são exemplos de comandos válidos:

ls -l
ls -laR => arquivo
wc -l <= arquivo &
ls | wc
cat -n <= arquivo => arquivonumerado
cat -n <= arquivo | wc -l => numerodelinhas
cat -n <= arquivo | wc -l => numerodelinhas &

E são exemplos de comandos não válidos:

ls -laR => arquivo | wc
ls -l | wc <= arquivo
=> comando

3 Criação de processos

Seguindo o modelo dos interpretadores de comando dos sistemas Unix/Linux, a cada novo programa a ser executado, é executado a chamada de função *fork()*, que cria um processo filho que é um clone do pai, e logo após, no processo filho, é executado *execvp()*, que procura o arquivo binário dos programas no sistema de arquivo do computador, os executa, e logo depois encerra o processo. Enquanto o filho executa o programa, o processo pai fica adormecido através da chamada de sistema *waitpid()*. Esse modelo de criação de processo é útil para proteger o interpretador de todos os erros que podem acontecer no novo programa.

O código abaixo indica como esse processo é realizado na linguagem C, para um caso simples, em que não existe redirecionamento da entrada e saída padrão, nem execução em background.

```
1  void fork_exec(char **argv){
2    pid_t pid;
3    int status;
4    switch(pid = fork()){
5      case -1:
6        printf("Fork falhou!\n");
7        exit(1);
8
9      case 0:   // processo filho
10       execvp(*argv, argv);
11       perror("Error ");
12       exit(1);
13
14     default:    //processo pai
15       while (wait(&status) != pid); // espera pelo filho
16   }
17 }
```

3.1 Execução em background

Como dito anteriormente, pode ser solicitado que o programa execute em background. Isso é possível removendo a linha 15 do código anterior, e preparando uma função para receber o sinal SIGCHLD emitido pelo processo filho quando o mesmo termina através da função *signal()* da biblioteca *signal.h*.

Esse modo de execução se não for bem utilizado pode gerar uma má aparência do interpretador de comandos. Por exemplo, digamos que o processo principal escreva na tela a mensagem: "Sim, mestre?"e fique interrompido aguardando o usuário digitar um texto. Se nesse momento o processo que está executando em background termina e exibe uma mensagem na tela, o processo principal vai continuar aguardando o usuário digitar um texto, mas o mesmo pode se sentir confuso, uma vez que a mensagem "Sim, mestre?"não está novamente na última linha de texto. Esse caso ocorre tanto no shell implementado quanto no shell original do linux, e uma solução é sempre redirecionar a saída padrão de processos em background para arquivos.

4 Redirecionamentos de entrada/saída padrão

O redirecionamento de entrada/saída padrão é necessário nos casos em que a entrada e/ou saída de um processo está conectada a um arquivo, ou ainda quando precisamos fazer a comunicação entre processos, ligando a saída de um com a entrada de outro.

Foram utilizadas as funções *close()* e *dup()* incluídas pela biblioteca *unistd.h* e a função *freopen()*. Utilizando a função *close()* desligamos a saída padrão do sistema e posteriormente utilizando a função *dup()* configuramos uma nova saída passando como parâmetro um ponteiro para um arquivo ou alguma posição de memória. A função *freopen()* nos permite configurar a saída ou entrada para um arquivo no momento que em que instanciamos sua abertura na linha de código.

4.1 Comunicação entre processos

A comunicação entre processos é necessária quando a entrada de um determinado processo depende da saída de outro. Neste caso é preciso criar um *pipe()* que faz a ponte de comunicação entre estes processos.

O pipe consiste de um vetor de duas posições onde a posição 1 indica a entrada e a posição 2 a saída. Alterando a entrada e saída padrões do sistema para as respectivas posições do pipe criamos um meio de comunicação entre os processos. Além disso o segundo processo precisa esperar que o primeiro termine e isto é feito utilizando as funções *wait()* e *waitpid()* vistas anteriormente.

O primeiro processo fecha sua saída padrão e a direciona para a posição 2 do pipe, o segundo processo espera o fim do primeiro e então troca sua entrada padrão pela posição 1 do pipe. Uma simples implementação onde é feita a comunicação entre dois processos pode ser vista no código abaixo. Neste código a saída do processo 1 é conectada a entrada do processo 2 utilizando o pipe. Mais descrições sobre o procedimento podem ser observadas nos comentários do código.

```
1 pid_t pid;
2 int pf[2]; // Pipe
3 pipe(pf); // Cria o pipe
4 switch(pid = fork()){ // Processo 1
5     case -1: exit(1); // Falha no processo 1
6     case 0: close(1); // Fecha a escrita padrao
7         dup(pf[1]); // Substitui pela escrita no pipe
8         close(pf[0]); // Fecha a leitura do pipe
```

```
9           execvp(comando1[0], comando1);
10  }
11  switch(pid = fork()){ // Processo 2
12      case -1: exit(1); // Falha no processo 2
13      case 0: close(0); // Fecha a leitura padrao
14          dup(pf[0]); // Substitui pela leitura no pipe
15          close(pf[1]); // Fecha a escrita do pipe
16          execvp(comando2[0], comando2);
17  }
```

5 Considerações finais

Neste trabalho prático, através da implementação de um shell, foi possível entender melhor como os sistemas operacionais criam e executam seus processos, como processos filhos são criados, e como os processos podem comunicar entre si. Foi possível também explorar recursos da linguagem C para criar chamadas de sistema.