PROPOSTA DE CRIAÇÃO DE UM SHELL PARA SISTEMAS UNIX

Almir SOUSA(1), Cláudio MONTEIRO(2), Francisco CHAGAS(3), Francisco HIRANO(4)
SOLTO – Grupo de Pesquisa e Desenvolvimento de Software Livre do Tocantins
Escola Técnica Federal de Palmas – ETF-Palmas
AE 310 SUL, Avenida NS 10,Centro, 77.021-090, Palmas-TO
(1)E-mail: almir@etfto.gov.br
(2)E-mail: ccm@etfto.gov.br
(3)E-mail: francisco@etfto.gov.br
(4)E-mail: willians@etfto.gov.br

RESUMO

O Unix por ser um grande originador de sistemas operacionais livres, tais como FreeBSD e Linux, tem estruturas bem padronizadas e livres para acesso de todos. Programar para Unix exige do programador um conhecimento aprofundado da organização deste, e, no caso de sistemas livres, como o FreeBSD e o Linux, sua documentação favorece este aprofundamento. Além disso, o uso de uma linguagem de programação adequada faz-se imprescindível. Assim, o uso do gcc, disponível para FreeBSD e Linux, num padrão ANSI, facilita a tarefa do desenvolvimento de um projeto de criação de um Shell para sistemas UNIX. Porém, a implementação de tal projeto, com objetivo didático, exige do programador uma experiência que vai além do ensinado em cursos tecnológicos na área de Informática pois, demanda a confecção de vários elementos básicos. Destes elementos destacamos: um editor de linha dinâmica num console com comportamento padronizado, tipo VT100, um interpretador de comandos que no mínimo consiga separar comandos internos de externos e, um executor de comandos capaz de lidar com pipes e redirecionamentos. Neste trabalho, apresentamos um Shell, construído no FreeBSD, satisfazendo esses requisitos mínimos.

Palavras-chave: Sistema Operacional; Shell; Editor de linha; Interpretador de comandos; Executor de comandos.

1. INTRODUÇÃO

O shell de comandos para sistemas Unix é um software independente que oferece comunicação direta entre o usuário e o sistema operacional. A interface de usuário não gráfica do shell de comandos é o ambiente propício para a execução de aplicativos e utilitários baseados em caracteres.

A criação deste software exige um conhecimento do sistema operacional utilizado. Em nosso caso, usamos o sistema operacional FreeBSD, versão 6.2, que é um sistema operacional livre do tipo Unix, descendente do BSD desenvolvido pela Universidade de Berkeley. Além disso, tivemos que aplicar conhecimentos avançados da linguagem C, no padrão ANSI, para superar dificuldades técnicas inerentes ao desenvolvimento de software básico. Finalmente, técnicas específicas foram usadas para a implementação do Editor de Linhas e do Interpretador de Comandos, sendo que algumas não disponíveis para o domínio público.

A criação deste protótipo de shell de comandos tem finalidade didática, além de tornar disponível no site da ETF-Palmas o programa fonte para nossos alunos e interessados. Nosso site é <www.cefet-to.org>.

Este trabalho é fruto do esforço de consolidação do SOLTO – Grupo de Pesquisa e Desenvolvimento de Software Livre do Tocantins, sediado na Escola Técnica Federal de Palmas, que pretende, dentre outros objetivos, a produção de software livre para todos.

O trabalho é apresentado em duas seções básicas – a da fundamentação teórica e da implementação do software.

2. FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta a fundamentação teórica para a construção do Shell.

2.1. Sistema Operacional e o Shell

O Sistema Operacional é um programa ou um conjunto de programas cuja função é servir de interface entre um computador e o usuário. É comum utilizar-se a abreviatura SO (em português) ou OS (do inglês "Operating System").

Segundo TANEMBAUM (2000), existem dois modos distintos de conceituar um sistema operacional: (i) pela perspectiva do usuário (visão "top-down"), é uma abstração do hardware, fazendo o papel de intermediário entre o aplicativo (programa) e os componentes físicos do computador (hardware); (ii) numa visão "bottom-up", de baixo para cima, é um gerenciador de recursos, i.e., controla quais aplicações (processos) podem ser executadas, quando e, que recursos (memória, disco, periféricos) podem ser utilizados.

Já para um usuário acessando um terminal textual num sistema operacional tipo Unix, o shell faz as vezes de um sistema operacional, pois é o intermediário desse usuário com os recursos controlados pelos processos disparados pelo próprio shell. Esse usuário não vê nem o Núcleo nem muito menos o Gerenciador de Processos, mas apenas o Shell que recebe comandos seus, interpreta-os, aciona programas e parâmetros adequados, disponibiliza seu ambiente para esses, redireciona entradas e saídas, quando necessário, e recebe resultados dos processos disparados.

Segundo TANENBAUM (2000), o shell, além de ser um interpretador de comandos "é a interface primária entre o usuário à frente do terminal e o sistema operacional".

Também para BORRO (2007) o shell do Sistema Operacional pode ser conceituado como "Shell, ou interpretador de comandos, é o programa disparado logo após o login responsável por "pegar" os comandos do usuário, interpretá-los e executar uma determinada ação."

O Shell não faz parte do Sistema Operacional, tomado, por nós, como o Núcleo, mas faz uso de várias chamadas de sistema (system calls) para desempenhar a contento suas funções. Além disso, ele se utiliza de arquivos de fluxos padrões ligados a processos controladores de terminais, para fazer a interação com o usuário. Tais processos são fundamentais para o funcionamento adequado do Shell, de forma que são reinicializados imediatamente após sua morte (respawn).

O Shell, embora não pertencente ao Sistema Operacional, é tão atrelado a este, que seu nome é plenamente justificado – casca do sistema. Num sistema monolítico como o FreeBSD, o Núcleo funciona como o Sistema

Operacional e contém além das funções que desempenham o papel de chamadas de sistemas, serviços de interrupção que permitem o gerenciamento de processos e dispositivos atrelados aos Sistemas de Arquivos (file systems). O Shell faz uso de algumas dessas funções para desempenhar adequadamente seu trabalho. Não fossem as chamadas de gerenciamento de processos disponíveis no Núcleo, o Shell teria que ter também o papel de gerenciador de processos. Abaixo, apresentamos na figura 1 as idéias até aqui apresentadas.

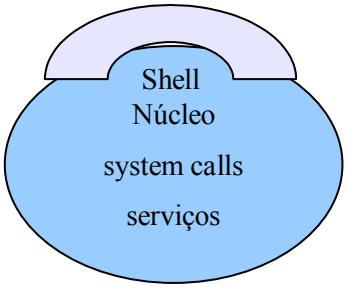

Figura 1: Núcleo e Shell.

2.2. Elementos Básicos do Shell

O programa que implementa o nosso shell tem três elementos básicos: o Editor de linhas dinâmicas, o Interpretador de comandos e o Executor de comandos, simplesmente chamados de Editor, Interpretador e Executor. Eles são acionados em sequência, como mostrado na Figura 2 abaixo:

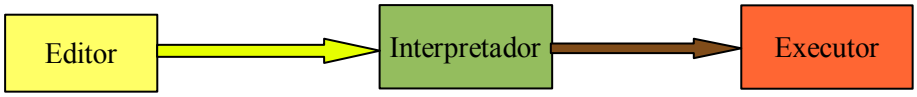

Figura 2: Elementos do Shell.

O Editor capta os comandos digitados pelo usuário e os passa para o interpretador. Nesta fase de desenvolvimento, apenas comandos normalmente usados fora de um script são analisados, i.e., não temos uma linguagem completa sendo interpretada, embora variáveis sejam manipuladas, memória de comandos executados, redirecionamentos e pipes sejam permitidos, daemons sejam reconhecidos e, comandos internos (builtin's) sejam isolados para execução pelo próprio Shell. Contudo, expressões regulares não são interpretadas.

O Interpretador faz o parsing dos comandos, transformando-os em vetores de strings apropriados para as chamadas do tipo exec. Como a sintaxe de um comando shell é muito simples, tipo uma ação seguida de parâmetros, a única dificuldade aqui é na detecção de redirecionamentos (< , >, >>) e pipes (|).

Já o Executor simplesmente dispara os processos correspondentes, executando a sequência fork-exec.

Outra característica importante de nosso Shell é que ele não interage com o Shell do Terminal através da chamada system(...). Todos os seus módulos implementam suas funções usando, quando necessárias, chamadas diretas ao sistema.

2.3. Processos

Para se entender melhor o funcionamento do Shell é preciso conhecer o conceito de processo, conceito este chave em estudos de sistemas operacionais. Um processo é um programa em execução. No FreeBSD e no Linux os processos passam a existir e serem manipulados após o carregamento do Núcleo, responsável por disparar o primeiro processo – o init. Após o init, qualquer processo tem que ser criado após um procedimento de clonagem de outro processo, chamado de forking (fork()), ou de substituição de um processo por outro, chamado aqui de exec (execl(), execle(), execlp(), execv(), execve() ou execvp()).

No nosso Shell, para permitir redirecionamentos e pipes, o executor depois de clonar o shell e antes de fazer a substituição por outro programa, toma o cuidado de reinicializar os fluxos padrões stdin, stdout e stderr.

Um processo tem dois identificadores importantes: pid (identificador do processo) e o ppid(identificador do pai do processo). O processo que cria outro é chamado de pai desse, enquanto que o processo criado é chamado de filho daquele.

3. A IMPLEMENTAÇÃO DO SHELL

A implementação do Shell é feita usando o compilador gcc, disponível no FreeBSD e no Linux. Na essência, temos três módulos: editor.c, parser.c e executor.c. Dos três o Editor é o mais técnico, enquanto que o parser é o mais conceitual, e o executor o mais imediato de se implementar.

Para se entender a implementação, é necessário compreender como um novo processo é criado num sistema tipo unix como o FreeBSD.

Primeiramente, um processo é clonado através do uso da primitiva fork(). Com essa primitiva, um processo se reproduz em um novo espaço, repassando todas as suas variáveis ao novo processo. O processo criado recebe um novo pid, enquanto seu pai é o processo reprodutor. Para que ambos saibam quem é quem, após o forking, a chamada fork() retorna o pid do filho para o pai, enquanto o filho recebe o valor zero.

A sequência em C, pode ser a seguinte:

```
{
int sonPid;            .
.
.                                    .
if((sonPid=fork())==0)
  {
    // aqui fica o filho
   // um novo programa pode ser acionado através da chamada execvp, por exemplo
   }
 else
  {
    // aqui segue o pai
  }
.
.
}
```

Código 1: Clonagem de um processo usando a função fork().

Depois, uma primitiva do tipo exec é chamada para sobrescrever o processo filho com outro programa.

Entremos agora nos detalhes de implementação de cada elemento do Shell.

3.1. O Editor de Linhas do Shell

Embora um editor de linhas pudesse ser facilmente implementado usando a chamada fgets (veja GPS (2007)), algumas exigências técnicas do tipo: recall de comandos já digitados, inserção de caracteres no meio do comando, e controle mais apurado de caracteres especiais, demandaram um esforço extra para uma implementação mais sofisticada. Além disso, a implementação feita pode ser usada para se conseguir um editor de programas simples, como o ee, nano ou pico.

Primeiramente, é necessário que a teclas digitadas sejam prontamente reconhecidas pelo editor, sem ecos indesejáveis no terminal. Usa-se para isso a função stty. Se pudéssemos usar um comando do Shell do sistema, seria uma tarefa fácil fazer isto em C, e.g.:

```
system("stty -echo raw");
```

Código 2: Colocando o terminal em modo de não espera e sem eco.

Para se retornar para o modo normal do Shell do sistema usaríamos:

```
system("stty echo -raw");
```

Código 3: Retomando o terminal para o modo de espera com eco.

Porém, não queremos usar de forma nenhuma este tipo de chamada para se ter um shell independente do Shell do sistema.

Assim, para usarmos qualquer rotina fora do núcleo, tivemos que nos valer da sequência fork-exec.

Um código alternativo poderia ser:

```
void chRawMode()
  {
   char *par[]={"stty","raw","-echo",NULL};
   int sonPid;
   int result;

   if(!(sonPid=fork()))
    {
     execvp("stty",par);
    }
   else
    waitpid(sonPid,&result,0);
  }
```

Código 4: Colocando o terminal em modo de não espera e sem eco sem o uso de system.

Observe no código acima a chamada "waitpid(sonPid, &result,0);" para esperar o processo filho terminar antes de continuar.

É incrível como a função chRawMode() acima abre as portas para a decodificação do teclado, inclusive com reconhecimento de teclas especiais.

Logo, a implementação de uma função chamada de "unsigned char pegaTecla()" foi possível, facilitando a construção de nosso editor. Foi também usado o processamento de sequências de escape, características de terminais do tipo VT100, xterm, etc. Uma sequência de escape é simplesmente um vetor de caracteres, começando com o caracter escape ('\27'), permitindo o reposicionamento do cursor, inclusive com limpeza do terminal. Uma destas sequências permite a implementação da função abaixo:

```
void gotoxy(int x,int y)
 {
  while(x > lineSize)
   {
    x -= lineSize;
    y++;
   }
  printf("\033[%d;%dH",y,x);
  curCol=x;
  curLine=y;
 }
```

Código 5: Função de posicionamento do cursor, usando sequências de escape.

Onde as variáveis lineSize, curCol e curLine controlam o tamanho da linha do terminal, a posição vertical e a horizontal do cursor, respectivamente. Depois disto, a implementação do editor reduz-se a um mero controle de movimentação do cursor. Constroe-se assim a função "int editString(int x, int y, char *s, int tam)". O par (x,y) tem que ser bem controlado para um visual agradável do shell.

Para se obter um visual agradável do Shell, a nível de comando, é muito importante obter a posição atual do cursor. Uma função chamada whereXY() pode ser construída usando "reports" de terminais padrões através da sequência de escape "ESCAPE [6 n". Infelizmente, não conseguimos colocar o terminal num modo padrão ANSI para a sequência funcionar. A sequência solicita do terminal sua posição atual, que é recebida através

de outra sequência de escape “ESCAPE [Linha; ColunaR”. Caso o terminal não tenha a habilidade de responder. a rotina fica presa, em códigos sem controle de “time-out”, como o nosso. De qualquer forma veja o Código 6 a seguir:

```
void whereXY (int *x, int *y)
 { char ln[10], col[10];
   int pos;
   printf("\033[6n");
   if(getchar()== '\033')
   {
    if(getchar()=='[')
    {
     ln[0]=0;
     pos=0;
     while((ln[pos]=getchar())!=';')
      pos++;
     ln[pos]=0;
     pos = 0;
     while((col[pos]=getchar())!='R')
      pos++;
     col[pos]=0;
     *y = atoi(ln);
     *x = atoi(col);
     curLine = *y;
     curCol = *x;
    return;
    }
   }
 printf("\nWhereXY com erro");
}
```

Código 6: Função de posicionamento do cursor, usando sequências de escape.

As funções básicas desenvolvidas para a implementação do Editor estão listadas abaixo na Tabela 1.

Tabela 1: Funções Básicas do Editor

Nome da rotina e tipo de retorno	Tipos de parâmetros	Função
void gotoxy	(int x,int y)	Posicionar o cursor
void whereXY	(int *x, int *y)	Pegar posição do cursor
void putcar	(unsigned char car)	Imprimir caracter
int pegueTecla()	Não tem	Pegar uma tecla digitada
void clrString	(int x, int y, char *s, int tam)	Imprime e limpa string e também a área de tela correspondente
int editString	(int x, int y, char *s, int tam)	Edita um string posicionada
int getCmd	(char *s, int tam)	Edita uma linha de comando

3.2. O Interpretador de Comandos do Shell

Agora vamos discutir a implementação do parser de nosso Shell.

Por não ser uma implementação completa da linguagem Shell, a nossa construção necessita conceitualmente apenas de dois elementos básicos: Um analisador léxico capaz de separar os átomos de uma linha de comandos, e um analisador sintático capaz de detectar redirecionamentos e de separar comandos múltiplos em pipes ou separados por ';'. Além disso, o analisador sintático deve reconhecer comandos internos (builtin's) para facilitar o papel do Executor.

As funções de classificação do Analisador Léxico são implementadas em se reconhecendo os separadores de átomos de comandos: espaço em branco, ';', '>', '<', “>>”, '| ', '&', e '-'. Também , aspas (”") e '$' são importantes para manipulação de variáveis e para implementação da função interna “echo”. Isto é implementado através de um autômato finito simples, como normalmente é sugerido em implementações de analisadores léxicos.

O Analisador Sintático solicita ao Léxico os elementos classificados para fazer sua análise. O primeiro elemento determina o comando; os outros são possíveis parâmetros. Assim, é importante determinar o início e o fim da linha do comando. Não tivéssemos pipes e comandos múltiplos numa linha, a tarefa seria fácil ao determinar o final de linha (CR). Porém implementamos pipes e comandos múltiplos. Para isto a definição clara de separadores de comandos tem que ser feita para o sucesso do parser.

O parser além da análise sintática faz uma conversão do comando para uma lista de strings que será usada na execução do comando na sequência fork-exec.

O parser é capaz também de detectar as teclas para cima e para baixo para poder manusear comandos já digitados, usando a Estrutura 1 abaixo para guardar a lista duplamente encadeada dos comandos.

```
struct cmdStack {
  struct cmdStack *previous;
  struct cmdStack *next;
  char *cmdStr;
 };
```

Estrutura 1: Lista de comandos digitados.

As funções básicas desenvolvidas para a implementação do Interpretador estão listadas abaixo na Tabela 2.

Tabela 2: Funções Básicas do Interpretador

Nome da rotina e tipo de retorno	Tipos de parâmetros	Função
int pegueComando	(char *origem, char *buffer, int *pos)	Pegue um comando inicial ou parâmetro
int takeAspasString	(char *origem, char *buffer, int *pos)	Retira um string entre aspas
Int takeApostrofeString	(char *origem, char *buffer, int *pos)	Retira um string entre apóstrofes
int takeHifenString	(char *origem, char *buffer, int *pos)	Retira um string após um hífen -
Int takeGreaterString	(char *origem, char *buffer, int *pos)	Retira um redirecionamento do tipo “>” ou do tipo “>>”
int takeLessString	(char *origem, char *buffer, int *pos)	Retira um redirecionamento do tipo “<” ou do tipo “<<”
int takePipeString	(char *origem, char *buffer, int *pos)	Retira um caracter de pipe “|”
int putArgument	(char **argv,char *buffer, int narq)	Ponha um argumento processado no vetor de parâmetros
int takeDaemonString	(char *origem, char *buffer, int *pos)	Retira um caracter de daemon “&”

Uma vez obtido um comando válido, o vetor de strings processado é passado para o Executor.

3.3. O Executor de Comandos do Shell

Na essência, o Executor é muito simples se o parser fizer seu papel sintático de separar e classificar os comandos digitados. A classificação é feita para saber se há redirecionamentos; se o comando segue um pipe "|" ou vem antes de um; se o comando é um daemon; ou se o comando é interno.

Quando há redirecionamentos, arquivos temporários são abertos para guardar ou receber outputs ou inputs, respectivamente. Se existe um pipe, automaticamente o processo anterior tem que redirecionar seu output e o posterior o seu input. Os redirecionamentos são feitos usando-se as chamadas

```
freopen(inFileName,"r", stdin);
freopen(outFileName,md, stdout);
```

Código 7: Código para implementação de redirecionamentos.

Um comando interno é executado pelo próprio Shell. Os comandos implementados são colocados no vetor de strings "char *builtinStr[NUMBUILTIN] = {"cd","clear","echo","exit","mkdir"}". Esta implementação tem apenas um caracter didático de se ensinar "como fazer".

A execução de um comando externo começa com a clonagem do Shell através de "sonPid=fork();".

Depois, os redirecionamentos são feitos como no Código 7 acima.

E em seguida, o processo filho é trocado pelo primeiro parâmetro do comando assim: "execvp(path,argv);".

Finalmente, se o processo não é um daemon ou está antes de um pipe "|", o pai tem que esperar a execução do filho através da linha de comando:

```
if(!(daemonFlag) || pipeFlag)
 {
  waitpid(sonPid,&result,0);
 }
```

Código 8: Espera de processos filhos pelo pai, quandos eles não são daemon's ou estão antes de pipes.

Os outros detalhes de implementação são meramente técnicos e não se justificam serem mostrados aqui.

4. CONSIDERAÇÕES FINAIS

Acabamos de apresentar os resultados de nossa pesquisa para implementar um Shell relativamente completo, usando idéias simples que podem ser usadas em componentes de Sistemas Operacionais em cursos tecnológicos. Pode-se a partir do fonte apresentado, confeccionar um Shell que permita a execução de comandos if .. fi, while ... done, etc. A interação do Shell com o Núcleo poderia ser mais otimizada, se por exemplo, o carregamento dos processos fossem feitos de uma forma mais direta pelo Shell.

Outra observação importante: O Editor poderia ser mais eficiente se houvesse um controle das impressões feitas pelos processos disparados.

REFERÊNCIAS

AHO, A.V., ULLMAN, J.D. **Compilers: Principles, Techniques and Tools.** Reading: Addison-Wesley, 1986.

BORRO, A. **Curso de Shell.** Disponível em: http://olinux.uol.com.br/artigos/258/1.html, 07 Set 2007.

GSP Services, Inc. **FreeBSD Man Pages.** Disponível em: http://www.gsp.com/support/man/, 08 Set 2007.

OLIVEIRA, R. , CARISSIMI, A. , TOSCANI, S. **Sistemas Operacionais.** Porto Alegre: Ed. Sagra Luzzato, 2001.

TANEMBAUM, A. **Sistemas Operacionais Modernos.** 2. ed. Rio de Janeiro: Prentice Hall do Brasil, 2003.
TANENBAUM, A.S., WOODHULL, A.S. **Sistemas Operacionais: projeto e implementação.** 2. ed. Porto Alegre: Bookman, 2000.

The Freebsd Documentation Project. Freebsd Handbook. Disponível em: http://www.freebsd.org/doc /en_US.ISO8859-1/books/handbook/, 08 Set 2007.

ZILLI, Daniel. **Engenheiro Linux.** Rio de Janeiro: Editora Brasport, 2003.

AGRADECIMENTOS

Agradecemos à Escola Técnica Federal de Palmas por disponibilizar tempo livre para a pesquisa e equipamentos para a implementação; ao professor Cláudio Castro Monteiro por ter nos acompanhado nesta empreitada; e a Deus por ter nos guiado e dado saúde para trabalhar e estudar.