Documentos

ISSN 1677-9274
Dezembro, 2009
99

Shells do Unix: Tutorial

Embrapa

ISSN 1677-9274
Dezembro, 2009

Empresa Brasileira de Pesquisa Agropecuária
Embrapa Informática Agropecuária
Ministério da Agricultura, Pecuária e Abastecimento

Documentos 99

Shells do Unix: Tutorial

Fábio Danilo Vieira

Embrapa Informática Agropecuária
Campinas, SP
2009

Embrapa Informática Agropecuária
Av. André Tosello, 209 - Barão Geraldo
Caixa Postal 6041 - 13083-886 - Campinas, SP
Fone: (19) 3211-5700 - Fax: (19) 3211-5754
www.cnptia.embrapa.br
sac@cnptia.embrapa.br

1ª edição on-line 2009

Dados Internacionais de Catalogação na Publicação (CIP)
Embrapa Informática Agropecuária

Vieira, Fábio Danilo

Shells do Unix : tutorial / Fábio Danilo Vieira. - Campinas : Embrapa Informática Agropecuária, 2009.

69 p. : il. - (Documentos / Embrapa Informática Agropecuária, ISSN 1677-9274 ; 99).

1. Unix. 2. Sistema operacional. 3. Programação. I. Título. II. Série.

CDD: 005.432 (21.ed.)

Autor

Fábio Danilo Vieira
Analista da Embrapa Informática Agropecuária
Caixa Postal 6041, Barão Geraldo
13083-886 - Campinas, SP
Fone: (19) 3211-5798
e-mail: fabiodv@cnptia.embrapa.br

Apresentação

O Unix e suas vertentes (Linux, FreeBSD, OpenBSD, Solaris, etc) sempre foi considerado pouco acessível por muitas pessoas, sejam elas profissionais da área de informática ou não.

Hoje em dia, com a modernização de ambientes gráficos, como KDE e GNOME, a utilização desses diversos “sabores” de sistemas operacionais derivados do Unix se tornaram um pouco mais populares, mesmo que ainda bem abaixo de seu concorrente principal, o MS Windows.

Entretanto, o ponto forte do Unix não está nesses ambientes gráficos mais requintados dos dias de hoje, e sim na sua velha e tradicional linha de comando e no que é possível se fazer através dela. E na interface destas linhas de comando estão diversos *Shells*, que são como espécies de tradutores do que o usuário digita, e o *kernel* do sistema, que é o “cérebro” do sistema operacional Unix e seus derivados.

Este tutorial, portanto, foi escrito com o intuito de desmitificar essa forte capacidade dos sistemas operacionais baseados em Unix e de auxiliar usuários que pouco conhecem este sistema a explorar as inúmeras funcionalidades que os *Shells* oferecem.

Kleber Xavier Sampaio de Souza
Chefe-Geral
Embrapa Informática Agropecuária

Sumário

Shells do Unix: Tutorial

Fábio Danilo Vieira

Capítulo 1

Shell: Interpretador de Comandos

A estrutura do Sistema Unix

Unix é um sistema operacional muito abrangente e complexo, que funciona em uma grande variedade de plataformas de *hardware*, desde PC´s a *Mainframes*. Ele possui muitos comandos com milhares de opções associadas. Entretanto, por trás de todos esses comandos, está uma estrutura que dá suporte para o funcionamento integral e preciso desses comandos, sendo que o *Shell* faz parte dessa estrutura. Consequentemente, para entender melhor o papel do *Shell* dentro do Unix, é antes necessário saber como o sistema operacional Unix está dividido em sua estrutura. Dessa forma, as partes do Unix podem ser, em relação ao seu funcionamento, classificadas em 3 níveis: o ***Kernel***, o ***Shell*** e os **Utilitários do Unix**. Veja a Figura 1, representativa dessa estrutura:

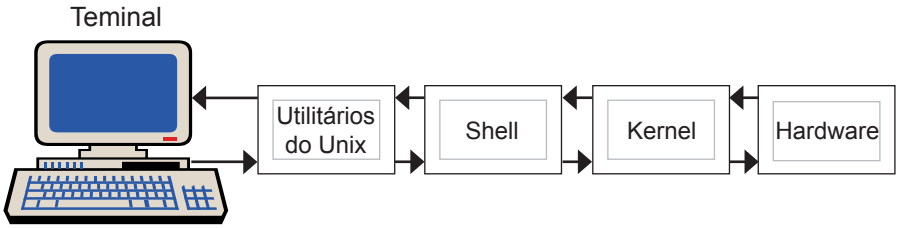


Figura 1. Componentes do sistema Unix.

Em seguida, serão explicadas todas as partes da estrutura do sistema operacional Unix.

O *Kernel*

O *Kernel* organiza as tarefas do sistema, além de administrar o armazenamento e transporte de dados. Ele é o "cérebro" do sistema Unix, controlando o *hardware* e dispositivos do computador quando um comando os solicita. Por exemplo, quando você digita um comando como "**cat** arq1.txt", o *Kernel* irá direcionar o computador para a leitura do arquivo "arq1.txt" no disco e, logo em seguida, exibi-lo na tela.

Todos os sistemas operacionais possuem um *Kernel*, claro que devem ter outros nomes, mas com funções semelhantes. Em seguida, lista-se um resumo das funções do *Kernel*:

- Estabelece o sistema de arquivos Unix, organizando e gerenciando o armazenamento de dados;
- Em relação à segurança, impede que usuários desautorizados acessem informações protegidas;
- Executa pedidos de entrada e saída, isto é, transfere dados de/para dispositivos de I/O (Entrada/Saída);
- Coordena várias funções internas do sistema operacional, como a alocação de memória e de outros recursos do sistema, para processos que estão sendo executados em um determinado momento.
- Garante que muitos programas ou aplicações possam rodar ao mesmo tempo sem que interfiram uns com os outros, pelo fato de o Unix ser um sistema multiusuário e multitarefa.

O *Shell*

Já em relação ao *Shell*, ele é o programa que interpreta os comandos digitados por um usuário. É uma parte do sistema operacional que funciona como um agente de ligação entre os comandos que a pessoa digita e as atividades (funções) que o *Kernel* deve realizar. No momento em que uma pessoa digita um comando, o *Shell* irá traduzir o nome daquele comando em linguagem de máquina, ou melhor, num conjunto de chamadas em linguagem de máquina, possibilitando ao *Kernel* realizar suas tarefas.

Nota-se, então, que cada comando no Unix nada mais é do que um nome fácil de se lembrar, sendo que esse nome de comando está ligado a um conjunto específico de chamadas do sistema.

O objetivo do *Shell* é deixar o sistema operacional mais amigável, ou seja, fazer com que o usuário, ao invés de se lembrar de um conjunto complicado de chamadas de sistema, lembre-se apenas do nome do comando.

É bom ressaltar que a grande maioria dos sistemas Unix possuem mais de um *Shell*, e esses *Shells* serão vistos e discutidos mais adiante.

Utilitários do Unix

Na camada mais externa da estrutura do Unix estão os *Utilitários* (Ferramentas e Aplicativos), que são programas pré-determinados a realizarem funções (tarefas) específicas. Dentro da variedade de sistemas operacionais Unix, existem muitos *Utilitários*, nem todos estão disponíveis em determinados sistemas.

Um usuário pode realizar muitas tarefas com esses utilitários, como: cálculos aritméticos, redirecionamento de saída para impressoras, desenvolvimento de programas, edição de um texto, canalização de comandos, etc.

O número de *Utilitários* dentro da família de sistemas operacionais Unix é imenso, e, além dessa grande variedade de aplicativos, há a possibilidade de combinação de comandos usando os pipelines (cujo símbolo é **|**), permitindo ao usuário obter um resultado de acordo com sua necessidade ou mais próximo dela e, também, de um modo mais rápido.

Os *Shells* disponíveis no Unix

Como já foi dito anteriormente, o *Shell* estabelece a interface entre o usuário do sistema e o *Kernel*. Também foi comentado, rapidamente, que a grande maioria dos sistemas Unix possuem mais de um *Shell* disponível junto a eles, o que permite ao usuário escolher com qual quer trabalhar, fazendo do Unix um sistema multiescolha, além de multitarefa e multiusuário.

Vale lembrar, só por questão de curiosidade, que o *Macintosh* também possui vários interpretadores de comandos. Caso a pessoa que o estiver

usando não quiser trabalhar com a interface padrão, poderá optar pelo *DiskTop*, *AtErase* ou *Square-One*.

Já em relação ao Unix, (quando inventado) foi projetado como um sistema operacional voltado para o programador, permitindo que este pudesse adequar o sistema de acordo com sua vontade. Consequentemente, não é surpresa alguma que vários *shells* tenham surgido no mercado, frutos da mente de alguns programadores fanáticos por Unix (e Linguagem C, usada na construção do Unix). É importante ressaltar que o código-fonte de algumas versões do Unix pode ser obtido facilmente através da internet, assim como o próprio sistema operacional.

O *Shell Bourne*

O precursor de todos esses *shells* existentes no mercado foi escrito por Ken Thompson, dos Laboratórios Bell da AT&T, por volta de 1979, como parte do projeto que envolvia o sistema de arquivos do Unix. Este ainda era complicado de se trabalhar. No entanto, depois de algum tempo, um outro componente da AT&T, *Steven Bourne*, se voltou para o *shell* escrito por Thompson e começou a modificá-lo e expandi-lo. A partir daí, o Unix teve um rápido crescimento e foi largamente distribuído entre instituições comerciais e universidades, e o *sh* ficou conhecido como o *Shell Bourne*. Este *shell* possui algumas características peculiares, pois é mais usado, principalmente hoje, para escrever *scripts* do que como um *shell* de comando e, está presente em praticamente todos os sistemas Unix existentes atualmente, além de possuir muitas variantes de *shell* baseadas nele.

O *Shell C*

Um dos mais importantes (se não o mais) *shell* do Unix, é o *Shell C*. Ele foi projetado por Bill Joy, da Universidade da Califórnia, em Berkeley. Quando estava desenvolvendo o editor de texto ***vi*** em linguagem C, Joy, admirador dessa linguagem, decidiu também criar um *shell* de comando que tivesse e compartilhasse muitas das estruturas da linguagem C, e que permitisse escrever scripts de *shell* mais poderosos e requintados, cujo nome ficou conhecido como *Shell C* ou *csh*. Neste *shell,* ele implementou algumas novidades em relação ao *Shell Bourne*, como:

- **Aliases de comando**: possibilitam aos usuários renomear e reconfigurar comandos facilmente, permitindo, assim, inserir uma *string* menor para um comando extenso.
- **Histórico de comando**: esse mecanismo faz com que o shell grave os comandos que um usuário utilizou, permitindo repeti-los sem precisar digitá-los inteiros novamente.
- **Controles de Serviço**: esses controles permitem aos usuários utilizar vários programas de uma só vez. Desse modo, pode-se usar um programa aplicativo num mesmo momento em que outro está sendo executado em segundo plano.

Pode-se notar, através desses aspectos, que o *Shell C* é bem interativo e flexível de se trabalhar, proporcionando mais facilidades e recursos para seus usuários.

O *Shell Korn*

O *Shell Korn*, *ksh*, foi desenhado como sendo uma extensão do *Shell Bourne*, com alguns recursos do *Shell C*, como aliases de comando (entretanto, num formato ligeiramente diferente), históricos de comando e controles de serviço. Ele foi desenvolvido por David Korn dos Laboratórios Bell da AT&T, no ano de 1982, com maiores aperfeiçoamentos realizados em 1986, 1988 e 1993. O *Shell Korn* está incluso de forma padrão, no Unix System V Release 4 e alguns outros sistemas, além de poder ser adquirido separadamente. Algumas propriedades desse shell são descritas abaixo:

- Edição interativa de linha de comando, inclusive com histórico de comando.
- Habilidade para extrair a parte de uma *string* especificada por um padrão, ou seja, pode-se procurar uma (ou algumas) determinada (s) palavra (s) dentro de um arquivo fornecendo ao sistema (com o *Shell Korn*) a descrição da palavra ou parte dela.
- Facilidade de se mover entre um diretório e outro, etc.

O *Shell Korn* está, cada vez mais, tornando-se popular. Com os recursos que possui, está se transformando numa alternativa para o *Shell C*.

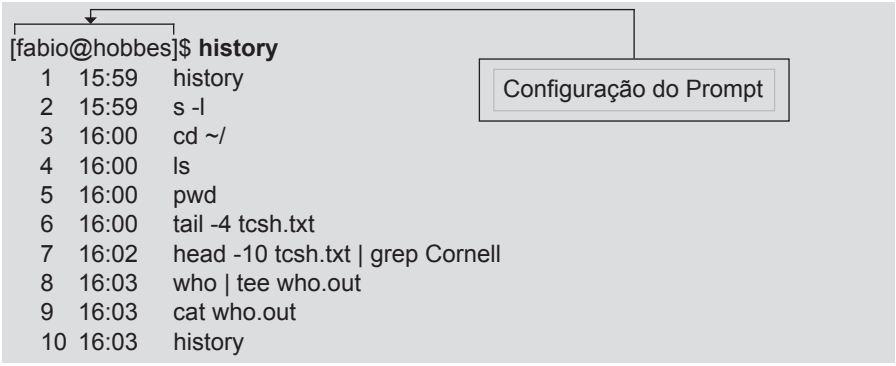
O *Shell tcsh* (versão modificada do *Shell C*)

Surgindo como uma versão modificada do *Shell C*, o *tcsh* possui fortes recursos, tanto como interpretador de comandos como para escrever *scripts*. Ele apresenta 95% de *csh* e 5% de novos recursos. Os mais importantes adicionais do *tcsh* em relação ao *Shell C* são:

- Leitura atenta da lista de histórico de comandos;
- Identificação de arquivos com os primeiros caracteres;
- Correção ortográfica de comando, arquivo e nomes de usuário;
- Desconexão automática depois de um determinado período inativo;
- Capacidade de monitorar *logins*, usuários e terminais;
- Novas variáveis de ambiente: HOST e HOSTTYPE

Exemplos de alguns recursos do *Shell tcsh*:

- Leitura atenta do histórico de comandos com o comando **history**:



- Novas variáveis de ambiente:

```
[fabio@hobbes]$ echo $HOST
hobbes
[fabio@hobbes]$ echo $HOSTTYPE
i386-linux
```

O símbolo **$** significa “valor de” e o comando **echo** exibe o valor de uma variável (esse comando será visto com mais detalhes no Capítulo 3).

O *Shell* "*Bourne-Again*" - *bash*

O *bash* é um *shell* que se encontra em todos os sistemas operacionais GNU. O projeto GNU visava desenvolver um sistema de softwares que deixasse o Unix mais completo, sendo que tal projeto foi criado pela "Free Software Foundation", uma organização fundada por Richard Stallman. A sigla GNU significa "GNU's Not Unix" (o "G" em GNU deve ser pronunciado). Alguns softwares desenvolvidos e distribuídos pelo projeto GNU são o editor **Emacs**, o compilador **gcc** e o depurador **gdb**.

O acrônimo "*Bourne-Again*" do *bash* é um trocadilho para Steven Bourne, o autor do *Shell Bourne*, sendo que o *bash* é uma reimplementação do *shell* original (Shell *Bourne*), tendo muitos novos recursos e não apresentando restrições de licença. Ele também possui implementações do *Shell* C e do *Shell Korn*.

Outros *Shells* do Unix

Alguns sistemas Unix ainda podem ter outros *shells*, além dos que já foram discutidos anteriormente. Geralmente, eles são adquiridos separadamente pelos proprietários de sistemas Unix. Logo a seguir, estão listados esses *shells* menos comumente usados:

- ***Shell rsh**:* é uma versão restrita do *Shell Bourne*, indicado para sites que tenham mecanismos de segurança. Houve discussões sobre alguns problemas de segurança envolvendo esse *shell.*
- ***Shell jsh**:* é também uma versão do *Shell Bourne*, que inclui os recursos de controles de serviço do *Shell C.*
- ***Shell MH*(*msh*)*:* esse *shell* foi desenvolvido por causa do programa de correio eletrônico MH, projetado pela *Rand Corporation.* Ele permite ter acesso instantâneo a qualquer correio eletrônico presente no sistema.

Identificando o *Shell*

Há muitas formas de se identificar o *shell* que se está usando. A primeira delas e, talvez a mais fácil, é entrar no arquivo etc/passwd (sendo que *etc* é um subdiretório do diretório raiz e *passwd* um arquivo pertencente ao

diretório *etc*) e ver o que está presente em sua conta. Veja o exemplo que se segue:

Exemplo:

```
[fabio@hobbes /]$ tail -8 etc/passwd
bene:5VgR1rD9qxT2w:500:500:Benedito Aparecido Cruz:/home/staff/bene:/bin/bash
kleber:bUXQIdilUbxiE:502:502::/home/staff/kleber:/bin/bash
rosa::503:503:Rosa Cristina Martins de Medeiros:/home/staff/rosa:/bin/bash
gonzal:1hI2zJUNyJdjE:504:504:Carlos Gustavo Gonzales,,,,:/home/staff/gonzal:/
bin/bash
rossano::505:505:Rossano Pablo Pinto,,,,:/home/staff/rossano:/bin/bash
brito::506:506:Antonio Brito,,,,:/home/staff/brito:/bin/bash
fabio:b.NlENts7oItc:501:501:Fabio Danilo Vieira:/home/staff/fabio:/bin/bash
joao:afltY24Vk/.xM:507:507:Joao da Silva:/home/staff/joao:/bin/bash
```

Veja no exemplo que os 8 últimos usuários do arquivo *etc/passwd* têm como *shell* padrão (ou *shell* de *login*) o *Shell bash* (o restante foi ignorado por ter informações compreendidas somente por um administrador de sistema).

No entanto, é bom conhecer outros modos de identificação, pois em alguns sistemas não tem o arquivo etc/passwd relacionado a segurança. Sendo assim, outra maneira bem simples de se identificar o *shell* consiste em observar o seu *prompt.* Se ele contém um %, provavelmente o *shell* que se está usando é o *Shell* C (csh) ou o *Shell* C modificado (tcsh), mas se o *prompt* for $, o *shell* em uso **pode ser** o *Shell Bourne*, o *Shell Korn* ou uma das variantes destes.

Há também um modo mais confiável de se obter o *shell* utilizado, que é pedir ao sistema operacional o programa que está sendo executado no momento. Através da linha de comando do exemplo a seguir, isso pode ser feito.

Exemplo:

```
[fabio@hobbes /]$ ps | grep $$
          326       p0 S    0:00 -bash
```

O comando **ps** exibe o que o sistema operacional está executando no momento e **grep $$** pesquisa por um nome de *shell* na lista de **ps**. Nesse exemplo, verifica-se que o *shell* utilizado é o *bash*.

Outra maneira de se conseguir identificar o shell utilizado é usar o comando **awk**, que é um programa que contém um kit de programação capaz de analisar e manipular arquivos de texto que tenham palavras, para ler o arquivo etc/passwd novamente. Veja o exemplo abaixo:

Exemplo:

```
[fabio@hobbes /]$ awk –F: ´{ if ($1 == “fabio”) print “seu shell é: ” $7 } ´< /etc/passwd
seu shell é: /bin/bash
```

Nesse exemplo, o comando **awk** verifica se o primeiro campo do arquivo passwd contém “fabio”. Caso contenha, ele irá imprimir o sétimo campo de tal arquivo, que possui o nome do *shell* relacionado a “fabio”.

Escolhendo um novo *Shell*

Foi visto, anteriormente. como identificar o *shell* que está sendo utilizado pelo sistema. Uma vez identificado esse *shell*, pode-se escolher um outro de acordo com a nossa preferência.

Antigamente, mudar de um *shell* para outro era bem complicado. A única alternativa era solicitar ao administrador do sistema para alterar o arquivo etc/passwd, e isso só era possível se ele tivesse tempo disponível. Felizmente, hoje, existe um programa simples (presente em quase todos os sistemas Unix) para alterar *shells* de *login,* que é o **chsh**, ou *change shell* (alterar *shell*). Esse comando não possui *flags*, não requer que nomes de arquivos sejam especificados e pode ser usado em qualquer local do sistema. É só digitar **chsh** e pressionar ENTER.

Antes de mais nada, vale dizer que o usuário também pode alternar de um *shell* para outro digitando apenas o nome do *shell* para o qual quer mudar. Ele não precisa, necessariamente, modificar o seu *shell* de *login* com o comando **chsh**, podendo usar outro *subshell* de acordo com sua necessidade.

Entretanto, antes de se mudar de *shell*, é preciso identificar os *shells* disponíveis no sistema. Há um local no sistema de arquivos do Unix em que estes estão guardados, e, por convenção, eles têm sh em algum lugar de seus nomes. Um dos locais é no diretório /bin (subdiretório do diretório raiz). Veja um exemplo de listagem desse diretório:

Exemplo:

```
[fabio@hobbes /]$ ls –lF /bin/*sh*
-rwxr-xr-x    1 root     root        56380     Dec 23    1996 /bin/ash*
-rwxr-xr-x    1 root     root       300668     Sep 3     1996 /bin/bash*
```

```
lrwxrwxrwx    1 root     root            3    Aug 27    1997 /bin/bsh -> ash*
-rwxr-xr-x    1 root     root       105356    Aug 12    1996 /bin/csh*
lrwxrwxrwx    1 root     root            4    Aug 27    1997 /bin/sh -> bash*
-rwxr-xr-x    1 root     root       233732    Dec 18    1996 /bin/tcsh*
```

Pode-se notar que há uma entrada em /bin/bsh e em /bin/sh para os *shells ash* e *bash*, respectivamente, embora seja realmente apenas um link apontando para o *shell ash* e *shell bash.*

Em outros sistemas, podem haver outros locais de armazenamento dos *shells.* Um exemplo disso pode ser visto abaixo:

Exemplo: (Tirado do livro “Aprenda em 24 horas Unix”)

```
% ls –lF /usr/local/bin/*sh*
-lrwxr-xr-x    1 root         8    Jul  26    14:46 /usr/local/ksh->bin/ksh*
--rwxr-xr-x    1 root    266240    Jan 19     1993 /usr/local/tcsh*
```

Evidentemente que podem existir outros *shells* no sistema, mas isso não influenciará no modo de alterar de um *shell* para o outro. Portanto, para modificar o *shell* de *login* para qualquer um dos *shells* disponíveis no sistema ou até mesmo só para verificar o *shell* que está se executando, usa-se o comando **chsh** (comentado anteriormente). Veja o exemplo:

Exemplo:

```
[fabio@hobbes fabio]$ chsh
Changing login shell for fabio
Old shell: /bin/bash
New shell: _
```

Nesse exemplo, o programa indica que bin/bash é o *shell* de *login* e pede para digitar o nome de um *shell* alternativo. Caso tente especificá-lo com algum outro nome de arquivo, o comando retorna uma mensagem dizendo que aquele nome é inaceitável como um novo *shell*. Confira o exemplo:

Exemplo:

```
[fabio@hobbes fabio]$ chsh
Changing login shell for fabio
Old shell: /bin/bash
New shell: ~/arq.txt
home/staff/fabio/arq.txt is unacceptable as a new shell.
```

O comando **chsh** não aceita essa mudança porque ele verifica no arquivo /etc/shells os nomes de *shells* válidos. Veja, em seguida, a leitura do arquivo /etc/shells em um sistema Unix:

Exemplo:

```
[fabio@hobbes fabio]$ cat /etc/shells
/bin/ash*
/bin/bash*
/bin/bsh -> ash*
/bin/csh*
/bin/sh -> bash*
/bin/tcsh*
```

Sabendo, então, os nomes dos *shells* válidos, pode-se alterar com precisão o *shell:*

Exemplo:

```
[fabio@hobbes fabio]$ chsh
Changing login shell for fabio
Old shell: /bin/bash
New shell: /bin/csh
[fabio@hobbes]$
```

Novo Prompt

Pode-se notar que a alteração obteve sucesso, isso porque o Unix não mostrou nenhuma mensagem de erro e também por causa da mudança da configuração do *prompt* (entretanto, este fator não deve ser sempre levado em consideração, pois, às vezes, o *prompt* pode permanecer o mesmo). O novo *shell* agora é o *Shell C*, o mais popular e usado *shell* do Unix, e por esse motivo, daqui para frente, ele será tratado de forma mais específica, principalmente nos próximos capítulos.

O ambiente Shell

O ambiente é um espaço na memória onde se pode colocar definições de variáveis do *shell* para que elas fiquem acessíveis a qualquer programa que for executado. Se não colocar a definição de uma variável no ambiente, apenas o *shell* irá conhecer o valor de tal variável.

A inserção da definição de uma variável no ambiente é feita em 2 etapas: na primeira define-se a variável do *shell* e, na segunda, se coloca-se a definição no ambiente Unix. Depois ela será chamada de variável ambiental (e continua sendo uma variável do *shell*).

Portanto, o *shell* é o principal programa interessado no ambiente, pois nele são armazenados os valores de **algumas** de suas relativas variáveis. Por

que **algumas**? Porque ele não procura no ambiente certas variáveis, e sim apenas a sua definição, ou seja, o que interessa para ele é apenas o valor que está **definido** para essas variáveis, e não seu valor **no ambiente**. Um exemplo é a variável *history*, do *Shell C,* cuja definição é lida pelo *csh*. Essa variável será vista com mais detalhes no próximo capítulo.

Já um exemplo de variável ambiental é a PATH, a qual todos os *shells* utilizam e onde, também, se define a rota de procura de programas.

Definindo variáveis

A definição de uma variável varia de *shell* para *shell* e, por este motivo, será mostrado como definir variáveis apenas nos dois principais *shells*: *Shell Bourne* e *Shell C*. Mas, os modos de como definir variáveis destes *shells* podem ser iguais a outros, principalmente em suas variantes.

Para se definir uma variável para o *Shell Bourne* e para o *Shell C*, usam-se as seguintes sintaxes de linhas de comandos:

```
$ variavel=valor          (Shell Bourne)
% set variavel=valor      (Shell C)
```

Nesses casos, variavel é o nome da variável, e valor pode ser tanto um número quanto uma sequência de caracteres. Uma variável do *shell* contendo um valor numérico é tratada pelo *shell* como se ela tivesse um valor texto ou alfanumérico. Observa-se também que o *Shell C* usa o comando **set** para definir variáveis.

Logo abaixo seguem alguns exemplos de definições de variáveis para o *Shell Bourne* e *Shell C*:

Exemplos:

No *Shell Bourne:*

```
$ usuario=´Jose da Silva´
```

No *Shell C:*

```
% set usuario=´Jose da Silva´
```

No *Shell Bourne:*

```
$ PATH=:/bin:/usr/bin
```

No *Shell C:*

```
% set PATH=:/bin:/usr/bin
```

No *Shell Bourne:*

```
$ arq1=/bin/vi  arq2=/usr/bin/gnuemacs
```

No *Shell C:*

```
% set arq1=/bin/vi  arq2=/usr/bin/gnuemacs
```

A variável usuario é definida como sendo “Jose da Silva”. A variável do *shell*, PATH, é *redefinida* com o valor “:/bin:/usr/bin”. Também se pode definir mais de uma variável por linha, com arq1 sendo “/bin/vi” e arq2 sendo “/usr/bin/gnuemacs”.

Para se exibir o valor de uma variável tanto do *Shell Bourne* quanto do *Shell C*, usa-se o comando **echo** mais o nome da variável precedida pelo sinal de cifrão ($), que significa “valor de”. Veja, em seguida, como exibir o conteúdo das variáveis definidas anteriormente:

```
$ echo $usuario
Jose da Silva
% echo $PATH
:/bin:/usr/bin
$ echo $arq1
/bin/vi
% echo $arq2
/usr/bin/gnuemacs
```

Apesar de se ter definido usuario, PATH, arq1 e arq2 como variáveis do *Shell Bourne* e do *Shell C*, elas ainda não são variáveis ambientais porque não foram colocadas no ambiente, mas isso será visto no próximo item.

Variáveis ambientais

Depois que se definem as variáveis para o *shell* pode-se transformá-las em variáveis ambientais.

Obs: em *Shell C,* define-se uma variável que se transforma em variável ambiental com apenas um comando, o qual será visto à frente.

As variáveis ficam disponíveis a cada processo de programa assim que sua execução começa. Um processo poderá ou não usar a informação guardada na variável ambiental. Portanto, as informações ambientais

fornecem uma maneira de se passar informações “globais” para qualquer programa que esteja em execução.

Tendo por base o que foi dito, define-se que variável ambiental é simplesmente uma variável do *shell* que foi inserida no ambiente. Mas, assim como a definição de uma variável varia de *shell* para *shell*, o modo de se colocar uma variável no ambiente também varia de um shell para outro. Como exemplos dessas diferenças, novamente irá se usar o *Shell Bourne* e o *Shell C*.

Para se inserir uma variável do *Shell Bourne* no ambiente, usa-se o comando **export**. Observe a sintaxe da linha de comando:

```
$ export variavel
```

Sendo assim, a definição de uma variável ambiental do *Shell Bourne* consiste em duas etapas:

- A criação
- Exportação da variável

Como exemplo, irá se definir a variável do prompt do *Shell Bourne*, PS1, e colocá-la no ambiente.

Exemplo:

```
$ PS1=“Aguardando ordem”; export PS1
Aguardando ordem _
```

Observe que foram colocados dois comandos na mesma linha, separados por ponto-e-vírgula, e o *prompt* foi modificado. O usuário também pode fazer com que PS1 volte ao seu valor original, digitando:

```
Aguardando ordem PS1=‘$’; export PS1
$ _
```

Para o *Shell C*, tem-se a possibilidade de definir e acrescentar uma variável no ambiente em apenas uma etapa, usando o comando **setenv**. Veja a sintaxe:

```
% setenv variavel valor
```

Como exemplo, também irá se redefinir o prompt do *Shell C*:

```
% setenv prompt ‘$’
$ _
```

Note que ele ficou igual ao *prompt* padrão do *Shell Bourne*, mesmo estando no *Shell C*. Portanto, quando for se identificar o *shell* em uso, a con-

figuração do *prompt* nem sempre é a maneira mais confiável de se fazer isso.

Para distingui-los de outras variáveis do *shell*, os nomes das variáveis ambientais do *Shell Bourne* são, geralmente, escritos com letras maiúsculas. No *Shell C* também há algumas variáveis com nomes em letras maiúsculas, mas o restante das variáveis estão mesmo em minúsculas. Veja, em seguida, uma lista de variáveis ambientais comumente usadas pelo *Shell Bourne* e *Shell C*. Observe também que não se deve utilizar os mesmos nomes para outras variáveis que venham a ser definidas, isso para evitar conflitos com os nomes reservados:

- HOME: seu diretório doméstico, obtido do quarto campo do arquivo de senha (*/etc/passwd*); O diretório onde seu *shell* está originalmente posicionado depois que se inicia uma sessão no sistema.
- PATH: rotas (os caminhos) para a procura de comandos, as quais consistem em uma lista de diretórios que o *shell* e outros comandos pesquisam para encontrar programas solicitados pelos usuários quando o caminho não for especificado.
- MAIL: indica o caminho para o seu arquivo de correspondência onde suas mensagens estão guardadas.
- TERM: modelo do seu terminal.
- SHELL: indica o *shell* que está sendo usado. Muito utilizada por programas como o editor de texto vi, por exemplo.
- PS1: *prompt* principal do *Shell Bourne* (e também do *Shell Korn*). O valor padrão é “$”.
- PS2: *prompt* secundário do *Shell Bourne*. O valor padrão é “>”.
- TES: separadores internos de campo, que podem ser os caracteres TAB, espaço e avanço de linha. Eles separam palavras na linha de comando do *shell.*
- TZ: especifica o fuso horário, cujo valor tem o formato *xxxnzzz*, onde *xxx* é a abreviação padrão do fuso horário local, *n* é a diferença em horas da hora de Greenwich (GMT) e *zzz* é a abreviação do horário de verão no fuso horário considerado, se houver. Por exemplo, PST8PDT é a especificação do fuso-horário da costa-oeste dos Estados Unidos, com 8 horas de diferença de Greenwich.

- LOGNAME: a identificação do usuário que está executando o *shell.*
- NAME: obtida do campo 5 do arquivo /etc/passwd (ou equivalente), ela é útil para programas de correio eletrônico e de impressão que, às vezes, precisam saber o nome real completo do usuário do sistema.
- USER: sinônimo de LOGNAME, ou seja, ela dá a identificação do usuário que está usando o *shell.*

Obs: antigamente, LOGNAME era uma variável pertencente ao *Shell Bourne* e USER ao *Shell C*, mas, hoje, ambos os shells utilizam essas duas variáveis.

No entanto, não é somente o *shell* que usa variáveis ambientais. Alguns comandos também utilizam-se delas. Um exemplo é o comando **more**. Quando esse comando está sendo executado, ele procura no ambiente pela variável MORE. Se tal variável for encontrada, seu valor é interpretado por esse comando, como opções de linha de comando. Caso o usuário, por exemplo, não queira ver as linhas em branco nos arquivos, ele pode atribuir à variável MORE a opção –s. Observe as linhas de comando:

No *Shell Bourne*:

```
$ MORE= -s      (atribui um valor a MORE)
$ export MORE  (exporta-a para o ambiente)
```

No *Shell C*:

```
% setenv MORE –s  (atribui um valor à variável e exporta-na para o ambiente)
```

No *Shell C* a definição e a colocação de uma variável no ambiente se dá em apenas uma etapa, enquanto que no *Shell Bourne* se dá em duas etapas.

Para se listar os vários aspectos de um ambiente, pode-se usar o comando **env**. Veja o exemplo:

Exemplo:

```
[fabio@hobbes]$ env
HISTSIZE=1000
HOSTNAME=hobbes
LOGNAME=fabio
HISTFILESIZE=1000
MAIL=/var/spool/mail/fabio
TERM=ansi
HOSTTYPE=i386
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
HOME=/home/staff/fabio
```

```
SHELL=/bin/bash
PS1=[\u@\h \W]\$
USER=fabio
OSTYPE=Linux
SHLVL=1
_=/usr/bin/env
```

Nesse exemplo, verifica-se que aparecem algumas variáveis desconhecidas, mas que não têm muita importância em relação à configuração necessária para o sistema funcionar adequadamente, comportando-se apenas como opções do próprio usuário ou administrador do sistema.

Arquivos de inicialização do *Shell*

Foi visto, no item anterior, como personalizar seu ambiente manualmente através de comandos, logo após ter se conectado ao sistema. Porém, pode-se colocar esses mesmos comandos dentro de um arquivo executável. Esse arquivo é chamado de *arquivo de inicialização do shell*, e ele é executado automaticamente toda vez que alguém se identifica no sistema.

Esse arquivo tem um nome especial, que é reconhecido somente pelo *shell* de *login* do usuário, e seu nome está armazenado juntamente com os dados de *login* do usuário no arquivo /etc/passwd ou equivalente. O *Shell Bourne* reconhece o arquivo de inicialização *.profile,* e o *Shell C* reconhece tanto o arquivo *.login* quanto o *.cshrc*.

O arquivo .cshrc do *Shell C* é executado quando o usuário se identifica no sistema e também toda vez que se cria um *subshel*l, isto é, quando o usuário muda de um outro *shell* qualquer para o *Shell C*.

Há muitos comandos que se pode colocar nos arquivos de inicialização .profile do *Shell Bourne* e .login do *Shell C*. Logo abaixo segue-se uma lista de possíveis inclusões que se pode realizar nesses arquivos:

- Definição de variáveis do *shell* e também sua colocação no ambiente.
- Redefinição da configuração do *prompt* do *shell*.
- Redefinição da rota de procura de comandos pela variável PATH.
- Definição de opções de configuração para o terminal usando o comando **stty** (*set tty drivers options,* ou definir opções do *driver tty*).

Existem também alguns comandos optativos que podem ser inseridos no arquivo *.login* ou *cshrc* do *Shell C*:

- Ativação da função de histórico de comandos.
- Definição de aliases (pseudônimos) para comandos.
- Redefinição de variáveis do *Shell C* e também sua inserção no ambiente.

Logo a seguir, tem-se alguns exemplos desses 3 arquivos: *.profile* do *Shell Bourne*, *.login* e *.cshrc* do *Shell C*:

Exemplos:

Exemplo 1:

```
$ cat .profile
PATH=:/bin:/usr/bin:/usr/lbin
TERM=kA
export PATH TERM
stty erase ‘^h’ kill ‘^x’ quit ‘^u’ intr ‘^c’
stty ixon
echo A hora de sua identificação é  ‘date’.
echo No momento há  who|ws –a  terminais em linha.
echo As variáveis do shell que estão definidas são:
set
echo A linha do terminal está assim configurada:
stty
```

Exemplo 2:

```
% cat .login
#  @(#)   $Revision: 62.2  $
setenv TERM vt100

stty erase “^h” kill “^u” intr “^c” eof “^d”
stty crtbs crterase

#  variáveis do shell

set history=100 savehist=50

#  define algumas variáveis de ambiente globais...

setenv EXINIT “: set ignorecase”
setenv NAME “Dave Taylor”
```

Exemplo 3:

```
% cat .cshrc
# .cshrc
```

```
# User specific aliases and functions

alias rm ‘rm -i’
alias cp ‘cp -i’
alias mv ‘mv -i’

setenv PATH=”.:/usr/sbin:/sbin:$PATH”
set prompt=\[`id -nu`@`hostname -s`\]\#\
```

Em primeiro lugar, quando aparece o caractere # mais um texto, significa que tal sequência refere-se a um comentário. Entretanto, pode-se ver que aparecem algumas variáveis e alguns comandos novos em relação ao que foi visto até aqui. Nos arquivos *.profile* e *.login* aparece o comando **stty** citado anteriormente, que configura opções de um terminal. No arquivo *.login*, por exemplo, a linha ***stty*** assegura que **^h** (as teclas *control* + *h*) equivale a apagar (*backspace*), **^u** equivale a excluir uma linha inteira de comando, **^c** envia uma interrupção para o programa que está sendo executado e **^d** indica o final de um arquivo. Já a segunda linha do **stty** assegura que a CRT (uma tecnologia usada na tela de um terminal padrão) seja capaz de apagar e excluir caracteres na tela. Outro comando é o alias, que surge no arquivo .cshrc. Apesar de, no próximo capítulo, ser bastante enfocado, por enquanto deve-se saber que o formato de execução é **alias** *word command* e que ele fornece um pseudônimo para linhas de comando muito extensas. No exemplo do .cshrc, ele permite que, digitando-se apenas **rm**, **cp** e **mv**, o flag **–i** já esteja incluso.

Em relação às variáveis, só no arquivo .login aparecem algumas diferentes das que foram comentadas até aqui. A variável history=100 permite ao sistema informar os últimos 100 comandos utilizados e savehist=50 grava e depois informa cinquenta deles, mesmo que se desligue o sistema e conecte-se novamente. E, finalmente, a variável EXINIT é utilizada pelo editor de texto **vi** e está definida para que tal editor possa realizar uma pesquisa de texto sem levar em consideração letras maiúsculas e minúsculas. Vale dizer que as variáveis *history* e *savehist* serão revistas no próximo capítulo.

Chega-se, então, ao final do primeiro capítulo, que teve como objetivo dar a definição da estrutura do sistema operacional Unix, explicar as funções do *shell* dentro do Unix, assim como comentar sobre os *shells* mais importantes e alguns ainda menos difundidos pela comunidade Unix. Foi visto como se faz para identificar o *shell* em uso, bem como mudar de um shell para outro. Explicou-se, também, sobre como definir suas variáveis e

exportá-las para o ambiente. Outro ponto importante foi sobre os arquivos de inicialização de shell, entre os quais estão o *.login* e o *.cshrc* do *Shell C* e o *.profile* do *Shell Bourne.*

Capítulo 2

Um pouco mais do *Shell C* e do *Shell Korn*

No capítulo anterior, foram apresentados alguns *shells* disponíveis no Unix. No entanto, como já foi dito anteriormente, o *Shell C* é o que está mais presente na grande maioria dos sistemas Unix existentes no mundo, e por esse motivo, será bastante enfatizado nesse capítulo. Além disso, também serão discutidos alguns recursos importantes do *Shell Korn*, que é uma alternativa para o *Shell C.*

Os mecanismos de histórico do *Shell C* e do *Shell Korn*

O mecanismo de histórico de comandos, tanto do *Shell C* quanto do *Shell Korn*, permite que o usuário possa relembrar e utilizar um comando anteriormente executado por ele no sistema. A partir do momento que um usuário se conecta ao sistema e digita o primeiro comando, este equivale ao comando 1, e o restante que tal usuário digitar será incrementado a esse primeiro comando. Sendo assim, para se fazer uma revisão ou repetir qualquer comando basta apenas tocar algumas teclas.

O *Shell C* utiliza o comando **history** para visualizar até os últimos *n* comandos digitados, sendo que esse *n* é o número de comandos que o usuário gostaria de lembrar quando executasse **history**. Para se definir *n* (número de comandos) utiliza-se uma variável do *Shell C* chamada *history* (não estranhem; ela, realmente, tem o mesmo nome do comando). Veja a sintaxe de linha de comando abaixo:

```
% set history=n
```

Depois que a variável history estiver definida, toda vez que o comando **history** for chamado, ele irá consultar essa variável e o valor de *n* atribuído a ela, para com isso poder exibir a quantidade de comandos desejada pelo usuário. Caso apareça uma quantidade menor, é porque não foi utilizado um número suficiente de comandos para atingir o valor de *n*.

Observe que essa variável não foi transportada para o ambiente, isso porque é um tipo de variável que só o *Shell C* pode reconhecer, portanto, ela não necessita ser definida no ambiente, mas *precisa* ser definida para o Shell C.

Já para o *Shell Korn*, o valor de *n* é único e padrão, e corresponde a 128. Sendo assim, o usuário irá visualizar "apenas" os últimos 128 comandos digitados por ele (caso ele tenha digitado tudo isso). Esse valor pode ser considerado pequeno em relação ao *Shell C*, no qual se pode atribuir valores bem maiores que esse. Para visualizar o histórico no *ksh,* também se utiliza o comando **history**, apesar de que nesse shell, **history** é um alias para a linha de comando **fc –l**.

Em seguida são exibidos alguns exemplos do que foi discutido desse mecanismo até aqui:

Exemplo de definição da variável history no *Shell C*:

```
[fabio@hobbes]$ set history=1000
```

Verificando as variáveis definidas para o Shell C, inclusive history:

```
[fabio@hobbes]$ set
argv    ()
cwd     /home/staff/fabio
history 1000
home    /home/staff/fabio
i       /etc/profile.d/*.csh
path    (/usr/local/bin /bin /usr/bin /usr/X11R6/bin /home/staff/fabio/bin /usr/X11R6/bin)
prompt  [fabio@hobbes]$
shell   /bin/csh
status  0
term    ansi
user    fabio
```

Exemplos de utilização do comando ***history***, que valem tanto para o *Shell C* quanto para o *Shell Korn*:

Antes de se utilizar o comando ***history***, serão digitados alguns comandos de exemplos para se observar a saída fornecida pelo comando de histórico:

```
[fabio@hobbes]$ ls -lF ← Lista-se o diretório por permissão e tipos de arquivos
total 1168
-rw-------          1 fabio    fabio      821    Mar 7    16:08    mbox
drwxr-xr-x         2 fabio    fabio      1024   Mar 4    20:41    novell/
drwxrwxr-x         2 fabio    fabio      1024   Mar 7    08:13    ocultos/
-rwxrwxr-x         1 fabio    fabio      3959   Mar 6    16:45    prog*
-rw-rw-r--         1 fabio    fabio      95     Mar 6    16:45    prog.c
drwxrwxrwx         2 fabio    fabio      1024   Mar 4    20:34    public/
-r--r--r--  1 fabio    fabio      183109 Mar 4    23:34    tcsh.1
```

```
-rw-rw-r--    1 fabio    fabio     493265  Mar 4    23:35    tcsh.lj
-rw-rw-r--    1 fabio    fabio     250862  Mar 5    00:13    tcsh.lt1
-rw-rw-r--    1 fabio    fabio     250862  Mar 5    00:09    tcsh.txt
-rw-rw-r--    1 fabio    fabio     48      Mar 4    20:28    testfile
-rw-rw-r--    1 fabio    fabio     85      Mar 6    16:03    who.out

[fabio@hobbes]$ wc prog.c
    11      15      95 prog.c

[fabio@hobbes]$ date
Sat Mar  7 18:09:39 EST 1998
```

Obtêm-se o número de linhas, palavras e caracteres do arquivo prog.c

Obtêm-se data e hora do sistema

Agora sim, se utilizará o comando **history**:

```
[fabio@hobbes]$ history
    1  18:03   set history=1000
    2  18:03   ls -lF
    3  18:08   wc prog.c
    4  18:09   date
    5  18:10   history
```

Observe que a primeira linha de comando da lista é o *set history=1000*, o qual foi executado no item anterior, e o restante são os exemplos utilizados para demonstrar a função do comando ***history***.

Para que esse comando possa ser usado permanentemente, pode-se adicionar a linha comando ***set history*** no arquivo *.login* ou *.cshrc* do *csh*. Se o usuário desejar que os comandos sejam lembrados pelo *shell,* mesmo que se desligue o sistema, acrescenta-se também a variável *savehist*. Um exemplo é o arquivo *.login,* visto no capítulo anterior, que possuía essas variáveis:

Exemplo do arquivo .login:

```
% cat .login
#  @(#)   $Revision: 62.2  $

setenv TERM vt100

stty erase "^h" kill "^u" intr "^c" eof "^d"
stty crtbs crterase

#  variáveis do shell

set history=100 savehist=50

#  define algumas variáveis de ambiente globais...

setenv EXINIT ": set ignorecase"
setenv NAME "Dave Taylor"
```

Variáveis history e savehist

Em relação ao *Shell Korn*, não se precisa fazer alterações em seus arquivos de inicialização, pois o comando ***history*** está pronto para ser usado imediatamente.

Como usar o histórico para economizar digitação

No item anterior foi visto como obter uma lista de histórico de comandos. Agora, veremos como usá-la.

Para se executar um comando de histórico no *Shell C*, utiliza-se o ponto de exclamação (!) no início de tal comando. Por exemplo, se o vigésimo quinto comando digitado foi o comando **date**, para executá-lo novamente, insere-se **!25** na linha de comando. Outra opção é inserir um ou mais caracteres do comando, por exemplo: **!d**, **!da** ou **!date**. Deve-se colocar caracteres suficientes para identificá-lo exclusivamente na lista de histórico.

Para se consertar um comando *anterior* (ou mesmo para reeditá-lo), digita-se: **^***padrão a ser alterado***^***padrão correto*. Por exemplo, caso o usuário tenha executado o comando **ls a*** (listar todos os arquivos que possuam nomes que comecem com a) e queira editá-lo novamente, para acrescentar mais a letra **t** junto à letra **a**, ele poderá usar a seguinte linha de comando: **^a*^at***.

O comando **!!** talvez seja o mais útil dentre todos os comandos de histórico do *Shell C*. Ele repete o último comando executado. Por exemplo, se o último comando usado foi **grep fabio /etc/passwd** (pesquisar por “fabio” no arquivo *passwd*), digitando-se **!!** e apertando ENTER, esse comando será executado novamente.

O *Shell C* possui outros comandos de histórico, como **!$**, que vai até a última palavra da linha de comando *anterior*, e **!***, que recupera todas as palavras da linha de comando *anterior*, menos a primeira. Um exemplo desse comando podia ser o caso do usuário ter usado, como última linha de comando, **ls /etc/passwd**, e, em seguida, modificá-la, trocando-se apenas a primeira palavra: **cat !*** (= cat /etc/passwd).

Já em relação ao *Shell Korn*, ele dispõe de quase todos os comandos de histórico do *Shell C*. Para se repetir comandos pelo número, utiliza-se a linha de comando **r***n*, onde *n* é o número do comando (por exemplo, **r***45*). Para se repetir comandos, usando, ao invés do número, o nome do

comando, digita-se **r***nome-do-comando* (Exemplo: **r***ls*). Sem quaisquer argumentos, **r** repetirá o comando anterior.

Em seguida, são transcritos alguns exemplos do que foi dito sobre comandos de histórico até aqui. Os exemplos serão baseados no *Shell C*, mas seus equivalentes em *ksh* serão lembrados.

1. Primeiro deve-se consultar a lista de histórico para usar os comandos que estão lá guardados:

```
% history
1   15:05   set history=1000
2   15:08   history
3   15:09   cat arq.txt
4   15:11   ls
5   15:14   date
6   15:14   history
```

Lembrando que, no *Shell Korn*, usa-se o mesmo nome de comando para obter o histórico.

2. Para se repetir o comando **date**, basta especificar seu número de comando:

```
% !5
date
Sat  Mar  14  13:05:03  Est  1998
```

No *Shell Korn*, o equivalente seria **r5**.

3. Para se repetir a linha de comando **cat arq.txt**, basta especificar sua primeira letra.

```
% !c
cat arq.txt
        Shell: interpretador de comandos
        Kernel: controla dispositivos de hardware e gerencia dados
%_
```

No *Shell Korn*, o equivalente seria **rc**.

4. Para executar o último comando do histórico, usa-se:

```
% !!
history
   1   15:05   set history=1000
   2   15:08   history
   3   15:09   cat arq.txt
   4   15:11   ls
```

```
5  15:14  date
6  15:14  history
```

O equivalente no *ksh* é **r**.

Pode-se usar centenas de exemplos desses comandos de histórico, mas os que foram dados e a explicação feita são suficientes para se utilizar plenamente o recurso de histórico de comandos. A tabela 1 a seguir exibe os comandos de histórico do *Shell C*:

Tabela 1. Comandos de histórico do *Shell C.*

Comando	Função
!!	Repete o comando anterior.
!$	Repete a última palavra do comando anterior.
!*	Repete todas as palavras do comado anterior, menos a primeira.
^a^b	Substitui a por b no comando anterior.
!n	Repete o comando n da lista de histórico.

Obs.: Tabela retirada do livro “Aprenda em 24 horas Unix

Como usar os aliases de comandos

Como já foi comentado no capítulo anterior, os aliases de comando possibilitam que se renomeie e reconfigure comandos de modo que se facilite o uso destes. Os aliases reduzem bastante o processo de digitação, assim como o mecanismo de histórico de comandos.

Para se usar o mecanismo de alias no *csh*, utiliza-se o formato padrão **alias** *palavra comando (ou sequência de comandos)*. Já no *ksh*, o formato utilizado é **alias** *palavra=comando (ou sequência de comandos)*.

Se for inserido, tanto no *Shell C* quanto no *Shell Korn*, somente o comando **alias,** sem quaisquer argumentos, será mostrada uma saída com uma lista de aliases que o usuário definiu (se tiverem aliases definidas). No *csh* se for usado a linha de comando **alias** *palavra*, será listado o alias atual, relacionado àquela palavra.

Alguns exemplos desse poderoso comando são listados e comentados em seguida:

1. Uma das linhas de comandos mais usadas do Unix é o **ls –FC**, que lista o conteúdo de um determinado diretório em várias colunas, informando os tipos de arquivos e os subdiretórios. Pode se criar um alias para ela:

```
% alias ls ´ls –FC´
```

Observa-se que o nome usado para substituir a linha de comando é o nome do próprio comando. Portanto, quando se for usar **ls** sozinho, ele substituirá **ls –FC**. Mas isso não impede que o comando **ls** seja usado com outras opções. Observe, então, a saída de **ls**:

```
% ls
mbox    ocultos/    prog.c    tcsh.1    testfile    vin@
novell/ prog*       public/   tcsh.lj   who.out
```

Com essa saída, é possível se identificar os subdiretórios (os que possuem uma barra na frente do nome) e os tipos de arquivos (*executáveis*: com asterisco na frente do nome; *textos*: sem nada embutido ao nome; *vínculos simbólicos*: com o símbolo arroba @ na frente do nome).

O equivalente uso do comando **alias** no *ksh* é **alias ls=´ls –FC´**.

2. Pode-se, também, criar aliases para sequência de comandos, ou seja, junções de dois ou mais comandos para fornecer uma determinada saída. Por exemplo, é possível se incrementar o alias anterior, fazendo com que, além de sair uma listagem que especifica os subdiretórios e arquivos, seja exibida também a quantidade de espaço usado em disco pelo conteúdo daquele diretório (comando **du**):

```
% alias ls ´ls –FC; du –s´
mbox    ocultos/    prog.c    tcsh.1    testfile    vin@
novell/ prog*       public/   tcsh.lj   who.out
1325
```

A saída fornece o total usado pelo conteúdo do diretório em kilobytes. Nesse caso, o total é 1325 kilobytes.

O equivalente em *ksh* é **alias ls=´ls –FC; du –s´**.

3. Para usuários que estão acostumados com os comandos do DOS e começaram a usar o Unix agora, o **alias** é um grande aliado para quem ainda não conseguiu se familiarizar com a sintaxe do Unix. Com os aliases, pode se recriar os comandos do DOS, renomeando seus equivalentes no Unix:

```
% alias DIR ´ls –lF´
% alias REN ´mv´
% alias COPY ´cp –i´
% alias DEL ´rm –i´
```

Testando alguns deles:

```
% DIR arq1.txt
-rw-rw----   1   fabio    fabio      220  Dec  3  14:15  arq1.txt
% REN arq1.txt  arq2.txt
% DIR arq2.txt
-rw-rw----   1   fabio    fabio      220  Dec  3  14:15  arq2.txt
```

Foram, nesse caso, recriados apenas os comandos mais usados do DOS, mas o usuário pode incrementar outros mais, de acordo com sua vontade.

Para que as definições de aliases fiquem permanentes, ou seja, para que não se precise defini-las toda vez que se entrar no sistema, deve-se inseri-los nos arquivos de inicialização de seus respectivos *shells*. No caso do *Shell C*, insere-se no arquivo *.cshrc* e. no Shell Korn, coloca-se no *.profile*. Num exemplo dado anteriormente do arquivo *.cshrc*, havia algumas linhas de aliases. Veja ele novamente:

```
% cat .cshrc
# .cshrc

# User specific aliases and functions

alias rm ‘rm -i’
alias cp ‘cp -i’
alias mv ‘mv -i’

setenv PATH=”.:/usr/sbin:/sbin:$PATH”
set prompt=\[`id -nu`@`hostname -s`\]\#\
```

Exemplos de algumas aliases.

Com essas linhas de aliases inseridas nesse exemplo do arquivo .cshrc, o usuário, quando utilizar um daqueles três comandos (**cp**, **rm** e **mv**), não precisará digitar o *flag* **–i**, pois ele já estará implicitamente incluso em qualquer um deles, o *flag* –i emite uma mensagem antes da execução, de fato, do comando em que é utilizado; geralmente, um pedido de confirmação para que se execute tal comando. Para se desativar um alias, devemos usar o comando **unalias** mais o nome do alias. Caso esse alias esteja configurado em um arquivo de inicialização de *shell,* o comando **unalias** o desativará temporariamente, enquanto o sistema não for desligado.

Percebe-se, com o que foi visto até agora, que seriam necessárias muitas horas para citar os possíveis usos do comando **alias**. Entretanto, o funda-

mento e as utilidades desse incrível comando ficou bem clara, e cabe ao usuário utilizá-lo de acordo com sua imaginação e necessidade.

Como personalizar *prompts*

Muitos usuários não gostam dos *prompts* que o Unix oferece. Realmente, alguns são bem grotescos e desagradáveis de se trabalhar. mas, alguns *shells* do Unix (a maioria deles), como o *csh* e o *ksh*, permitem ao usuário configurar seu próprio *prompt.*

A linha de comando usado no *Shell C* para se definir o prompt é **set** *prompt=“valor”*, enquanto que no *Shell Korn* é **PS1**=*“valor”*.

Logo a seguir, são relatados alguns exemplos de configuração de prompt no *Shell C* e seus equivalentes no *ksh*:

1. O primeiro exemplo é o mais simples de todos, pois o *prompt* é configurado como sendo um texto qualquer digitado pelo usuário. Nesse caso, o texto é “Olá, digite algo: “.

```
% set prompt=“Olá, digite algo: ”
Olá, digite algo: _
```

Seu equivalente no *Shell Korn* é **PS1=“Olá, digite algo: ”**.

2. Porém, há informações mais úteis para se colocar no *prompt* do que apenas simples textos ou sinais. Um exemplo bem útil seria utilizar a variável cwd, que informa o caminho do diretório de trabalho atual:

```
% set prompt=“cwd: ”
/users/fabio: _
```

Com isso, o *prompt* torna-se um pouco mais amigável. Lembre-se que, no *Shell Korn* a atribuição da variável cwd ao *prompt* seria assim: **PS1=“cwd”**.

3. Também se pode configurar o *prompt* para que ele informe o número do comando atual. Veja como:

```
% set prompt=“(\!)%”
(202)% _
```

O equivalente em *ksh* é **PS1=“(\!)$”**, sendo que o símbolo “$” é optativo e só foi colocado por ser padrão do *Shell Korn*.

4. Outra maneira de se fazer com que o *prompt* tenha uma utilidade maior é defini-lo de modo que ele informe o nome do computador atual, o nome do diretório de trabalho no momento e o número do comando atual. Observe: (exemplo tirado do livro “Aprenda em 24 horas Unix”)

```
%set prompt= “$system($cwd:t)(\!)%”
limbo(taylor)(26)% _
```

A variável cwd com a opção **:t** informa somente o nome do diretório atual, e não seu caminho inteiro, diminuindo um pouco mais o tamanho do *prompt.* No *Shell Korn* só trocaria **set prompt** por **PS1**.

Logo a seguir, são resumidos, numa tabela, os valores especiais para o *prompt* do sistema (tabela retirada do livro *“Aprenda em 24 horas Unix”)*:

Valor	Resultado
´cwd´	O diretório de trabalho atual.
\!	O número do comando atual.
$var	O valor de var.
^a^b	O final (última palavra) do valor de var.

O capítulo 2 chega a seu fim. Ele se destinou a dar maiores informações sobre alguns recursos importantes dos *Shells C* e *Korn*, como os aliases e histórico de comandos, por exemplo. Tendo em mãos a forma de utilizar tais recursos, o trabalho do usuário no Unix será bastante facilitado.

Capítulo 3

Programação de *scripts* de *Shell*

No capítulo anterior, alguns fortes recursos do *Shell C* e do *Shell Korn,* como o uso de aliases, do mecanismo de histórico de comandos e a definição de *prompts* personalizados, foram discutidos. Entretanto, para o uso de todos eles precisa-se se escrever muitas linhas de comandos, o que não é muito agradável, principalmente quando há muitos aliases ou variáveis a serem definidas. É nessa hora que os scripts de shell entram para resolver tal problema. Estes nada mais são que arquivos que contém uma série de comandos de shell. Quando alteradas as permissões dos *scripts* para inserir a permissão de execução, eles agem como qualquer outro comando do Unix.

Para se inserir a permissão de execução a um arquivo script, basta digitar a seguinte linha de comando:

```
% chmod 777 nome-do-arquivo
```

Para se criar programas-scripts, pode-se utilizar qualquer editor de texto disponível no Unix, como o **vi**, por exemplo.

O *shell,* no Unix, possui duas funções, que são: interpretar os comandos de linha digitados pelo usuário e atuar como linguagem de programação quando os comandos são inseridos num arquivo de lote (*scripts*).

Os *shells* mais utilizados do Unix para se desenvolver *scripts* são o *Shell C*, *Shell Bourne* e o *Shell Korn*, pelo fato de terem mais recursos que outros.

Definindo variáveis

A definição de uma variável dentro dos *scripts* se dá da mesma maneira que se faz na linha de comando, ou seja, quando se atribui um valor qualquer a uma determinada palavra, esta se torna uma variável (em alguns casos dentro do *script*, porém, com o uso de certos comandos, a definição de uma variável se dá com o uso de tais comandos, como por exemplo, com o comando **read**, que será visto mais à frente). Por exemplo, no *Shell C*, para se definir uma variável, como já se sabe, usa-se a seguinte linha de comando:

```
% set var=valor
```

No *Shell Bourne* seria **var=valor** (que também já foi explicado).

Portanto, nos *scripts* não é necessária uma predefinição de variáveis antes de sua utilização, a partir da primeira atribuição de valor ela já passa a existir.

Exemplos:

```
set var_num=10
word="MUNDO"
var_vetor=( um dois tres )
```

Outro modo comum de se fazer atribuição de variáveis é a partir de argumentos da linha de comando. O *shell* possui variáveis específicas para acessá-la, as quais possuem a forma **$***?*, sendo que*?* vai de 1 a 9. Observe o exemplo a seguir, que já considera a existência de um *script* de soma:

```
% script_soma  50  100
```

No exemplo acima, $1 é igual a 50 e $2, a 100, num *script* supostamente já criado para se somar números.

Nota-se que não há dificuldade alguma em se criar variáveis no Unix, seja através da linha de comando ou dentro dos *scripts*, e isso é uma grande vantagem quando se está fazendo um programa-*script*, principalmente quando ele for muito complexo.

Operadores

Dentro dos programas-scripts, pode se necessitar de determinadas operações para se gerar um ou mais resultados, como fazer comparações de variáveis, adicionar valores às variáveis, etc.

Logo a seguir estão relacionados alguns operadores comumente usados para escrever *scripts* de *shell*. Muitos deles já estão presentes em algumas linguagens de programação.

- Operadores de comparação usados pelo *csh*, *sh* e *ksh* (e suas variantes):

  z-gt maior que
  -lt menor que

-eq igual a
-o ou

- Operadores de comparação usados somente pelo *csh*

!= diferente
= atribui valor
== igual a
! negação booleana
&& e booleano
!! ou booleano
> maior que
< menor que
>= maior igual
<= menor igual

- Operadores para variáveis inteiras

+ adição
- subtração
++ soma 1
% módulo
* multiplicação
/ divisão
-- decrementa 1

- Operadores para bits

>> deslocamento para a direita
<< deslocamento para a esquerda
~ complemento
! negação lógica
| ou inclusive
^ ou exclusivo
& e

- Operadores que testam características de um arquivo ou diretório

-d verdadeiro se o arquivo for um diretório
-e verdadeiro se o arquivo existir
-f verdadeiro se o arquivo existe e é regular (texto)
-o verdadeiro se o usuário for o dono
-r verdadeiro se o arquivo pode ser lido pelo usuário

-w verdadeiro se o arquivo pode ser alterado pelo usuário
-x verdadeiro se o arquivo pode ser executado pelo usuário
-z verdadeiro se o arquivo estiver vazio

Entre esses operadores para arquivos e diretórios, os mais usados são os *flags* **–f** e **–d**, sendo que os outros são usados em situações extraordinárias.

Pode se ver que é uma lista extensa de operadores, que possibilitam criar diversos tipos de *scripts*, principalmente quando combinados com comandos de controle de fluxo, de *loop* e outros.

Comandos de controle de fluxo

Como o próprio nome já diz, os comandos de controle de fluxo controlam o fluxo de execução do programa, ou seja, geram saídas de acordo com as entradas.

Em seguida, são explicados alguns desses comandos para se escrever *scripts*.

if...fi

O **if** é o controle de fluxo mais utilizado na programação de scripts, e tem a seguinte forma:

```
if
   bloco-de-comandos
then
   bloco-de-comandos
fi
```

Um *bloco-de-comandos* é uma sequência de um ou mais comandos do *shell*. O primeiro bloco será sempre executado, e o segundo dependerá do resultado do primeiro. Veja alguns exemplos, lembrando que o símbolo **$** quer dizer “valor de” e o símbolo **#** significa que o que vem à frente dele é um comentário e não tem valor como parte da programação.

Exemplo 1:

```
var=10
if
```

```
   [ $var  -gt  0 ]
then
   echo $var é maior que zero    # emite-se uma mensagem
fi
```

Exemplo 2:

```
file="arquivo.txt"          # file recebe o nome do arq. arquivo.txt
if
   [ -f  $file ]
then
   echo $file é um arquivo regular
fi
```

O exemplo 1 é bem simples, no qual se utiliza o comando **if** para verificar se a variável var, que inicialmente vale 10, é maior que zero e, se for, como é, no caso aqui apresentado, transmite a mensagem afirmativa. Já no exemplo 2, se o arquivo (valor de file) for regular, **if** informa ao usuário, caso contrário ele não exibe nada. Porém, isso pode ser mudado, fazendo com que se exiba uma mensagem mesmo quando o arquivo não for regular, utilizando-se a palavra-chave **else** logo após **if**:

```
file=$1          # file recebe o argumento 1 da linha de comando
if
   [ -f  $file ]
then
   echo $file é um arquivo regular
else
   echo $file não é um arquivo regular
fi
```

É possível ainda realizar-se mais de uma comparação, usando outra palavra-chave, que é **elif**:

```
file=$1          # file recebe o argumento 1 da linha de comando
if
   [ -f  $file ]
then
   echo $file é um arquivo regular
elif
   [ -d  $file ]
then
   echo $file é um diretório
else
   echo $file não é um arquivo regular ou diretório
fi
```

Nesse último exemplo, **if** verifica se o arquivo é regular e caso seja, emite uma mensagem afirmando tal condição, senão, se o arquivo for, na verdade, um diretório, a mensagem relativa a essa condição será exibida, e, finalmente, se o arquivo não for regular nem diretório, uma mensagem referindo-se a esse resultado é que será mostrada na tela.

Pode-se, então, expandir qualquer instrução **if** para um número ilimitado de ramificações com **elif**.

case...esac

Quando se usa o **if** e nele há muitas ramificações (**elif**), o código ficará um pouco confuso, e é nesse momento que o uso do comando **case** se encaixa. Observe sua sintaxe, que é um pouco mais complicada que a do comando **if**, mas ainda compreensível:

```
case variável in
padrão 1) bloco-de-comandos;;
padrão 2) bloco-de-comandos;;
...
esac
```

Caso estivesse procurando valores possíveis para uma variável, poderia usar-se o **case**:

```
echo O que você quer:
read var          # aguarda entrada de dados
case $var in
veiculo) echo O preço é alto;;
casa) echo O preço é muito alto;;
*) echo O preço é desconhecido;;
esac
```

Nesse exemplo, **case** é executado após uma solicitação de entrada e dá o resultado (o preço no caso) de acordo com tal entrada. Essa lista poderia ser bem maior caso o algoritmo precisasse.

goto

O comando **goto** desvia o fluxo de execução para um ponto específico (label) de um *script*. Os nomes dos *labels* são sucedidos por dois pontos e devem aparecer sozinhos na linha.

Esse comando é muito usado em programas muito extensos, em que há a necessidade de dividi-lo em vários *labels* para melhor compreensão.

Observe o exemplo abaixo, com esse comando:

```
var=20
if
  [ $var  -gt  0 ]
then
  echo $var é maior que zero
  goto resultado
else
  echo $var não é maior que zero
fi

# o label que o comando goto chama
resultado:
  echo "Resultado positivo !"
  exit
```

Nesse exemplo, **if** verifica se o valor de var é maior que zero, se for, emite-se uma mensagem e redireciona-se o fluxo de execução para o label resultado, que também exibirá uma mensagem e terminará o programa com o comando **exit**. Só para esclarecer, o comando **echo** pode exibir mensagens tanto sem aspas como entre aspas; o resultado será sempre o mesmo.

switch...endsw

O comando ***switch*** possui uma sintaxe semelhante à do comando ***case***, só que em vez de executar um bloco de comandos, direciona o fluxo de execução para um *label* do *script,* sendo que esse *label* está relacionado a uma determinada condição do ***switch***. Dentro deste**,** existe o comando ***breaksw***, que direciona o controle para o comando seguinte, ao fim do ***switch*** *(**endsw**)* . Caso não haja nenhum *label* correspondente à condição especificada, o controle é transferido para o *label default,* se estiver definido, senão a execução continua após o comando ***endsw***. Veja uma exemplo de *script* com o controle de fluxo ***switch*** a seguir:

Exemplo:

```
echo "Opção 1 – Somar"
echo "Opção 2 – Sair"
read varop      # aguarda entrada de dados para varop
```

```
switch($varop)
  case 1:
        soma
        breaksw
  case 2:
        breaksw
  default:
        echo “Digite opção válida”
endsw
exit

soma:
  read varnum1
  read varnum2
  echo “O valor da soma é:”
  expr $varnum1 + $varnum2   # faz a soma das variáveis
)
```

No exemplo dado, primeiramente se lê a variável *varop*. Logo após, o comando ***switch*** avalia o valor de tal variável. Caso seja 1, será chamado o *label* soma, que irá ler (através do comando ***read***) duas variáveis e fazer a soma delas através do comando ***expr*** (que será comentado mais à frente). Caso *varop* seja 2, o controle de execução será direcionado para o comando ***exit***, e se *varop* não possuir nenhum dos dois valores, será executado o label default, o qual emitirá uma mensagem de opção inválida.

Expressões de *loop*

De vez em quando, aparecem programas em que há a necessidade de se executar várias vezes um mesmo conjunto de comandos e, em vez de escrevê-los várias vezes, utilizam-se os comandos de *loop* (ou de laços). E na programação, em geral, há dois tipos de *loop:* o *determinado* e o *indeterminado*.

Um *loop determinado* é aquele em que se sabe o valor exato de quantas vezes irá se executar os comandos, antes mesmo de se inseri-lo o *loop*. Já o *indeterminado* é aquele em que é necessário se executar um conjunto de comandos até que uma determinada condição não seja mais verdadeira.

Logo a seguir, serão explicados os comandos de *loop* mais usados nos *scripts* de *shell*, que são o **for...done** e o **while...done**.

for...done

O comando voltado para um *loop determinado* nos *scripts* de *shell* é o comando ***for***. Outras linguagens de programação (pelo menos a maioria delas), possuem esse comando. Sua sintaxe para o uso em *scripts* tem a seguinte forma:

```
for var-in-list
do
   bloco-de-comandos
done
```

Nos scripts de *shell,* esse comando é bastante utilizado para listas de arquivos, sendo bastante eficiente, pois essas listas, geralmente, são extensas. Veja um exemplo com o comando de *loop* ***for*** numa determinada lista de arquivos de um diretório:

Exemplo:

```
for var in ´ls´
do
   if
     [ -f  $var ]
   then
     echo $var é um arquivo regular
   fi
done
```

Nesse exemplo, a saída fornece uma lista com todos os arquivos regulares, sendo que var é a variável que irá ser usada dentro do *loop* e que irá fornecer o nome de cada arquivo regular da lista. Vale observar que a lista utilizada para se fazer tal pesquisa é obtida de uma comando **ls** no diretório corrente.

Se a parte *´in list´* for omitida do comando **for**, ele irá percorrer a lista de argumentos da linha de comando. Veja o exemplo:

Exemplo:

```
j=0
for i   # a variável i é definida dentro do laço for
do
   j=´expr $j + 1´          # faz a soma de j=j+1
   echo $i é o argumento $j
done
```

Nesse último exemplo, o comando **for** percorre os argumentos da linha de comando e identifica a ordem em que eles estão, ou seja, fornece a descrição do argumento e seu número de ordem na linha de comando.

while...done

No item anterior, quando se usou o comando **for**, já se sabia o número de vezes que um bloco de comandos seria executado. Mas, existem casos em que se precisa executar blocos de comandos até que algo aconteça, e é aí que o comando **while** se encaixa. A sintaxe do comando **while** é a seguinte:

```
while condição
do
    bloco-de-comandos
done
```

Um exemplo de *loop indeterminado* é quando se está esperando a chegada de um arquivo em seu diretório, mas não se sabe quando poderá chegar. Esse arquivo pode ser enviado pelo servidor ou por outro usuário que tenha permissão para gravar dados em sua área.

Com o uso do *loop* ***while,*** pode-se resolver essa situação. Dentro da *condição*, após o comando ***while****,* faz-se uma verificação se o primeiro arquivo é mais recente que o segundo; se for, aguarda-se mais 60 segundos e faz-se outra verificação. O primeiro arquivo, nesse caso, seria um arquivo qualquer de seu diretório, que seria usado apenas como ponto de verificação com o segundo arquivo, aquele é o arquivo que se está aguardando chegar em seu diretório. Para se fazer tal verificação, usa-se o *flag* **–nt**, que retorna verdadeiro se o primeiro arquivo for mais recente que o segundo. Veja como ficaria em termos de programação:

```
while
        [ prog.c  -nt  arqcheg ]
do
        sleep 60            # aguarda 60 segundos
done
```

Esse laço (*loop*) do programa, portanto, esperaria um minuto entre as verificações. Se o arquivo arqcheg ainda não estiver inserido no diretório, o programa esperará mais um minuto ainda.

Os loops **while** também podem ser usados de forma *determinada*, como o comando **for**. Observe o exemplo a seguir:

Exemplo:

```
# este script conta de 20 a 1
var_num=20
while
        [ $var_num  -gt  0 ]
do
        echo $var_num
        expr $var_num – 1          # subtrai 1 de var_num
done
```

Essa sequência anterior será executada 19 vezes, exibindo o valor da variável *var_num* na tela enquanto tal variável for maior que zero.

Outros comandos usados nos *scripts*

Na construção dos *scripts*, também são utilizados outros comandos auxiliares além dos de controle de fluxo e os de *loop*. Tais comandos auxiliares são tão úteis quanto os que foram explicados até aqui, pois são eles que dão o complemento aos comandos de fluxo e de *loop*. Os comandos auxiliares podem ser usados tanto nos *scripts* quanto em linhas de comandos.

Logo a seguir, serão explicados alguns deles.

clear

O comando ***clear*** limpa o conteúdo da tela e posiciona o cursor no canto superior esquerdo do monitor. Ele é um dos comandos mais usados nos scripts, principalmente no início de cada um, pois assim, a tela anterior é apagada e só o resultado do *script* é exibido.

echo

O comando **echo**, que já foi explicado e muito usado anteriormente, exibe *strings* ou conteúdo de variáveis de memória na tela. Quando for exibir o conteúdo de variáveis, utiliza-se o símbolo **$** antes delas. O comando **echo** exibe *strings* ou conteúdo de variáveis tanto entre aspas como sem aspas. Veja exemplos:

Exemplos:

1)

```
echo $variavel
```

ou

```
echo “$variavel”
```

2)

```
echo Isto é um teste
```

ou

```
echo “Isto é um teste”
```

O comando **echo** também possui alguns parâmetros importantes:

-n não gera avanço de linha após exibição
\\b retrocesso
\\f alimentação de página
\\n avanço de linha (default)
\\r carriage return (retorno do carro de impressão da impressora)
\\t tabulação

exit

O comando **exit** abandona o fluxo do *script*, ou seja, ele quebra a sequência de execução de *script* e retorna ao sistema operacional. Todo final de *script* deve possuir esse comando.

Quando usado diretamente na linha de comando, ele faz a desconexão do terminal com o servidor ou, se o usuário estiver usando um *subshell*, ele retorna ao *shell* de *login.*

expr

O comando **expr**, que também foi muito usado anteriormente, tem como função executar uma operação aritmética numa variável de memória ou diretamente em números fornecidos pelo usuário.

Abaixo estão listados dois exemplos, sendo que o primeiro é executado a partir da linha de comando e o segundo, dentro de um *script* qualquer:

Exemplos:

1)

```
$ expr 3 + 2
  5
$_
```

2)

```
# script de subtração
var=10
expr $var – 5
```

Operações possíveis com **expr**:

+ adição
* multiplicação
% resto
- subtração
\ divisão

read

Antes de mais nada, a sintaxe do comando **read** é a seguinte:

```
read nome-da-variável
```

O comando ***read*** fica aguardando uma entrada de dados pelo usuário e, logo após, atribui o valor dessa entrada à variável especificada.

Esse também foi um comando que se utilizou anteriormente. Veja mais um exemplo dele:

Exemplo:

```
# script que aguarda a entrada de uma expressão e
# verifica se é igual a palavra “acertou”

echo “Digite algo: ”
read algo
if
   [ $algo = “acertou” ]
then
   echo Você acertou a palavra!
else
   echo Você errou a palavra!
fi
exit
```

A variável que será usada como parâmetro do comando ***read*** não precisa ser definida anteriormente, pois ela receberá um valor só quando for feita uma entrada de dados e aí, irá possuir uma definição completa, como no exemplo que foi dado, no qual a variável *algo* não possui uma definição anterior, mas aguarda uma entrada de dados e depois efetua uma comparação para ver se é igual à palavra “acertou” ou não.

sleep

A sintaxe do comando **sleep** é a seguinte:

sleep *numero-de-segundos*

Esse comando aguarda um determinado tempo em segundos para continuar a execução normal do *script* (ou de comandos, caso seja utilizado a partir da linha de comando). Para quem se lembra, ele foi utilizado no exemplo da expressão de *loop* ***while...done***,em que se aguardava intervalos de 60 segundos para a gravação de um arquivo no diretório do usuário.

E, com a explicação desse último comando, chega-se ao final do capítulo 3, que tratou da programação básica para *shell* (válida tanto para o *Shell C, Shell Bourne, Shell Korn e* as variantes desses três). Foram tratados e explicados alguns comandos úteis para a programação de *scripts*, como os de controle de fluxo (***if...fi, goto,*** etc.), os de *loop* (***for...done, while... done***) e outros auxiliares *(**clear**, **exit, expr,*** etc.), além do uso de operadores e tratamento de variáveis. O principal objetivo desse capítulo não foi mostrar como ser um extraordinário programador de *scripts*, mas sim, explicar a finalidade e os fundamentos básicos de como se trabalhar e utilizar corretamente os *scripts* de *shell.*

Capítulo 4

Controle de Serviço

Já foi dito, anteriormente, que o Unix é um sistema operacional multitarefa, o que quer dizer que ele pode executar mais de um programa ao mesmo tempo. Um programa em execução no Unix é denominado *processo* (ou *serviço*). Mais precisamente, o sistema só pode executar um programa em um dado instante, entretanto, ele muda tão rapidamente de um processo para outro que, na maioria das vezes, parece que tudo é executado ao mesmo tempo. Quando o sistema estiver sendo usado por muitos usuários ou estiver com a carga pesada por alguma outra razão, os programas parecerão demorar mais do que o normal para serem executados, e, nesse momento, entram os comandos de controle de serviço. Vale ressaltar que tanto o *Shell C* quanto o *Shell Korn* possuem tais comandos, além de suas variantes.

Com os comandos de controle de serviço é possível suspender programas em execução e depois reiniciá-los; colocar programas para serem executados no segundo plano, enquanto se executam outros no primeiro; chamar programas do segundo para o primeiro plano ou mesmo encerrá-los. Enfim, uma série de controles que facilitam a vida do usuário.

Interrompendo serviços

Como já foi dito, no Unix, qualquer programa que esteja sendo executado se constitui num *processo*. No *Shell C* e no *Shell Korn*, os processos são conhecidos também como *serviços,* e o programa executado no momento é conhecido como *serviço atual*.

Tanto no *Shell C* quanto no *Shell Korn* (ou mesmo nas variantes destes), para se interromper um processo (ou serviço) em execução, deve-se pressionar **^z** (control + z). Para reiniciar o processo novamente, utiliza-se o comando ***fg***.

Esse tipo de controle de serviço é, geralmente, usado para comandos que geram saídas muito longas, como ***ls*** (listar o conteúdo dos diretórios), ***cat*** (exibem o conteúdo de um arquivo) e ***man*** (exibir uma documentação de ajuda do comando especificado), além de outros.

Em seguida, é transcrito um exemplo de comando em que a saída é muito longa. Esse comando é o ***man***, que, neste caso, irá mostrar a documentação do comando ***ls***.

Exemplo:

```
% man ls
LS(1)                                   LS(1)

NAME
       ls, dir, vdir - list contents of directories

SYNOPSIS
       ls  [-abcdfgiklmnpqrstuxABCFGLNQRSUX1] [-w cols] [-T cols]
       [-I pattern] [--all]  [--escape]  [--directory]  [--inode]
       [--kilobytes]  [--numeric-uid-gid]  [--no-group]  [--hide-
       control-chars] [--reverse] [--size] [--width=cols] [--tab-
       size=cols]  [--almost-all] [--ignore-backups] [--classify]
       [--file-type] [--full-time] [--ignore=pattern] [--derefer-
       ence]     [--literal]     [--quote-name]     [--recursive]
       [--sort={none,time,size,extension}]   [--format={long,ver-
      bose,commas,across,vertical,single-column}]
       [--time={atime,access,use,ctime,status}] [--help]  [--ver-
       sion]  [--color[={yes,no,tty}]]  [--colour[={yes,no,tty}]]
       [name...]

DESCRIPTION
       This documentation is no longer being maintained  and  may

--More-- _
```

No exemplo dado, observa-se que há mais informações a serem fornecidas sobre o comando **ls**, pois aparece a palavra **more** no final da tela (em alguns sistemas pode aparecer**:** em vez da palavra **more**), e não **END**, que indica o final da documentação de um comando. Sendo assim, se o usuário quiser terminar a execução, ele pode pressionar **q,** e o programa **man** será finalizado, mas, se ao invés disso, ele preferir interrompê-lo temporariamente e ver o restante da documentação de **ls** num outro momento, pode pressionar **^z**. Veja o que acontecerá:

```
% man ls
LS(1)                                   LS(1)

NAME
       ls, dir, vdir - list contents of directories

SYNOPSIS
       ls  [-abcdfgiklmnpqrstuxABCFGLNQRSUX1] [-w cols] [-T cols]
```

```
      [-I pattern] [--all]  [--escape]  [--directory]  [--inode]
      [--kilobytes]  [--numeric-uid-gid]  [--no-group]  [--hide-
      control-chars] [--reverse] [--size] [--width=cols] [--tab-
      size=cols]  [--almost-all] [--ignore-backups] [--classify]
      [--file-type] [--full-time] [--ignore=pattern] [--derefer-
     ence]    [--literal]    [--quote-name]    [--recursive]
      [--sort={none,time,size,extension}]  [--format={long,ver-
      bose,commas,across,vertical,single-column}]
      [--time={atime,access,use,ctime,status}] [--help]  [--ver-
      sion]  [--color[={yes,no,tty}]]  [--colour[={yes,no,tty}]]
      [name...]

DESCRIPTION
      This documentation is no longer being maintained  and  may

--More-- _
Stopped
```

A palavra ***Stopped*** (interrompido) é exibida após o usuário ter interrompido a execução do programa. Alguns sistemas podem exibir outras mensagens ou não exibir nada, mas a interrupção do programa será feita em todos.

Em seguida, o usuário poderá executar outros comandos, se desejar. Quando ele resolver olhar o restante da saída do comando ***man,*** interrompido com **^z**, poderá, então, usar o comando ***fg****,* que retornará ao ponto em que o programa foi suspenso.

```
% fg
man ls
LS(1)                                   LS(1)

NAME
      ls, dir, vdir - list contents of directories

SYNOPSIS
      ls  [-abcdfgiklmnpqrstuxABCFGLNQRSUX1] [-w cols] [-T cols]
      [-I pattern] [--all]  [--escape]  [--directory]  [--inode]
     [--kilobytes]  [--numeric-uid-gid]  [--no-group]  [--hide-
      control-chars] [--reverse] [--size] [--width=cols] [--tab-
      size=cols]  [--almost-all] [--ignore-backups] [--classify]
      [--file-type] [--full-time] [--ignore=pattern] [--derefer-
      ence]    [--literal]    [--quote-name]    [--recursive]
      [--sort={none,time,size,extension}]  [--format={long,ver-
      bose,commas,across,vertical,single-column}]
      [--time={atime,access,use,ctime,status}] [--help]  [--ver-
      sion]  [--color[={yes,no,tty}]]  [--colour[={yes,no,tty}]]
      [name...]
```

```
DESCRIPTION
       This documentation is no longer being maintained  and  may

--More-- _
```

A saída é a mesma que a anterior, isso porque foi o ponto em que o comando ***man*** foi interrompido, e a partir do qual poderá continuar. O nome do comando ***fg*** é uma abreviação para *foreground*, que quer dizer *primeiro plano*, sendo que primeiro plano é uma referência ao programa com o qual o monitor e o teclado estão trabalhando.

Primeiro plano / segundo plano

Às vezes, o usuário tem que utilizar programas que possuem um tempo de execução um pouco extenso, se for necessário realizar outras tarefas simultaneamente, pode-se interferir nesse execução. Entretanto, ele pode chamar um programa para ser executado no *segundo plano* (*background*), enquanto se utiliza de outros comandos no *primeiro plano* (*foreground*).

A grande vantagem de se rodar programas no segundo plano é que, ao contrário dos processos no primeiro plano, os processos do segundo plano não precisam que sua execução seja encerrada para que outro processo possa ser iniciado. Logo após ter inserido um processo no segundo plano, o *shell* em execução exibirá o *prompt* para sinalizar que se pode chamar um outro comando no plano principal (ou mesmo no segundo plano, novamente).

Antes de se explicar como executar programas em segundo plano, é importante ressaltar alguns pontos sobre o funcionamento desse recurso. Um deles é o fato de que o Unix, embora permita executar mais de um processo no segundo plano, possui um limite de execuções em *background*, que é, geralmente, entre 20 e 50 por usuário e entre 100 e 250 para o sistema todo. Outro ponto importante é que, quanto mais processos estiverem sendo executados em segundo plano, mais lento o sistema ficará, fazendo com que o tempo consumido seja igual ao da sua execução em primeiro plano. Um último ponto é sobre as saídas que os processos em segundo plano, quando encerrados, fornecem. Elas, geralmente, aparecem misturadas na tela com as saídas de processos executados no plano principal e até de outros processos em segundo plano. Dessa forma, pode-se evitar

tal problema redirecionando as saídas dos processos em segundo plano para arquivos em disco. Para se fazer isso, utiliza-se sinal de maior (**>**) mais o nome do arquivo para o qual a saída será enviada. Um exemplo de como se fazer redirecionamento é dado abaixo:

Exemplo:

```
% ls > saida.txt
```

É importante observar, também, que não se deve iniciar um processo no segundo plano que leia a entrada do teclado, isso porque, quando se estiver utilizando um programa no plano principal que também requeira entrada do teclado, haverá conflitos entre os dois processos.

Sendo assim, existem duas formas para se chamar programas em segundo plano: uma pode ser feita diretamente, ou seja, desde o início de sua execução e outra a partir do ponto em que um programa foi interrompido (**^z**).

Para se **iniciar** um programa em segundo plano, basta digitar o símbolo **&** no final da linha de comando. Com isso, o Unix irá gerenciá-lo desde o seu início até seu fim. Como exemplo, pode-se utilizar o comando **du,** executado no segundo plano para se verificar o espaço total em disco:

Exemplo:

```
% du –s / > arq.du &
290
%_
```

O número que foi mostrado como saída nada mais é que o *número do processo* (não é o espaço usado do disco) que está sendo executado e, que nesse caso, é 290. A opção **–s** do comando **du** faz com que ele exiba apenas o total usado em disco e não o espaço ocupado por cada arquivo e diretório também. Já a barra (**/**) faz com que ele pesquise o disco todo.

Alguns sistemas podem exibir, além do número do processo, o *número do controle de serviço*, que é o número de ordem dos **comandos de controle de serviço**. Veja:

Exemplo:

```
% du –s / > arq.du &
[1] 290
%_
```

O número do controle de serviço é o que está entre colchetes e que, neste caso, é 1. Tanto o número do processo quanto o número do controle de serviço são importantes para a explicação dos comandos restantes. Para se saber, então, o espaço em disco ocupado, é só consultar o arquivo arq.du depois de um certo tempo, logo após **du** ter sido executado em segundo plano. Observe:

```
% cat arq.du
26106
%_
```

Com o uso do comando **cat** para observar o conteúdo do arquivo arq.du, pôde-se obter o total de 26106 *kilobytes* usados em disco.

Existe também a possibilidade de se continuar a execução em segundo plano de programas interrompidos com **^z**. Em vez de chamá-los para o primeiro plano com o comando **fg**, utiliza-se o comando **bg** (*background*) para enviá-lo para o segundo plano e dar continuidade a sua execução.

O mesmo exemplo da linha de comando anterior pode ser usado, só que sem o símbolo **&** no final da linha:

Exemplo:

```
% du –s / > arq.du
Stopped
%_
```

Nota-se que o processo foi interrompido com **^z**, pois a palavra **Stopped** é exibida logo abaixo da linha de comando. Já se sabe que, para se trazer esse processo para ser executado no plano principal, utiliza-se o comando **fg**, mas também foi dito como levá-lo para o segundo plano, que é usando o comando **bg**. Veja o exemplo abaixo:

Exemplo:

```
% bg
[2] du –s / > arq.du
%_
```

Pelo exemplo, pode se ver que o comando **bg** exibe, como saída, o número do controle de serviço entre colchetes (n° 2) mais a linha de comando que foi interrompida e será executada em *segundo plano*. Em alguns sistemas esse número de controle de serviço pode ser trocado pelo número do processo.

É importante ressaltar que, até por uma questão de simplicidade, o artifício de se executar programas interrompidos em segundo plano é menos usado que o de se executar programas desde o início em segundo plano.

Obtendo informações sobre as tarefas que estão sendo executadas

Agora que já foi visto como interromper processos e executá-los em primeiro ou segundo plano, será, então, explicado como se obter informações sobre eles. Para se fazer isso, utiliza-se do comando **ps** ou do comando **jobs,** cujas saídas informam sobre o estado dos processos, mas somente dos que estão em segundo plano ou interrompidos.

O primeiro a ser explicado será o comando **ps**, mas, antes de tudo, serão fornecidos alguns exemplos de processos para que possam ser utilizados tanto na explicação de **ps** quanto na de **jobs**. O primeiro exemplo será de um comando executado e interrompido em seguida, o segundo será de um comando executado no plano principal, e o terceiro, de um comando sendo executado em segundo plano.

Exemplo 1: (Já usado anteriormente)

```
% man ls
LS(1)                                   LS(1)

NAME
       ls, dir, vdir - list contents of directories

SYNOPSIS
       ls  [-abcdfgiklmnpqrstuxABCFGLNQRSUX1] [-w cols] [-T cols]
       [-I pattern] [--all]  [--escape]  [--directory]  [--inode]
       [--kilobytes]  [--numeric-uid-gid]  [--no-group]  [--hide-
       control-chars] [--reverse] [--size] [--width=cols] [--tab-
       size=cols]  [--almost-all] [--ignore-backups] [--classify]
       [--file-type] [--full-time] [--ignore=pattern] [--derefer-
       ence]     [--literal]     [--quote-name]     [--recursive]
       [--sort={none,time,size,extension}]   [--format={long,ver-
       bose,commas,across,vertical,single-column}]
       [--time={atime,access,use,ctime,status}] [--help]  [--ver-
       sion]  [--color[={yes,no,tty}]]  [--colour[={yes,no,tty}]]
      [name...]
```

```
DESCRIPTION
      This documentation is no longer being maintained  and  may

--More-- _
Stopped
%_
```

Exemplo 2:

```
% cat prog.c
#include <stdio.h>
main()
{
int i;
            for ( i=0; i<=4; i++ )
                printf("Número: %d", i+1);
return;
}
%_
```

Exemplo 3: (Também já usado anteriormente)

```
% du –s / > arq.du &
[2] 660
%_
```

Considerando-se os exemplos dados, pode-se, então, utilizar o **ps**:

```
% ps
PID   TTY   STAT   TIME   COMMAND
639   P0     S      0:00    csh
652   P0     T      0:00    man ls
660   P0     R      0:04    du –s /
667   P0     R      0:00    ps
```

Observa-se que a saída só mostrou dados do comando interrompido e do que está sendo executado em segundo plano, além do nome do *shell* em execução (csh) e o nome do próprio comando **ps**, que acabou de ser utilizado. Sobre o comando **cat**, do exemplo 2, não foi exibido nenhuma informação, isto porque ele foi executado normalmente, isto é, em primeiro plano.

O significado de cada campo da saída de **ps** é explicado abaixo:

PID: número do processo

TTY: número do terminal

STAT: o estado atual do processo

TIME: tempo efetivo de execução do processo em minutos e segundos

COMMAND: nome do comando (ou parte da linha de comando)

O campo STAT, seus possíveis valores e seus respectivos estados são listados na tabela seguinte:

Possíveis valores de status do processo (STAT)

Valor	Significado
R	Executando
S	Dormindo (20 segundos ou menos)
I	Inativo (“dormindo” mais de 20 segundos)
T	Interrompido

Fonte: Taylor, Armstrone Junior, 1998.

No exemplo dado do comando **ps**, o estado do processo número 639 (**Shell csh**) é *dormindo*, ou seja, está a 20 segundos ou menos sem ser executado, mas não está *interrompido*. Caso ele tenha que interpretar um comando, por exemplo, voltará ao estado *executando*, mas, rapidamente, retornará ao estado *dormindo*. O processo 667 (o próprio comando **ps**) está no estado *executando*, pois quando a saída foi listada, **ps** ainda estava sendo executado. Já o processo de PID 652 (**man ls**) está como *interrompido*, como realmente deveria estar, enquanto que o do comando **du** (PID 660) está como *executando*, como também deveria estar, e o tempo de execução dele já chegou a 4 segundos. Caso seja digitado **ps,** novamente, mais o número de processo relativo ao comando **du** (660, neste caso), será exibido um tempo de execução maior, isto porque sua execução ainda está em progresso. Veja:

```
% ps 660
PID   TTY   STAT   TIME   COMMAND
660   P0     R       0:10   du –s /
%_
```

O tempo passado de execução do comando **du**, agora, é equivalente a 10 segundos. Quando sua execução estiver terminada, a saída será a seguinte:

```
% ps 660
PID   TTY   STAT   TIME   COMMAND
%_
```

A saída não mostrou nada, pois o processo já foi encerrado. Com isso, se for listar o conteúdo do arquivo arq.du (**cat**), será exibido o resultado de todo o processo, ou seja, o espaço usado em disco.

Como já se pôde notar, o comando **ps** não exibe o número do controle de serviço. Somente o número do processo é listado.

O comando **jobs**, ao contrário de **ps**, exibe o número do controle de serviço no lugar do número do processo. Esse comando fornece uma saída um pouco mais resumida do que o **ps**, mas com informações suficientes para se saber o estado de um processo. A saída de **jobs** para os exemplos dados é a seguinte:

```
% jobs
[1] + Stopped              man ls
[2]  - Running             du –s / > arq.du
```

A saída mostra os respectivos números (entre colchetes) de controle de serviço das linhas de comandos **man ls** e **du –s / > arq.du**, observando que eles obedecem à ordem de execução. Também é mostrado o estado em que cada processo está, sendo que o primeiro está interrompido e o segundo está em execução no segundo plano.

Outro fato que merece ser comentado neste momento, após a explicação de **ps** e **jobs**, é o de que, tanto o comando **fg** quanto o **bg**, também possuem parâmetros, ou seja, existe uma forma de executá-los sem ser sozinhos. Esses parâmetros podem ser o número relativo ao processo ou o número do controle de serviço, sendo que este último necessita ser especificado junto ao símbolo **%** antes dele. No caso de existirem vários processos interrompidos ou em segundo plano, o uso de parâmetros para **fg** ou **bg** é bastante útil e, utilizando-se os comandos **ps** ou **jobs**, é possível visualizar tais parâmetros.

Exemplo 1: (Utilizando o **ps**)

```
% ps
PID   TTY   STAT   TIME   COMMAND
639   P0     S      0:00    csh
652   P0     T      0:00    man ls
660   P0     R      0:04    du –s /
667   P0     R      0:00    ps
% fg 652
man ls

LS(1)                                   LS(1)
```

```
NAME
      ls, dir, vdir - list contents of directories

SYNOPSIS
      ls  [-abcdfgiklmnpqrstuxABCFGLNQRSUX1] [-w cols] [-T cols]
      [-I pattern] [--all]  [--escape]  [--directory]  [--inode]
      [--kilobytes]  [--numeric-uid-gid]  [--no-group]  [--hide-
      control-chars] [--reverse] [--size] [--width=cols] [--tab-
      size=cols]  [--almost-all] [--ignore-backups] [--classify]
      [--file-type] [--full-time] [--ignore=pattern] [--derefer-
     ence]    [--literal]    [--quote-name]    [--recursive]
      [--sort={none,time,size,extension}]  [--format={long,ver-
      bose,commas,across,vertical,single-column}]
      [--time={atime,access,use,ctime,status}] [--help]  [--ver-
      sion]  [--color[={yes,no,tty}]]  [--colour[={yes,no,tty}]]
      [name...]

DESCRIPTION
      This documentation is no longer being maintained  and  may

--More-- _

% fg 660 <-- Depois é só verificar o conteúdo do arquivo arq.du com cat
du –s / > arq.du
```

Exemplo 2: (Utilizando o **jobs**)

```
% jobs
[1] + Stopped              man ls
[2]  - Running             du –s / > arq.du
% fg %1 <-- mesma saída que o fg 652
% fg %2 <-- mesma saída que o fg 660
```

Terminando processos

Nos itens anteriores, foi explicado como trabalhar com processos interrompidos e em segundo plano e também como visualizar o estado em que estavam. Neste item, será abordado um comando para finalizar tais processos, ou seja, interrompê-los definitivamente. Esse comando é o **kill**, e sua sintaxe é bem simples:

kill *numero-do-processo*

ou

kill **%***numero-do-controle-de-serviço*

A necessidade de se utilizar esse comando pode aparecer em diversas situações, como no caso do usuário descobrir que um processo, em segundo plano, não está sendo executado corretamente ou no caso do sistema estar muito lento, por existir muitos processos interrompidos e outros em segundo plano.

Observe um exemplo simples abaixo, em que é criado um processo em segundo plano e depois ele é terminado:

Exemplo:

```
% man sort > sort.txt &
[1] 450
% kill 450
%_
```

ou

```
% kill %1
%_
```

Nesse exemplo, é pedido ao comando **man** para fornecer uma documentação do comando **sort,** com a saída redirecionada para o arquivo sort.txt, sendo todo processo executado em segundo plano. Feito isso, é exibido o número do controle de serviço mais o número do processo (lembrando que alguns sistemas podem listar somente o número do processo). Logo após, utiliza-se o comando **kill** para encerrar tal processo e, como parâmetro, pode ser passado tanto o número do controle de serviço (com o sinal **%** acompanhado) quanto o número do processo, que, nesse caso são, respectivamente, 1 e 450. Em alguns sistemas seria mostrada a mensagem **"450:Terminated"** (450 Terminado), mas em outros, nenhuma mensagem é exibida, devendo-se, então, usar os comandos **ps** ou **jobs** para saber se o processo realmente foi terminado. Usando o **ps** para verificar se os processos anteriores foram encerrados, obtém-se:

```
% ps
PID  TTY  STAT  TIME  COMMAND
405  P0   S     0:00  csh
700  P0   R     0:00  ps
```

Pela saída de **ps**, pode-se confirmar que o processo realmente foi encerrado.

É possível, também, ocorrer a necessidade do usuário querer se lembrar do número do processo ou do controle de serviço para finalizar um deter-

minado processo (interrompido ou em segundo plano). Ocorrendo, então, tal necessidade, ele pode utilizar o comando **ps** ou o comando **jobs** para se lembrar do número do processo ou de controle, respectivamente. Veja exemplos a seguir, os quais se utilizam dos exemplos de **ps** e **jobs** do item anterior:

Exemplos:

1. Pega a saída de **ps** e encerra um processo interrompido e outro em segundo plano:

```
% ps
PID   TTY   STAT   TIME   COMMAND
639   P0     S      0:00   csh
652   P0     T      0:00   man ls
660   P0     R      0:04   du –s /
667   P0     R      0:00   ps
% kill 652 660
%_
```

Verifica-se, então, a saída de **ps**:

```
% ps
PID   TTY   STAT   TIME   COMMAND
639   P0     S      0:00   csh
667   P0     R      0:00   ps
%_
```

Os processos foram, definitivamente, encerrados.

2. Utiliza-se, agora, a saída de **jobs** para encerrar os processos determinados:

```
% jobs
[1] + Stopped              man ls
[2]  - Running             du –s / > arq.du
% kill %1 %2
%_
```

Verificando-se a saída de **jobs** novamente:

```
% jobs
%_
```

Como não foi exibido nada, conclui-se que não há nenhum processo interrompido ou em segundo plano, portanto, o comando **kill** finalizou os processos especificados.

O capítulo 4 está, por aqui, finalizado. Nele foi discutido sobre um dos melhores recursos do Unix, que é o *controle de serviço*, presente nos *shells C* e *Korn*, além de algumas variantes destes e em *shells* mais recentes. Com os comandos de controle de serviço, como foi explicado, é possível se interromper a execução de comandos (processos), inserir processos em segundo plano, visualizar os estados dos processos ou mesmo encerrar processos interrompidos ou em segundo plano.

Conclusão

A estrutura do sistema operacional Unix é dividida em 3 partes: o *Kernel*, o *Shell* e os *Utilitários*. A ênfase deste trabalho é dada ao *Shell*, que é o interpretador de comandos do Unix.

No Unix existem vários *shells*, sendo que cada um deles possui características pertinentes e que os diferem uns dos outros. Não obstante, um *shell* também pode ter características que outros *shells* possuem. Os shells mais importantes do Unix são o *Shell C* (*csh*), *Shell Bourne* (*sh*) e o *Shell Korn* (*ksh*). Para se identificar o *shell* em uso, pode-se utilizar várias técnicas, sendo que uma delas é a linha de comando **ps | grep $$**. Já para se mudar o *shell* de *login*, ou seja, o *shell* padrão de quando se inicia o sistema, digita-se o comando **chsh**. No caso do usuário querer apenas alterar, momentaneamente, de um *shell* para outro, digita-se somente o nome do novo *shel*l. Os *shells* do Unix, como o *csh* e o *sh*, também possuem variáveis que servem para configurar o ambiente com o qual eles irão trabalhar, além de permitir a criação e a definição de outras pelo usuário. Essas variáveis também podem ser inseridas nos arquivos de inicialização de *shell*, nos quais pode-se estabelecer um ambiente inicial toda vez que se entrar no sistema.

Alguns *shells,* como os *Shells C* e *Korn* possuem alguns recursos interessantes em relação aos outros *shells.* Esses recursos são: o mecanismo de histórico de comandos, os *aliases* de comandos e a configuração de *prompts* personalizados. Com o mecanismo de histórico de comandos, é possível se lembrar de um programa anteriormente usado no sistema, digitando-se apenas alguns comandos curtos. Entre esses comandos está o **!!**, que executa o último programa utilizado. No entanto, no *csh*, para que esse mecanismo funcione bem, é preciso, antes, que a variável history esteja definida com o número de comandos que deverão ser lembrados para uma eventual execução de um comando de histórico. Com os aliases é possível dar pseudônimos a comandos (ou sequência de comandos) que oferecem uma certa dificuldade para o usuário, seja essa dificuldade de digitação ou de execução. Com esse recurso, pode-se dar nomes bem curtos para linhas de comandos muito extensas. Já o recurso de configuração de *prompts* permite ao usuário configurar o *prompt* de acordo com sua

vontade, possibilitando a inserção de textos, visualização do diretório de trabalho atual e número do processo atual. Esse recurso é de grande utilidade, pois a maioria dos sistemas Unix possuem *prompts* nada amigáveis.

O Unix também permite a construção de programas que contenham comandos de *shell*, chamados de *scripts de shell*. Aqueles mais utilizados para se escrever *scripts* são o *Shell C*, *Shell Bourne* e *Shell Korn*. Com esses *scripts*, é possível solucionar vários problemas com os quais o usuário pode se deparar durante seu trabalho no Unix. Dentro desses *scripts* pode-se colocar, além dos comandos usuais do shell, comandos de controle de fluxo, como o **if...fi**, e os comandos de *loop*, como o **for... done**. Também são utilizados alguns comandos auxiliares para incrementar ainda mais esses *scripts*, como o **expr**, por exemplo, que faz cálculos aritméticos. Tais comandos auxiliares também podem ser usados em linhas de comandos.

Como já foi dito, o Unix é um sistema operacional *multitarefa*, isto é, pode executar vários programas ao mesmo tempo. Na verdade, ele executa uma tarefa de cada vez, mas, com a rapidez com que ele as executa tem-se a impressão de que tudo está sendo processado no mesmo momento. No *Shell C* e *Korn*, um programa em execução é denominado *processo* ou *serviço*. Esses processos podem ser executados tanto no plano principal (primeiro plano) quanto no segundo plano. É possível, então,utilizar programas no primeiro plano enquanto outros estão sendo executados em segundo plano, agilizando um pouco mais o trabalho do usuário. Há a possibilidade, também, de se interromper processos e reiniciá-los, seja em primeiro ou em segundo plano (comandos **fg** e **bg**, respectivamente), além de poder visualizar os processos em execução e aqueles interrompidos (comandos **ps** e **jobs**) e encerrar processos com o comando **kill**.

Considerando-se o que foi explicado, conclui-se que a possibilidade que o Unix oferece de se trabalhar com diversos *shells*, facilita, de forma significativa, o trabalho de um usuário, principalmente quando se necessita utilizar um recurso presente somente em um *shell* diferente do que está sendo usado. Outro ponto importante é o fato de que alguns shells não agem apenas como simples interpretadores de comandos, mas também como linguagens de programação, possibilitando ao usuário construir seus próprios programas e utilizá-los como sendo comandos normais.

Por isso, além de outras razões, é que o sistema operacional Unix ainda está bem difundido entre empresas, instituições e profissionais de infor-

mática em geral, e com uma tendência ainda maior de se perdurar por um bom tempo no mundo da informática. Para quem nunca teve oportunidade de usá-lo, vale a pena tentar, pois, afinal, o Unix não é “apenas” um poderoso sistema operacional; ele faz parte da história da informática.

Referências

TAYLOR, D.; ARMSTRONG JÚNIOR, J.C. Aprenda em 24 horas UNIX. Rio de Janeiro : Ed. Campus, 1998. 672 p.

Literatura recomendada

ABRAHAMS, P.W.; LARSON, B.R. Unix for the impatient. 2nd ed. Reading : Addison-Wesley, 1996. 824 p.

ANUNCIAÇÃO, H. Unix para redes brasileiras. São Paulo : Ed. Érica, 1997. 196 p.

BOURNE, S.R. The Unix System V environment. Wokingham : Addison-Wesley, 1987. 378 p.

THOMAS, R.; YATES, J. Unix total. São Paulo : Ed. McGraw-Hill, 1988. 744 p.

Embrapa

Informática Agropecuária

Ministério da
Agricultura, Pecuária
e Abastecimento

BRASIL
UM PAÍS DE TODOS
GOVERNO FEDERAL