UNIX - um breve tutorial

Aleardo Manacero Jr.
UNESP/IBILCE/DCCE
S.J. do Rio Preto - SP

Conteúdo

Lista de Tabelas

Capítulo 1

Um pouco de história

Um dos sistemas operacionais mais usados hoje em dia quando o ambiente não é o de computação pessoal é o UNIX. Apesar de seu uso ter se ampliado muito nos últimos dez anos, este sistema já caminha para completar três décadas de existência, tendo suas origens no trabalho solitário de Ken Thompson, depois auxiliado por Brian Kernighan e Dennis Ritchie.

Thompson iniciou o desenvolvimento do UNIX a partir dos trabalhos que havia realizado no desenvolvimento do Multics, após a saída do Bell Labs daquele consórcio. Ele desenvolveu então um sistema de menor porte, escrito em *assembler*, que recebeu o nome de UNICS (em parte tirado do nome do Multics). Kernighan e Ritchie vieram em seu auxílio quando Thompson decidiu portar seu sistema para o PDP-11, que era muito mais moderno do que o PDP-7 onde nasceu o UNICS.

Para o PDP-11 decidiu-se usar outra linguagem como fonte, inicialmente o B e depois o C, ambos trabalhos de Ritchie. Em 1974 Ritchie e Thompson publicaram o primeiro artigo descrevendo o UNIX, trabalho este que lhes rendeu em 1984 o prêmio Turing oferecido pela ACM a grandes avanços em computação.

Nesta fase inicial o sistema era vendido a baixíssimo custo, vindo também com os fontes. Isso possibilitou não apenas a sua difusão como também o seu depuramento, pois cada usuário podia mexer no sistema como bem entendesse.

A Universidade de Berkeley passou a desenvolver sua própria versão de UNIX, que recebeu o nome de BSD, enquanto a AT&T passava a cobrar mais pelo software, que já vinha sem os seus fontes. Ambas rivais lançaram várias versões do UNIX, sendo que as mais recentes são o **4.3BSD** de Berkeley e o **System V/release-4** da AT&T.

Nestas versões já aparece o trabalho desenvolvido pela comissão de padrões do IEEE, que lançou o chamado POSIX, como forma de permitir um núcleo comum a qualquer fornecedor UNIX.

Hoje, cada uma das grandes empresas possui suas próprias versões de UNIX, como por exemplo, temos o AIX da IBM, o ULTRIX da Digital, o IRIX da Sillicon Graphics e o SunOS da Sun por exemplo. Para a linha PC temos o Linux, o FreeBSD, o Minix e o XINU entre outros. O mais conhecido, e que nós usaremos aqui, é o Linux, inicialmente desenvolvido por Linus Torvalds e que tem hoje um mercado bastante amplo.

Capítulo 2

Comandos básicos

2.1 Arquivos e diretórios

Em UNIX não existem diferenças entre arquivos, diretórios e mesmo periféricos. Todos podem ser tratados de uma mesma forma.

Nomes em UNIX não sofrem grandes limitações, podendo ter até 128 caracteres, começar por qualquer caracter, não seguem a linha "nome.ext", enfim, pode-se fazer qualquer coisa. Por exemplo, os nomes abaixo são todos corretos em UNIX:

nome.dat
.nomedat
.este.e.um.nome.correto.cheio.de.pontos.
todos_simbolos.a.seguir.sao_validos%!#()
0123456

Uma coisa a observar é que todo arquivo cujo nome começa com "." fica escondido a menos que se queira observá-lo com opções especiais de comandos.

Também deve-se observar que UNIX é sensível ao caso, isto é, os nomes a seguir representam arquivos distintos:

Arquivo1
arquivo1

2.2 Comandos básicos do UNIX

Apesar da maioria de vocês estar acostumado com ambientes totalmente gráficos (e quase todas as variantes de UNIX também possuem esses ambientes), é preciso conhecer alguns comandos de linha, uma vez que muitas vezes teremos que trabalhar remotamente em alguma máquina, em condições que o uso do ambiente gráfico se torna insustentável.

Comandos de linha são instruções curtas passadas ao computador para que ele execute alguma atividade, tal como apagar, copiar ou criar um arquivo. Ao longo desse texto estaremos preocupados única e exclusivamente com comandos de linha, uma vez que ambientes gráficos são muito simples de serem utilizados e pelo fato de vocês estarem já bastante acostumados ao Windows.

A tabela 2.1 apresenta uma primeira lista de comandos de linha.

UNIX	Descrição e opções
man	Descreve um dado comando
ls	Lista o conteúdo de um diretório *-l lista longa* *-i mostra as uid's dos arquivos* *-a mostra tambem os hidden files* *-F indica tipo do arquivo* *-C lista de forma compacta*
cd	Vai para um dado diretório
cp	Copia arquivos *-r copia recursivamente subdiretórios* *-p preserva data de criação*
rm	Apaga (remove) um arquivo *-r deleta recursivamente subdiretórios* *-i pede confirmação*
mv	Renomeia um arquivo ou move o mesmo entre diretórios
cat	Lista o conteúdo de um arquivo em uma passada
more	Lista o conteúdo de um arquivo página a página
mkdir	Cria um diretório
rmdir	Remove um diretório (se estiver vazio)
chmod	Modifica os atributos de um arquivo [ugoa] [+-=] [rwxXstugo]

Tabela 2.1: Comandos UNIX básicos

2.2.1 Exemplos de uso

- **man** man

  Descreve o funcionamento e sintaxe do comando *man*;

- **mkdir** *novodir*

  Cria o subdiretório *novodir* a partir do diretório atual;

- **cd** *subdir1/subsubdir*

  Passa do diretório atual para o diretório *subsubdir*, dentro do diretório *subdir1*, que está no direório atual. Observar que a separação dos nomes de diretórios em UNIX é feita com o símbolo / e não \, como no DOS/Windows;

- **cd** */export/home/master/comput/juca*

  Vai do diretório atual para o diretório indicado, sendo que como o parâmetro começa com o símbo / temos o caminho completo, a partir do diretório raiz da máquina.

- **cp** *source destination*

  Copia o arquivo *source* para *destination*;

- **cp** **doc ../outrodir*

  Copia todos os arquivos do diretório atual cujo nome termine com *doc* para o diretório *outrodir*, localizado a partir do diretório pai do atual;

- **cp** *../outrodir/* .*

  Copia todos os arquivos contidos em *outrodir* para o diretório atual (representado por .) ;

- **rm** *.*

  Remove todos os arquvos do diretório atual que tenham o caracter . em seu nome. Observar que em UNIX não existe o conceito de extensões nos nomes dos arquivos;

- **mv** *arq1 arq2*

  Muda o nome de *arq1* para *arq2*;

- bf mv *arq1 dir1*

  Move o arquivo *arq1* de seu diretório atual para o diretório *dir1*;

- **cat** *arquivo* ; **more** *arquivo*

  Mostram o conteúdo de *arquivo*, sendo que **cat** o faz de uma única vez, enquanto **more** o faz página (ou linha) por página (ou linha);

- **chmod** *u+x arq*

  Acrescenta a permissão de execução (x) para o dono (u) do arquivo *arq*. Outras opções seriam (-) para eliminar uma permissão, (r) para leitura, (w) para escrita, (g) para usuários do grupo do dono e (o) para os demais usuários;

- **chmod** *0644 arq*

  Deixa o arquivo *arq* com permissão de leitura e escrita para o dono ($6 = 110$ em binário) e apenas de leitura para os demais ($4 = 100$ em binário);

2.3 Outros comandos UNIX interessantes

Os comandos apresentados até aqui permitem que se faça muito pouco dentro do sistema. Os comandos da tabela 2.2 expandem um pouco mais o que podemos fazer em linha de comando.

COMANDO [opção]	DESCRIÇÃO
tail [-n] arquivo	mostra as últimas 10 linhas do arquivo ou as últimas *n* se existir o parâmetro [-n]
head [-n] arquivo	mostra as primeiras 10 linhas do arquivo ou as primeiras *n* se existir o parâmetro [-n]
history [p]	apresenta os ultimos p-1 comandos executados.
alias	personaliza os comandos. Por exemplo, *alias h history 20* faz com que o comando "h" seja entendido como "history 20"
pwd	apresenta o caminho completo ao diretório corrente.
who	mostra quem está "logado" naquela máquina
whoami (ou 'who am i')	mostra o nome (username) do usuário da sessão em que for executado
finger *user*	mostra os dados relativos ao usuário cujo nome (*user*) for dado no comando
talk user@machine	pede ligação com o usuário "user" logado na máquina "machine"
touch	cria ou atualiza (nova data) um arquivo
logout/login	encerram ou iniciam uma sessão

Tabela 2.2: Mais comandos de linha

2.4 Alguns atalhos

Existem certos caracteres que, quando usados na linha de comando, possuem significado especial. Eles servem como caracteres coringas na especificação do próximo comando a ser executado. Eles podem ser vistos na tabela 2.3

!!	⟶ re-executa o último comando
!n	⟶ re-executa o comando n
!!extensão	⟶ executa a última linha de comando acrescida da string extensão
^str1^str2	⟶ re-executa a última linha de comando trocando a string "str1" pela string "str2"
*	⟶ wildcard para n caracteres
?	⟶ wildcard para 1 caracter

Tabela 2.3: Atalhos para re-execução de comandos

Capítulo 3

Comandos mais interessantes

No UNIX existem muitos comandos que permitem ao usuário controlar o que está em execução na máquina ou ainda fazer buscas relativamente complexas dentro do sistema. Nesse capítulo apresentaremos alguns desse comandos, gastando um tempo maior em suas descrições.

3.1 ps e kill, uma dupla importante

Com o comando **ps** conseguimos descobrir quais os processos que estão executando no sistema. As várias opções do comando possibilitam examinar diferentes processos, segundo quem os iniciou, em que sessão. A partir dele podemos ver dados como o estado atual do processo, instante de início, porcentagem de CPU que está ocupando, memória, etc..

O seu parceiro, **kill**, pode ser usado para "matar" processos quando assim se desejar. Para tanto basta executar o comando "kill proc_number", onde proc_number é obtido a partir da saída do comando ps.

Detalhes sobre as opções disponíveis, bem como os significados de cada um dos dados de saída, podem ser encontrados através dos comandos "man ps" e "man kill", e não serão apresentados aqui por diferirem muito nos vários ambientes existentes.

3.2 Comandos internos ao arquivos *.login* e *.cshrc*

Estes arquivos são usados para a inicialização do sistema. Nos dois podemos fazer definição de *paths*, *aliases* e variáveis do ambiente. A diferença entre ambos é momento em que eles são executados.

O **.login** só é ativado no momento em que ocorre o login, não sendo executado em nenhum outro momento daquela sessão. Logo, nele devem estar os comandos que serão executados apenas uma vez no decorrer da sessão, independentemente da abertura de novas janelas.

O **.cshrc** é ativado toda vez que for aberta uma nova janela na sessão atual. Nele devem estar definidas todas as variáveis e *aliases* que se deseja manter em qualquer ponto da sessão

de trabalho.

3.3 Conversando com o mundo exterior

Existem comandos destinados exclusivamente para a comunicação com outras máquinas. Eles são usados para executar comandos remotos, fazer cópias de arquivos, transferir arquivos, abrir sessões remotas e também verificar o estado de máquinas e/ou usuários remotos. Na tabela 3.1 temos uma breve descrição destes comandos:

COMANDO	DESCRIÇÃO
ping maquina	mostra se uma máquina está ou não disponível para o mundo exterior
rsh maquina comando	executa um comando na máquina indicada
rusers	lista os usuários com sessões abertas nas máquinas conectadas com a máquina de onde partiu o comando
rcp f1 f2	faz a cópia remota de arquivos
rlogin maquina telnet maquina	abrem uma sessão remota
ftp maquina	abre uma sessão de transferência de arquivos entre a máquina onde se está logado e a màquina indicada pelo comando ftp

Tabela 3.1: Comandos para serviço remoto

Por razões de segurança muitas máquinas não aceitam/executam os comandos *rsh, rcp* e *rlogin*. Outras não aceitam também os comandos *telnet* e *ftp* quando originados por máquinas não conhecidas. Em particular, muitos sistemas adotam hoje a conexão remota apenas através de *ssh*.

3.4 Entrada e saída

O UNIX usa como entrada padrão o teclado e como saída padrão o monitor de vídeo. Como isso é extremamente inconveniente se formos executar tarefas de forma não-interativa, temos que ter meios de redirecionar tanto a saída como a entrada de comandos. Isto é feito através dos caracteres de redirecionamento listados a seguir:

" cmd > file "	$\longrightarrow$	redireciona a saída de cmd para file
" cmd < file "	$\longrightarrow$	redireciona a entrada de cmd para file
" cmd >& file "	$\longrightarrow$	redireciona a saída, inclusíve de erros, de cmd para file
" cmd >! file "	$\longrightarrow$	redireciona a saída de cmd para file, mesmo que file já exista
" cmd >> file "	$\longrightarrow$	redireciona a saída de cmd para file, acrescentando ao seu final caso file já exista

O comando pipe |

Com o comando pipe conseguimos redirecionar a saída do comando antes do pipe para a entrada do comando seguinte ao pipe. Assim, se pretendessemos listar os arquivos criados em setembro num dado diretório poderiamos fazer:

ls -l | grep Set

3.5 O comando grep

No exemplo anterior aparece o comando "grep". Ele serve como um seletor, onde sua saída apresenta todas as linhas onde ocorra uma dada string na entrada do programa. Por exemplo, "grep UNIX estearquivo" vai listar as linhas onde apareça a palavra UNIX no arquivo chamado *estearquivo*.

As várias opções para o comando grep incluem:

-i $\longrightarrow$ ignora maiúsculas e minúsculas.

-v $\longrightarrow$ seleciona as linhas onde não ocorre a string

3.6 O comando find

Ele é usado para procurar arquivos dentro do sistema. Esta ferramenta é bastante poderosa e uma descrição detalhada da mesma tomaria mais tempo do que o desejável aqui. Novamente devemos nos reportar ao comando "man find" para que os detalhes possam ser vistos. A título de exemplo podemos listar os seguintes casos:

find . -name estenome -print	$\longrightarrow$	lista as ocorrências de arquivos com "estenome" nos subdiretórios a partir do corrente
find . -newer estefile -print	$\longrightarrow$	lista todos os arquivos mais recentes que "estefile" nos subdiretórios abaixo do atual

Em especial, a opção *-print* não é necessária em algumas das implementações mais recentes do UNIX, como no Linux propriamente dito.

3.7 Executando em *background*

Uma das grandes vantagens em sistemas multi-tarefas é a possibilidade de que se execute programas em *background*. No UNIX isso pode ser feito de várias maneiras, dependendo do instante em que se quer iniciar a execução da programa. A seguir temos uma descrição das principais formas:

& $\longrightarrow$ é usado de duas formas distintas, iniciando a execução já em background ou após ter sua execução interativa suspensa. No primeiro caso faz-se "comando &", enquanto que no segundo caso, após iniciarmos a sua execução através de "comando", damos [**CTRL-Z**] seguido do comando **bg**

at $\longrightarrow$ é usado para iniciar uma tarefa em background num instante pré-determinado, como por exemplo, "at now + 1 minute comando" ou "at 23:00 comando" ou ainda "at now comando", nas máquinas IBM, ou como "at -f arquivo_script now". Além disso, olhando-se no manual do comando podem ser vistas outras opções como lista dos processos agendados para execução ou mesmo remoção de processos desta lista

3.8 Comandos *set* e *setenv*

Dentro do UNIX existe a possibilidade de definir variáveis que armazenam o modo de operação do sistema. Tais variáveis são chamadas **variáveis de ambiente**, sendo que seus valores podem ser definidos através dos comandos *set* e *setenv*.

Algumas variáveis de ambiente são extremamente importantes e devem ser definidas com bastante cuidado. Na lista a seguir apresentamos as principais variáveis de ambiente, seus significados e um exemplo de como devem ser definidas:

DISPLAY	$\longrightarrow$	define qual o monitor em que uma janela deve ser aberta. setenv DISPLAY nomemaquina:0.0
PATH	$\longrightarrow$	define em que diretórios deve-se procurar por um arquivo executável correspondente ao comando solicitado. set path = (/bin /usr/bin /etc .)
LD_LIBRARY_PATH	$\longrightarrow$	define em que diretórios devem ser procuradas bibliotecas para a execução de um programa. setenv LD_LIBRARY_PATH /usr/local/hagar/lib:/usr/X11R6/lib
MANPATH	$\longrightarrow$	define em que diretórios devem ser buscados os arquivos tipo troff contendo páginas de manual. setenv MANPATH /usr/man:/usr/openwin/man:/usr/X11R6/man

Os valores das variáveis de ambiente podem ser verificados de várias formas, sendo que os comandos *set*, *env* e *setenv*, quando executados sem outro parâmetro, retornam os valores das variáveis definidas para aquele ambiente. Já a execução do comando $NOME_VAR retorna o valor da variável NOME_VAR.

Capítulo 4

O editor vi

O **vi** é um editor bastante rudimentar em aparência, que pode ser usado a partir de qualquer terminal ligado numa máquina UNIX. Sua grande vantagem é não depender fortemente de ambientes de janela, como os outros editores disponíveis nos vários ambientes gráficos.

A seguir, na tabela 4.1, temos alguns dos comandos usados durante a edição no **vi**:

COMANDO (usado na linha de comando de arquivo)	DESCRIÇÃO
wq (ou x)	sai do **vi** salvando o texto
q	sai do **vi** sem salvar o texto (apenas se ele não foi alterado)
q!	sai do **vi** sem salvar o texto, ignorando qualquer alteração
w	salva o texto sem sair do **vi**

Tabela 4.1: Operações sobre arquivos no vi

Além dos comandos acima existem comandos para fazer a edição propriamente dita ou ainda algumas operações simples de busca sobre palavras no arquivo. Alguns desses comandos podem ser vistos na tabela 4.2, na próxima página. Recomenda-se fortemente que, além de uma consulta ao manual do vi, também se pratique bastante com o mesmo dada a sua grande flexibilidade e facilidade de uso remoto.

COMANDO (fora do modo de edição)	DESCRIÇÃO
dd	deleta a linha sob o cursor
dw	deleta a palavra a direita do cursor
x	deleta o caracter sob o cursor
r*char*	troca o caracter sob o cursor por *char*
R*text*ESC	troca o texto por *text*
cw*new*ESC	troca a palavra sob o cursor por *new*
easESC	acrescenta a letra *s* no final da palavra atual
u	desfaz a última edição
w	posiciona o cursor no início da próxima palavra
b	posiciona o cursor no início da palavra anterior
*n*G	vai para a linha *n* do arquivo
o	abre edição na linha seguinte ao cursor
i	abre edição na posição atual do cursor
ESC	sai do modo de edição
CTRL-F	avança uma página
CTRL-B	recua uma página
$	vai para o fim da linha atual
^	vai para o começo da linha atual
-	vai para o começo da linha anterior
/*string*	procura pela ocorrência de *string*
//	procura a próxima ocorrência da última *string* referenciada
:	vai para linha de comandos de arquivo

Tabela 4.2: Comandos de edição

Capítulo 5

Ambiente de comando - shell

O UNIX se diferencia de ambientes como o Windows e outros sistemas operacionais por permitir que o usuário escolha o padrão de comandos que mais lhe agrada. Assim, é possível configurar o sistema segundo determinados padrões, conhecidos como *shells*, que permitem que os comandos apresentados até aqui possuam sintaxes ligeiramente distintas entre si. Além disso existem certas capacidades (ou propriedades) que estão presentes em determinados ambientes e não em outros.

5.1 Histórico

Fazendo um retrospecto histórico pode-se dizer que inicialmente haviam dois ambientes (que chamaremos apenas como *shells* daqui por diante) diferentes, que são o Bourne shell (**sh**) e o C shell (**csh**). A diferença entre eles é que o csh possui uma maior facilidade para uso interativo, enquanto o sh apresenta menos erros.

Com o uso dos dois ambientes, muitas vezes tendo que mistura-los para obter bons resultados, começou-se uma busca por um ambiente perfeito. As primeiras tentativas ficaram bastante longe disso, como é o caso do **tcsh** e o *ksh* (de Korn shell). O primeiro era baseado no csh, tentando eliminar seus erros, mas os fabricantes optaram por se manterem fiéis ao csh. Já o segundo estava baseado no sh, acrescentando opções para interatividade. Seria o melhor dos dois mundos (Bourne e C), mas como é um sistema pago (para a AT&T), acabou não tendo o impacto previsto.

Com o advento do conceito de software livre, dentro do projeto GNU também se criou um novo *shell*, que adotou o padrão POSIX (e portanto o formato do sh), recebendo o nome de **bash** (Bourne again shell). Ele possui características muito semelhantes ao ksh, mas por ser livre (e gratuito) acabou por ser adotado pelo Linux e, hoje em dia, está disponível em vários outros sistemas (como no solaris).

Além desses *shells* foram criados alguns outros, sempre com a idéia de aperfeiçoar o que já existe. Entretanto grande parte deles acaba não sendo empregada pelos fornecedores de UNIX do mercado. Um deles em especial, o **rc**, foi inicialmente projetado para um outro sistema operacional (Plan 9), que foi desenvolvido pela AT&T mas não conseguiu penetração no mercado apesar de conceitualmente ser bastante superior ao UNIX. Como rc foi desenvolvido inteiramente a partir do zero ele consegue ser menor, mais rápido e melhor que os demais shells. Infelizmente, como originalmente foi criado para um sistema operacional natimorto acabou por ter pouco uso também.

5.2 Diferenças notáveis entre shells

A tabela 5.1 ilustra como os vários *shells* se diferenciam, ou melhor dizendo, quais são as características que cada *shell* apresenta ou deixa de apresentar. Essa tabela foi tirada da página www.looking-glass.org/shell.html, que foi, também, a fonte do material apresentado na seção anterior.

	SHELL				
Propriedade	sh	csh	ksh	bash	rc
Controle de tarefa	N	Y	Y	Y	N
Aliases	N	Y	Y	Y	N
Funções de *shell*	Y(1)	N	Y	Y	Y
História de comandos	N	Y	Y	Y	L
Edição de linha de comando	N	N	Y	Y	L
Complementação de nome de arquivo	N	Y(1)	Y	Y	L
Complementação de nome de usuário	N	Y(2)	Y	Y	L
Complementação de nome de máquina	N	Y(2)	Y	Y	L
Complementação de história	N	N	N	Y	L
Coprocessos	N	N	Y	N	N
Avaliação aritmética	N	Y	Y	Y	N
Prompt fácilmente personalizado	N	N	Y	Y	Y
Sintaxe base	sh	csh	sh	sh	rc
Gratuíto	N	N	N(3)	Y	Y
Verificação de *mailbox*	N	Y	Y	Y	F
Manuseia grandes listas de argumentos	Y	N	Y	Y	Y

Tabela 5.1: Diferenças básicas entre *shells*

Chaves para a tabela:
- Y propriedade existente no *shell*
- N propriedade não existente no *shell*
- F propriedade executável através de mecanismo de funções
- L precisa ligação com biblioteca para estar habilitado

Condições especiais:
- (1) versão original não incluia a propriedade, mas é padrão atualmente
- (2) propriedade recente e ainda não presente em muitas versões do *shell*
- (3) uma versão (pdksh) é disponível gratuitamente, embora sem todas as funcionalidades

5.3 Que shell devo usar?

A escolha por um *shell* deve ser feita levando em consideração alguns princípios básicos. O principal é que o *shell* deve estar disponível para o sistema em que o usaremos, o que parece óbvio. Outros princípios são:

- O que queremos fazer usando aquele *shell*? Precisamos de interatividade? Podemos admitir alguns erros? Etc.
- Conhecemos (com uma boa fluência) algum *shell*? Se sim é normalmente melhor não mudar ou, se a mudança for necessária, escolher um outro que se pareça com o atual. Se não devemos escolher o melhor para as nossas necessidades e disponibilidades.
- Temos tempo e meios para aprender um novo *shell*? Se sim podemos optar pela adoção do mais adequado ao nosso problema. Se não, é melhor trabalhar com o que conhecemos, sem maiores aventuras.

Capítulo 6

Usando e escrevendo scripts

Ao usar PC's voces já devem ter usado (ou até mesmo escrito) algum arquivo do tipo ".BAT", que nada mais é do que um arquivo contendo um conjunto de comandos que devem ser executados seqüencialmente para produzir o resultado desejado. O mais conhecido desses arquivos é o **AUTOEXEC.BAT**, executado toda vez que o computador é ligado. Em UNIX é possível construir arquivos semelhantes, isto é, arquivos que sejam um conjunto de comandos seqüenciais. Esses são os chamados *scripts* do UNIX.

Antes de examinarmos como construir *scripts* é preciso saber que cada *script* é escrito para ser executado em uma dada configuração do sistema. Essa configuração é o *shell* do sistema, que examinamos no capítulo anterior. Em cada máquina existem vários *shells* disponíveis para escolha pelo usuário, o que implica em diferentes *scripts* para diferentes *shells*. As diferenças na construção de *scripts* usando cada um desses shells são relativamente pequenas, consistindo em alguns detalhes de sintaxe dos comandos disponíveis. Nesse curso examinaremos apenas *scripts* escritos em C-shell, principalmente por terem uma estrutura bastante parecida com a linguagem C. O leitor fica convidado a transformar os *scripts* e regras aqui apresentados para o *shell* de sua preferência.

Antes de examinarmos como construir *scripts* é preciso fazer uma rápida apresentação de dois aplicativos para manipulação de arquivos bastante poderosos, tanto dentro de *scripts* quanto usados diretamente em linhas de comando. Esses aplicativos são os ccomandos **awk** e **sed**. Entretanto, para que se possa usar esses comandos em todo seu potencial é necessário um estudo bem mais profundo do que o que se faz aqui.

6.1 O comando awk

Esse comando é utilizado para a busca por padrões dentro de um arquivo e a realização de algum processamento sobre esses padrões. Ele pode ser usado para gerar relatórios ou filtrar textos, trabalhando bem tanto sobre números quanto sobre textos simples. Os autores de awk (Aho, Weinberger e Kernighan, por isso o seu nome!) procuraram torná-lo fácil de usar (para um programador C, evidentemente) e por isso não o fizeram eficiente o bastante. A sintaxe básica do comando awk é vista a seguir:

- awk [-Fc] -f program-file [file-list]
- awk [-Fc] '*awk-rule*' [file-list]

No primeiro caso o comando instrui o computador a executar regras de filtragem escritas em *program-file* sobre os arquivos contidos em *file-list*. No segundo caso é executada a regra *awk-rule* diretamente sobre os arquivos de *file-list*. Os argumentos usados nesses comandos podem ser descritos da seguinte forma:

-f program-file	faz com que awk leia as regras diretamente a partir do arquivo *program-file*.
-Fc	Usa o caracter indicado em **c** como separador de palavras no lugar dos separadores usuais (espaço e tab).

A sintaxe permitida nas regras é muito parecida com a sintaxe usada em testes de decisão da linguagem C. A grande diferença está na forma de tratamento das palavras dentro do arquivo. Para awk cada linha contém um determinado número de palavras, acessíveis através de uma numeração de parâmetros feita com o símbolo $. Dessa forma, a regra apresentada a seguir busca as ocorrências da string "time" dentro do arquivo *resultados* e imprime as linhas em que essa string é a terceira palavra da linha.

```
awk '{if ($3 == "time") print $0}' resultados
```

Observem que $0 identifica a linha como um todo. Se estivermos interessados apenas em alguns campos da linha, por exemplo o quarto e sexto campos, o comando seria:

```
awk '{if ($3 == "time") print $4, $6}' resultados
```

Observem agora que os campos quatro e seis aparecem separados por uma vírgula. Isso é necessário para o comando **print**, que faz a saída não formatada do awk. O resultado desse comando é um conjunto de pares de números separados por um único espaço em branco entre eles. Awk também admite saída formatada através do comando **printf**, com sintaxe semelhante ao C.

Dentro da sintaxe de awk existem alguns símbolos prédefinidos, tais como:

BEGIN	Identifica uma regra que será executada **apenas** antes da primeira linha examinada pelo comando
END	Identifica uma regra que será executada **somente** após a última linha examinada pelo comando
NF	variável que armazena o número de campos de uma linha
NR	variável que armazena o número de linhas lidas na execução do comando

A seguir podem ser vistos alguns exemplos de comandos awk, com as respectivas descrições funcionais.

6.1.1 Exemplos de comandos de uma linha

1. awk '{ if (NF > max) max = NF }
   END {print max}' arquivo

   Imprime o número máximo de campos em uma linha de um dado arquivo.

2. awk 'lenght ($0) > 80' arquivo

   Imprime as linhas com mais de oitenta caracteres.

3. awk '{nlines++}
   END { print nlines}' arquivo

   Imprime o total de linhas de um arquivo.

4. awk ' END {print NR}' arquivo

   Também imprime o número de linhas de um arquivo.

5. ls -l *files* | awk '{x += $5}; END {print "total bytes: ", x}'

   Imprime o total de bytes usados por *files*.

6.1.2 Exemplos de comandos com arquivos de regras

Para os exemplos a seguir, considere o seguinte arquivo de entrada:

```
$ more   cars
plym     fury       77   73    2500
chevy    nova       79   60    3000
ford     mustang    65   45    10000
volvo    gl         78   102   9850
ford     ltd        83   15    10500
chevy    nova       80   50    3500
fiat     600        65   115   450
honda    accord     81   30    6000
ford     thundbd    84   10    17000
toyota   tercel     82   180   750
chevy    impala     65   85    1550
ford     bronco     83   25    9500
```

1. Gerar um cabeçalho em um relatório do arquivo:

   $ more pr_header

   ```
   BEGIN    {
   print "Make     Model     Year     Miles     Price"
   print "_____________________________"
   }
            {print}
   ```

   Observem que a primeira regra (que imprime o cabeçalho) é executada apenas antes da análise da primeira linha, enquanto a segunda (que imprime uma linha do arquivo) é executada para todas as demais linhas.

   $ awk -f pr_header cars

2. Gerar um resumo sobre idade e preço dos carros

```
$ cat summary
BEGIN           {
                yearsum = 0; costsum = 0
                newcostsum = 0; newcont = 0;
                }
                {
                yearsum += $3
                costsum += $5
                }
$3 > 80         {newcostsum += $5; newcount++}
END             {
                printf "Average age of cars is %3.1f years \n", \
                        88 - (yearsum/NR)
                printf "Average cost of cars is $%7.2f \n", \
                        costsum/NR
                printf "Average cost of newer cars is $%7.2f \n", \
                        newcostsum/newcount
                }
```

Observem aqui o uso do comando printf para fazer saída formatada e do caracter \ para indicar a continuação do comando na linha seguinte.

$ awk -f summary cars

6.2 O comando sed

Esse comando é um editor não interativo, isso é, um editor usado dentro de scripts do shell. Um comando sed copia linhas da entrada usada para a saída padrão, editando-as durante o processo. A sintaxe básica do comando é vista a seguir:

- sed [-n] -f script-file [file-list]
- sed [-n] script [file-list]

As opções que aparecem nos dois casos tem o seguinte significado:

-f faz com que sed leia o seu *script* a partir de script-file.

-n inibe a cópia das linhas para a saída padrão.

Os próximos exemplos ilustram o que se pode fazer usando sed:

- Usando sed a partir da linha de comandos:

  $ **cat new**

  Line one.

  The second line.

  The third.

  This is line four.

  Five.

  This is the sixth sentence.

  This is line seven.

  Eighth and last.

  $ **cat demo1**

  s/line/sentence/gw temp

  Este comando troca as ocorrências da string "line" pela string "sentence" no arquivo de entrada e escreve as linhas em que isso ocorrer para o arquivo "temp".

  $ **sed -f demo1 new**

  ⋮

  $ **cat temp**

  The second sentence.

  This is the sentence four.

  this is sentence seven.

- Usando sed dentro de um script:

  sed -e 's/X//' > $jobname.jdf <<EOF1

  X

  XCLASS=2

  XLOGICAL CPU = 2,3,4,5,6,7,8,9,10,11,12,13

  XNUMBER OF PROCESSES = $nodenum

  XPROCESSES PER CPU = $procnum

  XPROGRAM=$jdfcomman

  X$jdfacpu

  X

  EOF1

  Este script cria um arquivo (chamado ' algumacoisa'.jdf) contendo as informações que aparecem após o caracter 'X' em cada uma das linhas até a linha marcada pela string 'EOF'.

6.2.1 Escrevendo scripts

Não existe uma receita para escrever *scripts* , assim como não existem receitas para se escrever programas em qualquer outra linguagem. O que se deve fazer é entender o que o *script* deverá executar, definir um algoritmo para a sua execução e, a partir dele, codificar o *script* usando os comandos que forem mais adequados para cada situação. Nesse sentido, seguem-se alguns exemplos de *script* para atividades diversas.

- Um *script* para executar comandos simples (*executa_backup*):

  #!/bin/csh -f

  # A linha acima indica que o *script* usara o shell csh e que o mesmo sera

  # iniciado em modo rapido

  # Linhas comecando com # (exceto as que comecarem com #!) sao comentarios

  # Espera cinco segundos apos meia-noite

  sleep 5

  # Submete ele mesmo para uma nova execucao as 24 horas

  at 24:00 executa_backup

  # Executa o comando de backup

  backup

- Um *script* para eliminar arquivos temporários:

  Script "ldir"

  Esse *script* faz a remoção de arquivos identificados por um segundo scritp (apresentado a seguir), considerando que seu arquivo de entrada (type1) contém nomes de arquivos a serem removidos.

  Script "limpatudo"

  Este *script* identifica (através do comando find) os arquivos temporários criados durante a última sessão de trabalho e os passa para o *script* ldir que fará, de fato, a remoção desses arquivos.

```
#!/bin/csh -f

set inlines = ‘ awk ’ END {print NR}’ type1‘
# atribui o numero de linhas do arquivo type1 para a variavel inlines
if ($inlines == ’0’) then
    exit # sai se o arquivo estiver vazio (inlines=0)
endif

# repete a sequencia abaixo ate que reste apenas uma linha no arquivo type1
while ($inlines != 1)
# atribui para cur_file o nome do primeiro arquivo em type1
    set cur_file = ‘ awk ’{if (NR == ’1’) print $0}’ type1‘
# remove esse arquivo
    /bin/rm $cur_file
# cria um arquivo auxiliar
    touch auxaux
# escreve nesse arquivo o conteudo de type1, exceto sua primeira linha
    awk ’{if (NR != ’1’) print $0 > ”auxaux”}’ type1
# copia o arquivo auxiliar para type1
    cp auxaux type1
# remove o auxiliar
    /bin/rm auxaux
# atribui para inlines o numero de linhas de type1
    set inlines = ‘ awk ’ END {print NR}’ type1‘
end

# atribui o arquivo da ultima linha de type1 para a variavel cur_file
set cur_file = ‘ awk ’{if (NR == ’1’) print $0}’ type1‘
# remove esse arquivo
/bin/rm $cur_file
```

limpatudo

```
#!/bin/csh -f

clear
echo " Apagando arquivos desnecessarios "
echo " Aguarde um momento "

# vai para o diretorio raiz do usuario
cd $HOME

# busca os arquivos mais recentes do que o arquivo touchday (criado no momento
# do login), salvando seus nomes (e caminhos) no arquivo daylist
find . -newer touchday ! -type d -print > daylist

# separe (grep) cada tipo de arquivo temporario para o arquivo type1
grep % daylist > type1
# chama o script  ldir, que remove os arquivos contidos em type1
    ldir type1
    /bin/rm type1
grep .bak daylist > type1
    ldir type1
    /bin/rm type1
grep .aux daylist > type1
    ldir type1
    /bin/rm type1
grep .toc daylist > type1
    ldir type1
    /bin/rm type1
grep .lof daylist > type1
    ldir type1
    /bin/rm type1
grep .BAK daylist > type1
    ldir type1
    /bin/rm type1
grep .dvi daylist > type1
    ldir type1
    /bin/rm type1
grep .log daylist > type1
    ldir type1
    /bin/rm type1

# remove o arquivo daylist
/bin/rm daylist

echo " "
echo " Operacao concluida "
sleep 3
```

Capítulo 7

Construíndo makefiles

Um tipo especial de *script* é conhecido como **makefile**, que é executado pelo comando *make*. Um makefile é na realidade um conjunto de instruções normalmente usadas para fazer o processamento de arquivos, principalmente durante a fase de compilação e instalação de softwares de tamanho elevado. O uso de makefiles é interessante por reduzir o tempo gasto com recompilações de um dado software em caso de erros ou modificações em parte dele. Makefiles podem ser bastante simples, com apenas um conjunto reduzido de ações a serem tomadas, ou bastante elaborados, contendo várias ações e definições de variáveis de ambiente.

Para um bom entendimento de como funciona um makefile aconselha-se a usar o comando "man make", com o qual se pode examinar todos os detalhes de um makefile. Aqui iremos apenas examinar três exemplos de complexidades diversas. O que se deve levar em conta nesses casos é que eles evitam que ações realizadas com sucesso sejam repetidas inutilmente.

7.1 Um makefile bastante simples

O exemplo a seguir apresenta um pequeno arquivo makefile, utilizado para a construção e manutenção de uma biblioteca chamada "libsim.a". Nessa biblioteca ficam armazenados os códigos objeto das funções contidas nos arquivos listados em **OBJECTS**, as quais são compiladas usando-se o compilador **gcc**.

O processo de execução desse makefile é composto basicamente em seguir as ações contidas na linha de comando identificada por **prog**, em que se diz primeiro ao computador para atualizar os arquivos listados por **OBJECTS**. Isso significa examinar no diretório corrente quais arquivos objeto estão disponíveis e, para cada um deles, verificar se a sua data de criação é anterior ao do arquivo fonte (".c") equivalente. Para os casos em que o objeto é mais velho que o fonte (ou não existir), é feita a compilação do fonte para que se gere um novo objeto. Nos demais casos não se faz nada com os arquivos e passa-se para o arquivo seguinte da lista. Completando-se a fase de compilação com sucesso, isto é, todos os arquivos sem erros de sintaxe, é criada (ou atualizada) a biblioteca "libsim.a", no diretório indicado no comando **ar**. Nele a opção "r" indica que os códigos objetos novos devem substituir suas antigas versões.

```
CC =            gcc

OBJECTS =       simulate.o put_in_queue.o semaphore.o \
                free_procs.o free_sem.o hold_event.o \
                init_graphs.o init_pmtr.o is_waiting.o \
                make_traces.o expon_fdp.o normal_fdp.o \
                wait_channel.o read_hard.o mount_graph.o \
                read_exec.o read_sync.o read_send.o \
                read_branch.o get_index.o comm_partner.o \
                find_sync.o make_synchr.o make_assynchr.o \
                put_func_tbl.o get_pr.o find_func.o \
                sync_partner.o still_running.o present_data.o \
                release_cli.o

prog :          $(OBJECTS); \
                ar r ../libs/libsim.a $(OBJECTS)
```

7.2 Um exemplo mais elaborado

Este *script* faz a compilação de um pequeno software, cujos arquivos fontes estao distribuidos entre três diretórios (aquele em que o makefile está, e os subdiretórios mono e parallel), criando três bibliotecas dentro do subdiretório libs e copiando o arquivo executável (chamado rtsim) para o diretório $HOME/bin (que é o subdiretório bin a partir do raiz do usuário).

```
# vou executar usando o ambiente bourne shell (sh)
SHELL =         /bin/sh

# os arquivos de trabalho serao encontrados a partir do diretorio atual
srcdir =        .

# os resultados (programa) irao para o diretorio $HOME/bin
prefix =        $$HOME/bin      # preciso de dois $ para indicar o valor correto de prefix
exec_prefix =   $(prefix)

# defino o comando cp como sendo o de instalacao
INSTALL =       cp
# defino o comando make para executar todos os makefiles
MAKE =          make

# defino o compilador como sendo o gcc
CC =            gcc
# definicao dos diretorios de arquivos com prototipos
DEFS =          -I/usr/local/casper/include -I/usr/local/casper/X11R6/include
# defino o grau de otimizacao (O2) e as opcoes para o gcc
CFLAGS =        -O2 $(DEFS)
# defino que a compilacao ira gerar apenas o arquivo objeto (-s)
LDFLAGS =       -s
```

```
# lista das bibliotecas externas a serem usadas na compilacao
EXTERN_LIBS =   -lm -L/usr/local/casper/X11R6/lib -lforms -lX11 -lsocket -lnsl
# lista das bibliotecas locais a serem usadas na compilacao
LOCAL_LIBS =    -L./libs -lbase -lmono -lparallel -lbase

# lista dos arquivos locais a serem compilados
OBJS =          def.o mono.o multi.o help_me.o dbl_to_char.o int_to_char.o

# lista de diretorios em que serao encontrados outros makefiles
DIRS =          mono parallel
cd =            cd

# aqui comeca a descricao dos comandos a serem executados pelo
# makefile. Nesse caso sao quatro comandos (all, prog, dir e compile),
# sendo que ao executar-se o comando make podemos passar qual e
# o programa que sera executado. Na falta desse parametro o sistema
# executa o comando all

# o comando all executa os demais comandos, na ordem indicada
all:
        prog dodirs compile

# o comando prog faz a compilacao dos arquivos listados em OBJS
# e cria a biblioteca libbase.a
prog :
        $(OBJS); ar r libs/libbase.a $(OBJS); \
        ranlib libs/libbase.a

# o comando dodirs executa os makefiles contidos nos diretorios marcados em DIRS
dodirs :
        for name in $(DIRS); \
        do \
        cd $$name; make; cd ..; \
        done
# o comando "for condicao do comando done" repete, para cada
# subdiretorio em DIRS, as acoes "cd subdiretorio", "make" (que
# executa o makefile contido nesse subdiretorio), e "cd .." para
# voltar ao diretorio em que esta este makefile

# o comando compile faz, de fato, a geracao do executavel (ao
# executar o comando "gcc ...") e o instala no diretorio previsto
compile:
        gcc $(CFLAGS) -o rtsim simula.c $(DEFS) $(EXTERN_LIBS) $(LOCAL_LIBS); \
        $(INSTALL) rtsim $(prefix) \
```

7.3 Um exemplo complexo

Não iremos aqui apresentar todos os detalhes de um makefile completo, uma vez que esses são, em geral, bastante complexos. No exemplo que se segue aparecem vários elementos típicos de um makefile bem elaborado, sendo portanto suficiente para que tomemos contato com o potencial de uso desse tipo especial de script. Como o mesmo é bastante extenso, iremos examiná-lo por partes, identificando o que é feito em cada trecho.

No primeiro trecho encontram-se as definições de qual shell executará os comandos de makefile (Bourne shell), em que diretórios ficam os fontes (srcdir), em que lugar deverão ser instalados o executável resultante (bindir) e os manuais (mandir) do software em instalação. Além disso, também são determinados quais objetos devem ser criados (OBJS), quais bibliotecas devem ser usadas na compilação (LIBS), qual o compilador (CC) e opções de compilação (DEFS, CFLAGS e LDFLAGS). Por fim, também são definidos os diretórios a serem usados durante a execução do makefile (DIRS).

```
SHELL =             /bin/sh

srcdir =            .

prefix =            /usr/local/casper
exec_prefix =       $(prefix)
bindir =            $(exec_prefix)/bin
datadir =           $(prefix)/lib
mandir =            $(prefix)/man/man1

INSTALL =           cp
INSTALL_PROGRAM=    $(INSTALL)
INSTALL_DATA =      $(INSTALL)

CC =                gcc
DEFS =              -I/usr/local/casper/X11R6/include
CFLAGS =            -O2
LDFLAGS =           -s

TERMFLAGS =

LIBS =              -L/usr/local/casper/X11R6/lib -lX11 -lm -lsocket

OBJS =              bitmap.o command.o contour.o eval.o graphics.o graph.o \
                    help.o internal.o misc.o parse.o plot.o readline.o \
                    scanner.o setshow.o specfun.o standard.o term.o util.o
                    gnubin.o binary.o

DIRS =              term demo docs docs/latextut
```

No segundo trecho do makefile encontramos partes de uma listagem de todos os arquivos encontrados na distribuição, tais como os arquivos fontes (de CSOURCE1 até CSOURCE8), arquivos de demonstração do programa (DEMOS), arquivos de texto e configuração (ETC) e também arquivos de documentação e manuais do programa (DOCS1, DOCS@ e DOCS3)

```
CSOURCE1 =  bf_test.c binary.c command.c setshow.c
CSOURCE2 =  help.c gnubin.c graphics.c graph3d.c internal.c
CSOURCE3 =  misc.c eval.c parse.c plot.c readline.c scanner.c standard.c
CSOURCE4 =  bitmap.c term.c util.c version.c
CSOURCE5 =  term/ai.trm term/amiga.trm term/aed.trm \
            term/apollo.trm term/gpr.trm term/hppj.trm term/compact.c
CSOURCE6 =  term/impcodes.h term/imagen.trm term/object.h \
            term/pbm.trm term/pslatex.trm term/gpic.trm
CSOURCE7 =  term/post.trm term/pstricks.trm term/regis.trm \
            term/v384.trm term/vws.trm term/x11.trm term/xlib.trm
CSOURCE8 =  contour.c specfun.c gplt_x11.c

DEMOS =     demo/1.dat demo/2.dat demo/3.dat demo/contours.dem \
            demo/scatter.dem demo/scatter2.dat demo/singulr.dem

ETC =       Copyright 0README README.gnu makefile.unx \
            README.3d README.mf configure configure.in Makefile.in

DOCS1 =     docs/makefile.org docs/checkdoc.c docs/doc2gih.c \
            docs/gnuplot.1 docs/lasergnu.1 docs/toc_entr.sty
DOCS2 =     docs/gnuplot.doc docs/gpcard.tex
DOCS3 =     docs/latextut/makefile.org docs/latextut/eg1.plt \
            docs/latextut/linepoin.plt docs/latextut/Makefile.in
```

Finalmente, o último trecho apresenta algumas das ações que podem ser realizadas ao se executa o comando make para o makefile em análise. Não faremos aqui um estudo detalhado de cada ação, bastando entender que ao executarmos um makefile temos a possibilidade de especificar quais ações devem ser tomadas, bastando acrescentar um parâmetro ao comando make, como "make all", "make install", "make check", "make gnuplot" ou "make distclean" por exemplo.

Neste trecho devem ser observados alguns detalhes interessantes sobre o potencial do makefile, como a possibilidade de mover o diretório atual para algum outro diretório de interesse, tal como "cd docs" encontrado em *doc*, ou de verificar se um dado arquivo existe, como em "test ! -f gnuplot_x11" dentro de *install*, ou ainda de remover arquivos usando o comando rm ou copia-los usando o comando cp, que havia sido previamente renomeado como INSTALL_PROGRAM.

```
all:            gnuplot gnuplot_x11 doc

gnuplot:        $(OBJS) version.o
                $(CC) -o $(OBJS) version.o $(LDFLAGS) $(LIBS)

doc:
                ( cd docs; $(MAKE) $(MFLAGS) gnuplot.gih )

gnuplot_x11:    gplt_x11.o
                $(CC) -o gplt_x11.o $(LDFLAGS) $(LIBS)

check:          all demo/binary1

install:        all $(LASERGNU)
                $(INSTALL_PROGRAM) gnuplot $(bindir)/gnuplot
                test ! -f gnuplot_x11 || $(INSTALL_PROGRAM)
                gnuplot_x11 $(bindir)/gnuplot_x11
                -$(INSTALL_DATA) $(srcdir)/docs/gnuplot.1
                $(mandir)/gnuplot.1
                (cd docs; $(MAKE) $(MFLAGS) install datadir=$(datadir))

clean:
                rm -f gnuplot gnuplot_x11 bf_test *.o core
                ( cd docs; $(MAKE) $(MFLAGS) clean )
                ( cd docs/latextut; $(MAKE) $(MFLAGS) clean )

distclean:      clean
                rm -f Makefile config.status
```