FMA 215

Aula 13: Programação em Shell Scripts

Prof. Jorge L. deLyra

19 de Agosto de 2002

1 Alguns Conceitos Relevantes

Além de servir como interface do sistema com o usuário e de poder ser utilizada tanto para a submissão ao sistema de comandos simples quanto para a execução de procedimentos mais complexos, que combinam vários comandos, no que chamamos de "programação de linha de comando", a shell pode ser utilizada como uma verdadeira linguagem de programação. Fazemos isto escrevendo certos arquivos, chamados "shell scripts", que serão lidos e interpretados pela shell, levando à execução dos procedimentos nele contidos. Qualquer dos comandos e estruturas que podem ser usadas para programação de linha de comando também podem ser usados em scripts, mas há algumas estruturas que podem ser usadas apenas em scripts.

Cada uma das shells interativas usuais tem a sua própria linguagem de programação. De forma geral elas têm estruturas parecidas umas com as outras, mas os detalhes diferem significativamente de uma para outra. Vamos examinar diretamente nesta aula a linguagem de programação da shell que usamos como padrão interativo, a `tcsh`.

É importante enfatizar que a programação em shell scripts com as shells interativas usuais não é um meio apropriado para todo e qualquer tipo de programação. Ela pode ser utilizada para procedimentos razoavelmente complexos, mas não se trata de um meio apropriado para a escrita de programas que sejam extremamente longos e complexos. De fato, para este tipo de uso há shells que foram criadas especialmente, como é o caso da Perl, as quais não são utilizadas interativamente de todo. Além disso, como trata-se de linguagens de programação interpretadas e não compiladas, as linguagens de shell não se prestam a programas cuja função seja a de executar procedimentos de processamento intensivo.

A programação em shell scripts é idealmente apropriada para funções que sejam excessivamente complexas para serem lembradas em detalhe e usadas diretamente na linha de comando, mas que não sejam mais longas do que algumas poucas páginas, preferencialmente uma única página. Elas são particularmente apropriadas para a execução de tarefas e rotinas no sistema, sejam elas administrativas ou de interesse particular do usuário.

2 Tarefas

Vamos começar escrevendo scripts para executar algumas tarefas que não são mais do que tarefas de programação de linha de comando como as que vimos anteriormente. Depois disto, vamos descrever e construir aos poucos um par de scripts que é muito útil para automatizar procedimentos de backup e de transporte de dados em disquetes ZIP.

1. Antes de mais nada, é preciso saber como escrever um arquivo contendo um programa de shell de forma que ele possa ser executado pela shell. A forma mais simples de fazer isto é simplesmente escrever um arquivo contendo linhas de comando, exatamente como você as utilizaria interativamente no terminal. Escreva um arquivo chamado `hello.tcsh` com o conteúdo que segue:

   ```
   #
   # Meu primeiro programa de tcsh!
   #
   echo "Hello world"
   ```

   Note que o nome do arquivo é arbitrário, só incluímos a terminação `.tcsh` para nosso próprio benefício, para ficar mais fácil lembrar que este arquivo contém um programa de `tcsh`. Observe o uso do caracter `#` para denotar comentários.

   Um cuidado que você deve tomar, apesar de que as versões mais recentes dos principais editores disponíveis no sistema parecem tomar conta disto automaticamente, é certificar-se de que haja um caracter `^J` no final da última linha do arquivo. Este é o caracter de controle que é produzido pela tecla `[Enter]` e que, do ponto de vista da shell, corresponde à instrução de execução do comando que está escrito naquela linha. Este caracter não é mostrado explicitamente pelos editores, é claro, a sua falta é invisível, mas ela faz com que a última linha não seja interpretada e executada pela shell e, portanto, que o seu programa não funcione corretamente.

Para ter certeza de que este caracter de controle esteja no arquivo, basta colocar o cursor do editor no início da linha vazia que segue a última linha do seu programa, antes de escrever o arquivo e sair do editor. Para verificar um arquivo depois de já ter saído do editor, mostre o conteúdo dele na tela com o comando `cat <arquivo>` e verifique onde a shell coloca o seu prompt depois de mostrar o conteúdo do arquivo. Se o prompt aparecer como uma continuação no final da última linha do programa, então o caracter `^J` está faltando no final daquela linha. Caso contrário, o prompt deve aparecer no lugar usual, o início da próxima linha.

2. Vamos agora descobrir como executar este programa. Há várias formas, antes de mais nada você pode submetê-lo à shell usando a linha de comando

   ```
   tcsh hello.tcsh
   ```

   Se for executada com um arquivo como argumento, a shell lê e interpreta o arquivo. Se não houver argumento ela entra em modo interativo, exatamente como a shell de sua sessão. Tente executar a shell sem argumento para ver o que acontece. Lembre-se de sair em seguida da nova shell que é criada, usando o comando `exit`, retornando à shell original de sua sessão.

3. Você deve ter observado que, apesar de ser extremamente simples, o programa demora um pouquinho para ser executado. Isto se deve ao fato de que a shell está executando os seus arquivos de inicialização antes de ler, interpretar e executar o programa. Estes arquivos são muito maiores e mais complexos que este pequeno programa e destinam-se primordialmente a configurar o seu ambiente interativo, ou seja, eles não são úteis neste contexto. Podemos tornar a execução mais rápida

com a opção `-f`, que faz com que a shell não leia arquivos de inicialização. Tente executar a linha de comando

```
tcsh -f hello.tcsh
```

e verificar este fato.

4. Bom, está tudo muito bem, mas seria ainda melhor se pudéssemos executar o programa digitando o nome do arquivo, como fazemos com todos os comandos, programas e aplicativos do sistema. Ora, não tema, com Linux não há problema! De fato, podemos fazer exatamente isto, modificando ligeiramente o nosso programa para incluir uma primeira linha contendo uma construção especial. Modifique o seu arquivo como segue:

```
#!/bin/tcsh -f
#
# Meu primeiro programa de tcsh!
#
echo "Hello world"
```

O "comentário" contido na primeira linha é uma estrutura especial que informa o sistema operacional de que o conteúdo do arquivo deve ser executado pelo programa cujo path completo segue o sinal de exclamação. Qualquer programa que aceite e interprete comandos poderia ser usado desta forma. Em nosso caso, trata-se da shell `tcsh`, com a opção `-f`, para tornar mais rápida a execução do programa.

O editor `emacs` detecta esta estrutura especial na primeira linha e automaticamente coloriza o buffer segundo as regras de sintaxe da shell. Depois de fazer a mudança, saia do editor e volte a entrar, já com o novo arquivo, para verificar este fato. A colorização ajuda muito na escrita de programas corretos e bem estruturados.

5. Caso você já tenha tentado digitar o nome do arquivo para executar o programa, deve ter sido cumprimentado pela desagradável mensagem de erro `hello.tcsh: Permission denied`. De fato, está faltando alguma coisa. O que falta é informar o sistema de que este é um arquivo

executável, sem o que ele nem tenta descobrir o que está dentro dele. Para ver isto, execute a linha de comando

`ls -l hello.tcsh`

e verifique que o arquivo não é, de fato, executável, como está descrito pelas letras logo no início da linha, que devem ser `-rw-rw-r--`. Para tornar o arquivo executável use a linha de comando

`chmod +x hello.tcsh`

e verifique, usando novamente o `ls -l`, que as letrinhas mudaram para `-rwxrwxr-x`, indicando que agora há autorização de execução para este arquivo. Muito bem, agora execute o programa digitando no terminal o nome do arquivo.

6. Um programa implementado através de um script de shell pode ler e manipular argumentos de linha de comando, exatamente como qualquer outro comando do sistema. Escreva um segundo programa chamado `echo.tcsh` contendo a estrutura mostrada abaixo. Não se esqueça de tornar o arquivo executável.

```
#!/bin/tcsh -f
#
# Meu segundo programa de tcsh!
#
echo $1 $2 $3 $4
```

A variável cujo valor é retornado por `$n`, onde `n` é algum número inteiro positivo, tem como valor o `n`-ésimo argumento que foi colocado na linha de comando depois do comando em si. Para verificar isto execute a linha de comando que segue:

`echo.tcsh primeiro segundo terceiro`

Tente repetir isto com números variados de argumentos na linha de comando, desde nenhum argumento até cinco ou mais.

7. Uma forma um pouco diferente de fazer este tipo de coisa é a que está abaixo. A estrutura `$#` retorna o número dos argumentos que foram

colocados na linha de comando, enquanto `$*` é o conjunto de todos os valores destes argumentos.

```
#!/bin/tcsh -f
#
# Meu segundo programa de tcsh!
#
echo Existem $# argumentos de linha de comando:
foreach arg ( $* )
    echo $arg
end
```

Faça as modificações necessárias no arquivo `echo.tcsh` e execute este novo programa. Observe que estamos usando aqui uma das estruturas de loop da shell, que já vimos na aula sobre programação de linha de comando.

Para quem programa em C, a estrutura `$n` é facilmente reconhecida como retornando os valores da variável vetorial `argv[n]`, enquanto `$#` retorna o valor da variável inteira `argc`. De fato, você pode usar nos scripts da shell `$argv[n]` em vez da abreviação `$n`, `$argv` em vez de `$*` e `$#argv` em vez de `$#`. Modifique o seu programa para utilizar estas formas completas das variáveis e verifique que ele funciona exatamente como antes.

8. Como você vê, você está agora em posição para escrever os seus próprios programas e comandos no sistema, armazenando-os de forma permanente em arquivos executáveis, para uso posterior. Assim, sem mais nem menos, você acaba de fazer uma aterrissagem forçada na terra dos programadores de sistema! Bom, agora precisamos descobrir como colocar algo de útil nestes programas de shell.

   Vamos adotar como projeto escrever um par de comandos para fazer e recuperar, de forma automática, backups de um subdiretório de sua conta. Os backups serão feitos em um disquete ZIP em formato `ext2`, a ser montado no diretório `/zip/ext2/`. Vamos assumir que a unidade ZIP seja interna, localizada no dispositivo `/dev/hdb`, como é o caso das que estão disponíveis na sala do Projeto Pró-Aluno. Vamos assumir também que exista em `/dev` um link chamado `/dev/zip` apontando para o dispositivo que de fato corresponde ao ZIP.

Se este esquema de trabalho não é apropriado para você por qualquer motivo, mude-o à vontade. Os detalhes aqui não são tão importantes, o que conta é ter um projeto e saber como programar para realizá-lo. Como simples exercício você pode até usar um floppy tradicional de 1.44 MB em vez do ZIP, limitando correspondentemente o tamanho do seu backup. As únicas diferenças importantes neste caso são os fatos de que não se pode particionar um floppy deste tipo e de que o dispositivo que corresponde a ele não é `/dev/zip` e sim algo como `/dev/floppy`, que em geral é um link para `/dev/fd0`.

Outra possibilidade interessante é simular uma partição de um ZIP usando um arquivo em um disco rígido, que pode ser montado como um filesystem usando-se o dispositivo "loopback". Este tipo de montagem tem de ser configurado e autorizado pelo gerente do sistema de forma que ela possa ser feita pelos usuários. Isto está autorizado nos terminais da sala Pró-Aluno, para um arquivo chamado `/temp/loopimage`. Aqui está a sequência de operações para criar e formatar um arquivo que simula uma partição de aproximadamente 96 MB. Primeiro vai-se para o diretório `/temp` e cria-se lá um arquivo contendo zeros binários com o tamanho correto, usando o comando

```
dd if=/dev/zero of=zip1.dd bs=1024 count=98303
```

Em seguida cria-se a estrutura de um filesystem do tipo ext2 dentro deste arquivo, usando o comando

```
mke2fs zip1.dd
```

O comando irá detetar que trata-se de uma arquivo comum e não de um dispositivo, solicitando confirmação da operação, que você deve dar digitando `y` quando solicitado. Você também pode colocar um label no filesystem, digamos a palavra "backup", usando o comando

```
e2label zip1.dd backup
```

Para poder montar no sistema o filesystem que o arquivo contém, você primeiro faz um link apontando para ele com o nome "loopimage",

```
ln -s zip1.dd loopimage
```

e depois o monta no diretório `/loop/ext2` usando o comando

```
mount /loop/ext2
```

Depois de ter enchido o filesystem com os seus conteúdos e de desmontá-lo do sistema, você pode escrever este filesystem num ZIP de verdade. Para isto, ele deve estar formatado com uma única partição, que deve ser a primeira, ocupando todo o disco. Use o comando `cfdisk` para fazer isto, se for necessário. Depois basta passar a imagem para um terminal onde exista uma unidade ZIP e usar neste terminal o comando

```
dd if=zip1.dd of=/dev/zip1
```

Depois disto você pode passar a tratar este ZIP como o faria normalmente. Não se esqueça de remover o link `loopimage` depois de terminar de fazer uso dele, para que outros usuários possam fazer a mesma coisa.

9. Obtenha um floppy ZIP de 100 MB vazio. Usando o programa `cfdisk`, particione o floppy colocando todo o espaço na partição 1. Caso o floppy ainda esteja particionado como vem da fábrica, você vai ter de primeiro deletar a partição 4, que é a que vem ativada por default. Para usar o programa de particionamento digite

```
cfdisk /dev/zip
```

e use os menus que aparecem, usando as teclas de cursor (não o mouse). Estamos assumindo que existam links apropriados no diretório `/dev/`, tais como os que aparecem a seguir:

```
lrwxrwxrwx   1 root   root     3 May  6 09:42 zip -> hdb
lrwxrwxrwx   1 root   root     4 Oct 12  1998 zip1 -> hdb1
lrwxrwxrwx   1 root   root     4 May  6 09:42 zip4 -> hdb4
```

Para que você possa fazer isto, você precisa estar autorizado no sistema a escrever no dispositivo que corresponde ao ZIP. Ou, se você estiver fazendo isto em seu próprio sistema residencial, pode fazê-lo como root. No caso dos sistema da sala Pró-Aluno, os dispositivos estão abertos, como mostra o output do comando `ls -l`:

```
brw-rw-rw-   1 root   disk        3,  64 Jul 20  1998 hdb
brw-rw-rw-   1 root   disk        3,  65 Jul 20  1998 hdb1
brw-rw-rw-   1 root   disk        3,  68 Jul 20  1998 hdb4
```

Dê também um "maximize" na partição 1, usando o botão apropriado. Note que o programa `cfdisk` não escreve, de fato, nada no disco até que você ordene explicitamente que ele o faça, usando o botão "write".

10. Tendo particionado o ZIP na partição 1, crie nela um filesystem `ext2`, usando a linha de comando

    ```
    mke2fs /dev/zip1
    ```

    Agora você já pode montar o ZIP em algum diretório. Como root você pode usar para isto uma linha de comando como

    ```
    mount /dev/zip1 /zip/ext2
    ```

    desde que já exista o diretório `zip`. Nos sistemas da sala Pró-Aluno já existe uma entrada de montagem no arquivo `/etc/fstab`, com opções que permitem que um usuário qualquer monte o ZIP usando simplesmente o ponto de montagem, como na linha de comando

    ```
    mount /zip/ext2
    ```

11. Seja qual for o seu caso, verifique que você pode de fato montar o ZIP e, tendo feito isto, vá para o diretório `/zip/ext2/` e veja o que há por lá. Exceto por um diretório especial, chamado `lost+found`, que é criado pelo comando `mke2fs`, o ZIP deve estar vazio. Verifique, usando `ls -al`, que os diretórios `..` e `lost+found` pertencem a root, enquanto o diretório `.` pertence a você. Volte para a raiz de sua conta.

    Estamos agora prontos para iniciar um sistema de backups neste ZIP. Vamos assumir, a título de exemplo, que haja um subdiretório de sua conta chamado `teste`, dentro do qual há alguns arquivos. Vamos adotar o padrão de que cada subdiretório de sua conta ganhará um backup em um subdiretório e mesmo nome do diretório `/zip/ext2/`, onde está montado o floppy. Ou seja, a raiz da sua conta estará mapeada na raiz do floppy e haverá um conjunto de subdiretórios correspondentes.

12. Para começar, precisamos criar o diretório do ZIP que corresponderá ao diretório `teste` em sua conta. Crie um diretório no ZIP usando a linha de comando

    ```
    mkdir /zip/ext2/teste
    ```

    Em seguida entre no subdiretório `teste` de sua conta. Uma forma simples e conveniente de fazer um backup de tudo que houver lá, usando o comando `cp`, é a que segue:

```
cp -af . /zip/ext2/teste
```

Isto fará uma cópia recursiva de tudo que houver dentro do diretório, sobrescrevendo arquivos que já existam no ZIP e preservando datas e ownerships dos arquivos. Ou seja, subdiretórios do diretório teste também serão copiados para o ZIP, com os seus conteúdos. Olhe as páginas do manual do comando `cp` para descobrir o significado de cada uma das opções utilizadas aqui.

13. Poderíamos sem dificuldade colocar estes dois comandos em um arquivo, criando assim um pequeno programa de shell que possa ser usado em outras ocasiões. Escreva um arquivo chamado `putzip` contendo estes comandos, obtendo algo como o que segue:

```
#!/bin/tcsh -f
#
# Um programa para fazer backups.
#
# Cria o diretorio.
mkdir /zip/ext2/teste
#
# Faz o backup.
cp -af . /zip/ext2/teste
```

Note o uso de comentários para explicar o que está sendo feito em cada linha. Habitue-se a colocar comentários em seus programas, é uma boa forma de documentá-los. Entretanto, este programa não é de fato muito útil, pois ele sofre de dois problemas: primeiro, se você tentar rodá-lo haverá um erro, pois o diretório já foi criado anteriormente; segundo, seja onde for que você estiver, ele colocará todos os backups no mesmo diretório `/zip/ext2/teste/` do ZIP, o que não é conveniente.

14. Para resolver o primeiro problema, podemos colocar em nosso programa `putzip` uma estrutura `if` que teste se o diretório a ser criado já existe, obtendo algo como o que segue:

```
#!/bin/tcsh -f
#
# Um programa para fazer backups.
#
# Cria o diretorio, se ele nao existe.
if ( ! -d /zip/ext2/teste ) mkdir /zip/ext2/teste
#
# Faz o backup.
cp -af . /zip/ext2/teste
```

Observe que esta é o mesmo tipo de estrutura `if` que usamos na linha de comando.

15. Como um primeiro passo para a solução do segundo problema, tendo em vista que o diretório do ZIP aparece em vários lugares neste programa, podemos usar uma variável, obtendo algo como o que segue:

```
#!/bin/tcsh -f
#
# Um programa para fazer backups.
#
# Define o diretorio do backup.
set bdir = /zip/ext2/teste
#
# Cria o diretorio, se ele nao existe.
if ( ! -d $bdir ) then
    echo "criando o diretorio no ZIP"
    mkdir $bdir
else
    echo "diretorio no ZIP ja existe"
endif
#
# Faz o backup.
cp -af . $bdir
```

A variável `bdir` representa aqui o "backup directory". Note que passamos a usar uma versão mais completa da estrutura do `if`, para permitir que o programa reporte o que está fazendo em cada caso. Esta estrutura mais completa só pode ser usada em scripts, ela não pode ser usada na linha de comando. A indentação dos comandos é feita para facilitar a leitura e, se você estiver usando o `emacs`, ela poderá ser feita de forma semi-automática com o uso da tecla `[Tab]`.

16. Temos ainda o problema de trocar a referência explícita ao diretório `teste` por algo que seja gerado de forma automática e que sirva em qualquer caso. Ou seja, se estamos em nossa conta no diretório

    `~/<diretorio>`,

    queremos que ele seja mapeado no diretório correspondente

    `/zip/ext2/<diretorio>`

    do ZIP. Podemos fazer isto usando o comando `echo $cwd`, que mostra o diretório onde estamos, pois a variável `cwd` é uma variável mantida pela shell que contém esta informação. Em seguida teremos de filtrar do resultado deste comando a parte que corresponde ao subdiretório de nossa conta. Num sistema onde as contas estejam localizadas em `/home/<username>` isto pode ser feito pela pipeline

    `echo $cwd | cut -d / -f 4-`

    Verifique este fato, executando esta pipeline. Execute inicialmente só o comando `echo $cwd` e depois a pipeline, para entender como ela funciona. Olhe as páginas do manual do comando `cut` (`man cut`) para entender o significado de cada uma das opções utilizadas.

17. Podemos agora utilizar esta estrutura em nosso programinha, para o que será necessário usar também as aspas reversas, que têm o efeito de executar a pipeline que está dentro delas e substituir o resultado no ponto indicado. Faça as mudanças necessárias no arquivo `putzip`, obtendo o que segue:

```
#!/bin/tcsh -f
#
# Um programa para fazer backups.
```

```
#
# Define o diretorio do backup, a
# _ partir do diretorio corrente.
set bdir = /zip/ext2/`echo $cwd | cut -d / -f 4-`
#
# Cria o diretorio, se ele nao existe.
if ( ! -d $bdir ) then
    echo "criando o diretorio no ZIP"
    mkdir -p $bdir
else
    echo "diretorio no ZIP ja existe"
endif
#
# Faz o backup.
cp -af . $bdir
```

Note que acrescentamos a opção `-p` no comando `mkdir`. O objetivo disto é prevenir um erro, no caso de você estar num subdiretório mais profundo em sua conta e algum diretório intermediário ainda não existir no ZIP. Olhe as páginas do manual do comando `mkdir` para entender como isto funciona em detalhe.

18. Neste ponto já temos um programa útil, cujo funcionamento é razoavelmente geral, para fazer backups de subdiretórios de nossas contas. Podemos colocar o programa `putzip` no subdiretório `~/bin/` de nossas contas, o que fará com que ele esteja no nosso "path" e possa ser executado a partir de qualquer lugar. Se você fizer isto, não se esqueça de executar o comando `rehash` para refrescar o conteúdo do seu path.

    Entretanto, ainda podemos melhorar o programa, tentando prevenir possíveis erros e tornar o seu funcionamento mais transparente. Por exemplo, se o floppy não estiver montado teremos um erro, pois neste caso o diretório `/zip/ext2/`, em vez de ser a raiz do ZIP, é simplesmente o de ponto de montagem, um subdiretório da raiz do sistema, onde, como meros usuários, não estamos autorizados a escrever. Não é difícil identificar se o floppy está ou não montado, pois o comando `mount` sem argumentos pode ser executado por qualquer usuário e mostra a coleção de filesystems que estão montados no sistema. Tente fazer isto.

19. Podemos filtrar a linha correspondente ao ZIP do output do comando `mount`, usando para isto a pipeline que segue:

    `mount | grep " /zip "`

    Observe as aspas e os espaços no alvo de procura do comando `grep`. Execute esta pipeline para ver o que acontece. Imediatamente em seguida, execute a linha de comando

    `echo $status`

    A variável `status` é outra variável mantida pela shell, que contém sempre o status de saída do último comando executado. Se o seu valor for `0`, isto significa que não houve erro no comando anterior. Neste caso, isto significa que o comando `grep` foi bem sucedido em achar o seu alvo `" /zip "`. Desmonte o ZIP com `umount /zip/ext2` e em seguida repita as duas linhas de comando acima, para ver a diferença. Desta vez, o `grep` não pôde achar o seu alvo, o que ocasionou um sinal de erro na saída, neste caso o valor `1`.

20. Se não estamos interessados na saída do comando `grep` em si, mas apenas no status de saída, podemos usar a opção `-q` (de "quiet", vamos sempre olhar as páginas do manual) do comando. Desta forma, o par de linhas de comando acima pode ser usado para detectar se o ZIP está ou não montado. Poderíamos construir o seguinte bloco de código para efetuar a montagem apenas no caso em que o ZIP não está montado:

    ```
    #
    # Verifica se o ZIP esta montado ou nao.
    mount | grep -q " /zip "
    #
    # Se ele nao estiver montado entao monta.
    if ( $status != 0 ) mount /zip/ext2
    ```

21. Podemos colocar neste bloco mais uma alternativa de deteção de erro, pois é possível que a montagem do ZIP não funcione, por exemplo porquê você esqueceu de colocar o disquete no drive. Para isto podemos melhorar um pouco a estrutura do `if`, usando dois, um dentro do outro, de forma a utilizar duas vezes o mecanismo de deteção de erro da variável `status`. Com isto obtemos um bloco como o que segue:

```
#
# Verifica se o ZIP esta montado ou nao.
mount | grep -q " /zip "
#
# Se ele nao estiver montado entao monta.
if ( $status != 0 ) then
    mount /zip/ext2
    #
    # Se a montagem falha, desiste.
    if ( $status != 0 ) then
        exit 1
    endif
endif
```

Observe o comando `exit` com argumento `1`, o que significa que, se chegarmos a este ponto do código, o programa terminará com status de erro de saída com valor `1`. Este valor será retornado à shell de sua sessão, que a colocará na variável `status`, para uso subsequente.

22. Colocando este bloco de código em nosso programa e fazendo mais alguns aperfeiçoamentos acabamos com algo como o que segue:

```
#!/bin/tcsh -f
#
# Um programa para fazer backups.
#
# Verifica se o ZIP esta montado ou nao.
mount | grep -q " /zip "
#
# Se ele nao estiver montado entao monta.
if ( $status != 0 ) then
    echo "`basename $0`: montando o ZIP"
    mount /zip/ext2
    #
    # Se a montagem falha, desiste.
    if ( $status != 0 ) then
        echo "`basename $0`: ERRO ao montar o ZIP"
```

```
        exit 1
    endif
endif
#
# Define o diretorio do backup, a
# _ partir do diretorio corrente.
set bdir = /zip/ext2/`echo $cwd | cut -d / -f 4-`
#
# Cria o diretorio, se ele nao existe.
if ( ! -d $bdir ) then
    echo "`basename $0`: criando o diretorio no ZIP"
    mkdir -p $bdir
else
    echo "`basename $0`: diretorio no ZIP ja existe"
endif
#
# Faz o backup.
echo "`basename $0`: fazendo o backup"
cp -af . $bdir
#
# Desmonta o ZIP.
echo "`basename $0`: desmontando o ZIP"
umount /zip/ext2
```

Observe que melhoramos um pouco as mensagens retornadas pelo programa. Dentro de um script como este a variável `0` tem sempre como valor o path completo do arquivo executável que contém o programa que está sendo executado. Já o comando `basename` extrai do seu argumento, que deve ser um path, o nome do arquivo, ou seja, a parte que está à direita do último caracter `/` no path. Tente testar isto, executando, por exemplo as linhas de comando

`basename $cwd`

e

`basename $HOME`

Desta forma estamos implementando um dos padrões informais do sistema, segundo o qual as mensagens de erro devem acusar qual foi o programa que as gerou. Finalmente, observe que tomamos o cuidado de desmontar o ZIP ao final do programa. Assim você pode colocar o ZIP no drive, fazer o backup de um diretório e retirar o ZIP, sem se preocupar com montagens e desmontagens. Desta forma, esperamos estar tornando o uso do programa mais fácil e transparente. Naturalmente, se estamos ou não tendo sucesso depende das circunstâncias e do gosto de cada um. O ponto de tudo isto é que você pode fazer as coisas você mesmo, da forma como quiser, de forma a customizá-las de acordo com as suas preferências pessoais.

3 Problemas e Dicas

1. Escreva um programa chamado `getzip` para fazer o inverso do `putzip`, ou seja, recuperar do backup todos os conteúdos de um determinado diretório. Tome todos os cuidados necessários e tente prever todos os possíveis erros. Em particular, tome cuidado com a possibilidade de sobrescrever acidentalmente arquivos que já existam em sua conta. Neste caso pode ser uma boa idéia não usar a opção `-f` do comando `cp`. O que se pode fazer em vez disso?

2. Considere o comando `e2label`, que permite colocar e ler labels em um filesystem `ext2`, leia as suas páginas de manual. Note que, uma vez colocados, os labels são visíveis dentro do `cfdisk`. Usando estes labels você pode ter vários floppies ZIP de backup, cada um para uma finalidade diferente. Além disso, a ausência do label correto pode ser usada para que você não escreva por acidente no floppy errado, possivelmente perdendo dados em algum floppy que não esteja sendo usado para backup e sim para outra coisa. Considere melhorar os seus programas para detectar os labels dos floppies e agir de acordo.

3. A maior parte dos programas existentes em `/etc/init.d/` são scripts de `sh`, a shell original do sistema, da qual a `bash` é a descendente moderna. De fato, é a `bash` que é usada para executar estes scripts, pois ela pode ser chamada também pelo nome de `sh`, caso em que emula de forma bem próxima a "Bourne shell" tradicional. Vá dar uma boa olhada nestes scripts do sistema e veja se você consegue identificar os elementos da linguagem de programação da `bash`. Será que você

consegue traduzir um de seus scripts de `tcsh` para a linguagem da `bash`?

4. Existem alguns scripts de `tcsh` nos diretórios

   `/root/bin/` e `/usr/local/cbin/`,

   dê uma olhada neles e veja se consegue descobrir suas funções. Deve haver também scripts de Perl e de outras shells nos diretórios

   `/usr/bin/` e `/usr/lib/cgi-bin/`,

   bem como em outros lugares. Usando uma combinação dos comandos `find`, `file` e `grep` você deve ser capaz de achar todos eles. Cace alguns por aí, leia-os e veja se consegue entender a função de scripts variados de várias shells, vamos ver que tipo de coisa as pessoas fazem com scripts de shell!

5. É possível escrever scripts de shell que atuam em um "cluster" de máquinas dentro de uma rede. Naturalmente, as shells remotas `rsh` ou `ssh` terão frequentemente um papel a desempenhar neste tipo de script. Considere como escrever scripts a serem executados na sala do Projeto Pró-Aluno, onde existem vários sistemas em rede. Considere possibilidades tais como descobrir quem está rodando que programas em cada uma das máquinas, ou que programas estão consumindo mais recursos dos vários sistemas. Lembre-se de que nem sempre todas as máquinas estão rodando Linux. Imagine meios de controlar os inevitáveis erros devido a este fato.

4 Uma Lista de Comandos Relevantes

Estão sendo colocados aqui apenas os comandos que não apareceram em nenhuma aula anterior.

`basename <path>` : retorna o nome do arquivo contido na path que é submetida como argumento.

`cfdisk /dev/<device>` : é um programa interativo para particionar o disco cujo dispositivo é submetido como argumento; o dispositivo precisa corresponder a um disco, que pode ser IDE (`/dev/hd?`), SCSI (`/dev/sd?`) ou um ZIP (que tanto pode ser IDE quanto SCSI).

`chmod <modo> <arquivo>` : moda as opções de controle de acesso do arquivo; o primeiro argumento é o modo ou a mudança de modo que se quer implementar; o segundo argumento é o arquivo (ou lista de arquivos) sobre o qual se quer que o comando atue.

`e2label <device> <label>` : coloca um label em um filesystem tipo `ext2`, que deve estar contido no dispositivo que é submetido como argumento.

`rehash` : refaz a lista de executáveis contidos nos diretórios que constam do path; é um comando built-in da shell `tcsh`.