Shell Script para Iniciantes

Piter PUNK
a.k.a. Roberto Freires Batista

2004

Sumário

Capítulo 1

Conhecendo o BaSH

Bem-vindos ao curso de *shell script*. Vamos começar mostrando um pouco o nosso grande companheiro, o *Bourne Again SHell* (um trocadilho em homenagem ao criador do `sh` original, Steven Bourne), conhecido também como `bash`. O `bash` é compatível com a `sh` original, assim como a Korn Shell (`ksh`) e a Z Shell (`zsh`). No mundo *NIX, existe uma segunda família de *shells*, descendente da `csh`, com uma sintaxe razoavelmente diferente. Exceto prova em contrário, a sintaxe revelada aqui é referente ao `bash`.

A principal utilidade de uma *shell* é fazer com que você possa utilizar o sistema operacional. É a *shell* que lê e interpreta os seus comandos, e então lê os outros programas ou realiza a operação solicitada (ou um *mix* destes dois). Sem a *shell*, o computador seria muito pouco útil (vamos lembrar que alguns sistemas de janelas, como o *X Window System*, são uma espécie de *shell*, em que os comandos são substituídos por hieroglifos, em um retorno aos tempos do Antigo Egito).

Não existe vida como nós a conhecemos fora da *shell*. Além das suas bordas, vivem programas malvados e sinais inescrupulosos, ávidos pelo sangue de inocentes. Mas, dentro da *shell* a vida é feliz, os usuários executam os seus comandos, os programadores seus ambientes de programação repletos de `make`´s, `cc`´s, `autoconf`´s, etc... E os administradores regem todo este sistema harmônico, através de comandos cabalísticos e *scripts* místicos...

O objetivo do curso é fazer com que você consiga criar estes *scripts* e automatizar as suas tarefas, facilitando a sua vida. O *shell scrip*t é um linguagem poderosa, já que une o `bash` à vasta gama de utilitários presentes em um sistema *NIX. E a sintaxe utilizada para o `bash` é extremamente simples.

1.1 *Prompt*, comando...

Um dos primeiros usos para a *shell* é ser utilizada como mecanismo de entrada e saída, nós passamos comandos para o sistema através da *shell*, e os comandos nos retornam suas saídas por ela. Um bom exemplo disto está logo a seguir, onde é executado o comando `ls` com a opção `-la` e é retornada a listagem dos arquivos deste diretório:

```
punk@rachael:~/lala$ ls -la
total 12
drwxr-xr-x    2 punk     users        4096 2003-12-08 00:27 ./
drwx--x--x   95 punk     users        8192 2003-12-07 22:28 ../
punk@rachael:~/lala$
```

Neste pequeno exemplo já podemos identificar um grande elemento do *shell*, o *prompt*. O *prompt* pode ser diferente de *shel*l para *shell* e é alterado até mesmo enquanto estamos utilizado a *shell*, ele serve para indicar que o sistema está pronto para receber o próximo comando. No exemplo acima, o *prompt* é: `punk@rachael:~/lala$`. Normalmente o *prompt* é simbolizado por um `#` ou `$`.

O segundo elemento que vemos é o comando a ser executado. Para um curso como este, é interessante que o aluno esteja familiarizado com alguns comandos básicos como o `ls`, `who`, `ps`, etc... caso contrário, pode ser que ele fique "boiando". Boa parte dos comandos a ser utilizados seguem a seguinte sintaxe:

```
comando -opções --opções-gnu argumento1 argumento2 ... argumentoN
```

As partes obrigatórias aí são o comando e o `argumento1` (e, dependendo do comando em questão, o `argumento2`). O restante é opcional, e é utilizado de acordo com o efeito final que queremos dar ao comando.

1.2 Juntando comandos...

A filosofia *NIX é a de que devem existir vários pequenos comandos e que união deles gere efeitos poderosos. Por exemplo, o `ls` serve para listar todos os arquivos de um determinado diretório. Mas existem alguns diretórios que possuem uma quantidade interminável de arquivos... como `/usr/bin` por exemplo. Para visualizar todos os arquivos deste diretório iríamos precisar de um paginador, para que pudéssemos ver uma página de cada vez...

Aí é que entra a idéia de "juntar" os comandos, utilizando *pipes* nós podemos direcionar a saída de um comando diretamente para outro:

```
$ ls /usr/bin | less
```

Pronto! Com o uso de um *pipe* (representado pelo `|`), redirecionamos a saída do `ls` para um `less`, que é um paginador. Poderíamos fazer algo semelhante para descobrir quantos diretórios temos em um outro determinado diretório... por exemplo

```
$ ls -l ~/ | grep -c "^d"
```

Neste caso, a saída do comando `ls` está sendo passada ao `grep`. Como isto está funcionando? Primeiramente, é listado o diretório `HOME` do usuário (o `HOME` é representado por um `~/`), todos nós sabemos que quando é utilizada a opção `-l`, é mostrada uma listagem detalhada, em que, entre outras coisas, temos a informação da natureza de um arquivo, um diretório é identificado por um "d", na primeira posição da linha em que aparece o arquivo. É este "d" que estamos contando com o comando `grep`.

1.3 Variáveis e aspas...

O *prompt*, que nós vimos na seção 1.1 é uma variável do sistema. Você pode mudar o *prompt* para o que você quiser. Algumas pessoas gostam de colocar o nome delas no *prompt*, outras gostam de colocar o nome do computador em que elas estão trabalhando... outras ainda gostam de colocar estes e mais zilhões de informações, enquanto existem alguns que vivem felizes com apenas um cifrão ou um sustenido como prompt.

Podemos ver o conteúdo de uma variável com o comando `echo`, que serve para mostrar alguma informação na tela, como o conteúdo de variáveis e textos:

```
punk@rachael:~$ echo $PS1
\u@\h:\w\$
```

Por incrível que pareça, este `\u@\h:\w\$` é o prompt. O `\u` é o nome do usuário, o `\h` é o *host* (ou máquina) em que ele está, o `\w` serve para mostrar o diretório de trabalho e o `\$` é um indicador, fica `$` quando se está usando um usuário comum e troca para `#` quando é o root do sistema. O `@` e o `:` são apenas um @ e um : mesmo.

Dar um valor para uma variável é extremamente simples, basta fazer:

```
punk@rachael:~$ PS1="umdoistrês "
umdoistrês echo $PS1
umdoistrês
umdoistrês
```

Pronto! Demos à variável `PS1` (que é o *prompt*) o valor de "umdoistrês " que, diga-se de passagem, é um nome horrível para um prompt. Outra variável de sistema interessate é o $HOME, que indica onde é o diretório pessoal de determinado usuário, podemos fazer:

```
umdoistrês echo $HOME
/home/punk
```

Podemos também dizer que:

```
umdoistrês echo "O prompt do $USER é $PS1 e o seu home é $HOME"
O prompt do punk é umdoistrês  e o seu home é /home/punk
```

Agora... e se quiséssemos escrever "$HOME"? Como fazer? Simples, poderíamos apenas mudar as aspas que estamos utilizando...

```
umdoistrês echo '$HOME é a variável que guarda o home de um usuário'
$HOME é a variável que guarda o home de um usuário
```

Mas, com isso, temos de novo um problema... e para colocar na mesma saída a palavra "$HOME" e o conteúdo da variável `HOME`? Uma solução seria o uso racional das aspas...

```
umdoistrês echo '$HOME='$HOME
$HOME=/home/punk
```

Ou então no uso do \, que é um caracter de escape, isso faz com que o *shell* não interprete o caracter que vem logo depois do \:

```
umdoistrês echo "\$HOME=$HOME"
$HOME=/home/punk
```

Questão de gosto, e de momento. Podemos também criar novas variáveis, com conteúdos variados... por exemplo:

```
umdoistrês MEU_NOME="Punk"
umdoistrês echo $MEU_NOME
Punk
```

Mas, às vezes, queremos que uma variável assuma o valor de um comando... Por exemplo, podemos fazer:

```
umdoistrês uname
Linux
```

E com isso sabemos sob qual Sistema Operacional estamos rodando. Talvez valha a pena guardar essa informação para ser utilizada um script. Se vale a pena, vamos armazenar a saída do comando `uname` em um variável (iremos chamar esta variável de `SO`).

```
umdoistrês SO=`uname`
umdoistrês echo $SO
Linux
```

Perceba o uso que demos às ‘ ‘, com elas podemos pegar a saída da execução de um comando. Para diminuir a confusão e facilitar a leitura, muita gente tem utilizado uma sintaxe mais nova, que substitui as ‘ ‘, por `$()`:

```
umdoistrês SO=$(uname)
umdoistrês echo $SO
Linux
```

Por convenção, as variáveis normalmente são escritas em maiúsculas, para ficarem destacadas em meio ao texto do *script* (que é escrito em minúsculas). Também são utilizadas as chaves `{}` para delimitar uma variável, por exemplo:

```
umdoistrês SO2="nada"
```

Agora, desejamos escrever “Linux2”, utilizando a variável `SO` que já tínhamos definida:

```
umdoistrês echo $SO2
nada
```

Aí entram as chaves:

```
umdoistrês echo ${SO}2
Linux2
```

Se este uso não é suficiente, as chaves podem ter uma utilidade muito mais nobre, vamos pensar na saída do comando `uname -a`:

```
umdoistrês uname -a
Linux rachael 2.4.22 #6 Tue Sep 2 17:43:01 PDT 2003 \
i686 unknown unknown GNU/Linux
```

Aí estão todos os dados disponíveis sobre a máquina na qual estamos, qual o sistema operacional, qual o nome dela, qual a versão, quando foi compilado, etc... Podemos querer endereçar cada um deles e não apenas o primeiro... para estes casos, existe como fazer vetores em `bash`:

```
umdoistrês UNAME=(‘uname -a‘)
umdoistrês echo ${UNAME[1]}
rachael
```

Atenção para o uso:

1. das aspas invertidas na definição do vetor UNAME. Em `bash`, cada um vetor é inicializado como uma série de valores, separadas por espaço dentro de (). Como a saída do comando `uname` já é separada por espaços, basta utilizar ela mesma, executando o comando entre ‘.

2. das chaves ao passar o elemento do vetor. Sem elas, o *shell* iria apresentar o conteúdo de UNAME e, ao lado, um [1] meio perdido.

1.4 Redirecionando...

Outro recurso interessante do *shell* é o de direcionar a saída de um comando para um arquivo. Por exemplo, podemos fazer:

```
punk@rachael:~$ echo Walla > arquivo_teste
punk@rachael:~$ cat arquivo_teste
Walla
```

Ou seja, a saída do comando `echo`, ao invés de aparecer na tela, apareceu diretamente no arquivo que selecionamos após o > (`arquivo_teste`). Preste bastante atenção no >, e verá que ele se parece muito com uma seta, indicando o fluxo para onde vai a informação.

Uma pequena experiência:

```
punk@rachael:~$ echo Salim > arquivo_teste
punk@rachael:~$ cat arquivo_teste
Salim
```

Pudemos ver que o > sobrescreve o arquivo anterior, se ele existir. Ou seja, é uma péssima idéia para um arquivo de *logs*, onde devemos sempre acrescentar alguma informação em um arquivo que já existe. Para fazer esse tipo de coisa, existe o comando >>, como vemos abaixo:

```
punk@rachael:~$ cat arquivo_teste
Salim
punk@rachael:~$ echo slovoboda >> arquivo_teste
punk@rachael:~$ cat arquivo_teste
Salim
slovoboda
```

Ta-dá! Colocamos os dados logo abaixo no nosso arquivo! Se fosse só isso, esse recurso de redirecionar para arquivo já iria ser a oitava maravilha do mundo moderno mas, como diziam os comerciais do 1406, não é só isso!

Podemos usar os sinais de < e << como entrada de dados também!!!

```
punk@rachael:~$ echo "texto todo em minúsculas" > arquivo_teste
punk@rachael:~$ tr a-z A-Z < arquivo_teste
TEXTO TODO EM MINúSCULAS
```

O comando `tr`, substitui uma cadeia de caracteres por outra. No nosso caso, substituímos as letras minúsculas `a-z`, pelas maiúsculas `A-Z`. E, como visto, utilizamos o conteúdo do arquivo `arquivo_teste` como entrada. Podemos usar também o <<, como veremos abaixo:

```
punk@rachael:~$ sh <<EOF
> cat /etc/passwd | grep ^root
> EOF
root:x:0:0::/root:/bin/bash
```

Quando dizemos `<<EOF` estamos avisando para ele usar como entrada tudo que escrevermos, até chegar um `EOF` (*End Of File*). Então, o que dissemos no comando acima foi: execute o `sh`, e use como entrada para ele os dados que forem sendo escritos até chegar um `EOF`. Atenção para o `>`, que é conhecido como `PS2`, e é o segundo prompt do shell (o `PS1` a gente já sabe qual é...)

1.5 Um pouco de aritmética...

Temos variáveis, redirecionamentos, atribuições... mas ainda não temos como fazer um simples 2+2. Experimente:

```
punk@rachael:~$ 2+2
-bash: 2+2: command not found
```

Não é bem assim que vamos conseguir fazer nossas operações matemáticas mas, se este não é o modo de realizar, qual seria?

O primeiro e mais tradicional é o uso de alguns comandos externos, como o `expr` e o `bc`. Podemos ver o uso dos dois logo abaixo:

```
punk@rachael:~$ expr 2 + 2
4
punk@rachael:~$ echo 2+2 | bc -l
4
```

O `bc` é um pouco mais difícil de trabalhar, sendo que precisamos passar a expressão matemática para ele via *pipe*... ou via arquivo. Mas ele é muito mais poderoso que o `expr`, que possui algumas limitações irritantes:

1. Só faz operações com inteiros e

2. Não aceita a multiplicação com o `*`, sendo necessário utilizar `\*`

Uma terceira maneira é utilizar `$(())`, podemos ver seu uso na linha abaixo:

```
punk@rachael:~$ echo $((2+2))
4
```

O `$(())` aceita várias operações como: adição (+), subtração (-), divisão (/), resto (%), multiplicação (*), exponenciação(**) além de várias operações lógicas, como E (&), OU (|), shifts ($<<$ e $>>$), etc...

A outra maneira é bem familiar para os usuários de BASIC, consiste no uso do comando `let`:

```
punk@rachael:~$ a=1
punk@rachael:~$ b=2
punk@rachael:~$ let a=a+b
punk@rachael:~$ echo $a
3
```

Todas as operações apresentadas para o `$(())` valem para o `let`, e várias outras que não estão explicitadas aqui. Muita atenção que o `let`, ao contrário de todos os outros comandos, não referencia a variável por $VARIÁVEL e sim, apenas por VARIÁVEL. Cuidado para não esquecer.

1.6 ...e um pouco de lógica

Temos operações lógicas permitidas pelos comandos "matemáticos", mas também temos duas funções importantes "embutidas" no `bash`, E e OU. Podemos utilizar estas duas funções, para selecionar quais comandos devemos executar, por exemplo:

```
punk@rachael:~$ ping -c1 www.linux.org && echo 'Rede OK!' ||\
echo 'Rede Suxou!'
PING www.linux.org (198.182.196.56) 56(84) bytes of data.
64 bytes from www.linux.org (198.182.196.56): icmp_seq=1 ttl=38 time=369 ms

--- www.linux.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 369.214/369.214/369.214/0.000 ms
Rede OK!
```

A resposta está meio suja, mas podemos entender o comando dado:

- `ping -c1 www.linux.org` -> Envia um `ping` para `www.linux.org`, o `-c1` serve para que seja enviado um e apenas um `ping` e depois disso o programa pare
- `&&` -> é o nosso E, com ele nós dizemos que o próximo comando só será executado se o anterior for bem sucedido
- `echo 'Rede OK!'` -> Informa que a rede está OK, já que o comando `ping` foi bem sucedido.
- `||` -> Se o comando anterior não foi bem sucedido, será executado o próximo comando, ou seja OU um OU o outro.
- `echo 'Rede Suxou!'` -> Avisa que a rede está suxada. Isso acontece, fazer o quê? C'est la vie...

Se quisermos deixar o resultado desta nossa linha de comando um pouco mais simpática, podemos fazer um redirecionamento da saída do `ping` para outro lugar... por exemplo:

```
punk@rachael:~$ ping -c1 www.linux.org >/dev/null && \
echo 'Rede OK!' || echo 'Rede Suxou!'
Rede OK!
```

Este `>/dev/null` direciona a saída do `ping` para lugar nenhum. É para onde vão várias coisas, como as reclamações que fazemos em *call-centers* e requisições ao administrador da rede, lembrem-se quando forem administradores de redirecionar todos os e-mails de usuários para o `/dev/null`.

Ao pingar um *host* que não existe, normalmente temos um erro:

```
punk@rachael:~$ ping www.plmo.ogre
ping: unknown host www.plmo.ogre
```

E, na nossa linha de comando também...

```
punk@rachael:~$ ping -c1 www.plmo.ogre >/dev/null && \
echo 'Rede OK!' || echo 'Rede Suxou!'
ping: unknown host www.plmo.ogre
Rede Suxou!
```

Dois problemas: o primeiro é que a nossa linha de comando não funciona direito quando pingamos algum servidor que não existe, mas isso se resolve pingando máquinas existentes; o segundo é mais grave... a mensagem de erro saiu no meio de tudo e sujou de novo a nossa tela, mesmo com o redirecionamento...

Isso acontece porque existem dois canais de saída: o *stderr* (2) e o *stdout* (1). Sendo que um é a saída de erros e o outro é a saída de mensagens. Podemos redirecionar os dois para o `/dev/null`:

```
punk@rachael:~$ ping -c1 www.plmo.ogre 2>/dev/null 1>/dev/null \
&& echo 'Rede OK!' || echo 'Rede Suxou!'
Rede Suxou!
```

Ou fazer isso com os dois ao mesmo tempo:

```
punk@rachael:~$ ping -c1 www.plmo.ogre &>/dev/null \\
&& echo 'Rede OK!' || echo 'Rede Suxou!'
Rede Suxou!
```

Também podemos redirecionar um para o outro, o que pode ser útil algumas vezes. Isso se faz com `2>&1`, por exemplo.

Capítulo 2

Programando

Ainda não é a programação própriamente dita, neste capítulo iremos dar uma rápida passagem pelas estruturas de iteração e seleção do `bash`, além de uma pincelada em outras áreas do conhecimento arcano. Depois de conhecer estas estruturas, vamos colocá-las seqüencialmente em arquivo, o resultado desse processo é o nosso *shell script*.

2.1 Separando o Joio do Trigo...

Bom, o método mais primitivo de seleção é utilizando o `&&` e o `||`, se o comando anterior foi bem sucedido, faça alguma coisa, se não foi, não faça. Isso pode ser um pouco incrementado com o uso do `[` (que às vezes é um *link* para o comando `test`). Veja o exemplo abaixo:

```
punk@rachael:~$ a=1 ; b=2
punk@rachael:~$ [ "$a" = "$b" ]&& echo iguais || echo diferentes
diferentes
punk@rachael:~$ [ ! "$a" = "$b" ]&& echo diferentes || echo iguais
diferentes
```

No primeiro verificamos se a variável `a` é igual à variável `b` e no segundo verificamos o oposto disto.

Podemos utilizar uma variedade interminável de testes como por exemplo:

```
punk@rachael:~$ [ "`ls GNUstep`" ]&& echo O diretório existe
O diretório existe
punk@rachael:~$ [ "`ls GNUstepa`" ]&& echo O diretório existe
/usr/bin/ls: GNUstepa: No such file or directory
```

```
punk@rachael:~$ [ "`ls GNUstepa &>/dev/null`" ]&& echo O diretório existe
punk@rachael:~$
punk@rachael:~$ [ -e GNUstep ]&& echo O diretório existe
O diretório existe
```

Vimos o teste utilizando um comando, um teste dando errado, um teste utilizando o `-e`, um teste com um redirecionamento... isso dá para ter um breve idéia do quanto esses testes são úteis e poderosos...

O próximo passo, é utlizar a trinca `if...then...fi`, como vamos ver logo abaixo:

```
punk@rachael:~$ if [ "$USER" = "punk" ]; then echo "Oi Punk"; fi
Oi Punk
```

Para quem está acostumado a ver isto de maneira mais estruturada, podemos fazer:

```
punk@rachael:~$ sh <<EOF
> if [ "$USER" = "punk" ]; then
>     echo "Oi Punk"
> fi
> EOF
Oi Punk
```

Que é semelhante ao formato que isto terá quando aparecer no seu script. Podemos ainda colocar alguns penduricalhos no `if...then...fi`, como o `else`:

```
punk@rachael:~$ sh <<EOF
> if [ "$USER" = "punk" ]; then
>     echo "Oi Punk"
> else
>     echo "Não é o Punk!"
> fi
> EOF
Oi Punk
```

Além do `if...then...else...fi`, temos outra estrutura de seleção muito interessante, o `case`, podemos dizer que ele é uma espécie de super`if`. Ele permite que comparemos uma variável com muitos valores de uma vez só... por exemplo:

```
case $OPCAO in
        1)
        echo "Você escolheu 1"
        ;;
        2)
        echo "Você escolheu 2"
        ;;
        *)
        echo "Você não escolheu nem 1 nem 2"
        exit
        ;;
esac
```

A variável `OPCAO` está sendo comparada com os valores `1` e `2`. Caso seja escolhida uma destas opções, o comando `echo` irá mostrar na tela a opção escolhida. Se não for nem `1` e nem `2`, será apresentada a mensagem “`Você não escolheu nem 1 nem 2`”. É isso que quer dizer aquele `*`: qualquer opção que não seja nenhuma das anteriores.

Mas a estrutura do `case` é muito comprida... acho que já está na hora de passarmos a escrever estes comandos dentro de algum arquivo.... o resultado disso serão os nossos *shell scripts* propriamente ditos.

2.2 A estrutura de um *shell script*

Primeiro, abra o seu editor de textos favorito (VIm! VIm! VIm!), ou talvez, se o seu editor favorito não estiver disponível, algum outro (elVIs! elVIs! elVIs!). O que interessa é que você saiba editar um texto nele. Agora vamos começar de verdade...

Em todo *shell script*, a primeira linha deve ser:

```
#!/bin/sh
```

Onde o `/bin/sh` é a *shell* utilizada para interpretar o *script*. Por exemplo, se você estiver com um `csh` *script* e quiser rodar no `bash`, o *csh script* deverá ter na primeira linha algo do tipo:

```
#!/bin/csh
```

Como todos os nossos scripts são compatíveis com o `bash`, utilize `/bin/bash` ou `/bin/sh` (que na maior parte dos Linux é um link para o `/bin/bash`). Depois desta linha, vem o *script* propriamente dito, com todos os seus comandos, estruturas e comentários.

Agora podemos tentar fazer algo de útil com o `case` que vimos um pouco acima, escreva o programa abaixo no seu editor de textos e salve como “script1”:

```
#!/bin/sh
echo ‘‘1) Opção 1’’
echo ‘‘2) Opção 2’’
read OPCAO
case $OPCAO in
        1)
        echo "Você escolheu 1"
        ;;
        2)
        echo "Você escolheu 2"
        ;;
        *)
        echo "Você não escolheu nem 1 nem 2"
        exit
        ;;
esac
```

Pronto! Agora temos um maravilhoso *shell script*. Antes de executá-lo, é interessante dar a ele a permissão de execução, isso é feito com o comando:

```
punk@rachael:~$ chmod +x script1
```

Agora pode executar...

```
punk@rachael:~$ ./script1
1) Opção 1
2) Opção 2
1
Você escolheu 1
```

O comando `echo` imprime na tela a lista de opções que queremos, e o `read` serve para que a possamos ler a resposta do usuário. No nosso caso, a resposta é guardada na variável `OPCAO`, que depois é interpretada pelo `case`.

Por enquanto os nossos *scripts* são extremamente curtos (é o que acontece normalmente com *scripts* para iniciantes... são todos curtos -;)), mas não será assim para sempre, podemos fazer um *script* com várias e várias funções... dúzias de variáveis e estruturas insanas... quando isso acontecer, você vai precisar comentar os seus programas... Vamos fazer isso com o programa que acabamos de “escrever”:

```
#!/bin/sh
#
# script1 - O primeiro script a gente nunca esquece
# Mostra opções na tela
#
echo “1) Opção 1”
echo “2) Opção 2”

# Lê a opção escolhida e guarda na variável OPCAO
#
read OPCAO
# Caso a opção seja 1 ou 2, imprime a opção escolhida
# pelo usuário. Se não for, avisa que não foi escolhida
# nem uma, nem a outra.
#
case $OPCAO in
        1)
        echo "Você escolheu 1"
        ;;
        2)
        echo "Você escolheu 2"
        ;;
        *)
        echo "Você não escolheu nem 1 nem 2"
        exit
        ;;
esac
```

Pronto! Agora ficou bem melhor. Quem encontrar o seu código daqui a duzentos anos, perdidos em um monte de papel e camadas de lixo, poderá identificar exatamente o que ele faz. Ou, se você não quiser esperar duzentos anos, qualquer um que encontrar um programa seu perdido em um HD irá conseguir saber o que ele faz e como alterá-lo. Isso sim é bem importante, principalmente se você estiver administrando um sistema e sair de férias (ou for despedido, nunca se sabe).

2.3 Passando parâmetros para os seus scripts

Existem duas maneiras de passar alguma informação nos seus scripts. A primeira a gente já viu, com o comando `read`, em que o usuário digita o que quer durante a execução do *script*. A outra maneira, bem mais eficiente se quisermos executar *scripts* de manutenção, é passar os argumentos diretamente na linha de comando, por exemplo:

```
punk@rachael:~$ ./meuscript
Os argumentos são:
punk@rachael:~$ ./meuscript um dois três quatro
Os argumentos são: um dois três quatro
```

O `bash` fornece várias maneiras de detectarmos em um script quais são os argumentos, quantos são e como endereçar cada um deles, o código do `meuscript`, está listado logo abaixo:

```
#!/bin/sh
echo Os argumentos são: $@
```

Ou seja, com isso nós já aprendemos que o `$@` mostra todos os argumentos da nossa linha de comando. Algumas outras variáveis úteis podem ser vistas na nova versão do `meuscript`.

```
#!/bin/sh
echo Você está executando o $0
echo Foram passados $# argumentos para o script
echo Os argumentos são: $@
echo O primeiro deles é o $1
```

Acho que o *script* em si não precisa de muitas explicações, veja o resultado da execução dele:

```
punk@rachael:~$ ./meuscript um dois tres quatro
Você está executando o ./meuscript
Foram passados 4 argumentos para o script
Os argumentos são: um dois tres quatro
O primeiro deles é o um
```

A explicação rápida é:

- `$#` - Quantidade de argumentos passados na linha de comando
- `$@` - Lista com os argumentos passados na linha de comando
- `$0`- Comando executado
- `$1`, `$2`, `$3`, etc... - Primeiro, Segundo, Terceiro, etc... argumento

Podemos agora até fazer um *script* que só aceite ser executado se houverem argumentos:

```
#!/bin/sh
if ! [ "$#" = "0" ]; then
        echo Você está executando o $0
        echo Foram passados $# argumentos para o script
        echo Os argumentos são: $@
        echo O primeiro deles é o $1
else
        echo Múmia! Precisa de um argumento!
fi
```

Esta é a nossa versão final do `meuscript`. Agora ele só executa se a quantidade de argumentos for diferente de `0`. E, caso não haja nenhum argumento devolve ao usuário uma mensagem de erro educada e explicativa.

2.4 De novo!

Além das estruturas de seleção, outras muito importantes para um programa são as estruturas de iteração. Elas fazem com que um determinado pedaço do código seja repetido durante uma quantidade "x" de vezes (onde essa quantidade "x" pode ser uma quantidade infinita).

O primeiro método a ser utilizado é o `for`. Com ele nós podemos passar por uma série de elementos em uma lista. A sintaxe dele é:

```
for variavel in lista; do
    comando1
    comando2
    ...
    comandoN
done
```

Onde `lista`, são várias opções separadas por espaços, *tab*s, ou *newline*s. Para termos uma idéia de como isso funciona, podemos fazer o seguinte:

```
#!/bin/sh
for i in `ls /tmp`; do
    echo Encontrado o arquivo $i
done
```

O que isso irá fazer? Primeiramente, será executado o `ls /tmp` (lembre que vimos que para passar o resultado de um comando para uma variável, usamos VARIAVEL=‘comando‘, neste caso, estamos passando o resultado do comando como lista para o `for`).

Depois de executado `ls /tmp`, será executado o `for`, sendo que a variável `i` irá assumindo o valor de cada elemento da lista, um por vez. Com isso, podemos ver que o `for` do `sh` não está limitado a utilizar apenas números, podendo usar qualquer coisa como índice. Bastando para isso colocar em uma lista os valores que queremos que `i` assuma, podemos usar o script abaixo como exemplo:

```
#!/bin/sh
echo "Os vertebrados se dividem em:"
for i in Mamíferos Aves Reptéis Anfíbios Peixes; do
        echo -e "\tClasse $i"
done
```

A saída dele é:

```
punk@rachael:~$ ./vertebrados
Os vertebrados se dividem em:
        Classe Mamíferos
        Classe Aves
        Classe Reptéis
        Classe Anfíbios
        Classe Peixes
```

Agora vem a grande dúvida: e para usar o `for` contando um número “x” de vezes? Isso é razoavelmente fácil. A primeira opção é bem simples:

```
for i in 1 2 3 4 5 6 7 8 9 10; do
    echo $i
done
```

Pronto! Isso irá contar de 1 a 10. O procedimento é realmente simples... contanto que sejam poucos os números. Se forem mais, podemos usar como auxiliar direto o comando `seq`. Veja o resultado dele na linha de comando:

```
punk@rachael:~$ seq 1 5
1
2
3
4
5
punk@rachael:~$
```

O comando `seq` conta sozinho a quantidade que quisermos, e conta bonitinho. Forçando um pouco a memória, podemos lembrar que um laço `for`, pode utilizar uma lista separada por espaços, *tab*s ou *newline*s... e a saída do `seq` se enquadra no último caso. Vamos fazer o nosso teste com o seguinte código:

```
for i in `seq 1 10`; do
    echo $i
done
```

Podemos ver o que o resultado é justamente o esperado, uma seqüência de números indo do 1 ao 10. Com isso já sabemos como fazer para ir do 1 ao 500 ou do 1000 ao 3 (sim! o `seq` também faz seqüências decrescentes).

Ainda existe uma outra alternativa para se fazer laços numéricos usando o `for` do `bash`. Para quem tem familiaridade com a linguagem C, isso pode até mesmo parecer natural, mas ainda acho a sintaxe tradicional muito mais intuitiva:

```
for ((i=1; i<11; i++)); do
    echo $i
done
```

Isto fará com que a contagem do `i` se inicie no 1, vá até que o `i` seja igual ou maior que 11 e o incremento é de um em um. Meio criptográfico para quem não conhece (embora quem já conheça talvez ache essa salada mais simples que a idéia da lista).

Outra maneira de fazer uma iteração é utilizando o comando `while`. A sintaxe dele é:

```
while condição; do
    comando1
    comando2
    ...
    comandoN
done
```

Ou seja, será executado o código um zilhão de vezes, enquanto a tal condição for verdadeira. Podemos construir com rapidez um exemplo didático (porém inútil) usando o `while`. Podemos vê-lo logo abaixo:

```
#!/bin/sh
while ! [ "$A" = "3" ]; do
        echo "1) lista arquivos"
        echo "2) lista processos"
```

```
        echo "3) sai"
        read OPCAO
        case $OPCAO in
                1)
                        ls
                ;;
                2)
                        ps
                ;;
                3)
                        A=$OPCAO
                ;;
        esac
done
```

O que este programa faz? Simples... enquanto `A` for diferente de `3`, ele irá apresentar um menu (os comando `echo` estão aí para isso...) e, depois, será lida uma opção. No caso dela ser `1`, serão listados os arquivos do diretório corrente. (Talvez o *script* fique mais interessante com um `| more` depois do `ls`, para deixar a listagem mais acessível. Se a opção for `2`, mostra os processos da *shell* corrente. E, no caso da resposta ser `3`, muda o valor de `A`, para `3` o que irá fazer com que o programa saia do laço `while` e, com isso, o nosso *script* chegará ao fim.

Se quisermos um laço infinito, o comando `while` está entre os mais votados. Basta que a condição seja sempre verdadeira... e temos um comando só para isso -;)

```
while true; do
      echo "Vou encher a tela de lixo! Me pare com CTRL+C"
done
```

O comando `true`, sempre devolve verdadeiro. Enquanto o comando `false` sempre devolve falso. Acredite, os dois tem utilidades e algum dia você vai precisar de um deles (além de usar o `true` em laços infinitos)

2.5 Extra! Extra!

Não pensei em nenhum lugar melhor para colocar essas informações, e elas são muito importantes para quem pretende programar seriamente utilizando *shell scripts*. O primeiro dos truques é o uso de funções. Existem duas maneiras de se declarar uma função:

```
function nome_da_funcao {
    comando1
    comando2
    ...
    comandoN
}
```

Ou

```
nome_da_funcao() {
    comando1
    comando2
    ...
    comandoN
}
```

Os dois métodos são equivalentes. É apenas uma questão de gosto preferir um ou o outro. É sempre bom lembrar que as funções devem estar todas declaradas antes de serem chamadas. Ou seja, normalmente colocamos elas no começo do programa. Podemos ver a declaração e o uso de funções no nosso exemplo:

```
#!/bin/sh
function funcao1 {
        echo Você chamou a função 1 com o argumento $1
        echo E o seu login é $USER
}
funcao2() {
        echo Você chamou a função 2 com o argumento $1
        echo E o seu home é $HOME
}
funcao1 argumento1
read -n1 -p "Qualquer tecla para continuar"
funcao2 argumento2
```

Temos aí duas funções, cada uma declarada de uma maneira. Atenção para o método como são passados os argumentos para a função. Eles são endereçados da mesma maneira que para um novo *script*, o primeiro argumento é `$1`, o segundo é `$2`, o terceiro é `$3` e por aí vai... Este foi o nosso tutorial ultra-rápido de como utilizar funções.

O segundo comando que julgo muito importante passar, é o `trap`. Ele consegue capturar alguns dos sinais para matar o seu programa. E qual a utilidade disso?

Uma delas é fazer com que o seu programa não saia com um simples CTRL+C, a outra, é fazer com que ele saia, mas antes disso coloque a casa em ordem.

Normalmente, a medida que um programa vai sendo executado ele vai criando arquivos temporários, arquivos de *lock*, arquivos de trabalho, diretórios estranhos, etc... e não se pode deixar todo esse monte de tralhas entupindo o sistema. Alguma hora a casa cai. Por isso é comum utilizarmos o comando `trap` em conjunto com uma função para a limpeza do sistema. Algo do tipo:

```
cleanup() {
    rm -rf /tmp/arquivo.tmp /tmp/arquivo2.tmp
}
trap 'cleanup' 2 15
```

Isso irá fazer com que o seu *script* intercepte os sinais 2 e 15 (`SIGINT` e `SIGTERM`). Quando um destes dois sinais chegar, será executada a função `cleanup` que vai limpar as sujeiras que o seu script deixa no sistema. Fechamos o capítulo com chave de ouro -:) Não é sempre que conseguimos um exemplo didático e útil ao mesmo tempo -:)

Capítulo 3

Trabalhando com Textos

Um dos principais usos de um shell script é no trabalho com textos. Desde os primórdios do UNIX, foram criadas várias ferramentas para manipulação de textos e *strings* e essas ferramentas foram algumas das responsáveis pela disseminação do sistema operacional. Como nós já dissemos anteriormente, a filosofia *NIX é a de diversos pequenos utilitários, cada um realizando bem uma tarefa, trabalhando unidos, para atingir o melhor resultado.

Neste capítulo faremos um uso intensivo destes pequenos utilitários e de pipes, para passar a saída de um destes comandos para outro. Como já é comum, não iremos "esgotar" cada comando, mostrando todas as opções, iremos apenas dar uma apresentação e suas principais opções.

3.1 Selecionando textos

Conseguir localizar os textos que nos interessam no meio de dezenas de arquivos é algo importante. Para isso existe o comando `grep`, com ele é possível procurar um determinado padrão em um arquivo, ou na saída de um comando (passada via *pipe*). Por exemplo:

```
punk@rachael:~$ free
             total       used       free     shared    buffers     cached
Mem:        256148     237784      18364          0      19200     136848
-/+ buffers/cache:      81736     174412
Swap:       265032         72     264960
```

O comando `free` mostra a quantidade de memória utilizada pelo sistema, com vários detalhes. Agora, vamos supor que quiséssemos ver apenas a linha em que é mostrado o uso da memória *swap*, como fazer? O comando `grep` resolve o problema:

```
punk@rachael:~$ free | grep Swap
Swap:       265032         72     264960
```

Pronto! Conseguimos mostrar com facilidade apenas a linha que nos interessa! E quando procuramos uma determinada informação em vários arquivos, como fazer?

```
punk@rachael:/etc$ grep -r "192.168.0.2" * 2>/dev/null
hosts:192.168.0.2             rachael.mylab rachael
rc.d/rc.inet1.conf:IPADDR[0]="192.168.0.2"
resolv.conf:nameserver 192.168.0.2
```

O comando utilizado acima mostra bem como fazer -;). A opção `-r` que está no `grep` serve para que o comando funcione de modo recursivo, ou seja, procure no diretório local e em seus subdiretórios. O `192.168.0.2` que está entre aspas é o padrão que estamos procurando (é o IP da minha máquina), o `*` serve como coringa, e indica todos os arquivos e, por fim o `2>/dev/null` redireciona todos os erros para o vazio.

A saída do comando mostra exatamente quais arquivos contém o padrão selecionado e o conteúdo do arquivo nesta determinada linha. Assim, se temos uma idéia do que estamos procurando, podemos editar diretamente o arquivo correto. Ou, se não temos uma idéia, podemos ter alguma dica de onde é melhor procurar -;)

Também é possível fazer construções complexas com o comando `grep` e o auxílio de algumas expressões regulares. Estas tais expressões regulares são alguns conjuntos de caracteres especiais que facilitam o trabalho de encontrar padrões. Alguns exemplos de expressão regular:

- `[0-9]` - Qualquer um dos números de 0 a 9 (0,1,2,3,4,5,6,7,8,9). De maneira análoga temos `[a-z]` e `[A-Z]`, que mostram o conjunto dos caracteres de a a z e de A a Z.

- `[FdGl]` - O padrão neste caso é F maiúsculo, o d minúsculo, o G maiúsculo e o l minúsculo novamente. Podemos combinar este tipo de padrão com o anterior: `[a-fl-z]`, casaria com qualquer letra de a a f e de l a z (mas deixaria de fora o g, h, i, j, k).

- `[^g-k]` - Qualquer caracter que NÃO seja g, h, i, j e k. Quando o `^` aparece dentro de `[]`, ele funciona como sendo uma negação.

- `^root` - Outro uso para o `^`, mas totalmente diferente: fora dos `[]`, o `^` representa o início de uma linha e, neste caso, estaríamos procurando uma linha que começasse com a palavra `root`.

- `root$` - O `$` representa o final de uma linha. Ou seja, o nosso padrão neste caso é uma linha que terminasse com a palavra `root`.
- `.*` - Qualquer coisa. Literalmente. Se quisermos por exemplo procurar algo que tenha `lala` e `lele` e algo (que não sabemos o que) entre essas duas palavras, podemos usar a expressão: `lala.*lele`. Se utilizarmos o `.` sozinho (sem o `*`) iremos casar qualquer caracter (mas apenas UM caracter).

Estes exemplos não cobrem nem 10% do que as expressões regulares são capazes de fazer, mas são suficientes para muitos usos. Vamos dar uma olhada no uso de expressões regulares utilizando um novo exemplo: quando o espaço em disco de uma máquina acaba, ou fica escasso, um monte de pequenos (e grandes) problemas podem acontecer. Para que isso seja evitado, é necessário que seja verificado constantemente a quantidade de disco utilizada e isso pode ser feito com o comando `df`.

Normalmente, só queremos ser avisados quando a ocupação do disco ultrapassa um determinado valor (digamos que uma ocupação igual ou maior a 70%, o que é um bom exemplo (embora na vida real seja mais interessante monitorar a partir dos 90%)):

```
punk@rachael:~$ df | grep "[7-9][0-9]%"
/dev/hda6               2038472   1474756    458492  77% /usr
/dev/hda7               2038472   1569020    364228  82% /opt
```

Pronto! Mostramos as duas partições que estão com mais de 70% de ocupação. Preste bastante atenção no padrão utilizado, pensamos em um primeiro caracter cobrindo os algarismos de 7 a 9 e um segundo caracter cobrindo de 0 a 9, isso cobre todos os números de 70-99. O que deixou um "buraco" na nossa linha de comando... e se a partição estiver 100% ocupada? Neste caso temos que usar mais alguns recursos de expressões regulares:

```
punk@rachael:~$ df | grep "\(100\|[7-9][0-9]\)%"
/dev/hda6               2038472   1474756    458492  77% /usr
/dev/hda7               2038472   1569020    364228  82% /opt
```

Agora está OK, cobrimos os valores de 70 a 99 e o 100 com apenas uma expressão regular... mas o que quer dizer esta barbaridade?

- `()` - Os parêntesis separam grupos distintos
- `|` - serve como sendo um OU para selecionar entre uma expressão (ou grupo de caracteres) e outro.

A nossa expressão é: `(100|[7-9][0-9])%`, ou seja o número 100 ou algo entre 70 e 99 e, em seguinda o símbolo de porcentagem. O que é exatamente o que queremos 100% ou 70 a 99%. Os \ são um "pequeno" detalhe, pequeno e desagradável. Apesar das expressões regulares em si serem razoavelmente padronizadas, alguns comandos já utilizavam os símbolos para outra coisa ou (às vezes) o próprio *shell* tem algum outro uso para os comandos indicados. Daí vem o uso dos \, para fazer com que alguns caracteres sejam interpretados da maneira correta. O lado ruim, é que cada comando utiliza os \ para caracteres diferentes, assim, o que precisa de um \ no `grep`, pode não precisar no `sed` ou no `awk` e vice-versa. O jeito é decorar ou apelar para a tentativa e erro.

Um último retoque na nossa expressão: uma máquina pode ter sistemas de arquivos locais e remotos, e os sistemas de arquivos remotos geralmente são gerenciados pela máquina remota (e mesmo que não fossem, quem tem que cuidar disso é o administrador da outra máquina e não você), os sistemas de arquivo locais começam com aquele `/dev`, vamos garantir que outros sistemas de arquivo não atrapalhem a nossa pobre vida:

```
punk@rachael:~$ df | grep '^/dev.*\(100\|[7-9][0-9]\)%'
/dev/hda6               2038472   1474756    458492  77% /usr
/dev/hda7               2038472   1569020    364228  82% /opt
```

Incluímos o `^/dev.*` que engloba toda linha que começar com `/dev` e for seguida de zero ou mais caracteres (quaisquer caracteres, em qualquer quantidade). Ficou bom, não acham?

Outro passo importante de se selecionar um texto é conseguir capturar apenas o que for interessante no texto, o que às vezes não é toda a saída fornecida pelo `grep`. Por exemplo, naquele comando que demos para localizar em quais arquivos havia a *string* "192.168.0.2". A saída era a seguinte:

```
punk@rachael:/etc$ grep -r "192.168.0.2" * 2>/dev/null
hosts:192.168.0.2               rachael.mylab rachael
rc.d/rc.inet1.conf:IPADDR[0]="192.168.0.2"
resolv.conf:nameserver 192.168.0.2
```

Mas, a informação que queríamos era em quais arquivos existem essa string. Na saída do `grep`, os arquivos são o pedaço anterior aos `:`, e isso que queremos. Para ter acesso a esta informação, vamos utilizar o comando `cut`:

```
punk@rachael:/etc$ grep -r "192.168.0.2" * 2>/dev/null | \
cut -f1 -d:
hosts
rc.d/rc.inet1.conf
resolv.conf
```

A sintaxe do `cut` é extremamente simples:

```
cut -fcampo -ddelimitador
```

No nosso caso, queremos o primeiro campo (`-f1`) separados por : (`-d:`). Para continuar o nosso treinamento, algo um pouco mais complicado, vamos tentar selecionar, na saída do comando `ifconfig`, o IP da nossa máquina:

```
punk@rachael:~$ /sbin/ifconfig eth0
eth0    Link encap:Ethernet HWaddr 00:50:BF:67:88:97
        inet addr:192.168.0.2 Bcast:192.168.0.255 Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:141120 errors:0 dropped:0 overruns:0 frame:0
        TX packets:243521 errors:0 dropped:0 overruns:0 carrier:0
        collisions:4479 txqueuelen:100
        RX bytes:15819409 (15.0 Mb)  TX bytes:84671582 (80.7 Mb)
        Interrupt:10 Base address:0xa000
```

O primeiro passo, é selecionar a linha em que se encontra o IP, podemos capturá-la utilizando como padrão o `inet addr`:

```
punk@rachael:~$ /sbin/ifconfig eth0 | grep "inet addr"
          inet addr:192.168.0.2  Bcast:192.168.0.255  Mask:255.255.255.0
```

Depois disso, o nosso próximo passo é selecionar a parte do comando que contém o IP. Para isso, vamos separar o segundo campo delimitado por :.

```
punk@rachael:~$ /sbin/ifconfig eth0 | grep "inet addr" | \
cut -f2 -d:
192.168.0.2  Bcast
```

E, nosso último passo, é separar o primeiro campo, delimitado por um espaço em branco:

```
punk@rachael:~$ /sbin/ifconfig eth0 | grep "inet addr" | \
cut -f2 -d: | cut -f1 -d' '
192.168.0.2
```

Pronto! Já temos o nosso IP (e agora vocês já sabem como fazer para selecionar apenas o IP da sua máquina dentro de um script), mais um exemplo de exemplo útil!

3.2 Organizando

Já conseguimos selecionar exatamente o que queremos na saída de um comando (ou diretamente de um arquivo: no caso do `grep`, ele aceita o nome do arquivo na própria linha de comando e, mesmo que não aceitasse, poderíamos passar o conteúdo do arquivo utilizando o comando `cat`). Mas, podemos precisar de uma pequena ajuda para colocar as coisas em ordem.

O comando `ps`, serve para mostrar quais os processos estão sendo executados pelo sistema. Além do nome dos processos em si, ele apresenta várias outras informações:

```
punk@rachael:~$ ps -edf | grep bash
punk       924   923  0 17:22 pts/1    00:00:00 -bash
punk      1110  1109  0 17:42 pts/2    00:00:00 -bash
punk      1196  1195  0 18:11 pts/0    00:00:00 -bash
punk      1350  1349  0 18:28 pts/3    00:00:00 -bash
```

Na primeira coluna temos o usuário dono do processo, depois do PID dele (é um número que identifica o processo no sistema), depois o PPID (Parent PID, que é o PID do processo "pai" deste processo), o uso da CPU, a hora em que foi iniciado e outras informações, finalizando com o nome do processo.

Podemos utilizar `ps -edo user`, para mostrar apenas a coluna com o nome do usuário, de todos os processos executados na máquina... porém, a saída desse comando é enorme:

```
punk@rachael:~$ ps -edo user | wc -l
     76
```

O `wc -l` serve para contar a quantidade de linhas que tem um arquivo (e, no nosso caso, a saída do `ps`), a saída fica deste tamanho porque normalmente, estão sendo executados dezenas de processos em uma máquina. E pior, um único usuário pode ser responsável por algumas dezenas destes processos...

Mas, como foi dito, o comando `ps -edo user` mostra apenas a lista com o nome do usuário responsável por cada processo, ou seja, no final das contas queremos apenas uma lista com o nome dos usuários, sem repetições. Para nós não está interessando quantos processos um determinado usuário está executando, nos interessa se o usuário está ou não executando um processo e, se estiver, queremos o nome dele.

Podemos diminuir muito esse problema com o uso do comando `uniq`, ele elimina as linhas repetidas de uma saída, podemos então usar:

```
punk@rachael:~$ ps -edo user | uniq
USER
root
rpc
root
lp
root
daemon
root
punk
root
punk
root
marina
punk
```

Infelizmente, nossa idéia não deu certo -:(, aparecem ainda várias vezes o nome de alguns usuários. A causa deste nosso pequeno problema é a ordem com que os processos são listados, em ordem de "chegada", vejamos a diferença:

```
punk@rachael:~$ cat teste
lala
lala
lele
lala
punk@rachael:~$ uniq teste
lala
lele
lala
```

O conteúdo do arquivo `teste`, são as palavras `lala, lala, lele` e `lala`, cada uma em uma linha. Quando usamos o comando `uniq`, ele apenas elimina onde as linhas idênticas que estão em seqüência. E é isto que ocorre com o a saída do nosso `ps -edo user`. Mas, para tudo existe solução, e podemos utilizar o comando `sort` para colocar uma saída de maneira ordenada, vejamos:

```
punk@rachael:~$ sort teste
lala
lala
lala
lele
```

Agora o arquivo está ordenado, e pronto para eliminarmos as linhas repetidas com o `uniq`.

```
punk@rachael:~$ sort teste | uniq
lala
lele
```

E achamos a solução para o nosso problema!!! Vamos agora aplicá-la à nossa lista de usuários:

```
punk@rachael:~$ ps -edo user | sort | uniq
daemon
lp
marina
punk
root
rpc
USER
```

O último "retoque" será eliminar onde está escrito USER (que na realidade é apenas o cabeçalho do comando `ps`) com um `grep` reverso:

```
punk@rachael:~$ ps -edo user | sort | uniq | grep -v USER
daemon
lp
marina
punk
root
rpc
```

Este é um bom exemplo para mostrar o quanto os pipes são úteis, passamos a saída do comando `ps` por outros três comandos, cada um exercendo uma função diferente e auxiliando o comando anterior para nos dar o resultado desejado.

Outra grande ferramenta para a manipulação e formatação de textos é o `awk`. Aliás, chamar o `awk` apenas de ferramenta é menosprezar os seus poderes... na realidade, o `awk` é uma linguagem de programação completa, com diversas capacidades. Basicamente, o que a sintaxe do `awk` é:

```
awk '/padrao/ { ação }' arquivo
```

Se não for listada nenhuma ação, o `awk` irá apenas imprimir as linhas que casarem com o padrão. Vamos ao nosso primeiro exemplo:

```
punk@rachael:~$ free | awk '/Swap/'
Swap:       265032          72     264960
```

Neste primeiro caso, utilizamos o `awk` como se fosse o comando `grep`. Ele apenas imprimiu a linha que casou com o padrão dado. Da mesma maneira que com o `grep`, podemos utilizar expressões regulares como padrão a ser encontrado:

```
punk@rachael:~$ df | awk '/\/dev.*(100|[7-9][0-9])%/'
/dev/hda6 2038472 1474756 458492 77% /usr
/dev/hda7 2038472 1569020 364228 82% /opt
```

Com a vantagem que diminuímos muito o uso do \ (lembra quando eu falei que cada comando precisava do \ em locais diferentes?). Uma grande vantagem do `awk` em relação ao `grep` é a existência do campo ação, com ele podemos fazer o `awk` realizar coisas diferentes para cada tipo de padrão encontrado:

```
punk@rachael:~$ df | awk '
> /^\/dev.*(100|[7-9][0-9])%/ {
> print "A partição "$1" está com "$5" de ocupação"
> }'
A partição /dev/hda6 está com 77% de ocupação
A partição /dev/hda7 está com 82% de ocupação
```

Neste nosso novo comando, quando encontramos o padrão desejado, imprimimos uma mensagem especial, o `$1` e o `$5` se referem à campos separados por espaços ou tabulações. Juntamos o `grep` e o `cut` em um só comando -:) Podemos usar o `awk` como sendo o `cut`, veja:

```
punk@rachael:/etc$ grep -r "192.168.0.2" * 2>/dev/null | \
awk -F: '{ print $1 }'
hosts
rc.d/rc.inet1.conf
resolv.conf
```

O `-F:` define o delimitador a ser utilizado, como o `-d` do comando `cut`. Para falar a verdade, é possível cumprir as funções de muitos outros comandos com o `awk` e, inclusive, criar novos. Vamos criar o nosso *script* `mem`, em seu editor favorito escreva:

```
#!/bin/sh
free | awk '
      BEGIN  {
```

```
        print "Memória"
}
/Mem:/ {
        print "\tTotal: "$2"\tBuffers: "$6"\tCacheada: "$7
}
/-\/+/ {
        print "\tUsada: "$3"\tLivre: "$4
}'
```

O que faz este nosso programa? Ele pega a saída do comando `free` e a transforma em algo um pouco mais inteligível, torne-o executável e veja o que acontece quando é chamado:

```
punk@rachael:~$ mem
Memória
        Total: 256148   Buffers: 22596  Cacheada: 134012
        Usada: 91136    Livre: 165012
```

Com isso acaba a famosa confusão em que o indivíduo acha que o Linux está usando toda a memória dele, culpa dos campos `used` e `free` que aparecem na saída do comando `free`. No caso do nosso `awk` a única novidade é o `BEGIN`, na realidade ele é um tipo de padrão especial que é executado no início do programa. Outro padrão semelhante é o `END`, que sempre é executado após todas as linhas haverem sido processadas.

3.3 Editando

Além de achar uma informação e formatá-la da maneira que achar mais agradável, uma necessidade razoável é alterar o conteúdo de um arquivo. Vamos tomar como exemplo o `/etc/hosts` que lista alguns IPs e nomes (normalmente os únicos IPs ali são o do *localhost* e o da sua máquina, os outros costumam estar sob responsabilidade de um DNS):

```
punk@rachael:~$ cat /etc/hosts | grep -v "^#\|^$"
127.0.0.1               localhost
192.168.0.2             rachael.mylab rachael
```

O `grep` ali faz com que todas as linhas de comentário (`^#`) e todas as linhas vazias (`^$`) não sejam mostradas, sendo mostrado apenas o que nos interessa. Pode ser necessário alterar o IP da nossa máquina, e isso iremos fazer com auxílio do comando `sed`:

```
punk@rachael:~$ cat /etc/hosts | grep -v "^#\|^$" | \
> sed -e "s/192.168.0.2/192.168.0.100/"
127.0.0.1             localhost
192.168.0.100         rachael.mylab rachael
```

Vejam que onde estava `192.168.0.2` foi trocado para `192.168.0.100`. Isso é feito pelo argumento entre aspas do `sed`.

`s/192.168.0.2/192.168.0.100/` quer dizer: substitua onde está 192.168.0.2 por 192.168.0.100. Poderíamos fazer isso com o nome do computador também (onde está `rachael`):

```
punk@rachael:~$ cat /etc/hosts | grep -v "^#\|^$" | \
> sed -e "s/rachael/pris/"
127.0.0.1             localhost
192.168.0.2           pris.mylab rachael
```

Oh! oh! Algo deu errado! Ele trocou apenas o primeiro `rachael` da linha, deixou o segundo lá, intocado. Mas é assim que o `sed` funciona, o padrão é trocar apenas a primeira ocorrência dentro de uma linha e depois procurar outra linha. Para mudar isso, precisamos indicar o que deve alterar todas as ocorrências que encontrar, isso é feito colocando um `g` no final do padrão:

```
punk@rachael:~$ cat /etc/hosts | grep -v "^#\|^$" | \
> sed -e "s/rachael/pris/g"
127.0.0.1             localhost
192.168.0.2           pris.mylab pris
```

Pronto! Agora tudo está correto, mudamos o nome da nossa máquina para `pris` (para quem não sabe, `pris` e `rachael` são Replicantes do filme *Blade Runner*). Podemos agora mudar o nome e o IP ao mesmo tempo:

```
punk@rachael:~$ cat /etc/hosts | grep -v "^#\|^$" | \
sed -e "s/rachael/pris/g" -e "s/192.168.0.2/192.168.0.100/"
127.0.0.1             localhost
192.168.0.100         pris.mylab pris
```

Vamos aproveitar e aprender mais uma sobre o `sed`, faça:

```
punk@rachael:~$ cat /etc/hosts | grep -v "^#\|^$" | \
sed -e "s/rachael/pris/g" -e "s/192.168.0.2/192.168.0.100/" \
-e "/127.0.0.1/d"
192.168.0.100         pris.mylab pris
```

Vemos que a nossa última adição apagou (ou deletou) a linha que continha o 127.0.0.1. Com isso já temos como trocar palavras e apagar linhas. A última coisa que nos falta é escrever isso no nosso arquivo de destino:

```
punk@rachael:~$ cat /etc/hosts | grep -v "^#\|^$" | \
sed -e "s/rachael/pris/g" -e "s/192.168.0.2/192.168.0.100/" \
-e "/127.0.0.1/d" > ~/hosts.new
```

Pronto! Tudo foi direcionado para o nosso novo arquivo `hosts.new` dentro do $HOME do usuário. Colocamos dentro do `HOME` do usuário apenas porque não podemos escrever no `/etc/hosts` original, que só pode ser sobrescrito pelo usuário `root`.

Aqui vai um toque especial, **NUNCA** (eu disse **NUNCA**) use o mesmo arquivo como sendo arquivo de entrada e de saída de um comando. Veja o que pode acontecer:

```
punk@rachael:~$ cat teste
lala
lele
lolo
punk@rachael:~$ sed -e "s/lolo/lili/" teste > teste
punk@rachael:~$ cat teste
punk@rachael:~$
```

O arquivo `teste` foi apagado. Existe uma maneira de burlar este problema:

```
punk@rachael:~$ cat teste2
fafa
fefe
fofo
punk@rachael:~$ cat teste2 | sed -e "s/fofo/fifi/" > teste2
punk@rachael:~$ cat teste2
fafa
fefe
fifi
```

Com este nosso comando, primeiro o arquivo `teste2` é lido e, apenas depois do arquivo lido é passado para o comando `sed` e este escreve no próprio arquivo original. Mas, mesmo assim esta não é uma boa idéia, o melhor processo para se editar um arquivo é o seguinte:

```
punk@rachael:~$ mv teste2 teste2.bak
punk@rachael:~$ cat teste2.bak | sed -e "s/fafa/fofo/" > teste2
```

Com isso nós mantemos o arquivo original, e não corremos o risco de causar nenhum dano a ele. Afinal, sempre é bom manter algum *backup*, não é?

Capítulo 4

Aparência é Tudo...

Depois de toda a nossa lógica feita, temos que arranjar uma maneira de "falar" com o usuário, para que tudo não fique um puro e perfeito caos e o usuário faça o que ele mais gosta de fazer: burrices, às vezes danosas. Para isso existem várias idéias e maneiras diferentes de deixar os *scripts* visualmente mais agradáveis, sendo que elas não são compartimentos estanques, sempre podem ser mescladas umas com as outras em busca de uma funcionalidade melhor...

4.1 O método COBOL

É um método trabalhoso, mas tem algumas vantagens:

1. é extremamente legível
2. você sabe *exatamente* o que vai aparecer na tela
3. funcionou durante mil anos, porque deixaria de funcionar agora?

O método consiste em desenhar todas as telas do programa e "imprimi-las" na tela conforme desenhadas, no momento necessário. Isso é feito com o comando `cat`. Alguns usuários mais avançados desta técnica, colocam as telas em arquivos texto, e vão colocando estes arquivos na tela na hora correta, ou então colocam todas as telas em um grande arquivão e através de `grep`s bem escolhidos e outra funções arcanas vão mostrando-as a medida que o usuário escolhe as suas opções.

Uma grande desvantagem, além do trabalho chato, é que as telas ficam forçosamente com o tamanho com que as criamos, isso não era problema quanto todos usávamos terminais com 80x25 caracteres, mas hoje em dia, época de interfaces gráficas, `xterm`s, `rxvt`s e onde até mesmo os terminais texto cederam aos encantos do *framebuffer*, corremos o risco de ter uma telinha ridiculamente pequena no canto superior esquerdo da tela e um gigantesco espaço em branco (ou, na maioria das vezes, “em negro”)

Para ilustrar este método, fizemos um pequeno script com apenas um trecho da tela, podemos vê-lo na seqüência:

```
#!/bin/sh cat <<EOF
==============================================
|                                            |
|     Demonstração Mané do Método COBOL      |
|                                            |
|     Montamos a tela inteira e depois       |
|     imprimimos com um cat                  |
|                                            |
==============================================
EOF
```

E, o efeito do script depois de executado é este:

```
punk@rachael:~$ ./teste
==============================================
|                                            |
|     Demonstração Mané do Método COBOL      |
|                                            |
|     Montamos a tela inteira e depois       |
|     imprimimos com um cat                  |
|                                            |
==============================================
```

Idêntico ao “desenhado”. O que serve como uma prova da eficiência deste método.

4.2 O método criptográfico

Enquanto o método COBOL requer muita força bruta e é extremamente legível, este método requer menos força bruta, mas a sua leitura está além da capacidade dos mortais. O texto é todo impresso através de caracteres de controle:

```
punk@rachael:~$ echo -e "\n\t\tMenu de Opções\n\n\t\t\
1)Opção 1\n\n\t\t2)Opção 2\n\n\t\t3)Sair\n\n\tEscolha \
a opção desejada: \c" ; read -n1 OPCAO
```

A linha anterior é um bom exemplo desta técnica. O `-e` do `echo`, faz com que os códigos de controle sejam executados. O `\n` faz um salto para o início da próxima linha, e o `\t` é uma tabulação. Enquanto o `\c` avisa para que o `echo` não salte uma linha. Para se ter uma idéia do quê acontece, o resultado é este:

```
                    Menu de Opções

                    1)Opção 1

                    2)Opção 2

                    3)Sair

            Escolha a opção desejada:
```

Apesar de um programa inteiro utilizando estes caracteres ficar bem enfadonho (e eles terem a mesma limitação das telas desenhadas em nossa era de terminais com dimensões variáveis), este método é muito conveniente para algumas mensagens de progresso, ou para se escrever *logs* com uma formatação agradável. O *script* abaixo mostra um uso deste método (desta vez, para saber o resultado, vai ter que escrever e rodar o *script*):

```
#!/bin/sh
if [ "$#" = "0" ]; then
        echo -e "\nPor favor, utilize:\n\n\t# $0 <arquivo>\n"
else
        echo -e "\nProcurando por $1\t\t\c"
        ACHEI=`find ./ -name "$1" 2>/dev/null`
        if [ "$ACHEI" != "" ]; then
                echo -e "Localizado  $ACHEI"
        else
                echo -e "\tFALHOU"
        fi
        echo
fi
```

Através dos códigos podemos também escrever textos coloridos, já que existem códigos de controle especiais para colocarmos um pouco de cor em nossos scripts. Mensagens de erro em vermelho costumam ficar bem chamativas.

A sintaxe para isso não é tão complicada quanto parece à primeira vista:

```
punk@rachael:~/scripts$ echo -e "\033[37;44;1mBranco no \
fundo azul\033[0m"
```

O `\033[` é o código da tecla ESC, depois vêm outros atributos, os que vão do 30 ao 37 se referem à cor de frente, os do 40 ao 47 se referem à cor de fundo e o `1m` e `0m`, se referem à cor em modo bold (negrito) ou normal. No *Bash-Prompt-HOWTO* tem um programa mostrando os códigos e suas respectivas cores. E várias outras maneiras de se usar caracteres de controle.

4.3 O bom e velho "linha e coluna"

Uma abordagem bem tradicional, e bem flexível. Podemos simplesmente posicionar o cursor onde quisermos na tela, basta indicar qual a linha e a coluna em que ele deve estar. A partir dali, começamos a escrever.

Por exemplo:

```
punk@rachael:~$ clear; tput cup 10 40; echo "oi"
```

Irá escrever "oi" aproximadamente no centro da tela. E logo em seguida aparece o *prompt*, Mas podemos melhorar um pouco isso, temos como saber exatamente quantas linhas tem a tela:

```
punk@rachael:~$ tput lines
24
```

De modo análogo, temos como saber a quantidade de colunas com o `tput cols`. Voltando ao nosso maravilhoso texto "oi" e ao *prompt* que insiste a aparecer no meio da tela... Basta depois de escrevermos o "oi", mandarmos o cursor para o final da tela:

```
punk@rachael:~$ clear; tput cup 10 40; echo "oi"; tput cup 24 0
```

E pronto! Agora temos o nosso "oi" solitário no meio da tela e o prompt lá em baixo. Mas essas funções de saber a quantidade de linhas e colunas podem ter outra importância, através delas é possível dimensionar as nossas telas para o tamanho de tela do terminal de nossos usuários, e dar uma aparência bem mais interessante, abusando de centralizações e menus proporcionais...

```
centraliza() {
        TAM=`expr length "$1"`
        COLS=`tput cols`
        COLUNA=$(((COLS-TAM)/2))
        LINHA=$2
        tput cup $LINHA $COLUNA
        echo $1
}
```

A função acima recebe a string ($1) e a linha em que ela deve aparecer ($2). A parte da centralização é feita automaticamente.

4.4 O menu de bolso

Se o que você quer é um menu rápido, e não está se importando tanto assim para as aparências, o `bash` já tem uma função feita sob medida, eu chamo-a de “menu” de bolso, de tão simples que é utilizá-lo, basta saber quais serão as opções e o que cada uma delas faz:

```
#!/bin/sh
select opcao in "Opção 1" "Opção 2" "Sair"; do
        case $REPLY in
                1)
                echo "Você escolheu 1"
                ;;
                2)
                echo "Você escolheu 2"
                ;;
                3)
                echo "Saindo"
                exit
                ;;
        esac
done
```

O `select` monta um menu numerado com a lista que houver logo após o `in`. E já deixa um prompt aguardando a entrada dos dados. O dado sendo válido (ou inválido), o menu será reapresentado ao usuário (eliminando a necessidade de fazer um loop infinito). A resposta do usuário é armazenada na variável `REPLY`, e pode ser facilmente tratada em um `case`, como vimos acima. É rápido, prático e eficiente.

4.5 Usando Dialog

O `dialog` é um programa simples de usar que oferece a você vários recursos "gráficos" para incrementar os seus *shell scripts*, de uma maneira um pouco mais automatizada que as vistas anteriormente. Iremos agora "mostrar" um pouco do `dialog`, mas ele não se resume apenas ao conteúdo deste capítulo, uma boa olhada no manual dele pode trazer grandes (e agradáveis) surpresas.

A estrutura de um comando no `dialog` é bem simples, mas por trás desta simplicidade está um ser extremamente mal... podemos colocar tantas opções em um texto do `dialog` que facilmente um comando ultrapassa 5 ou 6 linhas (e daí para mais). Um exemplo dos mais simples está logo abaixo:

```
dialog --tipo-de-caixa "Texto da caixa" <tamanho>
```

Dependendo do tipo-de-caixa, são necessárias várias outras opções. Vamos começar com uma simples, vamos brincar com uma `--msgbox`, onde você pode colocar informações do seu programa antes de executá-lo...

```
dialog --msgbox "\
        Este é um programa muito bem feito \
        e especial, e depois de clicar em \
        OK você entrará no maravilhoso mundo \
        do meu programa wonder-ultra-plus" 0 0
```

Irá aparecer uma caixa centralizada contendo o texto que você digitou e um botão de OK, uma *splash screen* bem profissional, não acham? O "0 0" do fim diz que a janela será autodimensioanda, mas você pode forçá-la a ter o tamanho que quiser trocando estes dois números: o primeiro é a altura e o segundo a largura da caixa.

Podemos agora melhorar um pouco mais a aparência da nossa caixa de mensagens, colocando um título nela:

```
dialog --title "Apresentação" --msgbox "\
        Este é um programa muito bem feito \
        e especial, e depois de clicar em \
        OK você entrará no maravilhoso mundo \
        do meu programa wonder-ultra-plus" 0 0
```

Já ficou melhor não é? Mas para ficar com um visual ainda mais profissional vamos colocar um título no fundo, com o nome do nosso programa:

```
dialog --backtitle "MeuPrograma 0.1" --title "Apresentação" ...
```

Onde estão os `...` é para colocar o resto do texto dos exemplos anteriores. Ah! Estas opções `--backtitle` e `--title` funcionam em todas as outras caixas do `dialog`. E iremos ver várias delas...

Continuando em ordem de complexidade, temos a caixa `--yesno` onde você coloca uma mensagem e o usuário responde esta mensagem com um "sim" ou um "não". Experimente:

```
dialog --yesno "Você deseja apagar tudo?" 0 0
```

Neste caso surge um novo problema... como saber o que o seu usuário respondeu? No caso do `--yesno`, ele devolve "0" para sim (sucesso) e "1" para não (fracasso), como a maior parte dos comandos do `bash`. Um programinha com a pergunta e a capacidade de receber a resposta seria semelhante a este:

```
dialog --yesno "Você deseja apagar tudo?" 0 0
if [ $? = 0 ]; then
   dialog --infobox "Apagando tudo..." 0 0
else
   dialog --infobox "A vida continua..." 0 0
fi
```

Com isso aprendemos a pegar os resultados do `--yesno` e de quebra aprendemos a usar uma `--infobox` -;)

Depois de duas opções, que tal tentar com mais? A caixa `--menu` serve para isso, ela vai apresentar um menu (!!!). É um pouco mais complicada que os outras que vimos até agora, tendo muito mais parâmetros:

```
dialog --menu "Selecione uma opção do menu" 0 0 3 \
         1 "Opção 1" \
         2 "Opção 2" \
         3 "Opção 3"
```

A primeira alteração visível, é que a parte do tamanho possui 3 argumentos ao invés de dois, os dois primeiros são os já conhecidos altura e largura, e o último é o tamanho da "lista" de opções.

Se você colocar o tamanho da janela da lista menor que a quantidade de opções existentes, você poderá "rolar" pelas opções (irá aparecer um indicador "v(+)" ou "^(+)" indicando que existem mais opções abaixo ou acima).

Outra alteração bem visível são as próprias opções! Você pode colocar um "atalho" e depois o texto da opção (E foi assim que fizemos, o atalho é o número e o texto está entre aspas). Para pegar a saída do "OK" e do "Cancel", use o mesmo processo que utilizamos para pegar a saída do `--yesno`, "0" para OK e "1" para "Cancel". O mais complicado é pegar a opção selecionada, para fazer isso precisamos:

- usar um arquivo temporário com a opção e depois ler do arquivo o que o usuário escolheu ou;
- Passar a saída diretamente para uma variável.

Para usar o primeiro dos métodos, é bem simples... apenas coloque no final do comando `dialog` um `2>/tmp/arquivotemporario`, como podemos ver abaixo:

```
dialog --menu "Selecione uma opção do menu" 0 0 3 \
         1 "Opção 1" \
         2 "Opção 2" \
         3 "Opção 3" 2>/tmp/arquivotemporario
```

Este método é fácil, mas é um pouco desagradável depender de um arquivo externo, é preferível utilizar a segunda opção, onde conseguimos passar a saída do `dialog` diretamente para uma variável:

```
OPCAO=‘dialog --stdout --menu "\
         Selecione uma opção do menu" 0 0 3 \
         1 "Opção 1" \
         2 "Opção 2" \
         3 "Opção 3"‘
```

Logo depois de executar o dialog, a saída dele estará na variável `OPCAO`, bem mais prático. A opção `--stdout` serve para direcionar a saída do `dialog` (que tradicionalmente vai para o *stderr*, para a saída padrão) e com isso, tornando possível capturá-la em variáveis.

Vamos passar agora para a `--radiolist`, podemos dizer que esta é um "modelo" diferente de menu, serão apresentadas várias opções para você selecionar e, como no `--menu`, apenas uma delas pode estar selecionada:

```
dialog --radiolist "Selecione uma opção do menu" 0 0 3 \
       1 "Opção 1" ON \
       2 "Opção 2" OFF \
       3 "Opção 3" OFF
```

Lembre que apenas uma das opções pode estar ativa, repare que também já deixamos uma opção selecionada "por default”. Esta é uma vantagem, já que fica bem claro qual opção nos recomendamos ao usuário. Para saber qual foi a escolha do usuário, temos os dois métodos já apresentados para a `--menu`.

Na nossa ordem de complexidade, vamos agora poder pegar várias opções em uma só tela, para isso temos a `--checklist`. A `--checklist` é uma parente da `--radiolist`, sendo ambas muito semelhantes, a grande diferença é que podemos selecionar mais de uma opção, enquanto na `--radiolist`, não custa repetir, podemos utilizar apenas uma:

```
dialog --checklist "O que pretende apagar?" 0 40 3 \
       usuarios "Todos os usuários" ON \
       programas "Todos os programas" ON \
       tudo "Tudo mesmo..." OFF
```

A saída desta caixa é o nome das várias opções selecionadas, cada uma delas entre aspas. Isso atrapalha um pouco para trabalhar com essas opções. Uma boa idéia é utilizar o `--separate-output` isso fará com que a saída seja dada uma opção por linha.

Por último, uma caixa sem opção, em que o usuário pode fazer o que quiser, a `--inputbox`. Este é o tipo de caixa que você vai escolher quando for pedir para o seu usuário escrever algo. Para coletar o que o seu usuário digitou, você já deve saber o que fazer.

```
dialog --inputbox "Escreva aqui alguma bobagem" 0 0
```

Usar o `dialog` é uma maneira fácil de fazer com que seus scripts tenham uma outra aparência, e impressionar o “usuário final”. Eles sempre ficam impressionados com cores, sombras e menus... até engolem sistemas que não funcionam... os nossos, além de funcionar, também são simpáticos -:)

Capítulo 5

Trabalhando conectado

A maior parte dos usuários de *shell scripts* são administradores de sistemas e administradores de rede. E, como administradores de rede, muitas vezes é necessário utilizar a rede para as tarefas diárias. Vamos dar uma olhada neste capítulo em alguns comandos e maneiras de se trabalhar utilizando a rede.

Nós vimos rapidamente no capítulo 1 um uso da rede, com o comando `ping`. Através dele, podemos ver se um site está vivo ou morto (ou se a nossa conexão de rede está funcionando). Mas, depois de verificar se a máquina está viva, seria interessante que soubéssemos disso... por exemplo, com um e-mail avisando. Este capítulo é um pouco mais complexo que os anteriores, afinal, além dos conceitos de *shell script* usamos muitos conceitos de redes.

5.1 Mandando e-mails

A maneira mais simples (e óbvia) de se mandar um e-mail é através do comando `mail`. Com ele podemos mandar uma frase, ou todo um arquivo texto com razoável facilidade:

```
punk@frankenstein:~$ echo "A máquina X faleceu" | \
mail -s "ALERTA" punk@rachael
```

Isso irá mandar uma mensagem com o conteúdo "A máquina X faleceu", com o assunto "ALERTA" para o endereço punk@rachael. O maior problema para esta abordagem é a necessidade de haver um servidor de e-mail (MTA) funcionando na máquina em que você envia o e-mail.

Podemos contornar isso de uma maneira "levemente" mais complicada, nos comunicando diretamente com um servidor de e-mail. Normalmente temos acesso a algum, nem que seja dos nossos e-mails pessoais.

Assim que conectarmos, iremos utilizar os comandos próprios do SMTP (que é o protocolo utilizado para enviar e-mails). Podemos verificar como isso funciona utilizando o comando `telnet`:

```
punk@frankenstein:~$ telnet rachael 25
Trying 192.168.0.2...
Connected to rachael.
Escape character is '^]'.
220 rachael.mylab ESMTP Sendmail 8.12.9/8.12.9; Wed, 3 Mar 2004 \
01:09:56 -0300
MAIL FROM: punk@frankenstein
250 2.1.0 punk@frankenstein... Sender ok
RCPT TO: punk@rachael
250 2.1.5 punk@rachael... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
La le li
.
250 2.0.0 i2349uYi001881 Message accepted for delivery
QUIT
221 2.0.0 rachael.mylab closing connection
```

Primeiro conectamos no servidor desejado (`rachael`) e na porta utilizada pelo SMTP (25). As respostas do servidor, no nosso caso, são as linhas com números no inicio, estes números são códigos alienígenas. Depois de conectado, enviamos os comandos necessários:

- `MAIL FROM: punk@frankenstein` - Avisa que o e-mail é para ser enviado como sendo de punk@frankenstein. Logo após o comando, o servidor nos responde que está OK. Algumas vezes, antes deste passo, é necessário utilizar o comando `HELO`, mas nem sempre isso é preciso.

- `RCPT TO: punk@rachael` - Agora passamos quem será o destinatário da mensagem (ou seja, quem vai receber) e determinamos que será punk@rachael, novamente o servidor nos dá o seu OK. Muitos MTAs necessitam que os e-mails sejam escritos entre < e >.

- `DATA` - A partir daqui, iremos escrever o conteúdo do nosso e-mail, ele termina com um "." sozinho em uma linha. Depois de tudo escrito e do "." digitado, o servidor nos avisa que a mensagem foi aceita

- `QUIT` - Finalmente, desconectamos do servidor.

Agora que já sabemos os comandos que devemos usar, está na hora de enviarmos o nosso e-mail, mas ele será enviado através de um simples `cat`... veja como:

```
punk@frankenstein:~# cat <<EOF >/dev/tcp/rachael/25
> MAIL FROM: punk@frankenstein
> RCPT TO: punk@rachael
> DATA
> La la la
> .
> QUIT
> EOF
```

E pronto! O e-mail foi enviado -:) A parte do `<<EOF` você já conhece mas, provavelmente, o `/dev/tcp/rachael/25` é novidade. não é? Esse redirecionamento apenas está enviando a saída do comando `cat` para a máquina `rachael`, na porta `25`, usando `tcp`. É um recurso muito simples porém pouco utilizado.

Com este conhecimento, podemos fazer o nosso próprio comando `mail`. Aproveitando essa nossa boa fase de genialidade, vamos chamar o nosso `mail` de `nossomail` (original, não?)

```
#!/bin/sh
#
# nossomail - script superinteligente para enviar e-mails
#
# Verifica a quantidade de argumentos, se for apenas
# um, pega a mensagem a ser enviada diretamente do stdin
#
# Caso contrário, usa o segundo argumento como mensagem
#
if [ "$#" = "1" ]; then
        MSG=`cat -`
else
        MSG="$2"
fi
FROM="$USER@$HOSTNAME"
TO="$1"
HOST=`echo $1 | cut -f2 -d@`
cat <<EOF >/dev/tcp/$HOST/25
MAIL FROM: $FROM
RCPT TO: $TO
DATA
```

```
$MSG
.
QUIT
EOF
```

Não existem maiores segredos no *script*, já que todos os conceitos já foram utilizados anteriormente (espero que vocês ainda lembrem dos outros capítulos). Podemos utilizar este nosso *script* de duas maneiras, uma delas é enviando a mensagem na própria linha de comando:

```
punk@frankenstein:~$ ./nossomail punk@rachael "Teste"
```

A outra maneira é enviando a mensagem através de um `pipe`:

```
punk@frankenstein:~$ echo TESTE | ./nossomail punk@rachael
```

Pronto! Mais um exemplo que também é um *script* útil!

5.2 Trocando arquivos

Além de mandar e-mails, um outro uso típico para uma rede é a transferência de arquivos. Existe uma série de comandos interessantes para utilizarmos em nossos *scripts* e manipularmos arquivos em rede.

O primeiro da lista é o `wget`. Ele pode ser utilizado para baixar *sites* inteiros, tanto via *http* (Web) como por *ftp* (ftp é ftp mesmo) e é muito útil. Podemos imaginar um uso bem simples, todo dia, a filial da sua empresa em Laodicéia do Oeste disponibiliza um arquivo com o *backup* daquele dia. *Backups* costumam ser muito grandes para se enviar por e-mail (e, só para constar, um arquivo enviado por e-mail aumenta de tamanho). Ah, esses *backups* devem ser gravados na sede da empresa...

A melhor coisa a fazer é aproveitar o horário da madrugada, quando não costuma haver ninguém trabalhando e muito menos usando a banda e trazer o arquivo. Vamos usar o `wget`.

```
wget http://laodiceia.xingling.com.br/backups/backup_de_hoje
```

Pronto! Este é o comando. Basta digitá-lo e o *backup* será trazido até você, como que por mágica. Isso irá baixar o arquivo `backup_de_hoje` que está no diretório `backups` dentro da máquina `laodiceia.firma-xingling.com.br` através de `http`. Este nosso primeiro comando tem um pequeno problema: onde ele vai guardar o arquivo que baixar?

Se você digitá-lo no *prompt*, ele vai baixar o arquivo no diretório em que estiver. Mas, costuma-se colocar *backups* em lugares específicos. Muito mais agradável seria usar:

```
wget -O /backup \
     http://laodiceia.xingling.com.br/backups/backup_de_hoje
```

Pronto! Agora ficou bom. Se o arquivo fosse disponibilizado via `ftp`, poderíamos usar o `wget` também, apenas trocando onde está `http://` por `ftp://`.

E se fosse o caso inverso? E se nós tivéssemos que enviar um arquivo para a filial de Otonópolis do Sul? Talvez um relatório... bom, se for por `ftp` podemos usar o ótimo `ncftp`. Ele agrega vários comandos utilíssimos para transferências de arquivos via *scripts*. No nosso caso de enviar arquivos, o comando apropriado é o `ncftpput`, cuja sintaxe é:

```
ncftpput -u usuario -p senha maquina-remota caminho arquivo-local
```

Se quiséssemos enviar o arquivo `Exemplo1` para o diretório `aulas_shell/rede` da máquina `maquina_remota`. Ou seja, para o `ftp://maquina_remota/aulas_shell/rede/` iríamos fazer assim:

```
ncftpput -u usuario -p senha maquina_remota aulas_shell/rede Exemplo1
```

Onde indica o `usuario` e a `senha`, você coloca o usuário e senhas apropriados. Podemos também utilizar o `ncftp` para baixar arquivos, através do `ncftpget`. Ele possui uma sintaxe mais simples que a do `ncftpput`.

```
ncftpget -u usuario -p senha ftp://exemplo.servidor/dir/arquivo
```

Vale a pena dar uma boa explorada nesses comandos, consulte a *man-page* deles e o *help*. Muitos problemas "insolúveis" podem ser resolvidos por eles, experiência própria.

5.2.1 HTML...

Existem momentos em que não queremos baixar o arquivo HTML em si, e sim a página já renderizada. Dependendo do que queremos fazer, pode ser muuuito mais simples encontrar uma determinada informação na página já formatada.

Para esses casos, uma boa opção seria utilizar o `lynx`. Como exemplo, irei utilizar esta página:

```
<HTML>
   <HEAD>
       <TITLE>Exemplo</TITLE>
   </HEAD>
   <BODY>
       <OL>
            <LI>Um item <LI>outro item
            <LI>mais um
            <LI>e mais outro
       </OL>
   </BODY>
</HTML>
```

Vamos supor que o nosso objetivo fosse pegar o segundo item (o "`outro item`") como faríamos esse *script*? Impossível não é, mas com certeza seria bem mais simples com a página já devidamente formatada. Veja como ela fica depois de renderizada pelo `lynx`:

```
punk@rachael:~$ lynx -dump exemplo.html
    1. Um item
    2. outro item
    3. mais um
    4. e mais outro
```

Agora qualquer um que conheça um pouco de *shell script* já sabe como pegar o segundo item -;)

```
punk@rachael:~$ lynx -dump exemplo.html | grep 2 | cut -f2 -d.
 outro item
```

Aproveitamos para retirar o `2.` da frente. Para redirecionar a página já formatada para um determinado arquivo, basta utilizar o `>`, como já visto anteriormente.

Outro uso muito importante do `lynx` é para acessar páginas. Com um pouco de conhecimento, é possível enviar formulários já preenchidos através da linha de comando, bastando colocar o conteúdo dos campos diretamente na URL... mas isso varia de formulário para formulário... e já estamos no final da apostila. Espero que tenha o conteúdo dela tenha sido útil para dar uma pequena introdução aos *shell scripts*. Agora é colocar a mão na massa...

Apêndice A

Usando "O" editor de textos: VI

A.1 Descrevendo o VI...

O VI é o editor de textos padrão do mundo *NIX. Qualquer sistema operacional que seja compatível com o UNIX, ou que seja nele inspirado possui a sua versão do VI. Isto tem um motivo, além de ser um dos editores de texto mais tradicionais do mercado, é também um dos mais poderosos.

Normalmente, as distribuições Linux não vem com o VI original, mas sim com clones como o VIm, elVIs e nVI. Ou, às vezes, com todos eles -;). Todos são idênticos nas funcionalidades básicas, mudando um pouco nas funções mais avançadas.

O VI possui dois modos de trabalho, o modo de edição (onde você escreve o texto propriamente dito) e o modo de comando, onde podemos passar comandos para ele. Normalmente, quando abrimos o VI (ou o seu clone preferido), ele está no modo de comando. Nos nossos exemplos, iremos usar o `elvis`, mas boa parte do que se aplica, se aplica a outros clones...

A.2 Editando textos...

Para sair do modo de comando e entrar no modo de edição, aperte `i`. Isso, a letra `i`. Com isso você irá entrar no modo de inserção (tem outras maneiras de entrar no modo de edição, mas essa é a mais comum).

A partir de agora, você pode escrever o seu texto à vontade. Para escrever, basta ir escrevendo e, para navegar no texto, o `elvis` permite que você use as setas e os comandos *PgUp*, *PgDn*, *Home* e *End*. Você pode apagar o texto que escreveu utilizando o *Backspace*.

Para realizar algumas outras coisas, é necessário voltar ao modo de comando. Para isso, pressione a tecla *ESC*. No modo de comando, existem vários atalhos de teclas para facilitar a edição de textos.

A.2.1 Navegando pelo texto

Apesar das setinhas funcionarem no `elvis`, elas não funcionam no `vi` padrão, então é bom conhecer como fazer os movimentos do jeito "certo". Todos estes comandos de movimentação devem ser dados no modo de comando.

Os mais básicos são:

- `j`,`k`,`h`,`l` -> respectivamente: para baixo, para cima, para a esquerda e para a direita. Você pode substituir o `l` pela barra de espaços
- `enter` -> Vai para o início da próxima linha (nããão...)
- `0` -> Começo da linha (atenção, `0` é um zero e não um "o" maiúsculo)
- `$` -> Final da linha
- `w,b` -> Caminham pelas palavras, `w` para frente e `b` para trás.
- `:/,?` -> Procura por um padrão (o uso é `:/padrão ou ?padrão`), o `:/` vai para frente e o `?` para trás. Para os preguiçosos, o `n` repete a última busca.

Existem outros, mas estes são suficientes para navegarmos no texto e ainda fazermos alguns truques -;).

A.2.2 Marcando textos

Você pode marcar áreas retangulares do texto com o *Ctrl+V*. Apenas pressione essas teclas e selecione o texto desejado com as setas. Após selecionar o texto, pressione `y` para copiar. Mova o cursor até a posição em que deseja copiar o texto e aperte `p`. Se, ao invés de copiar o texto selecionado você preferir apagá-lo, troque o `y` pelo `d`.

Se não gostar de demarcar retângulos, pode marcar linhas inteiras utilizando o atalho *Shift+V* e depois movendo o cursor pelas linhas desejadas. Se quiser copiar apenas a linha em que está, utilize o `yy`.

Um recurso muito útil para quem programa, é utilizar o $>$ ou o $<$ sobre o texto selecionado. Isso faz com que ele aumente ou diminua uma indentação, tornando o código muito mais legível. Acredite, isso é uma mão na roda enquanto editamos um *shell script*.

A.2.3 Apagando

Bom, você viu que apagar textos marcados é apertando `d`. Para apagar outras coisas também é assim... E ainda é possível aproveitar outros recursos... por exemplo, se você apertar a letra `d`, e depois o número `5` e por fim a seta para a direita, irá apagar 5 caracteres para a direita. Se fizer isso e apertar a seta para a esquerda, irá apagar 5 caracteres para a esquerda e, se apertar a seta para cima, ou para baixo, irá apagar 5 linhas nessas direções.

Normalmente, o que se mais utiliza é o `d`(seta para o lado) para apagar apenas um caracter. Outro muito utilizado é o `D` que apaga a partir da posição do cursor até o final da linha e o `dd` que apaga a linha inteira.

Um texto apagado, pode ser colado utilizando o comando `p`.

Se você estiver usando o `d` para apagar o final de uma linha para colar uma linha na outra, está usando o comando errado... para fazer isso, use o `J` de `Join` (ou seja, juntar). Leia e lembre disso, irá lhe poupar dores de cabeça futuras...

A.3 Saindo do VI e outros adicionais...

Como sempre, sobram alguns comandos que a gente não sabe onde deve colocar... eu resolvi que devo colocar aqui -;)

A.3.1 Sair do VI

Para sair do VI, salvando o arquivo que você escreveu, use `:wq`, se não quiser salvar, use `:q!` é, com a `!` no final mesmo. Saber sair do VI é muito importante, normalmente é a primeira coisa que acontece com os marinheiros de primeira viagem: ficam presos. Agora você não irá cair nessa. (OK, mas pode cair em outras armadilhas...)

A.3.2 Lendo um arquivo no VI

Esta é outra boa... enquanto você estiver editando um arquivo, pode precisar do conteúdo de algum outro... para isso, use o `:r nome_do_outro_arquivo`, o tal `nome_do_outro_arquivo` será carregado logo abaixo do cursor.

Você também pode fazer um truque! O `:!` é utilizado para executar um comando, com `:r!date` por exemplo, você irá inserir o resultado do comando `date` no seu arquivo! (Dica boa!!!)

A.3.3 Substituições

Existem várias outras maneiras, mas escolhi uma opção especial para nós que somos "quase" *experts* no `sed` (afinal, vimos alguns parágrafos sobre ele no Capítulo 3). Ela tem mais ou menos a mesma sintaxe do `sed`.

```
:s/padrão1/padrão2/
```

Isso irá substituir onde está `padrão1` pelo `padrão2`. Mas apenas na primeira ocorrência na linha em que estiver o cursor... usando: `:s/padrão1/padrão2/g`, você substitui o texto na linha inteira e, usando um `%` antes do `s`, você realiza substituições no texto inteiro!!!

Como foi possível ver utilizando o `%`, é possível limitar a ação do comando de substituição (bom, no caso específico do `%`, nós limitamos a substituição ao texto inteiro). Mas podemos fazer diferente:

```
:1,3s/padrão1/padrão2/g
```

Isso irá substituir `padrão1` por `padrão2` nas linhas de `1` a `3`. E, atenção para a dica: `.` é a linha atual, e `$` é o final do arquivo. Dá para fazer alguns intervalos interessantes utilizando esses dois. Ah! E se você quiser repetir a sua última substituição, use `:&`. E, para desfazer a sua última ação, use o comando `u`.

A.4 Configurações...

Algumas configurações interessantes. Quando são utilizadas com freqüência, são colocadas em um arquivo do tipo `.vimrc`, `.exrc`, etc... Para colocá-las enquanto está editando um arquivo, você deve primeiro passar para o modo de comando e fazer:

```
:set autoindent
```

Isso irá ativar a indentação automática (útil para quem está programando). Para desativar a indentação, use:

```
:set noautoindent
```

Se quiser colocá-los em um arquivo, basta ir colocando os comandos um abaixo do outro, sem os dois pontos iniciais. Alguns arquivos `.vimrc` acabam ficando quilométricos... para quem usa o `vim`, dentro da documentação do programa costuma ficar o `vimrc_example.vim` que é um ótimo e bem completo exemplo...

Algumas das opções que podemos usar são

- `autoindent` -> Seta a indentação automática
- `number` -> Numera todas as linhas. Os números não são salvos junto com o arquivo.
- `showmatch` -> Quando você escreve um (, [ou {, ele detecta o correspondente },] ou). Outro útil para programadores...
- `list` -> Mostra as tabulações como `^I` e os finais de linha como `$`. Ótimo para descobrir o que é uma tabulação e o que é espaço. Também para encontrar espaços perdidos no final das linhas (e que dependendo do lugar causam problemas exóticos).

Existem dúzias de outras opções destas... utilize `:set all` para ver todas, e decidir se quer utilizá-las. Muitas vezes, a configuração *default* é mais que suficiente.

Apêndice B

Comandos úteis...

Depois de ler todo o conteúdo deste livro, podemos esquecer algumas coisinhas, ou precisar rapidamente localizar um comando. Aqui estão listados alguns dos comandos mais utilizados em *shell scripts*, separados por função.

Lembrando que o conteúdo presente aqui é apenas um pequeno resumo e, sequer, chega aos pés da documentação destes comandos, que pode ser visualizada e consultada através do comando `man`. Acostume-se com ele, ele é seu amigo. Para usar, basta fazer:

```
# man nome_do_comando
```

Com isso você terá acesso a uma quantidade incrível de informações. Comandos mais complexos como o *sed* e o *awk* não aparecem neste apêndice. Apesar de muito utilizados, é necessário praticamente um livro para explicar cada um deles.

B.1 ...para manipular arquivos...

B.1.1 ls - Listar arquivos

O comando `ls` lista os arquivos contidos em um diretório (ou no diretório em que estamos). Por exemplo:

```
ls /tmp
```

Irá listar todos os arquivos do `/tmp`. Existem uma série de opções que podem ser dadas em conjunto com o comando `ls`. Podemos citar algumas:

- `-l` -> lista os arquivos e seus atributos
- `-a` -> lista os arquivos ocultos

- `-R` -> lista recursivamente o diretório desejado e seus subdiretórios
- `-h` -> coloca os tamanhos de arquivos em formato "humano", 4k ao invés de 4096, 1M ao invés de 1048576 e assim por diante.

Estas opções podem ser combinadas de diferentes formas. Ex: `ls -laRh`

B.1.2 rm - Remover arquivos

Remove um arquivo (ou lista de arquivos). É bom lembrar que nem sempre (tradução: quase nunca) é possível recuperar um arquivo apagado, por isso, faça backups e tome cuidado. Opções úteis:

- `-i` -> Pergunta antes de remover o arquivo.
- `-r` -> Recursivo, apagando o interior dos diretórios e os próprios.
- `-f` -> Força a remoção do arquivo (CUIDADO)

E, para não ficarmos órfãos, um exemplo de uso:

```
rm -rf arquivo1 arquivo2 diretorio3
```

B.1.3 mv - Movendo e Renomeando arquivos

Move um arquivo de um local para outro na árvore de diretórios. É possível usar este comando como um "truque" para renomear arquivos, movendo-os para o nome novo:

```
mv nome_velho nome_novo
```

Ou, na sua outra forma:

```
mv /caminho/antigo/do/arquivo /lugar/novo/do/arquivo
```

As duas formas aceitam as opções `-f` e `-i`, sendo que a primeira faz com que o comando `mv` sobrescreva o arquivo novo, caso já exista um arquivo com este nome, e a segunda pergunta antes de sobrescrevê-lo.

B.1.4 cp - Copia arquivos

O comando `cp`, copia arquivos e diretórios. Podendo (inclusive) trocar o nome destes arquivos e diretórios no processo da cópia, ou manter o mesmo nome, desde que os arquivos sejam copiados para uma localização diferente da original. O modo de uso é simples:

```
cp arquivo_original copia_do_arquivo
```

Ou

```
cp arquivo_original /outro/lugar/para/o/arquivo_original
```

Claro que é possível copiar o arquivo para outro diretório com um nome diferente, basta especificar qual seria esse nome. Além disso, o comando `cp` possui várias opções:

- `-f` -> Sobrescreve os arquivos de destino, caso possuam os mesmos nomes
- `-i` -> Pergunta antes de sobrescrever os arquivos.
- `-p` -> Copia preservando todos os atributos, grupos, permissões, etc...
- `-R` -> Copia recursivamente, copiando diretórios, subdiretórios e seus conteúdos.
- `-a` -> Tenta manter a maior quantidade de informação possível dos arquivos copiados.

B.1.5 file - descobre o tipo do arquivo

Não precisa de maiores explicações. Basta usar:

```
file nome_do_arquivo
```

E o comando retorna o tipo de arquivo com que estamos trabalhando.

B.1.6 mkdir - Cria diretório

Para criar um diretório, usamos o comando `mkdir`, com a seguinte sintaxe:

```
mkdir nome_do_diretorio
```

é possível especificar também um caminho inteiro:

```
mkdir /caminho/para/o/nome_do_diretorio
```

E, se este caminho não existir, é possível criá-lo utilizando a opção `-p`:

```
mkdir -p /caminho/para/o/diretorio/nome_do_diretorio
```

Para remover o diretório criado, podemos usar o `rmdir` ou, o já conhecido, `rm`.

B.2 ...para trabalhar com textos...

B.2.1 cat - O texto na tela

O principal uso do `cat` é mostrar o conteúdo de um arquivo na tela. Claro, com os redirecionamentos que aprendemos, podemos mandar este conteúdo para qualquer lugar que quisermos.

Uma opção interessante do `cat` é a `-n`, que numera as linhas que são apresentadas pelo comando `cat`, facilitando selecionar apenas uma determinada linha. Existe um comando "irmão" do `cat`, o `tac` que mostra o arquivo em modo "reverso", com as últimas linhas sendo mostradas primeiro (e, obviamente, as primeiras sendo mostradas por último)

B.2.2 head - Mostra o início do arquivo

O comando `head` imprime as primeiras 10 linhas de um arquivo. Se utilizarmos a opção `-N` (onde `N` é um número) serão impressas as primeiras `N` linhas do arquivo. Como no exemplo:

```
head -5 arquivo
```

B.2.3 tail - Mostra o final de um arquivo

Praticamente idêntico ao `head`, mas trabalhando com o final do arquivo ao invés de com o começo. Uma boa opção é a `-f` para quando estiver observando arquivos de log. Neste caso o `tail` não pára, ele continua mostrando as últimas linhas do arquivo e as linhas que forem sendo adicionadas nele. O uso do `tail` é:

```
tail -7 arquivo
```

B.2.4 grep - Encontrando o que se procura

O comando `grep` é utilizado principalmente para se localizar um texto específico (ou uma expressão regular) dentro de um (ou mais) arquivos. O comando é extremamente poderoso e possui uma série de opções.

A sintaxe "default" é:

```
grep [-opções] ‘‘texto_ou_expressão_regular’’ arquivo
```

Ao invés de `arquivo`, podemos passar a saída de outros comandos para o `grep`, via `pipe`. Uma explicação mais detalhada do uso do comando e de expressões regulares pode ser encontrada no Capítulo 3.

B.2.5 cut - Corta um pedaço do texto

A sintaxe mais comum do `cut` é:

`cut -f`*n* `-d`*delim* `arquivo`

Onde *n* é o número do campo e *delim* é o delimitador utilizado. Por exemplo, o comando:

`echo “um dois três” | cut -f2 -d’ ’`

Irá retornar `dois` que é o segundo campo delimitado por espaços. Também é possível mostrar vários campos de uma só vez:

- `-f2-4` -> Irá mostrar os campos dois, três e quatro
- `-f3-` -> Irá mostrar todos os campos, a partir do terceiro
- `-f-4` -> Mostra todos os campos, do início até o quarto campo.

Lembrando que, como vimos no exemplo, ao invés de `arquivo` podemos utilizar a saída de outro comando, através do `pipe`.

B.2.6 sort - Ordena um texto

O comando `sort` coloca um texto em ordem. É possível ordenar de várias maneiras um arquivo, em ordem alfabética, em ordem numérica, em ordem reversa, etc... a sintaxe é:

`sort [-opções] arquivo`

E, as tais `opções` podem ser:

- `-n` -> coloca em ordem numérica
- `-r` -> ordem reversa
- `-d` -> ordem do “dicionário”

Entre várias outras que podem ser vistas na `manpage` do `sort`. O companheiro inseparável do `sort` é o `uniq`, que elimina linhas repetidas em um arquivo.

B.2.7 tr - troca caracteres

O comando `tr` troca caracteres isolados ou em grupos. Com ele é possível trocar todas as maiúsculas por minúsculas, apagar determinado caractere ou caracteres repetidos. A sintaxe padrão é:

```
tr [-opções] grupo1 [grupo2]
```

Por exemplo, para trocar maiúsculas por minúsculas:

```
tr A-Z a-z
```

Lembre que a entrada do `tr` é via `pipe`. Opções úteis incluem:

- `-d` -> apagar os caracteres encontrados no `grupo1`
- `-s` -> apaga caracteres repetidos

B.2.8 fold - "enquadra" textos

O comando `fold` é usado para "enquadrar" textos em um determinado tamanho. Por exemplo, podemos passar um texto inteiro para 40 colunas com o comando:

```
fold -w 40 arquivo
```

Bastante útil no trabalho com textos, afinal, nunca se sabe com qual formato deles chegam...

B.3 ...sobre o sistema e...

B.3.1 ps - mostra os processos no sistema

O comando `ps` mostra os processos em execução no sistema (juntamente com várias informações). Para ajudar a confundir, ele suporta duas sintaxes diferentes. Ou seja, você pode escrever seus comandos de duas maneiras diversas e ter o mesmo resultado (ou algo muito parecido).

O `ps` possui várias e várias opções. Não iremos mostrar (nem de longe) todas as elas. As opções no estilo SysV são utilizadas com um "-" na frente das opções. As opções BSD são sem o "-". Por exemplo:

```
# ps ef
```

Irá mostrar os processos do usuário, em forma de árvore e com todas as variáveis de ambiente logo após o nome do processo que está sendo executado. Já o comando:

```
# ps -ef
```

Irá mostrar todos os processos em execução na máquina com vários dados detalhados. Este tipo de comportamento torna particularmente difícil passar uma lista de opções. Algumas interessantes são:

- `ax` -> Mostra todos os processos em execução.
- `x` -> Todos os processos do usuário.
- `u` -> Mostra o usuário responsável pelo processo (entre outras coisas).
- `-e` -> Todos os processos em execução (o mesmo do `ax`).
- `-f` -> Expõe os processos em formato de árvore.

A `manpage` mostra vários outros. Inclusive, o suficiente para sequer confundir realmente bastante. Use as opções obscuras com cuidado. Se você gosta do `ps -f`, divirta-se com o `pstree`.

B.3.2 free - informações sobre a memória do sistema

Mostra a quantidade de memória livre no sistema. Este comando mostramos com detalhes no meio do texto, mas mostramos também a maior parte dos outros que aparecem neste apêndice...

```
punk@rachael:~$ free
             total    used     free    shared    buffers     cached
Mem:        256148  130316   125832         0       8636      76512
-/+ buffers/cache:   45168   210980
Swap:       265032       0   265032
```

Na primeira linha, temos o total de memória, a quantidade utilizada (incluindo os *buffers* e o *cache*). Na segunda linha temos algo mais próximo da realidade, mostrando a memória utilizada sem trabalhar com *cache* e o *buffer*. Por fim, na última linha, temos a utilização da memória *swap*.

Existem algumas opções para alterar a saída do programa:

- `-o` -> Não mostra a linha "`-/+ buffers/cache:`"
- `-t` -> Mostra o total de memória, somando a *swap* com a memória RAM.
- `-m` -> A quantidade de memória é mostrada em Megas, ao invés de Kilobytes.

B.3.3 uname - identificação do sistema

Mostra praticamente tudo sobre a sua máquina. O nome da máquina, o sistema operacional, a arquitetura, compilação do kernel, versão do kernel, etc... por exemplo:

```
punk@rachael:~$ uname -n -r -m -o
rachael 2.4.22 i686 GNU/Linux
```

Com isto estamos mostrando o nome da máquina (`-n`), a versão do kernel (`-r`), o tipo de *hardware* (`-m`) e o sistema operacional (`-o`). Caso queira ver todas as informações disponíveis, é possível utilizar a opção `-a`.

B.3.4 date - a data do sistema

O comando `date` apresenta a data e hora do sistema. É bem útil para fazer arquivos com o nome do dia (por exemplo, arquivos de backup). Escrever simplesmente `date`, retorna algo do tipo:

```
Fri Mar 26 00:16:49 BRT 2004
```

Mas é possível configurar a saída do comando `date` para coisas mais civilizadas (ou, pelo menos, melhores para colocar em nome de arquivo). Faça assim:

```
punk@rachael:~$ date +%Y_%m_%d
2004_03_26
```

Estes `%algumacoisa`, tem um significado especial, como podemos ver abaixo:

- `%Y` -> Ano, com quatro dígitos, se quiser com dois, use o `%y`
- `%m` -> Mês, de forma numérica
- `%b` -> Nome do mês, abreviado. Se quiser o nome inteiro, use `%B`.
- `%H` -> hora (com dias de 24 horas corridas... e não 12 horas AM e 12 horas PM)
- `%M` -> minutos

Dica de amigo, guarde a data no formato que lhe agradar em uma variável... você pode precisar disso várias vezes durante um *script*.

B.3.5 mount - Vendo os sistemas de arquivos montados

O comando `mount` mostra todos os dispositivos montados (ou sistemas de arquivos remotos), seus pontos de montagem, o tipo de sistema de arquivos e as opções de montagem.

Você também pode usar o comando `mount` para montar um sistema de arquivos. A sintaxe é a seguinte:

```
mount [-t tipo_de_fs] [-o opções] dispositivo ponto_de_montagem
```

Recomendação: ler a `manpage` antes de começar a trabalhar com as `opções` do comando. Várias delas só funcionam em determinados sistemas de arquivos. Se for utilizado apenas o `mount`, você verá as informações descritas no primeiro parágrafo deste item.

B.4 ...para outras coisas

Alguns comandos que não se enquadram nos descritos acima, mas são muito úteis mesmo assim:

B.4.1 ping - Verifica se uma determinada máquina está viva

Ótimo para examinar se uma máquina está respondendo, se está acessível ou para ter alguma idéia de como está a rede. Use o comando `ping` assim:

```
ping nome_da_maquina
```

Ele começará a fazer infinitos "pings" para a máquina desejada. O `ping` infinito não é dos melhores para usarmos em *scripts*. Geralmente utilizamos algumas opções:

- `-c n` -> Limita o número de "pings". Serão realizados apenas `n` "pings".
- `-w n` -> Espera no máximo `n` segundos para receber a resposta do ping enviado.

Com estas duas é mais fácil detectar se uma determinada máquina está viva ou morta (o `-w` ajuda bastante a evitar uma espera praticamente "eterna").

B.4.2 mcookie - Gera um código (teoricamente) único

Este comando é muito útil para se nomear arquivos temporários. Apenas faça, no começo do seu programa:

```
TEMP=”seu-programa-‘mcookie‘.tmp”
```

Com isso, você tem a certeza de que seu arquivo temporário não vai apagar o de outro usuário (ou vice-versa). Arquivos como `/tmp/temp.tmp` pedem para ser apagados ou para algo horrível acontecer com eles.

B.4.3 zcat - mostra um arquivo compactado com o gzip

Você pode guardar um arquivo compactado e mostrá-lo na tela diretamente, sem a necessidade de descompactá-lo antes do uso. A sintaxe é simples:

```
zcat nome_do_arquivo
```

Para abrir arquivos do `bzip2`, você pode utilizar o `bzcat`.

B.4.4 zgrep - procura por um texto no interior de um arquivo compactado com o gzip

Bom, da mesma maneira que o `zcat`, o `zgrep` foi feito para trabalhar diretamente com arquivos compactados. Ele (internamente) utiliza o comando `grep`, tendo a mesma sintaxe.

E, de novo, temos o clone do `zgrep` para quando for usar arquivos `.bz2`, o `bzgrep`.

B.4.5 tee - escreve algo na saída padrão e em um arquivo ao mesmo tempo

Útil para arquivos de log. Com o comando `tee`, a mesma saída pode ser mandada tanto para a saída padrão como para um arquivo, assim, você precisa mostrar as tuas mensagens apenas uma vez. A opção `-a` garante que o arquivo não seja apagado pelo próprio `tee` toda vez que for invocado, com ela, as mensagens são colocadas sempre no final do arquivo.

Um exemplo:

```
echo “Tudo fracassou! Erro mortal” | tee -a /var/log/script.log
```

Com isso, a frase passada pelo comando `echo` será apresentada na tela e no arquivo `/var/log/script.log`.

Referências Bibliográficas

[1] JARGAS, A. M. 2001, *Expressões Regulares - Guia de Consulta Rápida.* São Paulo, Novatec Editora

[2] RAIMUNDO, R. M. 2000, *Curso Básico de Programação em POSIX-Shell Script.* Rio de Janeiro, Book Express

[3] ROBBINS, A. D. 1997, *Effective AWK Programming 2ed.* Seattle, Specialized Systems Consultants, Inc

[4] SAADE, J. 2001, *Bash - Guia de Consulta Rápida.* São Paulo, Novatec Editora