PROGRAMAÇÃO SHELL

ORGANIZAÇÃO: RUBENS QUEIROZ DE ALMEIDA
http://www.dicas-l.com.br/download/ProgramacaoShell.pdf

Sumário

Sumário

Sumário

Sumário

1. Introdução

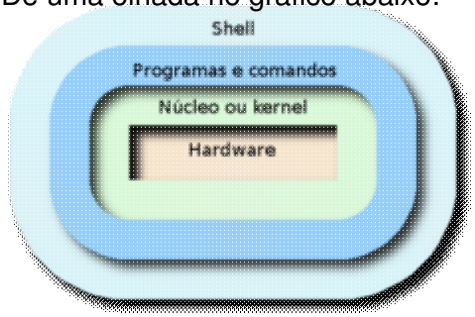
Nesta semana serão apresentados conceitos gerais de programação shell. O que é uma shell, como funciona, exemplos básicos e outras informações. Resumidamente, iremos aprender os conceitos que nos fornecerão os fundamentos para prosseguir com as atividades das demais semanas.

Este material foi baseado, em grande parte, na série de artigos escritos por Júlio Neves para a revista Linux Magazine. Esta série de artigos pode também ser encontrada no site do autor.

Embora existam diversos sabores de shell, neste curso nós iremos tratar da shell Bash (*Bourne Again Shell*).

2. O Ambiente Linux

Para você entender o que é e como funciona o Shell, primeiro vou te mostrar como funciona o ambiente em camadas do Linux. Dê uma olhada no gráfico abaixo:



Neste gráfico dá para ver que a camada de hardware é a mais profunda e é formada pelos componentes físicos do seu computador. Envolvendo esta, vem a camada do kernel que é o cerne do Linux, seu núcleo, e é quem bota o hardware para funcionar, fazendo seu gerenciamento e controle. Os programas e comandos que envolvem o kernel, dele se utilizam para realizar as tarefas aplicativas para que foram desenvolvidos. Fechando tudo isso vem o Shell que leva este nome porque em inglês, Shell significa concha, carapaça, isto é, fica entre o usuário e o sistema operacional, de forma que tudo que interage com o sistema operacional, tem que passar pelo seu crivo.

3. O Ambiente Shell

Bom já que para chegar ao núcleo do Linux, no seu kernel que é o que interessa a todo aplicativo, é necessária a filtragem do Shell, vamos entender como ele funciona de forma a tirar o máximo proveito das inúmeras facilidades que ele nos oferece.

O Linux por definição é um sistema multiusuário - não podemos nunca esquecer disto ? e para permitir o acesso de determinados usuários e barrar a entrada de outros, existe um arquivo chamado /etc/passwd que além fornecer dados para esta função de "leão-de-chácara" do Linux, também provê informações para o login daqueles que passaram por esta primeira barreira. O último campo de seus registros informa ao sistema qual Shell a pessoa vai receber ao se "logar" (ARGH!!!).

Quando eu disse que o último campo do /etc/passwd informa ao sistema qual é o Shell que o usuário vai receber ao se "logar", é para ser interpretado ao pé-da-letra, isto é, se neste campo do seu registro estiver prog, a pessoa ao acessar o sistema receberá a tela de execução do programa prog e ao terminar a sua execução ganhará imediatamente um logout. Imagine o quanto se pode incrementar a segurança com este simples artifício.

Lembra que eu te falei de Shell, família, irmão? Pois é, vamos começar a entender isto: o Shell, que se vale da imagem de uma concha envolvendo o sistema operacional propriamente dito, é o nome genérico para tratar os filhos desta idéia que, ao longo dos anos de existência do sistema operacional Unix foram aparecendo. Atualmente existem diversos sabores de Shell, dentre estes eu destaco o sh (Bourne Shell), o ksh (Korn Shell), bash (Bourne Again Shell) e o csh (C Shell).

3.1. Principais Sabores de Shell

3.1.1. Bourne Shell (sh)

Desenvolvido por Stephen Bourne da Bell Labs (da AT&T onde também foi desenvolvido o Unix), este foi durante muitos anos o Shell default do sistema operacional Unix. É também chamado de Standard Shell por ter sido durante vários anos o único e até hoje é o mais utilizado até porque ele foi portado para todos os ambientes Unix e distros Linux.

3.1.2. Korn Shell (ksh)

Desenvolvido por David Korn, também da Bell Labs, é um superset do sh, isto é, possui todas as facilidades do sh e a elas agregou muitas outras. A compatibilidade total com o sh vem trazendo muitos usuários e programadores de Shell para este ambiente.

3.1.3. Boune Again Shell (bash)

Este é o Shell mais moderno e cujo número de adeptos mais cresce em todo o mundo, seja por ser o Shell default do Linux, seu sistema operacional hospedeiro, seja por sua grande diversidade de comandos, que incorpora inclusive diversos instruções características do C Shell.

3.1.4. C Shell (csh)

Desenvolvido por Bill Joy da Berkley University é o Shell mais utilizado em ambientes *BSD e Xenix. A estruturação de seus comandos é bem similar à da linguagem C. Seu grande pecado foi ignorar a compatibilidade com o sh, partindo por um caminho próprio.

Além destes Shells existem outros, mas irei falar contigo somente sobre os três primeiros, tratando-os genericamente por Shell e assinalando as especificidades de cada um que porventura hajam.

4. O funcionamento do Shell

O Shell é o primeiro programa que você ganha ao se "logar" no Linux. É ele que vai resolver um monte de coisas de forma a não onerar o kernel com tarefas repetitivas, aliviando-o para tratar assuntos mais nobres. Como cada usuário possui o seu próprio Shell interpondo-se entre ele e o Linux, é o Shell quem interpreta os comandos que são teclados e examina as suas sintaxes, passando-os esmiuçados para execução.

— Êpa! Esse negócio de interpreta comando não tem nada a haver com interpretador não, né?

— Tem sim, na verdade o Shell é um interpretador (ou será intérprete) que traz consigo uma poderosa linguagem com comandos de alto nível, que permite construção de loops (laços), de tomadas de decisão e de armazenamento de valores em variáveis, como vou te mostrar.

Vou te explicar as principais tarefas que o Shell cumpre, na sua ordem de execução. Preste atenção nesta ordem porque ela é fundamental para o entendimento do resto do nosso bate papo.

4.1. Exame da Linha de Comandos

Neste exame o Shell identifica os caracteres especiais (reservados) que têm significado para interpretação da linha, logo após verifica se a linha passada é uma atribuição ou um comando.

4.1.1. Atribuição

Se o Shell encontra dois campos separados por um sinal de igual (=) **sem espaços em branco entre eles**, identifica esta seqüência como uma atribuição.

Exemplos

```
$ ls linux
linux
```

Neste exemplo o Shell identificou o ls como um programa e o linux como um parâmetro passado para o programa ls.

```
$ valor=1000
```

Neste caso, por não haver espaços em branco (já dá para notar que o branco é um dos caracteres reservados) o Shell identificou uma atribuição e colocou **1000** na variável `valor`.

Jamais Faça:

```
$ valor = 1000
bash: valor: not found
```

Neste caso, o Bash achou a palavra `valor` isolada por brancos e julgou que você estivesse mandando executar um programa chamado `valor`, para o qual estaria passando dois parâmetros: `=` e `1000`.

4.1.2. Comando

Quando uma linha é digitada no prompt do Linux, ela é dividida em pedaços separados por espaço em branco: o primeiro pedaço é o nome do programa que terá sua existência pesquisada; identifica em seguida, nesta ordem, opções/parâmetros, redirecionamentos e variáveis.

Quando o programa identificado existe, o Shell verifica as permissões dos arquivos envolvidos (inclusive o próprio programa), dando um erro caso você não esteja credenciado a executar esta tarefa.

Resolução de Redirecionamentos

Após identificar os componentes da linha que você teclou, o Shell parte para a resolução de redirecionamentos. O Shell tem incorporado ao seu elenco de vantagens o que chamamos de redirecionamento, que pode ser de entrada (stdin), de saída (stdout) ou dos erros (stderr), conforme vou te explicar a seguir.

Substituição de Variáveis

Neste ponto, o Shell verifica se as eventuais variáveis (parâmetros começados por $), encontradas no escopo do comando, estão definidas e as substitui por seus valores atuais.

Substituição de Meta Caracteres

Se algum metacaractere (*, ? ou []) foi encontrado na linha de comando, neste ponto ele será substituído por seus possíveis valores. Supondo que o único arquivo no seu diretório corrente começado pela letra `n` seja um diretório chamado `nomegrandeprachuchu`, se você fizer:

```
$ cd n*
```

Como até aqui quem esta trabalhando a sua linha é o Shell e o comando (programa) `cd` ainda não foi executado, o Shell transforma o `n*` em `nomegrandeprachuchu` e o comando `cd` será executado com sucesso.

Passa Linha de Comando para o kernel

Completadas as tarefas anteriores, o Shell monta a linha de comandos, já com todas as substituições feitas, chama o kernel para executá-la em um novo Shell (Shell filho), ganhando um número de processo (PID ou Process IDentification) e permanece inativo, tirando uma soneca, durante a execução do programa. Uma vez encerrado este processo (juntamente com o Shell filho), recebe novamente o controle e, exibindo um prompt, mostra que está pronto para executar outros comandos.

5. Decifrando a Pedra da Roseta

Para tirar aquela sensação que você tem quando vê um script Shell, que mais parece uma sopa de letrinhas ou um hieróglifo vou lhe mostrar os principais caracteres especiais para que você saia por ai como o Jean-François Champollion decifrando a Pedra da Roseta (dê uma "googlada" para descobrir quem é este cara, acho que vale a pena).

5.1. Caracteres para remoção do significado

É isso mesmo, quando não desejamos que o Shell interprete um caractere especial, devemos "escondê-lo" dele. Isso pode ser feito de três formas distintas:

5.1.1. Apóstrofo ou plic (')

Quando o Shell vê uma cadeia de caracteres entre apóstrofos ('), ele tira os apóstrofos da cadeia e não interpreta seu conteúdo.

```
$ ls linux*
linuxmagazine
$ ls 'linux*'
bash: linux* no such file or directory
```

No primeiro caso o Shell "expandiu" o asterisco e descobriu o arquivo `linuxmagazine` para listar. No segundo, os apóstrofos inibiram a interpretação do Shell e veio a resposta que não existe o arquivo `linux*`.

5.1.2. Contrabarra ou Barra Invertida (\)

Idêntico aos apóstrofos exceto que a barra invertida inibe a interpretação somente do caractere que a segue.

Suponha que você acidentalmente tenha criado um arquivo chamado `*` (asterisco) - que alguns sabores de Unix permitem - e deseja removê-lo. Se você fizesse:

```
$ rm *
```

Você estaria fazendo a maior encrenca, pois o `rm` removeria todos os arquivos do diretório corrente. A melhor forma de fazer o pretendido é:

```
$ rm \*
```

Desta forma, o Shell não interpretaria o asterisco, e em conseqüência não faria a sua expansão.

Faça a seguinte experiência científica:

```
$ cd /etc
$ echo '*'
$ echo \*
$ echo *
```

Viu a diferença? Então não precisa explicar mais.

5.1.3. Aspas (")

Exatamente igual ao apóstrofo exceto que, se a cadeia entre aspas contiver um cifrão ($), uma crase (`), ou uma barra invertida (\), estes caracteres serão interpretados pelo Shell.

Não precisa se estressar, eu não te dei exemplos do uso das aspas por que você ainda não conhece o cifrão ($) nem a crase (`). Daqui para frente veremos com muita constância o uso destes caracteres especiais, o mais importante é entender o significado de cada um.

6. Caracteres de redirecionamento

A maioria dos comandos tem uma entrada, uma saída e pode gerar erros. Esta entrada é chamada Entrada Padrão ou *stdin* e seu default é o teclado do terminal. Analogamente, a saída do comando é chamada Saída Padrão ou *stdout* e seu default é a tela do terminal. Para a tela também são enviadas por default as mensagens de erro oriundas do comando que neste caso é a chamada Saída de Erro Padrão ou *stderr*. Veremos agora como alterar este estado de coisas.

Vamos fazer um programa gago. Para isto faça:

```
$ cat
```

O `cat` é uma instrução que lista o conteúdo do arquivo especificado para a Saída Padrão (stdout). Caso a entrada não seja definida, ele espera os dados da stdin. Ora como eu não especifiquei a entrada, ele está esperando-a pelo teclado (Entrada Padrão) e como também não citei a saída, o que eu teclar irá para a tela (Saída Padrão) fazendo desta forma, como eu havia proposto um programa gago. Experimente!

6.1. Redirecionamento da Saída Padrão

Para especificarmos a saída de um programa usamos o > (maior que) ou o >> (maior, maior) seguido do nome do arquivo para o qual se deseja mandar a saída.

Vamos transformar o programa gago em um editor de textos (que pretensão heim!). :-)

```
$ cat > Arq
```

O `cat` continua sem ter a entrada especificada, portanto está aguardando que os dados sejam teclados, porém a sua saída está sendo desviada para o arquivo `Arq`. Assim sendo, tudo que esta sendo teclado esta indo para dentro de `Arq`, de forma que fizemos o editor de textos mais curto e ruim do planeta.

Se eu fizer novamente:

```
$ cat > Arq
```

Os dados contidos em `Arq` serão perdidos, já que antes do redirecionamento o Shell criará um `Arq` vazio. Para colocar mais informações no final do arquivo eu deveria ter feito:

```
$ cat >> Arq
```

ATENÇÃO Como já havia lhe dito, o Shell resolve a linha e depois manda o comando para a execução. Assim, se você redirecionar a saída de um arquivo para ele próprio, primeiramente o Shell "esvazia" este arquivo e depois manda o comando para execução, desta forma você acabou de perder o conteúdo do seu querido arquivo.

Com isso dá para notar que o >> (maior maior) serve para inserir texto no final do arquivo.

6.2. Redirecionamento da Saída de Erro Padrão

Assim como o default do Shell é receber os dados do teclado e mandar as saídas para a tela, os erros também serão enviados para a tela se você não especificar para onde deverão ser enviados. Para redirecionar os erros use `2> SaidaDeErro`. Note que entre o número `2` e o sinal de maior (>) não existe espaço em branco.

Preste atenção! Não confunda `>>` com `2>`. O primeiro anexa dados ao final de um arquivo, e o segundo redireciona a Saída de Erro Padrão (stderr) para um arquivo que está sendo designado. Isso é importante!

Suponha que durante a execução de um script você pode, ou não (dependendo do rumo tomado pela execução do programa), ter criado um arquivo chamado `/tmp/seraqueexiste$$`. Para não ficar sujeira no seu disco, ao final do script você colocaria uma linha:

```
$ rm /tmp/seraqueexiste$$
```

Caso o arquivo não existisse seria enviado para a tela uma mensagem de erro. Para que isso não aconteça deve-se fazer:

```
$ rm /tmp/seraqueexiste$$ 2> /dev/null
```

Sobre o exemplo que acabamos de ver tenho duas dicas a dar:

Dica # 1

O `$$` contém o PID, isto é, o número do seu processo. Como o Linux é multiusuário, é bom anexar sempre o `$$` ao nome dos dos arquivos que serão usados por várias pessoas para não haver problema de propriedade, isto é, caso você batizasse o seu arquivo simplesmente como `seraqueexiste`, o primeiro que o usasse (criando-o então) seria o seu dono e todos os outros ganhariam um erro quando tentassem gravar algo nele.

Para que você teste a Saída de Erro Padrão direto no prompt do seu Shell, vou dar mais um exemplo. Faça:

```
$ ls naoexiste
bash: naoexiste no such file or directory
$ ls naoexiste 2> arquivodeerros
$
$ cat arquivodeerros
bash: naoexiste no such file or directory
```

Neste exemplo, vimos que quando fizemos um `ls` em `naoexiste`, ganhamos uma mensagem de erro. Após, redirecionarmos a Saída de Erro Padrão para `arquivodeerros` e executarmos o mesmo comando, recebemos somente o prompt na tela. Quando listamos o conteúdo do arquivo para o qual foi redirecionada a Saída de Erro Padrão, vimos que a mensagem de erro tinha sido armazenada nele.

Faça este teste ai.

Dica # 2

— Quem é esse tal de `/dev/null`?

— Em Unix existe um arquivo fantasma. Chama-se `/dev/null`. Tudo que é mandado para este arquivo some. Assemelha-se a um Buraco Negro. No caso do exemplo, como não me interessava guardar a possível mensagem de erro oriunda do comando rm, redirecionei-a para este arquivo.

É interessante notar que estes caracteres de redirecionamento são cumulativos, isto é, se no exemplo anterior fizéssemos:

```
$ ls naoexiste 2>> arquivodeerros
```

a mensagem de erro oriunda do `ls` seria anexada ao final de `arquivodeerros`.

7. Redirecionamento da Entrada Padrão

Para fazermos o redirecionamento da Entrada Padrão usamos o < (menor que).

— E prá que serve isso? - você vai me perguntar.

— Deixa eu te dar um exemplo que você vai entender rapidinho.

Suponha que você queira mandar um mail para o seu chefe. Para o chefe nós caprichamos, né? então ao invés de sair redigindo o mail direto no prompt da tela de forma a tornar impossível a correção de uma frase anterior onde, sem querer, escreveu um "nós vai", você edita um arquivo com o conteúdo da mensagem e após umas quinze verificações sem constatar nenhum erro, decide enviá-lo e para tal faz:

```
$ mail chefe < arquivocommailparaochefe
```

O teu chefe então receberá o conteúdo do `arquivocommailparaochefe`.

8. Here Document

Um outro tipo de redirecionamento muito louco que o Shell te permite é o chamado *here document*. Ele é representado por << (menor menor) e serve para indicar ao Shell que o escopo de um comando começa na linha seguinte e termina quando encontra uma linha cujo conteúdo seja unicamente o label que segue o sinal `<<`.

Veja o fragmento de script a seguir, com uma rotina de ftp:

```
ftp -ivn hostremoto << fimftp
    user $Usuário $Senha
    binary
    get arquivoremoto
fimftp
```

Neste pedacinho de programa temos um monte de detalhes interessantes:

1. As opções que usei para o ftp (-ivn) servem para ele ir listando tudo que está acontecendo (?v de verbose), para não perguntar se você tem certeza de que deseja transmitir cada arquivo (?i de interactive), e finalmente a opção ?n serve para dizer ao ftp para ele não solicitar o usuário e sua senha, pois esses serão informados pela instrução específica (user);
2. Quando eu usei o << fimftp, estava dizendo o seguinte para o intérprete: — Olhe aqui Shell, não se meta em nada a partir daqui até encontrar o label fimftp. Você não entenderia nada, já que são instruções específicas do comando ftp e você não entende nada de ftp. Se fosse só isso seria simples, mas pelo próprio exemplo dá para ver que existem duas variáveis ($Usuário e $Senha), que o Shell vai resolver antes do redirecionamento. Mas a grande vantagem desse tipo de construção é que ela permite que comandos também sejam interpretados dentro do escopo do here document, o que também contraria o que acabei de dizer. Logo a seguir explico como esse negócio funciona. Agora ainda não dá, está faltando ferramenta.
3. O comando `user` é do repertório de instruções do ftp e serve para passar o usuário e a senha que haviam sido lidos em uma rotina anterior a esse fragmento de código e colocados respectivamente nas duas variáveis: `$Usuário` e `$Senha`.
4. O `binary` é outra instrução do ftp, que serve para indicar que a transferência de `arquivoremoto` será feita em modo binário, isto é, o conteúdo do arquivo não será interpretado para saber se está em ASCII, EBCDIC, ...
5. O `get arquivoremoto` diz ao ftp para pegar esse arquivo em `hostremoto` e trazê-lo para o nosso host local. Se fosse para mandar o arquivo, usaríamos o comando `put`.

Um erro muito freqüente no uso de labels (como o `fimftp` do exemplo anterior) é causado pela presença de espaços em branco antes ou após o mesmo. Fique muito atento quanto a isso, por que este tipo de erro costuma dar uma boa surra no programador, até que seja detectado. Lembre-se: um label que se preze tem que ter uma linha inteira só para ele.

— Está bem, está bem! Eu sei que dei uma viajada e entrei pelos comandos do ftp, fugindo ao nosso assunto que é Shell, mas como é sempre bom aprender e é raro as pessoas estarem disponíveis para ensinar...

9. Redirecionamento de Comandos

Os redirecionamentos que falamos até aqui sempre se referiam a arquivos, isto é mandavam para arquivo, recebiam de arquivo, simulavam arquivo local, ... O que veremos a partir de agora redireciona a saída de um comando para a entrada de outro. É utilíssimo e quebra os maiores galhos. Seu nome é pipe (que em inglês significa tubo, já que ele encana a saída de um comando para a entrada de outro) e sua representação é uma barra vertical (|).

```
$ ls | wc -l
  21
```

O comando `ls` passou a lista de arquivos para o comando `wc`, que quando está com a opção `-l` conta a quantidade de linha que recebeu. Desta forma, podemos afirmar categoricamente que no meu diretório existiam 21 arquivos.

```
$ cat /etc/passwd |sort | lp
```

Esta linha de comandos manda a listagem do arquivo `/etc/passwd` para a entrada do comando `sort`. Este a classifica e manda-a para o `lp` que é o gerenciador do spool de impressão.

10. Caracteres de Ambiente

Quando quer priorizar uma expressão você coloca-a entre parênteses não é? Pois é, por causa da aritmética é normal pensarmos deste jeito. Mas em Shell o que prioriza mesmo são as crases (`) e não os parênteses. Vou dar exemplos de uso das crases para você entender melhor.

Eu quero saber quantos usuários estão "logados" no computador que eu administro. Eu posso fazer:

```
$ who | wc -l
  8
```

O comando `who` passa a lista de usuários conectados para o comando `wc -l` que conta quantas linhas recebeu e lista a resposta na tela. Pois bem, mas ao invés de ter um oito solto na tela, o que eu quero é que ele esteja no meio de uma frase.

Ora para mandar frases para a tela eu uso o comando `echo`, então vamos ver como é que fica:

```
$ echo "Existem who | wc -l usuários conectados"
Existem who | wc -l usuários conectados
```

Hi! Olha só, não funcionou! É mesmo, não funcionou e não foi por causa das aspas que eu coloquei, mas sim por que eu teria que ter executado o `who | wc -l` antes do `echo`. Para resolver este problema, tenho que priorizar esta segunda parte do comando com o uso de crases, fazendo assim:

```
$ echo "Existem `who | wc -l` usuários conectados"
Existem 8 usuários conectados
```

Para eliminar esse monte de brancos antes do 8 que o `wc -l` produziu, basta tirar as aspas. Assim:

```
$ echo Existem `who | wc -l` usuários conectados
Existem 8 usuários conectados
```

Como eu disse antes, as aspas protegem tudo que esta dentro dos seus limites, da interpretação do Shell. Como para o Shell basta um espaço em branco como separador, o monte de espaços será trocado por um único após a retirada das aspas.

Antes de falar sobre o uso dos parênteses deixa eu mandar uma rapidinha sobre o uso de ponto-e-vírgula (;). Quando estiver no Shell, você deve sempre dar um comando em cada linha. Para agrupar comandos em uma mesma linha teremos que separá-los por ponto-e-vírgula. Então:

```
$ pwd ; cd /etc; pwd; cd -; pwd /home/meudir /etc/ /home/meudir
```

Neste exemplo, listei o nome do diretório corrente com o comando `pwd`, mudei para o diretório `/etc`, novamente listei o nome do diretório e finalmente voltei para o diretório onde estava anteriormente (`cd -`), listando seu nome. Repare que coloquei o ponto-e-vírgula (;) de todas as formas possíveis para mostrar que não importa se existe espaços em branco antes ou após este caractere.

Finalmente vamos ver o caso dos parênteses. Veja só o caso a seguir, bem parecido com o exemplo anterior:

```
$ (pwd ; cd /etc ; pwd;) /home/meudir /etc/ $ pwd /home/meudir
```

— Quequeiiisso minha gente? Eu estava no `/home/meudir`, mudei para o `/etc`, constatei que estava neste diretório com o `pwd` seguinte e quando o agrupamento de comandos terminou, eu vi que continuava no `/home/meudir`, como se eu nunca houvesse saído de lá!

— Ih! Será que é tem coisa de mágico aí?

— Tá me estranhando, rapaz? Não é nada disso! O interessante do uso de parênteses é que ele invoca um novo Shell para executar os comandos que estão no seu interior. Desta forma, realmente fomos para o diretório /etc, porém quando todos os comandos dentro dos parênteses foram executados, o novo Shell que estava no diretório /etc morreu e voltamos ao Shell anterior cujo diretório corrente era /home/meudir. Faça outros testes usando cd, e ls para você firmar o conceito.

Agora que já conhecemos estes conceitos veja só este exemplo a seguir:

```
$ mail suporte << FIM
> Ola suporte, hoje as ?date "+%H:%M"?
> ocorreu novamente aquele problema
> que eu havia reportado por
> telefone. Conforme seu pedido
> ai vai uma listagem dos arquivos
> do diretorio:
> "ls -l"
> Abracos a todos.
> FIM
```

Finalmente agora temos conhecimento para mostrar o que havíamos conversado sobre *here document*. Os comandos entre crases (`) serão priorizados e portanto o Shell os executará antes da instrução mail. Quando o suporte receber o e-mail, verá que os comandos date e ls foram executados imediatamente antes do comando mail, recebendo então uma fotografia do ambiente no momento em que a correspondência foi enviada.

O prompt primário default do Shell, como vimos, é o cifrão ($), porém o Shell usa o conceito de prompt secundário, ou de continuação de comando, que é enviado para a tela quando há uma quebra de linha e a instrução não terminou. Esse prompt, é representado por um sinal de maior (>), que vemos precedendo a partir da 2ª linha do exemplo.

Para finalizar e bagunçar tudo, devo dizer que existe uma construção mais moderna que vem sendo utilizada como forma de priorização de execução de comandos, tal qual as crases (`). São as construções do tipo $(cmd), onde cmd é um (ou vários) comando que será(ão) executado(s) com prioridade em seu contexto.

Assim sendo, o uso de crases (`) ou construções do tipo $(cmd) servem para o mesmo fim, porém para quem trabalha com sistemas operacionais de diversos fornecedores (multiplataforma), aconselho o uso das crases, já que o $(cmd) não foi portado para todos os sabores de Shell. Aqui dentro do Botequim, usarei ambas as formas, indistintamente.

Vejamos novamente o exemplo dado para as crases sob esta nova ótica:

```
$ echo Existem $(who | wc -l) usuários conectados
Existem 8 usuários conectados
```

Veja só este caso:

```
$ Arqs=ls
$ echo $Arqs
ls
```

Neste exemplo eu fiz uma atribuição (=) e executei uma instrução. O que eu queria era que a variável `$Arqs`, recebesse a saída do comando `ls`. Como as instruções de um script são interpretadas de cima para baixo e da esquerda para a direita, a atribuição foi feita antes da execução do `ls`. Para fazer o que desejamos é necessário que eu priorize a execução deste comando em detrimento da atribuição e isto pode ser feito de qualquer uma das maneiras a seguir:

```
$ Arqs=`ls`
```

ou:

```
$ Arqs=$(ls)
```

Para encerrar este assunto vamos ver só mais um exemplo. Digamos que eu queira colocar dentro da variável $Arqs a listagem longa (ls -l) de todos os arquivos começados por arq e seguidos de um único caractere (?). Eu deveria fazer:

```
$ Arqs=$(ls -l arq?)
```

ou:

```
$ Arqs=`ls -l arq?`
```

Mas veja:

```
$ echo $Arqs -rw-r--r-- 1 jneves jneves 19 May 24 19:41 arq1 -rw-r--r-- 1 jneves jneves 23 May 2
```

— Pô, saiu tudo embolado!

— Pois é cara, como eu já te disse, se você deixar o Shell "ver" os espaços em branco, sempre que houver diversos espaços juntos, eles serão trocados por apenas um. Para que a listagem saia bonitinha, é necessário proteger a variável da interpretação do Shell, assim:

```
$ echo "$Arqs"
-rw-r--r-- 1 jneves jneves 19 May 24 19:41 arq1
-rw-r--r-- 1 jneves jneves 23 May 24 19:43 arq2
-rw-r--r-- 1 jneves jneves 1866 Jan 22 2003 arql
```

— Olhe, amigo, vá treinando esses exemplos, porque, quando nos encontrarmos novamente, vou lhe explicar uma série de instruções típicas de programação Shell. Tchau! Ahh! Só mais uma coisinha que eu ia esquecendo de lhe dizer. Em Shell, o "jogo da velha" (#) é usado quando desejamos fazer um comentário.

```
$ exit # pede a conta ao garcon
```

- O termo Shell é mais usualmente utilizado para se referir aos programas de sistemas do tipo Unix que podem ser utilizados como meio de interação entre o usuário e o computador.
- É um programa que recebe, interpreta e executa os comandos de usuário, aparecendo na tela como uma linha de comandos, representada por um prompt, que aguarda na tela os comandos do usuário.

11. Resumos

Relacionamos a seguir algumas tabelas resumo de diversas funcionalidades do shell Bash e suas explicações.

11.1. Redirecionamentos de Saída

>	Redireciona a saida de um comando para um arquivo, destruindo seu conteúdo.
>>	Redireciona a saída de um comando para um arquivo, mantendo intacto o seu conteúdo.
2>	Redireciona a saída de erros para um arquivo, destruindo seu conteúdo (anexa ao fim do arquivo).

11.2. Redirecionamentos de Entrada

<	Avisa ao Shell que a entrada não será feita pelo teclado, mas sim por um arquivo
<<	Indica ao Shell que o escopo do comando começa na linha seguinte e termina quando encontrar um rótulo (label) definido.

11.3. Redirecionamentos Especiais

|	Passa a saída de um comando para entrada de outro. Conhecido como "pipe"
tee	Passa a saída de um comando para a saída padrão (tela) e também para um arquivo.

11.4. Testes em Arquivos

-d	é um diretório
-e	o arquivo existe
-f	é um arquivo normal
-L	o arquivo é um link simbólico
-r	o arquivo tem permissão de leitura
-s	o tamanho do arquivo é maior que zero
-w	o arquivo tem permissão de escrita
-x	o arquivo tem permissão de execução
-nt	o arquivo é mais recente (NewerThan)
-ot	o arquivo é mais antigo (OlderThan)
-ef	o arquivo é o mesmo (EqualFile)

11.5. Argumentos em Shell Scripts

$0	indica o comando emitido
$$	

	PID do processo executando a shell. Bastante útil para criação de arquivos temporários.
$#	número de argumentos fornecidos
$?	código de retorno. Códigos iguais a zero indicam sucesso, ao passo que valores diferentes indicam algum tipo de erro
$*	Todos os argumentos fornecidos como uma string separada por brancos
$_	Último parâmetro do comando anterior
$1, ..., $n	argumentos enviados à shell

11.6. Comparação Numérica

-lt	é menor que *(LessThan)*
-gt	é maior que *(GreaterThan)*
-le	é menor igual *(LessEqual)*
-ge	é maior igual *(GreaterEqual)*
-eq	é igual *(EQual)*
-ne	é diferente *(NotEqual)*

Exemplo

```
#!/bin/bash
if [ $# -ne 1 ];
then
echo "Sintaxe: $0 arquivo"
exit 1
fi
echo "O nome do arquivo é $1"
```

Verifica se foi fornecido como argumento o nome de um arquivo. Se não existir, encerra o processamento

Exercícios

Explique quais são as condições testadas nas sentenças abaixo:

```
[ -r lista.txt ]

[ -r $1.db ]

[ ! -w databases.txt ]

[ -x ns.sh ]

[ -f tmp.txt ]

[ $1 -lt $2 ]

[ $1 -gt 0 ]
```

```
[ $1 -eq $# ]
```

11.7. Classes de Caracteres

Os colchetes são usados para indicar grupos de caracteres. O símbolo "^" após o colchete "[" significa negação.

[abc]	a, b, c
[0-9]	qualquer número decimal
[a-z]	qualquer letra minúscula
[A-Z]	qualquer letra maiúscula
[a-zA-Z]	qualquer letra
[^0-9]	qualquer caractere que não seja um número decimal

11.8. Âncoras

^echo	caractere echo no começo da frase
[A-Za-z]+$	existência de uma palavra no final da linha
^globo$	linhas contendo apenas a palavra "globo"

11.9. Lista de Comandos

cmd1; cmd2; ...	execução sequencial
cmd1 &	execução assíncrona
cmd1 && cmd2 ...	execute cmd2 se cmd1 tiver código de saída 0 - exit(0)
cmd1||cmd2	executa cmd2 apenas se cmd1 tiver um código de retorno diferente de zero.

1. Introdução

Os conceitos que serão abordados nesta semana já foram exibidos, embora de forma indireta, nas semanas anteriores. Nesta semana iremos aprender a trabalhar com as diversas estruturas de controle de laço existentes em shell scripts: `if`, `for`, `while`, `case`, e `until`.

Para a confecção deste material utilizamos documentação do Projeto de Documentação do Linux (The Linux Documentation Project. Utilizamos também diversos artigos escritos por Júlio Cezar Neves e Rodrigo Bernardes Pimentel.

2. Condicionais

Expressões condicionais permitem decidir se determinada ação deve ou não ser tomada. A decisão é baseada na avaliação do resultado de uma expressão.

Expressões condicionais possuem muitos formatos. O mais básico deles é:

```
Se 'expressão'
então
      ação
```

A ação só é executada se a expressão avaliada for verdadeira. '2<1' é uma expressão que resulta em uma afirmativa falsa, ao passo que '2>1' é uma expressão verdadeira.

Expressões condicionais possuem outros formatos, tais como:

```
Se expressão
então
      ação 1
caso contrário
      ação 2
```

A ação 1 só é executada se a expressão testada for verdadeira. Se falsa, a ação 2 é executada em seu lugar.

Ainda uma outra forma de expressão condicional:

```
Se expressão 1 então
      ação 1
ou então
      se expressão 2
então
      ação 2
```

Neste caso, acrescentamos a condicional *ou então* (ELSE IF), que faz com que a ação 2 seja executada se a expressão 2 for verdadeira.

Falando agora sobre sintaxe:

A base para a construção `if` em bash é:

```
if [expressão];
then
  código a ser executado se a expressão testada for verdadeira
fi
```

Exemplo

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo "a expressão avaliada é verdadeira"
fi
```

O código a ser executada se a expressão entre colchetes for verdadeira, pode ser encontrada após a diretiva `then` e antes da diretiva `fi`, que indica o final do código executado condicionalmente.

Exemplo if .. then ... else

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo "a expressão avaliada é verdadeira"
else
   echo "a expressão avaliada é falsa"
fi
```

Exemplo: Condicionais com variáveis

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
    echo "a expressão avaliada é verdadeira"
else
    echo "a expressão avaliada é falsa"
fi
```

3. Loops for, while e until

Nesta seção explicaremos o uso das estruturas de laço `for`, `while` e `until`.

O laço `for` é um pouco diferente, em bash, de outras linguagens de programação. Basicamente, ele lhe permite fazer uma iteração através de uma série de 'palavras' dentro de uma string.

O laço `while` executa um trecho de código se a expressão de controle é verdadeira, e para apenas quando ela é falsa ou uma interrupção explícita é encontrada dentro do código em execução.

O laço `until` é quase igual ao `while`, com a diferença de que o código é executado enquanto a expressão de controle for falsa.

Se você suspeita que os laços `while` e `until` são similares, você acertou.

Exemplo da construção for

```
[1]   #!/bin/bash
[2]   for i in $( ls ); do
[3]       echo item: $i
[4]   done
```

Na segunda linha, nós declaramos a variável **i** de forma a que ela assuma os valores resultantes da execução do comando `ls` (`$( ls )`.

A terceira linha poderia conter mais comandos, se necessário, ou então poderiam ser colocadas mais linhas antes do `done` que sinaliza o final do laço.

A diretiva `done` indica que o código que usou os valores assumidos pela variável `$i` terminou e `$i` pode assumir um novo valor.

Este script não faz muito sentido, mas um uso mais útil seria usar o laço para encontrar apenas determinados arquivos que atendessem a um critério específico.

Um outro tipo de laço, usando a construção `for`:

```
#!/bin/bash
for i in `seq 1 10`;
do
        echo $i
done
```

Sempre tem mais de um jeito de se fazer a mesma coisa.

Um exemplo da construção while

```
#!/bin/bash
COUNTER=0
while [  $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done
```

Este código é executado enquanto o valor da variável `COUNTER` for menor que 10. A variável `COUNTER` é incrementada em 1 ao final de cada iteração.

Um exemplo da construção until

```
#!/bin/bash
COUNTER=20
until [  $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

Neste exemplo, a variável `COUNTER` é decrementada em uma unidade a cada iteração. Quando chegar atingir um valor inferior a dez, o laço é finalizado. Se executarmos este script temos:

```
COUNTER 20
COUNTER 19
COUNTER 18
COUNTER 17
COUNTER 16
COUNTER 15
COUNTER 14
COUNTER 13
COUNTER 12
COUNTER 11
COUNTER 10
```

Quando a variável `COUNTER` chega a dez, na próxima rodada o valor é 9, e o laço não é executado.

4. case

Por Rodrigo Bernardo Pimentel

Quando se quer testar uma série de condições, pode-se usar `if`, `elif` e `else`. Porém, quando o teste é para um mesmo valor de variável, a repetição pode-se tornar inconveniente:

```
if [ "$REPOSTA" = "sim" ]; then
        faz_coisas

elif [ "$RESPOSTA" = "nao"]; then
        exit 0

elif [ "$RESPOSTA" = "depende_do_tempo" ]; then
        faz_coisas_se_o_tempo_permitir

elif [ "$RESPOSTA" = "depende_do_humor" ]; then
        faz_coisas_se_o_humor_permitir

  ...
```

E por aí vai. As checagens são necessárias, afinal de contas precisamos reagir diferentemente a cada uma das condições. Mas a repetição de `elif [ "$RESPOSTA" = "..."]; then` torna-se cansativa.

Para esse tipo de problema, existe uma construção em bash chamada `case` (existe também em outras linguagens como C). Sua estrutura é:

```
case "$variavel" in
   primeira_opcao)
           comando1
           comando2
           ...
           ;;

   segunda_opcao)
           outro_comando
           ainda_outro
           ...
           ;;
   ...

   *)
           ultimo_comando

esac
```

Isso testa `$variavel` com cada uma das opções. Quando achar uma adequada, executa os comandos até `;;` e sai do loop (não tenta outras opções mais adiante). O nosso exemplo com `if` acima ficaria:

```
case "$RESPOSTA" in
        sim)
                faz_coisas
                ;;
```

```
        nao)
                exit 0
                ;;

        depende_do_tempo)
                faz_coisas_se_o_tempo_permitir
                ;;

        depende_do_humor)
                faz_coisas_se_o_humor_permitir
                ;;

        *)
                echo 'NDA!'

esac
```

Notem que `*` é um "catchall", ou seja, se o valor da variável `RESPOSTA` não for `sim`, `nao`, `depende_do_tempo` ou `depende_do_humor`, serão executandos os comandos após o `*)` (que não precisa terminar em `;;`).

Normalmente, os scripts de inicialização do sistema (em `/etc/init.d` ou `/etc/rc.d/init.d`, dependendo da distribuição) necessitam de um parâmetro, um dentre `start`, `stop`, `status`, `restart` ou `reload` (às vezes mais). A situação ideal para se usar um `case`, certo? O pessoal das distribuições também acha. Portante, se você quiser exemplos de `case`, procure nos scripts de inicialização da sua distribuição!

5. Bash - Estruturas Básicas

> Leia com atenção o tutorial a seguir, de autoria de Rodrigo Bernardo Pimentel, para consolidar os conhecimentos sobre estruturas de laço (loop) em bash.

É comum queremos executar a mesma função, ou uma função parecida, sobre uma série de argumentos, um por vez. Em bash, há duas estruturas básicas pra isso: `for` e `while`.

O `for`, em bash, é diferente do `for` em linguagens como C ou Perl. Nessas linguagens, o `for` é simplesmente um `while` mais completo. Em bash, o for atua sobre uma seqüência de parâmetros (não necessariamente numéricos ou seqüenciais). Por exemplo:

```
[rbp@muppets ~]$ for i in 1 2 3 4 5; do echo $i; done
1
2
3
4
5
[rbp@muppets ~]$
```

Ou,

```
[root@muppets ~]$ for login in rbp sizinha queiroz; do adduser $login; done
[root@muppets ~]$
```

Você pode inclusive usar o for a partir de uma saída de comando. Por exemplo, digamos que você tenha um arquivo com uma série de nomes de usuários a serem acrescentados ao sistema, como no exemplo acima:

```
[root@muppets ~]$ for login in `cat /tmp/usuarios`; do adduser $login; done
[root@muppets ~]$
```

O `while`, por sua vez, tem funcionamento semelhante à maioria das linguagens procedurais mais comuns. Um comando ou bloco de comandos continua a ser executado enquanto uma determinada condição for verdadeira. Por exemplo, imitando o exemplo acima:

```
[rbp@muppets ~]$ while [ $i -le 5 ]; do echo $i; i=$(($i + 1)); done
1
2
3
4
5
[rbp@muppets ~]$
```

Aos poucos:

```
while [ $i -le 5 ]
```

O `while` deve ser seguido de `verdadeiro` ou `falso`. Se for `verdadeiro`, o bloco é executado e o teste é repetido. Se for falso, não.

No caso, `[ $i -le 5 ]` é uma forma de usarmos o comando `test`. Esse comando testa uma expressão e retorna verdadeiro ou falso. Poderia ser escrito como

```
while test $i -le 5
```

`$i -le 5` é a expressão testada. O programa `test` aceita alguns operadores. Entre eles:

-lt (less than)	primeiro argumento é menor do que o segundo
-le (less or equal)	primeiro argumento é menor ou igual ao segundo
-gt (greater than)	primeiro argumento é maior do que o segundo
-ge (greater or equal)	primeiro argumento é maior ou igual ao segundo
=	primeiro argumento é igual ao segundo
!=	primeiro argumento é diferente do segundo

O programa `test` pode também fazer testes `unários`, ou seja, com só um argumento. Por exemplo,

```
arq="/tmp/arquivo"
tmp=$arq
i=1
while [ -e $tmp ]; do
        i=$(($i+1))
        tmp=$arq$i
done
touch $tmp
```

Esse scriptzinho (note que não o fiz na linha de comando, mas indentado, para usá-lo a partir de um arquivo; funciona dos dois jeitos) só sai do loop quando achar um nome de arquivo que não exista. Ou, de forma mais didática, "enquanto existir (-e) o arquivo cujo nome está na variável $tmp, ele continua executado o bloco de comandos".

Alguns dos operadores unários mais comuns são:

-e	arquivo ou diretório existe
-f	é um arquivo (em oposição a ser um diretório)
-d	é um diretório
-x	arquivo tem permissão de execução para o usuário atual
-w	arquivo tem permissão de escrita pelo usuário atual
-r	arquivo tem permissão de leitura pelo usuário atual

Para mais detalhes, `man test`.

Continuando:

```
do echo $i
```

Logo após um `while <teste>`, é preciso iniciar o primeiro comando com `do`. Os seguintes (se houver), não.

A próxima linha mostra um exemplo de algo que não tem nada a ver com a estrutura do `while` em si, mas é um truquezinho legal de bash:

```
i=$(($i + 1))
```

A construção `$((expressão))` é um operador matemático em bash. Isso é expandido para o resultado da expressão. Por exemplo,

```
[rbp@muppets ~]$ echo $((2 + 3)) =
5
[rbp@muppets ~]$ echo $((2 - 3))  # Funciona com números negativos =
-1
[rbp@muppets ~]$ echo $((2 * 3)) =
6
[rbp@muppets ~]$ echo $((10 / 2)) =
5
[rbp@muppets ~]$ echo $((3 / 2))  # Não usa casas decimais =
1
[rbp@muppets ~]$  =
```

Mas, como diz um amigo meu, voltando...

```
done
```

Isso termina "oficialmente" o loop `while`. A propósito, como pode ter sido notado, termina o `for` também.

Como foi dito acima, o while espera "verdadeiro" ou "falso". Eu nunca disse que esperava isso só do programa `test` :)

Com efeito, qualquer expressão pode ser usada, e seu valor de retorno será utilizado para determinar se é "verdadeiro" ou "falso". Para quem não sabe, todo programa em Unix retorna um valor ao terminar sua execução. Normalmente, se tudo correu bem o valor retornado é 0 (zero). Se há algum erro, o valor retornado, diferente de zero, indica o tipo de erro (veja as manpages dos programas; `man fetchmail` seção `exit codes` é um bom exemplo). Portanto, ao contrário do que programadores de C ou Perl poderiam achar intuitivo (dentro de um `while`, ou uma condição em geral), um programa que retorna 0 é considerado "verdadeiro" aos olhos do `while`.

Assim, podemos fazer:

```
while w | grep -qs rbp; do
        sleep 5s
done
echo 'rbp acaba de sair do sistema!'
```

Nesse exemplo, o while checa o retorno da expressão `w | grep -qs rbp`. Isso retorna "verdadeiro" quando o grep acha `rbp` na saída do comando `w`. Toda vez que achar, espera 5 segundos e checa de novo. Quando não achar, sai do loop e mostra um aviso de que a última sessão do rbp foi fechada.

Pra finalizar: se você quiser fazer um loop infinito, pode usar `:` (dois pontos) como condição sempre verdadeira:

```
while : ; do
        echo 'Emacs rules!'
done
```

Isso vai imprimir uma constatação sábia infinitamente, até você usar C-c (Ctrl + C). Normalmente, isso é utilizado com alguma condição de parada.

6. Definição de funções em shell scripts

Para encerrar esta seção, falaremos um pouco sobre funções, que irão nos ajudar a tornar nosso código mais limpo, elegante, eficiente e fácil de entender.

Uma aplicação comum é quando se requer que um usuário pressione a tecla <ENTER> ou forneça algum dado para o script. Poderíamos, por exemplo, definir uma função chamada `TecleEnter`:

```
#!/bin/bash

TecleEnter ( ) {
echo Tecle ENTER
read x
}
```

É fato conhecido que shell scripts começam simples e terminam enormes, abrangendo várias páginas de código. A definição de funções para tarefas tais como menus, verificação de argumentos, etc, torna o código muito mais legível e fácil de ser analisado.

Você pode inclusive criar programas e colocar as funções em um arquivo separado. No início do programa você emite um comando para carregar as funções, como abaixo:

```
#!/bin/bash

. /usr/local/etc/func.sh

TecleEnter
```

7. Exemplos Práticos

Listamos a seguir, alguns exemplos práticos de shell scripts que utilizam estruturas de controle de laço. Leia os códigos com atenção e reproduza-os em seu computador, observando com atenção os resultados.

Um alerta, muitos dos exemplos que se seguem, utilizam conceitos e comandos que serão apresentados nas próximas duas semanas. Mesmo assim, para oferecermos exemplos reais do poder que shell scripts possuem, optamos por inserir estes exemplos neste ponto. Como estes exemplos são muito bem explicados e detalhados, o estudo deste material poderá contribuir significativamente para a compreensão do material das próximas semanas e também para dar-lhe mais fluência em programação bash.

7.1. Here Strings

Por Julio Cezar Neves

Primeiro um programador com complexo de inferioridade criou o redirecionamento de entrada e representou-o com um sinal de menor `(<)` para representar seus sentimentos. Em seguida, outro sentindo-se pior ainda, criou o *here document* representando-o por dois sinais de menor `(<<)` porque sua fossa era maior. O terceiro, pensou: "estes dois não sabem o que é estar por baixo"... Então criou o *here strings* representado por três sinais de menor `(<<<)`.

Brincadeiras a parte, o *here strings* é utilíssimo e, não sei porque, é um perfeito desconhecido. Na pouquíssima literatura que sobre o tema, nota-se que o *here strings* é freqüentemente citado como uma variante do *here document*, com a qual discordo pois sua aplicabilidade é totalmente diferente daquela.

Sua sintaxe é simples:

```
$ comando <<< $cadeia
```

Onde cadeia é expandida e alimenta a entrada primária `(stdin)` de comando.

Como sempre, vamos direto aos exemplos dos dois usos mais comuns para que vocês próprios tirem suas conclusões.

Uso #1

Substituindo a famigerada construção `echo "cadeia" | comando`, que força um `fork`, criando um subshell e onerando o tempo de execução. Vejamos alguns exemplos:

```
$ a="1 2 3"
$ cut -f 2 -d ' ' <<< $a # normalmente faz-se: echo $a | cut -f 2 -d ' '
2
$ echo $NomeArq
Meus Documentos
$ tr "A-Z " "a-z_" <<< $NomeArq  Substituindo o echo ... | tr ...
meus_documentos
$ bc <<<"3 * 2"
6
```

```
$ bc <<<"scale = 4; 22 / 7"
3.1428
```

Para mostrar a melhoria no desempenho, vamos fazer um loop de 500 vezes usando o exemplo dados para o comando `tr`:

```
$ time for ((i=1; i<= 500; i++)); { tr "A-Z " "a-z_" <<< $NomeArq >/dev/null; }

real    0m3.508s
user    0m2.400s
sys     0m1.012s
$ time for ((i=1; i<= 500; i++)); { echo $NomeArq | tr "A-Z " "a-z_" >/dev/null; }

real    0m4.144s
user    0m2.684s
sys     0m1.392s
```

Veja agora esta seqüência de comandos com medidas de tempo:

```
$ time for ((i=1;i<=100;i++)); { who | cat > /dev/null; }

real    0m1.435s
user    0m1.000s
sys     0m0.380s
$ time for ((i=1;i<=100;i++)); { cat <(who) > /dev/null; }

real    0m1.552s
user    0m1.052s
sys     0m0.448s
$ time for ((i=1;i<=100;i++)); { cat <<< $(who) > /dev/null; }

real    0m1.514s
user    0m1.056s
sys     0m0.412s
```

Observando este quadro você verá que no primeiro usamos a forma convencional, no segundo usamos um *named pipe* temporário para executar uma substituição de processos e no terceiro usamos *here string*. Notará também que ao contrário do exemplo anterior, aqui o uso de *here string* não foi o mais veloz. Mas repare bem que neste último caso o comando who está sendo executado em um subshell e isso onerou o processo como um todo.

Vejamos uma forma rápida de inserir uma linha como cabeçalho de um arquivo:

```
$ cat num
1       2
3       4
5       6
7       8
9       10
$ cat - num <<< "Impares Pares"
Impares Pares
1       2
3       4
5       6
7       8
9       10
```

Uso #2

Outra forma legal de usar o *here string* é casando-o com um comando read, não perdendo de vista o que aprendemos sobre `IFS` (veja mais sobre esta variável no Papo de Botequim). O comando `cat` com as opções -vet mostra o <ENTER> como `$`, o <TAB> como `^I` e os outros caracteres de controle com a notação `^L` onde `L` é uma letra qualquer. Vejamos então o conteúdo de uma variável e depois vamos ler cada um de seus campos:

Também podemos ler direto para um vetor (array) veja:

```
$ echo $Frutas
Pera:Uva:Maçã
$ IFS=:
$ echo $Frutas
Pera Uva Maçã        # Sem as aspas o shell mostra o IFS como branco
$ echo "$Frutas"
Pera:Uva:Maçã        # Ahhh, agora sim!
$ read -a aFrutas <<< "$Frutas"  # A opção -a do read, lê para um vetor
$ for i in 0 1 2
> do
>     echo ${aFrutas[$i]}        # Imprimindo cada elemento do vetor
> done
Pera
Uva
Maçã
```

7.2. IFS - Inter Field Separator

Por Julio Cezar Neves

O shell tem uma variável interna chamada `IFS` - *Inter Field Separator* (será Tabajara? :) - cujo valor default podemos obter da seguinte forma:

```
$ echo "$IFS" | od -h
0000000 0920 0a0a
0000004
```

O programa `od` com a opção `-h` foi usado para gerar um dump hexadecimal da variável. E lá podemos ver:

Valor Hexadecimal	Significado
09	<TAB>
20	Espaço
0a	<ENTER>

Ou seja os separadores entre campos (tradução livre de IFS) default são o `<TAB>`, o espaço em branco e o `<ENTER>`. O IFS é usado em diversas instruções, mas seu uso é muito comum em par com o `for` e/ou com o `read`. Vejamos um exemplo semelhante ao dado no Cantinho do Shell de 15/12/2006, que me inspirou a escrever este artigo.

```
$ cat script1.sh
#!/bin/sh
```

```
while read linha
do
    awk -F: '{print $1}' /etc/passwd > /dev/null
done < /etc/passwd
```

Como podemos ver este script não faz nada (sua única saída foi enviada para /dev/null para não deturpar os tempos de execução), mas vamos usá-lo para avaliação dos tempos de execução.

Este script foi alterado, trocando o `awk` pelo `cut`, e ficando com a seguinte cara:

```
$ cat script2.sh
#!/bin/sh
while read linha
do
     echo $linha | cut -f1 -d: > /dev/null
done < /etc/passwd
```

Mais uma outra alteração, usando 99% de intrínsecos do Shell (exceto o comando echo) desta vez tomando partido do IFS:

```
$ cat script3.sh
#!/bin/sh
IFS=:
while read user lixo
do
     echo $lixo > /dev/null
done < /etc/passwd
```

Neste último exemplo, transformamos o separador padrão em dois-pontos (:) e usamos sua propriedade em conjunto com o `read`, isto é, o primeiro campo veio para a variável `user` e o resto para a variável `lixo`.

Em seguida, fiz um script usando Shell quase puro (novamente o echo foi o vilão). Repare a construção `${linha%%:*}`, que é um intrínseco (built-in) do Shell que serve para excluir da variável linha o maior casamento com o padrão especificado (:* - que significa "de dois-pontos em diante"), ou seja, excluiu de linha tudo a partir do último dois-pontos, contado da direita para a esquerda.

Dica: quem não conhece o macete acima, não pode deixar de ler a seção referente a expansão de parâmetros em: http://twiki.softwarelivre.org/bin/view/TWikiBar/TWikiBarPapo009

```
$ cat script4.sh
#!/bin/sh
while read linha
do
     echo ${linha%%:*} > /dev/null
done < /etc/passwd
```

Para finalizar, adaptei o script escrito pelo incansável Rubens Queiroz que, exceto pelo `awk`, é Shell puro.

```
$ cat script5.sh
#!/bin/sh
for user in `awk -F: '{print $1}' /etc/passwd`
do
    echo $user > /dev/null
done
```

Agora, o mais importante: reparem os tempos da execução de cada um deles:

```
$ time script1.sh

real    0m0.123s
user    0m0.032s
sys     0m0.032s
$ time script2.sh

real    0m0.297s
user    0m0.152s
sys     0m0.084s
$ time script3.sh

real    0m0.012s
user    0m0.004s
sys     0m0.004s
$ time ./script4.sh

real    0m0.012s
user    0m0.004s
sys     0m0.008s
$ time ./script5.sh

real    0m0.014s
user    0m0.012s
sys     0m0.004s
```

Reparem que estas diferenças de tempo foram obtidas para um arquivo com somente 29 linhas. Veja:

```
$ wc -l /etc/passwd
29 /etc/passwd
```

Um outro uso interessante do IFS é o que vemos a seguir, primeiramente usando o IFS default que como vimos é <TAB>, Espaço e <ENTER>:

```
$ Frutas="Pera Uva Maçã"
$ set - $Frutas
$ echo $1
Pera
$ echo $3
Maçã
```

Agora, vamos alterar o IFS para fazer o mesmo com uma variável qualquer, e para tal vamos continuar usando o famigerado /etc/passwd:

```
$ Root=$(head -1 /etc/passwd)
$ echo $Root
root:x:0:0:root:/root:/bin/bash
$ oIFS="$IFS"
$ IFS=:
$ set - $Root
$ echo $1
root
$ echo $7
/bin/bash
$ IFS="$oIFS"
```

Senhores, neste artigo pretendi mostrar duas coisas:

- O uso do IFS que, infelizmente para nós, é uma variável pouco conhecida do Shell e
- Que quanto mais intrínsecos do Shell usamos, mais veloz e performático fica o script.

7.3. Mala direta por email

Por Rubens Queiroz de Almeida

O exemplo de hoje mostra como montar um programa para envio de mala direta, por email, para diversos destinatários. Utiliza-se, para montar a lista de destinatários, um banco de dados em modo texto. A mensagem está embutida dentro da própria shell, utilizando um recurso chamado ***in here documents***, ou, traduzindo, documento embutido.

O banco de dados para este exemplo consiste de linhas em que os campos são separados pelo caractere `**:**`.

O objetivo do shell script é enviar, para uma lista de assinantes, o novo número de seu telefone celular. As mensagens devem ser personalizadas, com o nome do destinatário.

Banco de dados: Arquivo lista.txt

```
Rubens Queiroz de Almeida:queiroz@dicas-l.com.br:98761234
José Amâncio Bueno:amancio@example.com.br:99998888
Maria Aparecida Antunes:aparecida@example.com:81772976
```

O programa

```
#!/bin/bash

IFS=:

while read nome email telefone
do
echo $nome ... $email ... $telefone

/usr/sbin/sendmail $email << EOF
Subject: [EXAMPLE] Mudança do número de telefone
From: Rubens Queiroz de Almeida <queiroz@dicas-l.com.br>
To: $nome <$email>

A/C $nome

Gostariamos de lhe comunicar que o novo número de seu
telefone celular é $telefone.

Atenciosamente,

Serviço de Atendimento ao Cliente
Example.Com
EOF
done < lista.txt
```

Passemos agora à explicação dos campos.

```
IFS=:
```

IFS significa *INPUT FIELD SEPARATOR*. No nosso banco de dados em modo texto, o caractere que separa os campos é `**:**`.

```
while read nome email telefone
```

Esta diretiva sinaliza o início de um laço que irá ler, linha a linha, todo o conteúdo do arquivo lista.txt. O primeiro campo será atribuído à variável `nome`, o segundo à variável `email` e o terceiro à variável `email`.

```
echo $nome ... $email ... $telefone
```

Esta diretiva não faz nada de útil, apenas ecoa para a tela o valor das três variáveis à medida em que são lidas. Apenas para acompanhamento da execução do programa.

```
/usr/sbin/sendmail $email << EOF
```

Esta linha invoca o sendmail para o envio da mensagem. Aqui usamos o ***IN HERE DOCUMENT***. O texto que será processado pelo programa `sendmail` vai da linha seguinte até encontrar, no começo da linha, os caracteres `EOF`.

Uma mensagem eletrônica consiste do cabeçalho, seguida de uma linha em branco e em seguida do corpo da mensagem. Podemos colocar quantas linhas de cabeçalho desejarmos, dependendo do que necessitarmos. Neste caso, identificamos o assunto da mensagem e o remetente (`Subject` e `From`). Importante colocar também o campo `To:`, especificando, como fizemos, o nome completo do destinatário e o email. Fazemos isto para evitar que apareça na mensagem o texto `undisclosed recipients`, que sinaliza que a mensagem foi enviada em lote, para dezenas ou centenas de pessoas. Poderíamos colocar outros campos, como por exemplo `Reply-To`, quando desejarmos que a resposta seja enviado para uma pessoa que não seja o remetente.

O que vem em seguida é a mensagem. Note bem os pontos em que foram inseridas as variáveis. Utilizamos aqui as variáveis `$nome` e `$telefone`.

IMPORTANTE: Como você está enviando mensagens para diversas pessoas, antes de fazer o envio real, faça um teste para certificar-se de que tudo está funcionando corretamente. A não ser que você seja realmente um programador genial, a chance de cometer erros é grande. Para fazer o teste, substitua a chamada ao `sendmail` por

```
/usr/sbin/sendmail queiroz@dicas-l.com.br << EOF
```

Observe que eu retirei a variável `$email` desta linha. As mensagens serão enviadas apenas para meu endereço, para que eu possa verificar se tudo está realmente correto. É claro que você não precisa enviar a mensagem de teste para milhares de endereços. Para testar, crie um novo arquivo `lista.txt` contendo apenas uns cinco ou dez endereços. É mais do que suficiente.

As aplicações desta receita são inúmeras e podem ser usadas em diversos contextos. Aqui vale a imaginação do programador.

Meus agradecimentos, mais uma vez, ao *Papai do Shell*, Júlio Neves, que foi quem me ensinou este truque.

7.4. Geração de arquivos de índice em html

Por Rubens Queiroz de Almeida

Em dos sites que mantenho, chamado Contando Histórias, eu criei uma página onde relaciono todo o conteúdo do site. Esta página é gerada através de um shell script que conta o número de mensagens existentes, divide este número por dois, e monta uma tabela com duas colunas. Para entender melhor o que é feito, nada melhor do que visitar a página de arquivo do site.

Vamos então ao script e à explicação de seu funcionamento.

```
#!/bin/bash

homedir=/html/contandohistorias

cd $homedir/html

# O laço que se segue trabalha
# sobre todos os arquivos do diretório
# /html/contandohistorias/inc que
# tenham a terminação "inc". Estes são arquivos
# no formato html, gerados pelo software txt2tags
# (txt2tags.sourceforge.net).

for file in *.inc
do

# O arquivo php final é formado a partir do nome do
# arquivo terminado em "inc". Este nome é atribuído
# à variável $php, definida no próximo comando

  php=`echo $file | sed 's/inc/php/'`

# No arquivo html a primeira linha contém o título
# da mensagem, formatada como título de nível 1
# (H1). O título é extraído desta linha com o
# comando sed e em seguida convertido em uma
# referência html, para ser usada mais tarde na
# montagem do arquivo geral.

  sed -n 1p $file | sed 's:<H1>::;s:</H1>:</A>:' \
  | sed "s:^:<BR><A HREF=/historias/$php>:" >> /tmp/idx.tmp

# Usamos o comando tac para inverter a ordem das
# mensagens, deixando as mais recentes em primeiro
# lugar na listagem.

  tac /tmp/idx.tmp > /tmp/idx.$$ && mv /tmp/idx.$$ /tmp/idx.tmp
done

cp /tmp/idx.tmp $homedir/inc/listagem.inc

# Fazemos a seguir a contagem de linhas do arquivo
# idx.tmp, que representa o total de mensagens já
# enviadas. A variável $half é obtida dividindo
# por 2 o número total de linhas do arquivo

lines=`wc -l /tmp/idx.tmp | awk '{print $1}'`
```

```
half=`expr $lines / 2`

# Usamos agora o comando split para partir o
# arquivo em dois. A diretiva "-l" sinaliza que
# a divisão do arquivo deve ser feita levando-se
# em conta o número de linhas (lines).

split -l $half /tmp/idx.tmp

# o comando split gera dois arquivos "xaa" e
# "xbb". Estes dois arquivos formarão as duas
# colunas da tabela.

mv xaa $homedir/inc/coluna1.inc
mv xab $homedir/inc/coluna2.inc

# A seguir, fazemos a construção do arquivo
# php final, através da inclusão dos diversos
# elementos da página: cabeçalho (Head.inc), barra
# de navegação (navbar.inc), barra da esquerda
# (esquerda.inc), e as duas colunas da tabela
# (coluna1.inc e coluna2.inc).


echo "<?PHP include(\"/html/contandohistorias/inc/Head.inc\"); ?>
<div id=top>
<H1>Contando Histórias</H1>
<?PHP include(\"/html/contandohistorias/inc/navbar.inc\"); ?>
</div>

<div id=mainleft>
<?PHP include(\"/html/contandohistorias/inc/esquerda.inc\"); ?>
</div>

<div id=maincenter>
<h1>Arquivo Lista Contando Histórias</h1>
<table>
<tr valign=top>
<td>
<?PHP include(\"/html/contandohistorias/inc/coluna1.inc\"); ?>
</td>
<td>
<?PHP include(\"/html/contandohistorias/inc/coluna2.inc\"); ?>
</td>
</tr>
</table>
</html>
</body>" > $homedir/arquivo.php

rm /tmp/idx.*
```

7.5. Catálogo de Telefones via Web

Por Rubens Queiroz de Almeida

Este script foi desenvolvido para uso na Web (cgi).

Nós temos um banco de dados no formato texto, onde os campos são separados pelo caractere `:`.

Em nosso exemplo de testes utilizamos os seguintes dados:

```
Rubens Queiroz de Almeida:queiroz@dicas-l.com.br:99876543:Campinas:SP
Maria Auxiliadora Neves:maria@example.com.br:98987676:Jundiaí:SP
José Expedito Silva:ze@example.com.br:88887777:Boa Esperança:MG
Marcone Ferraz:ferraz@example.com.br:77775454:Porto Alegre:RS
```

O script oferece um campo de busca, onde podemos digitar um ou mais caracteres. A busca é feita com o comando `egrep`. As linhas do banco de dados que contiverem os campos pesquisados são então formatadas para apresentarem os resultados no seguinte formato:

```
Nome: José Expedito Silva
Email: ze@example.com.br
Telefone: 88887777
Cidade: Boa Esperança
```

O campo `email` é um link com o formato `mailto`, para que se possa enviar uma mensagem para a pessoa clicando-se em seu endereço eletrônico.

Para ver como o script funciona, você pode utilizar o script de demonstração disponível no site da Dicas-L.

A seguir, o script com comentários:

```
#!/bin/sh

# Definição dos programas e arquivos de dados utilizados

GREP=/bin/egrep
SED=/bin/sed
DB=/var/www/cgi-bin/fonebook.dat

# Todo script cgi deve se iniciar pelas duas linhas que seguem

echo Content-type: text/html
echo

# A seguir fazemos um teste para verificar se o comando
# grep existe e, caso exista, verificamos se foi passado
# um número de argumentos ao programa que seja diferente
# de zero. Fazemos também uma formatação da página com
# diretivas HTML. É muito importante prestar atenção no uso
# das aspas duplas (").  Se não foi dado nenhum argumento
# de pesquisa, é exibida a página contendo o campo de busca
# (primeira parte do segundo if). Caso contrário, é realizada
# a consulta e os resultados, se houverem, são exibidos.

if [ -x $GREP ]; then
   if [ $# = 0 ]; then
     echo "
     <H1 style=\"text-align:center\">Consulta ao Cadastro Telefônico</H1>
     <TABLE ALIGN=CENTER BORDER=0 CELLSPACING=0 CELLPADDING=0 COLS=2 WIDTH=800>
     <TR><TD>
     <CENTER> </CENTER>
     <ISINDEX prompt=\"Forneça argumento para busca: \">
     <CENTER>
     <P>Após fornecido o argumento de busca, pressione a tecla <B>
        &lt;ENTER></B> ou <B>&lt;RETURN></B>.
```

```
      <BR> 
      </CENTER>
      <TR><TD>
      </TABLE>"
    else
      echo "
      <TITLE>Resultado da Busca</TITLE>
      <CENTER>
      <H1>Resultado da Busca para: $*</H1>
      </CENTER>
      <UL>"

# Atribuímos à variável busca todos os valores fornecidos
# pelo usuário e formatamos substituindo os espaços em
# branco pelo caractere "|" para submeter estes dados ao
# comendo grep

      busca=`echo $*|sed 's/ /|/'g`
      $GREP -i -w "$busca" $DB > /tmp/fonebook.$$

# A seguir, fazemos um teste para verificar se o arquivo
# com os resultados existe e é diferente de zero. Se for
# igual a zero não obtivemos sucesso na busca e o usuário
# deve ser notificado.

      if [ -s /tmp/fonebook.$$ ]; then

# O nosso banco de dados no formato texto usa como separador
# dois pontos ":".  Precisamos informar esta fato através
# da variável IFS (INPUT FIELD SEPARATOR).

        IFS=":"
        while read nome email celular cidade estado
        do
          echo "<LI><B>Nome</B>: $nome</A>
                <BR><B>Email</B>: <A HREF=mailto:$email</A>$email</A>
                <BR><B>Telefone</B>: $celular<BR>
                <B>Cidade</B>: $cidade
              <HR COLOR=RED NOSHADE>"
        done < /tmp/fonebook.$$
        echo "</UL>"

      else
         echo "<P>Sua busca não gerou resultados<BR>
               <A HREF=/cgi-bin/fonebook.cgi>Realizar nova busca</a>"
      fi
      echo "
      <P><FONT SIZE=-1><A HREF=#TOP>De volta ao topo desta página</A>

      </BODY>
      </HTML>"
      fi
else
        echo "Aconteceu um problema na busca... "
fi
```

7.6. Seleção aleatória de texto para exibição em páginas Web

Por Rubens Queiroz de Almeida

Em dois dos meus sites, Aprendendo Inglês e Contando Histórias, eu uso um script cgi que seleciona, dentre o acervo publicado no site, uma mensagem aleatória.

Para ver como funciona, visite o site Aprendendo Inglês ou o site Contando Histórias. O mecanismo é o mesmo usado no script para gerar os números da loteria, com apenas algumas pequenas modificações.

```
#!/bin/bash

homedir=/var/www/

cd $homedir/html

# Neste ponto eu defino os limites inferior e superior
# para a seleção das mensagens. O limite inferior é
# 1 e o superior é igual ao número de arquivos existentes
# no diretório. Eu obtenho este número através da combinação
# dos comandos ls e wc.

LOWER=1
LENGTH=`ls *.inc | wc -l`

# Aqui entra o perl novamente, para selecionar um número
# aleatoriamente, entre os limites fornecidos

Random=`perl -e "print int(rand($LENGTH+1))+$LOWER;"`

# Atribuimos à variável Display o nome do arquivo
# selecionado aleatoriamente. O comando ls lista
# todos os arquivos do diretório e o sed seleciona,
# da lista completa, apenas  o nome do arquivo.
# A variável Random contém o número aleatório gerado
# e o sed imprime apenas a linha correspondente a este
# número, que é o nome do arquivo que o script cgi
# exibirá na tela

Display=`ls | sed -n -e ${Random}p`

# A seguir vem a montagem da página html. Como padrão,
# temos as duas linhas com os comandos "echo" e em seguida
# o conteúdo da página gerada. Os comandos "cat" logo a seguir
# ecoam para a tela o conteúdo de três arquivos:
# 1. head.inc: cabeçalho da página html
# 2. $Display: o arquivo selecionado aleatoriamente
# 3. rodape.inc: rodapé da página html

echo "Content-type:text/html";
echo

cat $homedir/inc/head.inc

cat $Display

cat $homedir/inc/rodape.inc

echo "</body></html>"
```

Este foi um exemplo muito simples, para finalidades didáticas. Este script pode ficar muito sofisticado, criando páginas html com css e qualquer outro recurso que estiver à mão.

1. Introdução

Esta semana iniciamos o estudo de um assunto fundamental para bons programadores shell. Conhecer alguns dos comandos mais usados juntamente com alguns exemplos de utilização. Basicamente, um shell script consiste de alguns comandos, frequentemente passando informações uns para os outros, alguns estruturas de controle e mais nada. Os comandos podem ser considerados blocos que conversam entre si. De acordo com a filosofia de sistemas Unix, cada comando deve fazer apenas uma coisa. E bem, muito bem. Por definição, devem ser simples e fáceis de entender. O maior poder reside justamente na combinação de funcionalidades. O conceito de pipes, ou tubulações, é vital ao Unix e à programação shell. Cada comando faz uma parte do trabalho (e faz bem feito). A saída gerada é então enviada a um outro programa, que por sua vez fará aquilo que conhece, e assim por diante.

De tudo isto, reforço mais uma vez a importância de se conhecer os comandos do Unix/Linux e suas principais funcionalidades. Este documento apresenta apenas alguns deles, os mais usados. Mas existem muitos outros e mesmo os comandos aqui apresentados podem ser usados de formas infinitamente diferentes das apresentadas aqui. Esta desenvoltura virá com o tempo, em que aprenderemos gradativamente, a empregar mais recursos, mais comandos, de forma mais eficaz.

Leia com atenção as explicações que se seguem e procure reproduzir, em seu computador, os exemplos apresentados. Tente também imaginar formas diferentes de realizar as mesmas tarefas. Procure entender tudo o que acontece e procure também imaginar situações práticas, que possam lhe ajudar a resolver os problemas do seu dia a dia.

A programação shell é muito simples, podemos fazer programas desde o primeiro dia. Mas ao mesmo tempo é excitante, pois seu potencial é praticamente ilimitado e a cada dia podemos aprender mais.

2. Documentação

Sistemas GNU/Linux possuem uma documentação abundante e de boa qualidade. A primeira coisa que deve ser aprendida ao iniciarmos nosso estudo dos comandos do sistema, é aprender como obter ajuda.

Dentre os comandos que nos oferecem ajuda, temos o `man`, que formata as páginas de documentação do sistema para visualização. Por exemplo, para sabermos a sintaxe do comando `ls`, basta digitarmos:

```
man ls
```

Entretanto, se não sabemos o nome do comando, fica dificil obter ajuda. Para resolver este problema, existe o comando `apropos`, que nos permite localizar o comando desejado especificando, como argumento, palavras que sirvam para contextualizar as funções do comando que procuramos.

Vejamos um exemplo:

```
apropos list | head
dir (1)              - lista o conteúdo do diretório
hosts.equiv (5)      - listagem de máquinas e usuários que são concoradada...
ls (1)               - lista o conteúdo do diretório
securetty (5)        - arquivo que lista os terminais (ttys) nos quais o supe...
services (5)         - lista de serviços da rede Internet
signal (7)           - lista de sinais disponíveis
suffixes (7)         - lista de sufixos de arquivos
vdir (1)             - lista o conteúdo do diretório
```

Na coluna da esquerda, temos o nome do comando, e na coluna da esquerda, a sua descrição. Uma vez localizado o comando que buscamos, podemos então, com o comando `man`, obter informações mais detalhadas.

Da mesma forma, se quisermos obter uma descrição simplificada da funcionalidade de um comando, similar ao conteúdo gerado pelo comando `apropos`, podemos usar o comando `whatis`:

```
% whatis ls
ls (1)          - lista o conteúdo do diretório
ls (1)          - list directory contents
```

O comando `man` pode gerar um arquivo formatado, próprio para impressão. Para isto usamos a diretiva `-t`. Esta diretiva faz com que o comando man redirecione a saída gerada para um formatador que por sua vez irá criar um arquivo postscript.

```
% man -t tcpdump > tcpdump.ps
```

Em sistemas FreeBSD e Linux, o formatador utilizado é o groff, que normalmente faz parte da instalação básica destes sistemas. Os arquivos postscript resultantes são bastante mais agradáveis de se ler e podem ser visualizados com o comando gs (ghostscript) ou mesmo impressos em uma impressora postscript.

3. O Comando ls

Existem diversos comandos para obter informações a respeito do sistema de arquivos. O mais usado e útil é o comando `ls`.

Um dos comandos mais utilizados em sistemas unix é o comando ls. Forma uma dupla inseparável com o comando "cd".

Embora simples de se usar, existem algumas características do comando "ls" que podem nos ajudar a economizar tempo e trabalhar mais produtivamente.

Geralmente trabalhamos com o comando ls da seguinte forma:

```
% ls
CCUECMAG   XF86Config.gz    lib
DSC        a                links
DicasL     a2ps.tar.gz      mail
EFR        amanda           mirror
FOB        bin              packages
```

Desta forma, listamos os arquivos do diretório corrente pelo nome. O comando ls, sem nenhum argumento, não lista todos os arquivos. Os arquivos iniciados por "." são omitidos. Estes arquivos são criados pelos aplicativos que usamos, tal como netscape, elm, pine, shells e outros. São omitidos visto que não precisamos vê-los toda vez que listamos nosso diretório home. Veja só:

```
% ls -a
.            .pine-debug2     TMP
..           .pine-debug4     XF86Config.gz
.Xauthority  .pinerc          a
.acrorc      .procmail        a2ps.tar.gz
.addressbook .procmailrc      amanda
... (linhas omitidas)
```

Como você pode ver, a listagem ficou consideravelmente maior.

Tomemos agora apenas as primeiras duas linhas da listagem anterior:

```
.          .pine-debug2     TMP
..         .pine-debug3     XF86Config.1.gz
```

As entradas "." e ".." indicam respectivamente o diretório corrente e o diretório um nível acima. Todos os diretórios em sistemas Unix contém estas duas entradas. Ou seja, são perfeitamente dispensáveis de qualquer listagem.

O comando

```
% ls -A
```

gera uma listagem completa, inclusive com os arquivos escondidos, porém não exibe as entradas para o diretório corrente "." e o diretório acima "..".

Outro problema quando se emite apenas o comando ls sem argumentos. é que não conseguimos identificar o tipo de arquivos. Para remediar este problema podemos emitir o comando:

```
% ls -l
lrwxrwxrwx   1 queiroz  supsof        29 Aug 18 12:33 efr -> /www/ns-home/docs/r
-rw-r--r--   1 queiroz  supsof   1307554 Aug 11 08:54 efr01.zip
-rw-r--r--   1 queiroz  supsof      8031 Sep  4 10:55 email.txt
-rw-r--r--   1 queiroz  supsof     13358 Sep 11 15:28 formmail.pl
drwxr-xr-x   2 queiroz  supsof       512 Apr 22  1996 lib
dr-xr-xr-x   2 queiroz  supsof       512 Sep 20  1997 links
```

O primeiro caracter indica o tipo de arquivo:

l	link
d	diretório
-	arquivo regular

Existem outros tipos de arquivos, que não iremos detalhar neste momento.

Entretanto, se a minha intenção é apenas saber o tipo de arquivo, a opção "-l" me fornece muito mais informação do que eu realmente preciso.

O comando

```
% ls -F
CUECMAG/      XF86Config.gz     lib/
DSC/          a                 links/
DicasL@       a2ps.tar.gz       mail/
EFR/          amanda/           mirror/
FOB/          bin/              packages/
FOTB/         conteudo.html@    recode-3.4/
JAVA/         dicas-l.tar.gz    src/
Mail/         dicas.pl          thankyou.html
Manaus/       dicasl.new.tar.gz tmp/
TMP/          email.txt         y/
```

me fornece a mesma informação, porém de forma muito mais sucinta. Os nomes de diretórios são seguidos por "/", os links por "@" e os arquivos regulares são apresentados normalmente.

Podemos também combinar as diretivas "-A" e "-F", para podermos também incluir os arquivos escondidos na listagem:

```
cshrc            .ssh/         dicas.pl
cshrc.220997     .subscribers  dicasl.new.tar.gz
desksetdefaults  .tt/          efr@
dt/              .twinrc*      efr01.zip
dtprofile*       .wastebasket/ email.txt
elm/             .xinitrc      formmail.pl
exrc             CCUECMAG/     links/
fm/              DSC/          mail/
forward          DicasL@       mirror/
gimprc           EFR/          packages/
```

Normalmente a saída do comando ls é formatada em múltiplas colunas. Todavia, se a saída for redirecionada para um outro comando, é exibida apenas uma coluna, o que pode ser inconveniente.

Podemos todavia forçar que, mesmo utilizando um pipe, a saída seja formatada em colunas:

```
% ls -C | more
```

Este comando possui ainda outro inconveniente. A saída gerada é disposta alfabeticamente, de cima para baixo. Ou seja, se estivermos analisando a saída na tela de um computador, podemos ter algo do tipo:

```
00index.txt        970903.html        980227.doc
970303.doc         970903.src         980227.html
970303.html        970903.txt         980227.src
970303.src         970904.doc         980227.txt
970303.txt         970904.html        980228.doc
970304.doc         970904.src         980228.html
970304.html        970904.txt         980228.src
970304.src         970905.doc         980228.txt
970304.txt         970905.html        980301.doc
970305.doc         970905.src         980301.html
970305.html        970905.txt         980301.src
970305.src         970906.doc         980301.txt
970305.txt         970906.html        980302.doc
970306.doc         970906.src         980302.html
970306.html        970906.txt         980302.src
970306.src         970907.doc         980302.txt
970306.txt         970907.html        980303.doc
970307.doc         970907.src         980303.html
970307.html        970907.txt         980303.src
970307.src         970908.doc         980303.txt
970307.txt         970908.html        980306.doc
970310.doc         970908.src         980306.html
 --More--
```

Ou seja, se estivermos analisando a saída na tela de um computador, podemos ter algo do tipo:

```
00index.txt        970903.html        980227.doc
970303.doc         970903.src         980227.html
970303.html        970903.txt         980227.src
970303.src         970904.doc         980227.txt
970303.txt         970904.html        980228.doc
970304.doc         970904.src         980228.html
970304.html        970904.txt         980228.src
970304.src         970905.doc         980228.txt
970304.txt         970905.html        980301.doc
970305.doc         970905.src         980301.html
970305.html        970905.txt         980301.src
970305.src         970906.doc         980301.txt
970305.txt         970906.html        980302.doc
970306.doc         970906.src         980302.html
970306.html        970906.txt         980302.src
970306.src         970907.doc         980302.txt
970306.txt         970907.html        980303.doc
970307.doc         970907.src         980303.html
970307.html        970907.txt         980303.src
970307.src         970908.doc         980303.txt
970307.txt         970908.html        980306.doc
970310.doc         970908.src         980306.html
 --More--
```

Veja o final da primeira coluna. O arquivo é 970310.doc. O próximo na lista deveria ser 970310.html. Na segunda coluna o primeiro arquivo é 970903.html. Ou seja, a saída funciona da seguinte forma:

```
|   ^
|   | e assim por diante
V---|
```

Este comportamento entretanto pode ser mudado. Podemos fazer com que a saída seja algo do tipo:

```
------->
<-------
------->
```

o que é mais conveniente para visualização na tela de um computador. O comando para tal é:

```
% ls -AFx | more
```

4. O Comando mv

Para mover comandos de um destino para outro, ou mesmo para renomear arquivos, utilizamos o comando `mv`. A sintaxe é bastante simples:

```
mv arquivo1 arquivo2
```

ou ainda

```
mv arquivo1 /tmp/arquivo2
```

O comando `mv` possui a diretiva `-i`, que invoca o modo interativo. Desta forma, se o arquivo destino já existir, o usuário é solicitado a tomar uma ação:

```
mv -i arquivo1.txt arquivo2.txt
mv: sobrescrever `arquivo2.txt'?
```

Como por padrão, sistemas derivados do Unix assumem que o usuário sabe o que está fazendo, se a diretiva `-i` não for empregada, o arquivo destino, se já existir, será sobrescrito silenciosamente, sem alertar o usuário.

5. O comando rename

O comando `rename` permite a alteração do nome de diversos arquivos, ao mesmo tempo. Vejamos alguns exemplos:

Com este comando é possível alterar facilmente a extensão de diversos arquivos:

```
rename .htm .html *.htm
```

O comando acima irá substituir todos os arquivos terminados em .htm por .html.

Mas não é só isto:

```
% ls
jose.txt joseovaldo.txt josenir.txt
% rename jose joao jose.???
% ls
joao.txt  josenir.txt  joseovaldo.txt
% rename jose ricardo *ovaldo.txt
% ls
joao.txt  josenir.txt  ricardoovaldo.txt
```

Para converter todos os caracteres em caixa alta dos arquivos para caixa baixa:

```
rename 'y/A-Z/a-z/' *
```

Para remover a extensão `.bak` de todos os diretórios de um arquivo:

```
rename 's/\.bak$//' *.bak
```

O comando `rename` possui a seguinte sintaxe:

```
rename [opções] perlexpr [ files ]
```

Fornecemos ao comando algumas diretivas sobre as ações a serem efetuadas. A sintaxe válida para isto é o formato aceito pelo comando perl, como `s/\.bak$//` e `y/A-Z/a-z/` dos exemplos acima. A última diretiva são os arquivos sobre os quais as ações serão aplicadas.

6. O comando cp

Para copiar arquivos, usamos o comando `cp`. Da mesma forma que o comando `mv`, a sintaxe é:

```
cp arquivo1.txt arquivo2.txt
```

A diretiva `-i` invoca o modo interativo, que solicita a intervenção do usuário quando necessário.

7. O comando rm

Para apagar arquivos, usamos o comando `rm`. Por padrão, o comando não questiona o usuário sobre suas ações. Por esta razão, é muito importante que se tome extremo cuidado ao usá-lo. O comando

```
rm -rf /
```

irá apagar todo o sistema operacional. Sem perguntas.

Da mesma forma que recomendado para os comandos `cp` e `mv`, recomenda-se que o comando `rm` seja usado no modo interativo, com a diretiva `-i`. Uma grande quantidade de sistemas operacionais GNU/Linux atuais já ativam esta diretiva por padrão.

8. Os comandos cat e tac

O comando `cat` é muitissimo utilizado para exibir o conteúdo de arquivos e também para diversas outras funções. O nome `cat` deriva de *catenate*, que significa *concatenar*, juntar.

Vejamos alguns exemplos:

```
cat arquivo1.txt arquivo2.txt arquivo3.txt > arquivo123.txt
```

Reúno, em um único arquivo, o conteúdo dos arquivos `arquivo1.txt`, `arquivo2.txt` e `arquivo3.txt`

Com o comando `cat`, podemos também numerar as linhas de um arquivo. O comando:

```
cat -n teste
```

irá exibir na tela todas as linhas do arquivo `teste` precedidas por uma numeração. Caso você queira salvar o arquivo com a numeração, basta redirecionar a saída do comando para um outro arquivo:

```
cat -n teste > teste01
```

Às vezes, criamos, acidentalmente ou não, arquivos com caracteres não imprimíveis. Com o comando `cat` podemos localizar estes arquivos.

Tomemos a seguinte seqüência de comandos:

```
% touch a  a
% ls
a  a
```

Temos no diretório, aparentemente, dois arquivos com o mesmo nome. O carácter <CTRL>-X não aparece na listagem, está invisível Mas como isto é possível? Neste caso é evidente que os dois arquivos não possuem o mesmo nome; um deles chama-se "a" e o outro chama-se <CTRL>-Xa. Mas como fazer para identificar o problema? O comando cat pode ser usado para isto:

```
% ls | cat -v
^Xa
a
```

A opção -v faz com que o comando cat exiba também os caracteres invisíveis.

Outra dica, às vezes a listagem de um diretório pode parecer um pouco estranha, como em:

```
% ls
    a  a
```

Porque a listagem não começa na primeira coluna? Vejamos porque:

```
% ls | cat -v
^A
^X
^Xa
a
```

Existem dois arquivos "invisíveis", ou seja, os seus nomes são compostos de caracteres não exibíveis (existe esta palavra?), no caso `^A` e `^Xa`. Da mesma forma que o comando "cat -v" foi utilizado neste caso, para listar diretórios, ele pode ser usado para listar aqueles arquivos onde tudo supostamente deveria estar funcionando mas não está. Neste caso o culpado pode ser um destes caracteres invisíveis em um local onde não deveria estar.

O comando `tac`, como podemos ver, é o inverso do comando `cat`, ou seja, ele exibe na tela o conteúdo de um arquivo, mas a posição de todas as linhas é revertida. A primeira linha se torna a última e assim por diante.

Vejamos um exemplo. No site www.dicas-l.com.br, existe uma listagem de todas as dicas já veiculadas. O mais interessante é que os leitores vejam as mensagens em ordem cronológica reversa, ou seja, as mensagens mais recentes devem aparecer primeiro. A seguir reproduzo um trecho do script de manutenção do site:

```
#-------------------------------------------------------------------------
# Montagem do Histórico
#-------------------------------------------------------------------------
echo "<LI><A HREF=`echo $TIP_FILE.php`>$title</A>" >> $IncDir/listing
tac $IncDir/listing > $IncDir/gnitsil.inc
```

Com o comando `echo`, colocamos ao final do arquivo `listing`, a dica do dia. Com o comando `tac`, fazemos a reversão da ordem das linhas do arquivo `listing` e geramos um novo arquivo, chamado `gnitsil.inc`, que possui o mesmo conteúdo, porém em ordem reversa.

9. O comando xargs

Por Júlio Neves

Existe um comando, cuja função primordial é construir listas de parâmetros e passá-la para a execução de outros programas ou instruções. Este comando é o xargs e deve ser usado da seguinte maneira:

```
xargs [comando [argumento inicial]]
```

Caso o comando, que pode ser inclusive um script Shell, seja omitido, será usado por default o echo.

O xargs combina o argumento inicial com os argumentos recebidos da entrada padrão, de forma a executar o comando especificado uma ou mais vezes.

Exemplo:

Vamos prodrurar em todos os arquivos abaixo de um determinado diretório uma cadeia de caracteres usando o comando find com a opção *type f* para pesquisar somente os arquivos normais, desprezando diretórios, arquivos especiais, arquivos de ligações, etc, e vamos torná-la mais genérica recebendo o nome do diretório inicial e a cadeia a ser pesquisada como parâmetros. Para isso fazemos:

```
$ cat grepr
#
# Grep recursivo
# Pesquisa a cadeia de caracteres definida em $2 a partir do diretorio $1
#
find $1 -type f -print|xargs grep -l "$2"
```

Na execução deste script procuramos, a partir do diretório definido na variável $1, todos os arquivos que continham a cadeia definida na variável $2.

Exatamente a mesma coisa poderia ser feito se a linha do programa fosse a seguinte:

```
find $1 -type f -exec grep -l "$2" {} \;
```

Este processo tem duas grandes desvantagens sobre o anterior:

1. A primeira é bastante visível: o tempo de execução deste método é muito superior ao daquele, isso porque o grep será feito em cada arquivo que lhe for passado pelo find, um-a-um, ao passo que com o xargs, será passada toda, ou na pior das hipóteses, a maior parte possível, da lista de arquivos gerada pelo find;
2. Dependendo da quantidade de arquivos encontrados que atendem ao find, poderemos ganhar aquela famosa e fatídica mensagem de erro "Too many arguments" indicando um estouro da pilha de execução do grep. Como foi dito no item anterior, se usarmos o xargs ele passará para o grep a maior quantidade de parâmetros possível, suficiente para não causar este erro, e caso necessário executará o grep mais de uma vez.

ATENÇÃO! Aê pessoal do linux que usa o ls colorido que nem porta de tinturaria: nos exemplos a seguir que envolvem esta instrução, você devem usar a opção --color=none, senão existem grandes chances dos resultados não ocorrerem como o esperado.

Vamos agora analisar um exemplo que é mais ou menos o inverso deste que acabamos de ver. Desta vez, vamos fazer um script para remover todos os arquivos do diretório corrente, pertencentes a um determinado usuário.

A primeira idéia que surge é, como no caso anterior, usar um comando find, da seguinte maneira:

```
$ find . -user cara -exec rm -f {} \;
```

Quase estaria certo, o problema é que desta forma você removeria não só os arquivos do cara no diretório corrente, mas também de todos os outros subdiretórios "pendurados" neste. Vejamos então como fazer:

```
$ ls -l | grep " cara " | cut -c55- | xargs rm
```

Desta forma, o grep selecionou os arquivos que continham a cadeia cara no diretório corrente listado pelo ls -l. O comando cut pegou somente o nome dos arquivos, passando-os para a remoção pelo rm usando o comando xargs como ponte.

O xargs é também uma excelente ferramenta de criação de one-liners (scripts de somente uma linha). Veja este para listar todos os donos de arquivos (inclusive seus links) "pendurados" no diretório /bin e seus subdiretórios.

```
$ find /bin -type f -follow | \
xargs ls -al | tr -s ' ' | cut -f3 -d' ' | sort -u
```

Muitas vezes o /bin é um link (se não me engano, no Solaris o é) e a opção -follows obriga o find a seguir o link. O comando xargs alimenta o *ls -al* e a seqüência de comandos seguinte é para pegar somente o 3º campo (dono) e classificá-lo devolvendo somente uma vez cada dono (opção -u do comando sort).

Você pode usar as opções do xargs para construir comandos extremamente poderosos. Para exemplificar isso e começar a entender as principais opções desta instrução, vamos supor que temos que remover todos as arquivos com extensão .txt sob o diretório corrente e apresentar os seus nomes na tela. Veja o que podemos fazer:

```
$ find . -type f -name "*.txt" | \
xargs -i bash -c "echo removendo {}; rm {}"
```

A opção -i do xargs troca pares de chaves ({}) pela cadeia que está recebendo pelo pipe (|). Então neste caso as chaves ({}) serão trocadas pelos nomes dos arquivos que satifaçam ao comando find.

Olha só a brincadeira que vamos fazer com o xargs:

```
$ ls | xargs echo > arq.ls
$ cat arq.ls
arq.ls arq1 arq2 arq3
$ cat arq.ls | xargs -n1
arq.ls
arq1
arq2
arq3
```

Quando mandamos a saída do ls para o arquivo usando o xargs, comprovamos o que foi dito anteriormente, isto é, o xargs manda tudo que é possível (o suficiente para não gerar um estouro de pilha) de uma só vez. Em seguida, usamos a opção -n 1 para listar um por vez. Só para dar certeza veja o exemplo a seguir, quando listaremos dois em cada linha:

```
$ cat arq.ls | xargs -n 2
arq.ls arq1
arq2 arq3
```

Mas a linha acima poderia (e deveria) ser escrita sem o uso de pipe (|), da seguinte forma:

```
$ xargs -n 2 < arq.ls
```

Outra opção legal do xargs é a -p, na qual o xargs pergunta se você realmente deseja executar o comando. Digamos que em um diretório você tenha arquivo com a extensão .bug e .ok, os .bug estão com problemas que após corrigidos são salvos como .ok. Dá uma olhadinha na listagem deste diretório:

```
$ ls dir
arq1.bug
arq1.ok
arq2.bug
arq2.ok
...
arq9.bug
arq9.ok
```

Para comparar os arquivos bons com os defeituosos, fazemos:

```
$ ls | xargs -p -n2 diff -c
diff -c arq1.bug arq1.ok ?...y
....
diff -c arq9.bug arq9.ok ?...y
```

Para finalizar, o xargs também tem a opção -t, onde vai mostrando as instruções que montou antes de executá-las. Gosto muito desta opção para ajudar a depurar o comando que foi montado.

Então podemos resumir o comando de acordo com a tabela a seguir:

Opção	Ação
-i	Substitui o par de chaves ({}) pelas cadeias recebidas
-nNum	Manda o máximo de parâmetros recebidos, até o máximo de Num para o comando a ser executado
-lNum	Manda o máximo de linhas recebidas, até o máximo de Num para o comando a ser executado
-p	Mostra a linha de comando montada e pergunta se deseja executá-la
-t	Mostra a linha de comando montada antes de executá-la

10. Os comandos head e tail

Os comandos `head` e `tail` são usados, respectivamente, para visualizar as primeiras e as últimas linhas de um arquivo. Ambos os comandos aceitam a diretiva `-n`, que é usada para alterar o seu comportamento padrão, que é exibir as dez primeiras ou as dez últimas linhas de um arquivo.

Vejamos alguns exemplos:

```
tail /etc/hosts

tail -n 1000 arquivo1.txt

head /etc/hosts

head -n 20 arquivox.txt
```

11. O comando mkdir

O comando `mkdir` é usado para criar um diretório:

```
mkdir /teste
```

Podemos usar o comando `mkdir` também para criar uma árvore completa de diretórios. Para isto, precisamos usar a diretiva "-p". O comando:

```
mkdir -p /home/users/cs/ano1/graduacao/jose
```

é equivalente aos comandos

```
mkdir /home
mkdir /home/users
mkdir /home/users/cs
mkdir /home/users/cs/ano1
mkdir /home/users/cs/ano1/graduacao
mkdir /home/users/cs/ano1/graduacao/jose
```

e bem menos trabalhoso :)

12. Permissões em UNIX/Linux

Por Daniel Duclos

O UNIX possui um sistema de permissões que define os direitos dos usuários e dos programas sobre todos os arquivos e diretórios. Essas permissões são controladas pelo comando chmod. Para este tutorial utilizaremos a sintaxe da versão GNU desse programa.

12.1. Sistema Básico de permissões

O sistema básico de permissões é dado em três níveis principais: dono do arquivo (user, em inglês), grupo de usuários ao qual pertence o arquivo (group) e o restante (others). Dos termos em inglês utilizaremos u para representar o dono do arquivo, g para grupo e o para os outros.

Cada nível pode ter, por sua vez, três tipos de permissões: leitura (read, representado pela letra r), escrita (write, w) e execução (execute, representado pela letra x). A nomenclatura ugo e rwx, derivadas dos nomes em inglês para os níveis e permissões é comumente usada na Internet e será adotada neste tutorial.

O sistema garante o acesso de leitura, gravação ou execução dependendo do valro de um bit associado ao arquivo. Cada bit pode ter o valor de 0 ou 1. Existem então 9 bits para representar as permissões, pois são três níveis (ugo) vezes três permissões possíveis (rwx).

Vamos observar as permissões de um arquivo de exemplo:

```
$ ls -l index.html
-rw-r--r--  1 daniduc webmasters 4632 2004-12-17 13:39 index.html
$
```

Repare no `-rw-r--r--`. Essas são as permissões do arquivo. As três primeiras são para o dono do arquivo (daniduc), as três seguintes representam os direitos do grupo (webmasters) e as três últimas dos demais usuários. Vamos destrinchá-las.

Analisemos agora as permissões do dono do arquivo, representadas por são rw-. O traço representa o bit inativo (valor zero) e a letra o bit ativo (valor 1). Então vemos que o dono tem permissão de ler e escrever no arquivo, mas não executá-lo. Veja um teste:

```
$ ./index.html
bash: ./index.html: Permissão negada
$
```

O sistema avisou que não há permissão para executar o arquivo, como esperado.

Assim, concluimos que o arquivo pode ser lido e escrito pelo seu dono e pelos membros do grupo que o possui, e os demais apenas podem lê-lo.

Agora veremos como alterar essas permnissões. Para isso, é preciso entender como se referir numericamente à essas permissões. A teoria é muito simples. Vamos montar uma tabela de valores possíveis para os bits de leitura, gravação e execução, que podem estar desligados (representados por 0) ou ligados (1)

```
rwx
000
001
010
011
100
101
110
111
```

Ora, podemos converter os valores binários em decimais, e então a tabela ficaria assim:

```
rwx - Valor octal
000 - 0
001 - 1
010 - 2
011 - 3
100 - 4
101 - 5
110 - 6
111 - 7
```

Assim fica fácil visualizar o valor numérico que representa nossa escolha de permissões. Se queremos que um arquivo tenha permissão de leitura, gravação mas não de escrita teremos o primeiro e o segundo bits ligados e o terceiro desligado, ou seja, 110, que em octal é 6. Agora basta aplicar o mesmo sistema para cada um dos níveis. Para representar as permissões do nosso arquivo de exemplo (rw-, rw-, r--) ficaria 110, 110, 001, ou seja 664.

Agora, suponha que eu queira dar a permissão de escrita para todo mundo no arquivo index.html. Bastaria dar o comando:

```
$ chmod 666 index.html
```

Veja como fica:

```
$ ls -l index.html
-rw-rw-rw-  1 daniduc webmasters 4632 2004-12-17 13:39 index.html
```

Para retirar a permissão de escrita de todos e do grupo fica:

```
$ chmod 644 index.html
$ ls -l index.html
-rw-r--r--  1 daniduc webmasters 4632 2004-12-17 13:39 index.html
```

12.2. Diretórios

Um diretório deve ter permissão de execução para que se possa entrar nele. Além disso, as permissões de um diretório tem precedência sobre as dos arquivos que ele contém. Veja um exemplo:

Primeiro temos um diretório pertencente ao usuário daniduc

```
$ ls -ld /dir/
drwxr-xr-x  2 daniduc daniduc 72 2005-04-19 03:14 /dir/
$
```

Agora o usuário daniduc cria um arquivo dentro de /dir e em seguida dá à ele permissões totais para todos os usuários:

```
daniduc $ cd /dir
daniduc $ touch teste
daniduc $ ls -l teste
-rw-r--r--  1 daniduc daniduc 0 2005-04-19 03:14 teste
daniduc $ chmod 777 teste
daniduc $ ls -l teste
-rwxrwxrwx  1 daniduc daniduc 0 2005-04-19 03:14 teste
```

Teoricamente, outro usuário pdoeria remover esse arquivo. Mas vamos ver que o usuário carla não consegue isso:

```
carla $ ls -l teste
-rwxrwxrwx  1 daniduc daniduc 0 2005-04-19 03:14 teste
carla $ rm -f teste
rm: imposível remover `teste': Permissão negada
carla $
```

Isso se dá por causa das permissões do diretório no qual o arquivo teste está contido:

```
drwxr-xr-x  2 daniduc daniduc 72 2005-04-19 03:14 /dir/
```

Vamos alterar as permissões do diretório para que todos tenham controle sobre ele, mas manter o arquivo teste exatamente igual:

```
daniduc $ chmod 777 /dir
daniduc $ -> ls -ld /dir
drwxrwxrwx  2 daniduc daniduc 72 2005-04-19 03:14 /dir/
daniduc $
```

Vamos ver agora se a carla consegue seu intento:

```
carla $ ls -l teste
-rwxrwxrwx  1 daniduc daniduc 0 2005-04-19 03:14 teste
carla $ rm -f teste
carla $ ls -l teste
ls: teste: Arquivo ou diretório não encontrado
carla $
```

12.3. Bits extras

12.3.1. Sticky Bit

Além dos bits básicos já vistos até agora, há ainda mais dois extras. O primeiro deles é o "sticky bit", associado à diretórios e representado pela letra t. Caso este bit esteja ativado (com valor 1), o diretório não pode ser removido, mesmo que com permissão 777. Além disso, os arquivos criados dentro desse diretório só podem ser apagados pelo seu dono. Isso é muito útil em diretórios temporários, como o tmp, onde todos podem escrever:

```
$ ls -ld /tmp/
drwxrwxrwt  9 root   root    45056 2005-04-19 04:04 /tmp/
$
```

Para ativar o sticky bit utiliza-se o valor 1 À frente das permissões já vistas:

```
daniduc $ chmod 1777 /dir/
daniduc $ ls -ld /biboca/
drwxrwxrwt  2 daniduc daniduc 48 2005-04-19 03:30 /dir/
daniduc $
```

12.3.2. SUID

Normalmente quando um arquivo é executado ele roda com os privilégios do usuário que o está executando. Mas há momentos em que isso precisa ser contornado. Por exemplo, para um usuário alterar sua senha ele chama o comando passwd. Esse comando precisa remover a senha antiga e inserir a senha nova no arquivo /etc/shadow. Porém, esse arquivo normalmente só pode ser alterado pelo root. Como fazer isso, se o passwd foi chamado pelo usuário comum?

A resposta é ativar o bit SUID. Com esse bit ativado o programa passa a rodar com as permissões do usuário dono do arquivo, e não mais de quem o invocou. O bit SUID é representado pela letra s logo após a área de permissões do usuário. Veja:

```
daniduc $ ls -l /usr/bin/passwd
-rwsr-xr-x  1 root root 26616 2004-11-02 19:51 /usr/bin/passwd
daniduc $
```

Para se ativar o bit SUID com o comando chmod utiliza-se o valor 4 á frente das permissões básicas (ugo):

```
daniduc $ touch teste
daniduc $ ls -l teste
-rw-r--r--  1 daniduc daniduc 0 2005-04-19 03:50 teste
daniduc $ chmod 4755 teste
daniduc $ ls -l teste
-rwsr-xr-x  1 daniduc daniduc 0 2005-04-19 03:50 teste
daniduc $
```

12.3.3. GUID

Similarmente ao BIT SUID, o GID faz com que o arquivo seja executado com os privilégios do grupo ao qual pertence e não do usuário que o executa. O bit GID é representado pela letra s logo após o conjunto de permissões do grupo. Para ativar o GID, utilize o 2 no comando chmod:

```
daniduc $ touch teste
daniduc $ ls -l teste
-rw-r--r--  1 daniduc daniduc 0 2005-04-19 04:06 teste
daniduc $ chmod 2755 teste
daniduc $ ls -l teste
-rwxr-sr-x  1 daniduc daniduc 0 2005-04-19 04:06 teste
daniduc $
```

ATENÇÃO: Não é aconselhável, por questões de segurança, ativar os bits SUID e GID em novos arquivos, especialmente se eles pertecerem ao root.

12.4. MÁSCARA

Quando criamos um arquivo ele é criado com permissões totais, ou seja, 666. Note que eles jamais são criados com permissão de execução. Isso em geral não é seguro, claro. Esse modo de criação pode ser alterado com o comando umask. Similarmente ao chmod, no qual se especifica quais bits devem ser desligados, o comando umask determina quais bits devem ser *desligados*.

Se você quer que os arquivos sejam criados sem permissão de escrita para o grupo e para os outros, então o comando ficaria assim:

```
umask 022
```

Nenhum bit foi desligado para o dono, e o bit de escrita (010, ou 2 em octal) desligado para grupo e outros.

Esse comando está em geral incluido no arquivo /etc/profile ou arquivo equivalente que determina o ambiente para todos os usuários.

13. awk

Um utilitário bastante útil para manipulação de strings é o awk. O nome (bastante estranho por sinal) é derivado das iniciais de seus três criadores, Aho, Kernighan, e Weinberger. Funciona com pipes ou diretamente com arquivos.

Por exemplo, suponhamos que queiramos fazer alguma formatação em cima da saída do comando ls:

```
% ls -l /tmp
total 184
srwxrwxrwx   1 queiroz  supsof     0 May  5 18:12 FvConSocke
-rw-r--r--   1 root   system     193 Apr 29 14:00 SM_OPO13zqd
-rw-r--r--   1 root   system     220 Apr 25 16:31 XX
-rw-r--r--   1 root   system     949 Apr 25 15:28 a
-rw-rw-rw-   1 root   system     0 Apr 25 19:12 errdemon.1708
  ....
```

Se não estivermos interessados em todos estes campos, podemos fazer uma seleção com o programa awk:

```
% ls -l /tmp | awk '{print $9}'

FvConSocke
SM_OPO13zqd
XX
  ....
```

Se quisermos fazer uma listagem dos usuários de uma máquina em ordem alfabética podemos utilizar o arquivo /etc/passwd. A diferença é que o arquivo /etc/passwd possui o caracter ":" como delimitador de seus campos. Para especificar o caracter delimitador utilizamos a diretiva "F:", como exemplificado abaixo:

```
% awk -F: '{print $1}' /etc/passwd | sort > list.txt
```

A saída do comando awk é redirecionada para o comando sort que faz a ordenação e o resultado é gravado no arquivo list.txt.

Ilustraremos a seguir mais alguns exemplos interessantes do comando `awk`.

Para se imprimir apenas o último campo de um arquivo com o comando awk, podemos utilizar o comando:

```
awk '{print $NF}' arquivo.exemplo
```

A variável NF significa número de campos. Quando precedida por "$" indica o último campo, à semelhança de $1, $2, etc.

Se quisermos imprimir a contagem do número de campos de um arquivo:

```
awk '{print NF}' arquivo.exemplo
```

Se quisermos imprimir apenas as linhas que contenham mais de dez campos:

```
awk -F: 'NF > 10 {print}' arquivo.exemplo
```

Ou, se quisermos imprimir apenas as linhas que possuam exatamente 10 campos:

```
awk -F: 'NF == 11 {print}' arquivo.exemplo
```

Se quisermos imprimir apenas o segundo campo de registros que contenham a palavra teste:

```
awk '/teste/ {print $2}' arquivo.exemplo
```

O pacote gawk (de Gnu AWK), mantido e distribuído pela Free Software Foundation, traz também um livro excelente sobre o uso do comando awk, com muitos exemplos.

Além deste manual, existe também um cartão de referência.

A documentação é distribuída separadamente do código fonte do programa gawk. O livro e o cartão de referência encontram-se em ftp://ftp.gnu.org/gnu/gawk/gawk-3.0.3-doc.tar.gz. Caso este link não seja mais válido quando você ler este documento, por favor nos avise e procure no Google, que é bem fácil de achar.

Vale a pena ler. Primeiramente porque o material é excelente e segundo porque não custa nada ;-)

14. expr

Por Alexandre de Abreu

O comando `expr` é muito conhecido pelos programadores shell(bash, ksh, sh, etc.), mas, a sua utilização, na maioria das vezes, restringe-se ao comando abaixo:

```
contador=`expr $contador + 1`
```

Este comando irá incrementar o valor da variável contador em 1. Outras formas de alcançar o mesmo objetivo em Bash seriam:

```
let contador++
contador=$((contador+1))
contador=$(bc <<< "$contador + 1")
```

É verdade que existem várias outras maneiras de fazer, mas o foco deste documento é apresentar as funcionalidades do comando `expr`, um poderoso processador de expressões.

Esse comando pode ser utilizado para processar expresões matemáticas através de operadores lógicos como `|`, `>`, `>=`, `&`, que representam o `OU`, `E`, `MAIOR QUE`, etc., assim como também pode utilizar operadores aritméticos como visto no exemplo acima com o operador de adição ou `+`. Outras possibilidades são `%` ou módulo, `*` ou multiplicação.

Abaixo veremos exemplos de expressões aritméticas utilizando alguns operadores assim como os parênteses para agrupar e definir a ordem de execução dos comandos:

```
$ expr \( 30 + 2 \) \* \( 13 % 2 \) - \( 2 \* \( 20 - 8 \) \)
8
$ expr \( 30 + 2 \) \* \( 13 % 2 \) - \( 2 \* \( 20 - 8 \) \) / 2
20
$ expr \( \( 30 + 2 \) \* \( 13 % 2 \) - \( 2 \* \( 20 - 8 \) \) \) / 2
4
```

Todos sabemos que o domínio do tópico `Expressões Regulares(RegEx)` é de grande valia e utilidade para qualquer programador, principalmente aqueles que utilizam linguagens scripting como Perl, PHP, Shell, Python, Ruby, utilizadas na manipulação de Strings.

O `expr` possui suporte à RegEx, assim como o comando `grep`. Validar expressões torna-se um trabalho viável mesmo sem o `GNU grep/egrep`, ique nem sempre estão disponíveis em algumas versões/sabores do UNIX.

A sintaxe para o processamento de uma String ou validação contra um padrão(pattern) utilizando expressões regulares através do comando `expr` pode ser feita de duas maneiras:

```
expr   STRING : REGEXP

expr   match STRING REGEXP
```

Adotaremos aqui a primeira sintaxe mostrada acima. Para tentarmos entender como funcionam as

RegEx juntamente com o comando `expr`, nada melhor que utilizar exemplos. A seguir incluimos alguns comandos com pequenos comentáros, alguns deles acompanhandos do comando equivalente utilizando o `GNU grep`:

1. Cadeias de caracteres ou Strings que começam por `D` precedidas ou não de um ou mais espaços ou caracteres de tabulação(TAB)

   ```
   $ expr " Dicas" : '[[:blank:]]*D'
   2

   $ expr "Dicas" : '[[:blank:]]*D'
   1

   $ expr "   Dicas" : '[[:blank:]]*D'
   4
   ```

   Primeiramente, deve-se deixar claro que, como na maioria dos casos no mundo UNIX, a String é tratada de modo "Case Sensitive, `ou seja ``D` é diferente de `d`.

   O caracter `^`, que geralmente determina o início de uma cadeia de caracteres é implícito, ou seja, o `expr`, por si só, já entende que o padrão acima descrito, utilizando a RegEx '[[:blank:]]*D', é equivalente ao utilizado através do comando `grep`:

   ```
   $ echo "   Dicas" | grep "^[[:blank:]]*D"
     Dicas
   ```

   O comando acima seria o equivalente ao último comando `expr` mostrado, veja que ele utiliza o caractere `^` para determinar o início da linha ou da cadeia de caracteres.

   Voltando ao comando: `expr " Dicas" : '[[:blank:]]*D'`

   Verificamos que a saída padrão é igual a 4, pois, de acordo com a String que foi testada contra o padrão em RegEx acima existem 4 caracteres que estão de acordo com a cadeia `" Dicas"`: 3 espaços e o caracter `D`.

   O `expr` imprime na saída padrão o número de caracteres que estão de acordo com o padrão testado, no último caso, 4. Veremos logo abaixo como imprimir os caracteres que estão de acordo com o padrão especificado na linha de comando, o que é bem mais interessante e útil.

   Vejam que a String `" Dicas"` está entre aspas duplas e não entre aspas simples. É interessante sempre adotar este procedimento, pois, ao utilizar variáveis, o valor da variável é utilizado durante a expressão:

   ```
   $ STR="   Dicas"
   $ expr "$STR" : '[[:blank:]]*D'
   4
   ```

2. Como validar uma expressão?

   Qual seria o comando `expr` para validar uma String que pode ser resultado de uma entrada de usuário utilizando o `read`, uma linha de um arquivo ou saída padrão de um comando?

Utilizando `grep`, teremos:

```
$ STR="localhost.localdomain"
$ echo "$STR" | grep -q "localhost" && echo OK
OK

if grep -q "localhost" /etc/hosts; then
 echo "Existe"
else
 echo "Não existe"
fi
```

Estes exemplos são bem simples, o parâmetro `-q` do comando `grep` suprime a saída padrão. O Bash ainda possibilita a utiliação da forma abaixo para `Pattern Matching`:

```
$ STR="localhost.localdomain"
$ [[ $STR = l* ]] && echo OK
OK
```

Mas, ainda assim não será suficiente para alguns casos, como veremos a seguir. Um exemplo mais interessante seria a validação de um nome de diretório/arquivo ou uma String em relação ao padrão resultante do comando `date` abaixo:

```
$ date '+%d%m%y-%T'
260405-11:52:52
```

O expr pode ser empregado nesta situação. Abaixo veremos que a representação `[[:digit:]]` equivale aos números de 0 a 9 ou `[0-9]`. Vejamos como seria o comando correto para validação:

```
$ STR=``date '+%d%m%y-%T'``
$ expr "$STR" : '[[:digit:]]\{6\}-[[:digit:]]\{2\}:[[:digit:]]\{2\}:[[:digit:]]\{2\}'
15
$ echo $?
0
```

Como vimos, o comando `expr` acima retorna 0, ou seja, quando há um `matching` ele retorna `true` ou verdadeiro: variável $? igual a 0. O valor de retorno pode ser armazenado em uma variável e posteriormente verificado através dos comandos abaixo:

```
STR="Dicas-Linux eh legal"

expr "$STR" : '.*Linux.*' > /dev/null

if [ $? -eq 0 ]; then
 echo "Encontrado"
else
 echo "Nada encontrado"
fi
```

O padrão acima corresponde a qualquer cadeia que contenha a palavra `Linux`. O caractere `.` equivale a qualquer caractere. O retorno será verdadeiro, logo, será mostrado na tela a palavra `Encontrado`.

Como retornar a cadeia que está de acordo com o padrão especificado? Resposta: Utilizando parênteses. Vamos a um exemplo simples, mostrando o comando utilizado no começo deste documento:

```
$ expr "   Dicas" : '\([[:blank:]]*D\)'
  D
```

Este comando retorna os caracteres da String que estão de acordo com o padrão especificado através da RegEx e demarcados pelos parênteses. Note que os parênteses devem ser `escapados` com contra-barras para que não sejam entendidos como um caractere literal `(` ou `)`.

3. Como retirar da String abaixo o número de identificação do processo(PID)?

```
# tail -1 /var/log/secure
Apr 26 09:27:01 localhost sshd[2549]: error: Address already in use.
                ^^^^
```

Uma RegEx válida para esta situação seria:

```
'[[:upper:]][[:alpha:]]\{2\} [[:digit:]]\{2\} [:[:digit:]]\{8\} [[:alnum:]]\{1,\} [[:alnu
```

Note que é possível especificar o número de ocorrências para cada representação(dígitos, alfa-numéricos, etc.) indicando este número entre chaves com contra-barras: `\{2\}`, `\{1,\}`. Este último quer dizer `um ou mais`.

Ao executar o comando abaixo vimos que ele retorna o número de caracteres:

```
$ expr "Apr 26 09:27:01 localhost sshd[2549]: error: Address already in use." : '[[:upper
38
```

Mas, se adicionarmos os parênteses com contra-barras na sub-cadeia que desejamos obter (PID), teremos:

```
$ expr "Apr 26 09:27:01 localhost sshd[2549]: error: Address already in use." : '[[:upper
2549
```

Este documento teve por finalidade mostrar uma das funcionalidades do comando `expr` com relação a processamento de Strings. Esta ferramenta faz parte do pacote `coreutils` ou `fileutils` dependendo da distribuição Linux. Faz parte também da maioria dos sistemas operacionais UNIX (testei em AIX, HP-UX e SunOS/Solaris).

Lógico que, para tirar o máximo desta funcionalidade é necessário um bom conhecimento sobre Expressões Regulares. Para quem ainda não tem tanto conhecimento neste tópico, a melhor referência é o livro Expressões Regulares - Uma Abordagem Divertida, de autoria de Aurélio Marinho

Jargas.

15. O comando wc

O comando `wc` conta o número de linhas, palavras e bytes em arquivos

```
wc [opções] [arq1] [arq2] ? [arqN]
```

Por padrão, todas as informações (bytes, linhas e palavras) são exibidas

-c	imprime apenas a contagem de bytes
-w	imprime apenas a contagem de palavras
-l	imprime apenas a contagem de linhas

16. O comando sort

O comando sort, na sua forma mais simples, serve para ordenar o conteúdo de um arquivo. Tomemos o arquivo: ***arq1***

```
x
a
h
j
k
```

O comando abaixo, executado sobre o arquivo arq1, irá gerar a saída exibida abaixo:

```
% sort arq1
a
h
j
k
x
```

Além desta função, o comando sort também pode ser utilizado para combinar dois arquivos diferentes. Os arquivos sobre os quais o comando sort irá atuar já devem ter sido previamente ordenados: ***arq1***

```
aa
yy
```

arq2

```
bb
zz
```

O comando

```
% sort -m arq1 arq2
```

irá exibir na tela

```
aa
bb
yy
zz
```

A saída do comando sort, em todos os exemplos apresentados, tem sido redirecionada para a tela. Caso queiramos redirecionar esta saída para um arquivo para processamento posterior, temos duas opções equivalentes:

```
% sort arq1 arq2 > arq3
```

ou

```
% sort arq1 arq2 -o arq3
```

O comando sort, também oferece inúmeras facilidades interessantes. Tomemos o arquivo abaixo como exemplo:

arq1

```
1:2:3:4:5:6
1:1:3:4:5:6
1:4:3:4:5:6
1:2:3:4:5:6
1:0:3:4:5:6
1:2:3:4:5:6
1:7:3:4:5:6
1:2:3:4:5:6
1:0:3:4:5:6
1:9:3:4:5:6
```

O comando abaixo

```
% sort -t: +1 -n arq1
```

irá gerar a seguinte saída

```
  |
  v
1:0:3:4:5:6
1:0:3:4:5:6
1:1:3:4:5:6
1:2:3:4:5:6
1:2:3:4:5:6
1:2:3:4:5:6
1:2:3:4:5:6
1:4:3:4:5:6
1:7:3:4:5:6
```

Observar que o segundo campo, indicado pela seta, está ordenado numericamente em ordem crescente. Os campos deste arquivo são separados por ":". O tipo de separador é indicado pela diretiva "-t:". Em seguida à diretiva "-t" poderíamos indicar qualquer tipo de separador. O campo a ser ordenado é indicado pela diretiva "+1". Para o comando sort a contagem dos campos inicia-se por 0, desta forma, o valor "+1" irá indicar na realidade o segundo campo do arquivo. A ordenação também pode ser feita numericamente, do maior para o menor valor:

```
% sort -t: +1 -nr arq1

  |
  v
1:9:3:4:5:6
1:7:3:4:5:6
1:4:3:4:5:6
1:2:3:4:5:6
1:2:3:4:5:6
1:2:3:4:5:6
1:2:3:4:5:6
1:1:3:4:5:6
1:0:3:4:5:6
1:0:3:4:5:6
```

Uma outra característica interessante do comando sort é a possibilidade de fazer as comparações sobre os argumentos convertidos para minúsculas (diretiva -f). Tomemos os arquivos `arq1` e `arq2`:

arq1

```
AA
XX
```

arq2

```
bb
kk
```

O comando `sort` abaixo

```
% sort arq1 arq2
AA
XX
bb
kk
```

irá gerar uma saída onde a ordenação será feita primeiramente sobre as letras maiúsculas e em seguida as minúsculas, ou seja, A-Z e em seguida a-z. Já o comando abaixo

```
% sort -f arq1 arq2
AA
bb
kk
XX
```

irá realizar a ordenação dos arquivos independentemente das palavras estarem grafadas em maiúsculas ou minúsculas.

O comando `sort` pode também ser utilizado para ordenar arquivos removendo eventuais linhas duplicadas. Tomemos o arquivo `arq1`:

arq1

```
joao
maria
jose
maria
joao
heitor
```

O comando

```
% sort -u arq1
```

irá gerar a saída abaixo

```
heitor
joao
jose
maria
```

A diretiva "-u" fez com que a saída gerada contivesse apenas uma ocorrência de cada uma das linhas.

17. O comando cut

O comando `cut` serve para extrair (cortar) partes selecionadas de uma sequencia de caracteres. Tomando como exemplo um arquivo, que contenha a linha:

```
Cantinho do Shell|Rubens Queiroz de Almeida|http://www.Dicas-L.com.br/cantinhodoshell/|cs
```

Temos acima um registro, contendo quatro campos, separados pelo caractere "|".

Imprimir o segundo campo:

```
cut -d "|" -f 2 arquivo.txt
Rubens Queiroz de Almeida
```

Imprimir apenas os caracteres nas posições 10 **e** 20:

```
cut -c 10,20 arquivo.txt
du
```

Imprimir os caracteres nas posições de 10 **a** 20:

```
cut -c 10-20 arquivo.txt
do Shell|Ru
```

Nos dois exemplos anteriores, observar bem o uso dos caracteres "," e "-". A vírgula indica um conjunto de caracteres e o hifen identifica uma faixa de caracteres.

18. O comando tar

18.1. Introdução

O comando tar (*tape archive*), é utilizado para arquivar diversos arquivos em um só. As suas diretivas principais são:

-c, --create	Criar um arquivo
-C, --directory	mudar para o diretório
-f, --file	arquivo a ser criado
-t, --list	listar os conteúdos de um arquivo
-v, --verbose	exibir todas as ações realizadas
-x, --extract	extrair arquivos
-z, --gzip	compactar ou descompactar o arquivo
-T, --files-from	obter a lista dos arquivos que farão parte do arquivo a partir do arquivo especificado
-X, --exclude-from=FILE	lista de arquivos a serem excluídos

Vejamos alguns exemplos:

1. Criar um arquivo backup do diretório `/etc`:

   ```
   tar cvzf /tmp/etc.tar.gz /etc
   ```

2. Descompactar o arquivo criado no diretório `/root`:

   ```
   tar xvzf /tmp/etc.tar.gz -C /root
   ```

3. Listar o conteúdo do arquivo /tmp/etc.tar.gz

   ```
   tar tvzf /tmp/etc.tar.gz
   ```

4. Criar um backup de todos os arquivo do diretório `/home` que tenham sido modificados nos últimos sete dias:

   ```
   find . -mtime -7 | tar cvzf /tmp/backup.tar.gz -T -
   ```

   Observe que, neste exemplo, a lista dos arquivos é obtida através do comando `find` e alimentada, através do pipe, para o comando `tar`.

   O comando `tar` possui um número muito maior de opções e uma leitura atenta de sua documentação certamente lhe será muito útil.

18.2. Backups com GNU/TAR

O GNU Tar (Tape Archive), comum em sistemas livres como FreeBSD e Linux, nos oferece a facilidade de se criar um backup de arquivos selecionados>

```
# find . -name \*.pm | tar cvzf backup.pm.tar.gz -T -
```

O comando find irá encontrar os arquivos terminados em .pm (perl modules) e o comando tar receberá esta lista por meio do pipe (caractere "|"), indicada pela flag "-T". O sinal "-" significa "standard output", ou o que veio pelo pipe, que foi gerado pelo comando find.

O arquivo criado, backup.pm.tar.gz irá conter apenas arquivos terminados em .pm.

18.3. Fatiamento de arquivos para transferência em links de baixa qualidade

Autor: Roberto Baronas

A transferência de grande quantidade de dados entre servidores UNIX distantes, servidos por links de telecomunicação de baixa qualidade às vezes é um problema que nos causa grandes dissabores, principalmente devido às frequentes interrupções que possam ocorrer.

No meu caso, uma solução caseira que resolveu em definitivo o problema de transferência de quatro arquivos de +-600Mb entre o Brasil e a Bolívia, num link de baixa velocidade foi a seguinte:

Usei o comando split conforme mostrado abaixo, para fatiar os arquivos de 600Mb em aproximadamente oito arquivos de 80Mb cada:

```
split -b 80m o92dk1.tar dk1
```

Esse comando, pegou o arquivo o92dk1.tar com aproximadamente 600Mb de tamanho e o quebrou em fatias com o tamanho máximo de 80Mb, no caso, oito fatias, prefixadas pelo string dk1.

O resultado do que ocorreu pode ser visto abaixo:

```
-rw-r--r-- 1 oracle dba 649144320 Feb 3 15:17 o92dk1.tar
-rw-r--r-- 1 root sys 83886080 Feb 15 16:52 dk1aa
-rw-r--r-- 1 root sys 83886080 Feb 15 16:52 dk1ab
-rw-r--r-- 1 root sys 83886080 Feb 15 16:52 dk1ac
-rw-r--r-- 1 root sys 83886080 Feb 15 16:52 dk1ad
-rw-r--r-- 1 root sys 83886080 Feb 15 16:52 dk1ae
-rw-r--r-- 1 root sys 83886080 Feb 15 16:53 dk1af
-rw-r--r-- 1 root sys 83886080 Feb 15 16:53 dk1ag
-rw-r--r-- 1 root sys 61941760 Feb 15 16:53 dk1ah
```

A restauração pôde ser feita via comando cat, da seguinte forma:

```
cat dk1aa dk1ab dk1ac dk1ad dk1ae dk1aaf dk1ag dk1ah > novo_arquivo
```

Assim, pudemos, transmitir as fatias, e em caso de queda de link, perderiamos somente a fatia que estava sendo transmitido no momento da queda. As fatias já transferidas estavam salvas.

Para testar a integridade dos arquivos, bastou utilizar o comando cksum no arquivo original o92dk1.tar e no arquivo de destino arquivo_novo. Os números do resultado devem ser os mesmos para ambos os arquivos. Abaixo temos um exemplo:

```
cksum o92dk1.tar
```

```
97395000 649144320 o92dk1.tar

cksum arquivo_novo
97395000 649144320 arquivo_novo
```

O valor verificador encontrado foi o número 97395000 para os dois arquivos, o que provou a integridade dos mesmos.

Finalmente, temos um exemplo do script que foi utilizado no servidor de destino para efetuar a transferência dos arquivos:

```
ftp -n -i -v arcturus.ho.u2236.unilever.com << eod >>zz_ftp.log 2>&1
user usuario senha
pwd lcd /diretorio_de_origem
cd /diretorio_de_Destino
get dk1aa
get dk1ab
get dk1ac
get dk1ad
get dk1ae
get dk1af
get dk1ag
get dk1ah
eod
```

Sistema Operacional utilizado: HP-UX 11i ou 11.11

18.4. GNU/Linux - Backup de Arquivos Pessoais

Para fazer um backup de todos os arquivos pertencentes a um determinado usuário, podemos conjugar os comandos **tar** e o comando **find**, como abaixo:

```
$ tar  -cvf archive.tar ``find  . -user queiroz -print``
```

Compactando:

```
$ tar  -cvzf archive.tar.gz ``find  . -user queiroz -print``
```

Uma aplicação interessante é executar este comando antes de remover um usuário do sistema de forma a se manter um backup de seus dados.

1. Introdução

Nesta semana damos continuidade ao estudo dos comandos mais comuns utilizados na programação shell. Como na semana anterior, os comandos são explicados com exemplos. Estes exemplos envolvem, na maior parte das vezes, outros comandos. Caso precise de mais informações sobre algum comando, para entender os exemplos, consulte a pagina de documentação, com o comando `man comando`.

Para melhor fixação dos conceitos, reproduza em seu computador todos os comandos apresentados juntamente com os exemplos de aplicação.

2. O comando date

Exibe a data do sistema. Pode ser configurado de diversas formas, para exibir apenas partes da informação de data.

Para ilustrar uma das possibilidades de uso do comando `date`, tomemos o script a seguir, que faz o backup com o comando rsync, de um servidor remoto. Toda a atividade é registrada em um arquivo de log, guardado em `/root/logs`. O arquivo de log do dia 4 de março de 2008, será gravado com o nome `ccuec.20080304.log`.

```
#!/bin/sh

cd /work/backup/www.ccuec.unicamp.br

rsync -avz -e ssh                  \
    --delete                       \
    --exclude "/proc"              \
    --exclude "/var/spool/mail"    \
    --exclude "/var/spool/postfix" \
    --exclude "/tmp"         \
    www.ccuec.unicamp.br:/ . > /root/logs/ccuec.`date +%Y%m%d`.log
```

Para conhecer em mais profundidade as funcionalidades do comando `date`, podemos ler o tutorial de autoria de Fabiano Caixeta Duarte, que reproduzimos a seguir.

Ao escrever scripts para tratar logs ou fazer estatísticas, é comum termos que lidar com datas em diversos formatos. Ocorre que nem sempre a data se encontra no formato desejado. Para isto, podemos nos valer de um utilitário chamado date.

O comando date tem como funções principais exibir e alterar a data do sistema. Mas com ele podemos fazer mais do que isto: podemos converter o formato de exibição da data atual ou de uma data qualquer especificada.

Entretanto, este utilitário tem diferentes versões, escritas por autores diferentes. O Linux adotou a versão escrita por David Mackenzie e o FreeBSD utiliza uma versão originária do Unix da AT&T. As diferentes versões possuem sintaxes divergentes. Vamos aqui abordar ambas.

Nem tudo é diferente :)

A simples execução do comando `date`, retorna a data atual no seguinte formato:

```
$ date
Sat Sep 17 18:00:00 BRT 2005
```

Uma vez que ambos os fontes do comando `date` utilizam a função `strftime` (`time.h`), os caracteres utilizados para formatação da data são os mesmos.

Veja alguns exemplos:

%d	Dia
%m	Mês em representação numérica

%Y	Ano representado em quatro dígitos
%F	Equivalente a %Y-%m-%d (formato frequentemente utilizado para inserção em bancos de dados)
%A	Dia da semana por extenso

O sinal de + indica que utilizaremos os caracteres de formatação. Para exibirmos o dia da semana, por extenso, da data atual, façamos:

```
$ date +%A
Saturday
```

Importante ressaltar que o comando date respeita a variável de ambiente LANG. Isto significa dizer que:

```
$ echo $LANG
en_US.iso8859-1
$ date +%A
Saturday
$ LANG=pt_BR.iso8859-1
$ date +%A
sábado
```

Passeando no tempo

Podemos precisar de provocar um deslocamento para o passado ou futuro na data a ser formatada.

No linux

Para informar ao comando date o deslocamento de uma data, utilzamos o parâmetro -d ou --date. Podemos especificar o deslocamento em dias(day), semanas(week), meses(month) ou anos(year).

Seguem alguns exemplos:

```
$ date -d "1 month"
Mon Oct 17 18:00:00 BRT 2005
$ date -d "2 week"
Sat Oct  1 18:00:00 BRT 2005
```

Para "voltarmos no tempo" utilizamos a variante `ago`.

```
$ date -d "3 year ago"
Tue Sep 17 18:00:00 BRT 2002
```

No FreeBSD

O deslocamento é feito a partir do parâmetro -v, com a seguinte sintaxe:

```
[sinal][número][unidade_de_tempo]
```

- O sinal indica se o deslocamento é negativo (passado) ou positivo (futuro)
- O número indica a quantidade de unidades de tempo a serem deslocadas
- A unidade de tempo é indicada por um dos caracteres (y, m, w, d, H, M, S), ou seja, (ano, mês, semana, dia, hora, minuto, segundo)

Os mesmos exemplos, conforme a sintaxe a ser utilizada no FreeBSD:

```
$ date -v+1m
Mon Oct 17 18:00:00 BRT 2005
$ date -v+2w
Sat Oct  1 18:00:00 BRT 2005
$ date -v-3y
Tue Sep 17 18:00:00 BRT 2002
```

Datas específicas

Nos casos em que vamos tratar informações oriundas de logs, as datas podem não estar no formato necessário ao nosso propósito.

Vamos supor que iremos alimentar um banco de dados com as informações contidas em um log com o seguinte formato:

```
Sep 17 18:26:14 servidor sm-mta[87357]: j8HMQ9tJ087356: stat=Sent
```

Precisamos desta data no formato 2005-09-17.

No Linux

Novamente o parâmetro -d nos será útil, mas de uma maneira diferente. Desta vez, passaremos a data extraída do log da seguinte maneira:

```
date -d "Sep 17" +%F
```

Observe alguns exemplos dos formatos aceitos:

- "17 Sep"
- "Sep 17"
- "17 September"
- "17 Sep 2005"

Importante ressaltar que os espaços em branco podem ser substituídos por um - ou até serem suprimidos. Outros caracteres não podem ser utilizados como separadores.

No FreeBSD

Aqui vemos uma maior versatilidade no padrão de entrada, com o uso de um parâmetro com dois argumentos, para determinar a data a ser formatada.

O parâmetro -f aceita como argumentos o formato de entrada e a data alvo. No caso apresentado no nosso log, o comando ficaria assim:

```
date -f "%b %d" "Sep 25" +%F
```

Assim, podemos usar qualquer formato de entrada para a conversão, basta que o especifiquemos corretamente.

O log de acesso do squid (ferramenta de proxy e cache), por exemplo, contém as datas em timestamps (segundos desde 01/01/1970). O comando a seguir converte este timestamp para algo humanamente compreensível:

```
$ date -f %s 1126992004 +%d/%m/%Y
17/09/2005
```

Referências

- man date
- man strftime
- Datas Presentes e Futuras

3. O comando df

Relata o espaço de disco usado pelo sistema de arquivo. A forma de uso mais comum envolve o uso da diretiva `-h`, que converte as unidades de alocação de espaço em um formato mais legível para seres humanos.

```
% df -h
Sist. Arq.      Tam   Usad Disp  Uso% Montado em
/dev/hda1       9,4G  5,8G  3,6G  63% /
tmpfs           379M  8,0K  379M   1% /lib/init/rw
udev             10M  104K  9,9M   2% /dev
tmpfs           379M     0  379M   0% /dev/shm
/dev/hda3        28G  9,5G   19G  35% /work1
/dev/hdb2       218G  146G   73G  67% /work
/dev/hdd1       115G   91G   24G  80% /home
```

Sem a diretiva `-h`:

```
Sist. Arq.       1K-blocos    Usad Dispon.   Uso% Montado em
/dev/hda1         9767184   6057532   3709652  63% /
tmpfs              387424         8    387416   1% /lib/init/rw
udev               10240       104     10136   2% /dev
tmpfs              387424         0    387424   0% /dev/shm
/dev/hda3        29060676   9946512  19114164  35% /work1
/dev/hdb2       228124040 152103016  76021024  67% /work
/dev/hdd1       120050028  95225368  24824660  80% /home
```

A unidade usada é blocos de 1k, o que torna a compreensão das informações um pouco mais difícil.

4. O comando dd

De uma forma simplificada, o comando `dd` faz a conversão e cópia de arquivos. Entretanto, o comando `dd` possui várias outras funções interessantes além da cópia pura e simples de arquivos. Um função que julgo bastante útil é a conversão de caracteres.

Por exemplo, para se converter todos as letras maiúsculas de um documento para letras minúsculas, executar o comando abaixo:

```
dd if=arquivo1 of=arquivo2 conv=lcase
```

Este comando irá converter todos as letras maiúsculas do arquivo1 em letras minúsculas e gerar um outro arquivo chamado arquivo2 com o resultado do processamento.

Da mesma forma, se quisermos converter todas as letras do arquivo2 para maiúsculas:

```
dd if=arquivo2 of=arquivo3 conv=ucase
```

Outra aplicação interessante deste comando seria renomear todos os arquivos em um determinado diretório com seu nome equivalente em letras minúsculas:

```
#!/bin/sh
for file in `ls`
do
mv $file `echo $file | dd conv=lcase`
done
```

Com o comando `dd`, podemos também fazer a recuperação de dados em disquetes ou fitas com erros, o comando `dd` pode ser muito útil. Normalmente, com comandos como `tar` e outros, sempre que são encontrados erros o processo de leitura é interrompido e não conseguimos recuperar nada, o que frequentemente é uma tragédia.

O comando

```
dd bs=1440b conv=sync,noerror if=/dev/fd0 of=/tmp/imagem.disquete
```

irá ler todo o disquete, gravando o resultado em /tmp/imagem.disquete. A opção `conv=noerror` fará com que os setores defeituosos sejam ignorados e tudo que puder ser lido seja gravado no arquivo de saída. É sempre melhor ter uma parcela de nossos dados do que dado nenhum, certo?

Um exemplo bastante ilustrativo da potencialidade do comando `dd` é o backup da MBR. O MBR (Master Boot Record) é a informação, no primeiro setor de um disco rígido ou disquete, que identifica como e onde está localizado o sistema operacional, de modo a que possa ser carregado pelo computador (boot). Adicionalmente, o MBR contém informações de particionamento do disco rígido (**Master Partition Table**). Finalmente, a MBR também inclui um programa que lê o setor de boot da partição que contém o sistema operacional que será carregado na memória do sistema (e por ai vai).

De tudo isto, deduzimos facilmente que a MBR contém informações muito importantes para o funcionamento do sistema. Ter uma cópia desta informação é **muito importante**, mas quase ninguém se lembra disto.

É importante, em procedimentos de backup, guardar uma cópia do MBR (Master Boot Record), para facilitar a recuperação de todas as informações nele contidas.

Este backup, em sistemas *nix, pode ser realizado com o seguinte comando:

```
dd if=/dev/hda of=mbr.backup bs=512 count=1
```

Para restaurar:

```
dd if=mbr-backup of=/dev/hda bs=512 count=1
```

Como a tabela de partições ocupa os últimos 66 bytes da MBR, caso se queira preservar esta informação, podemos variar a sintaxe do comando `dd`:

```
 dd if=mbr-backup of=/dev/hda bs=446 count=1
```

A identificação do disco neste exemplo, `/dev/hda`, pode variar. Discos SCSI, por exemplo, são identificados por `/dev/sdX`, onde `X` pode ser `a`, `b`, etc.

É claro que uma estratégia de backup vai muito além de se salvar a MBR.

Para saber mais, leia o documento sobre clonagem de sistemas GNU/Linux, onde eu falo em detalhes sobre este assunto. Os procedimentos podem ser adaptados para diversas finalidades. Afinal de contas, uma clonagem nada mais é do que uma forma de backup.

Este documento está em http://www.dicas-l.com.br/lic

Para saber mais sobre a MBR, leia o documento **Understanding and working with the MBR**, que está em http://www.geocities.com/rlcomp_1999/procedures/mbr.html,

5. O comando pwd

Indica o diretório corrente

6. O comando find

O comando find é extremamente poderoso e flexível para descobrir arquivos que atendem a determinadas especificações.

Por exemplo, suponhamos que queiramos descobrir todos os arquivos que não possuem dono em nosso sistema. Esta situação é extremamente comum, visto que usuários são criados e apagados diariamente e ficam vagando pelo sistema e podem eventualmente vir a comprometer a segurança. O comando

```
find / -nouser -print
```

irá gerar uma listagem com todos os arquivos do sistema que não pertencem a ninguém.

Caso queiramos simplesmente apagar estes arquivos (não recomendável!!!) basta redirecionar a saída deste comando para o comando xargs, da seguinte forma:

```
find / -nouser -print | xargs rm
```

O mais recomendável é gerar um backup destes arquivos, para em seguida apagá-los:

```
find . -cpio /dev/rmt0 -nouser
```

Para restaurar estes arquivos

```
cpio -idmv < /dev/rmt0
```

O comando cpio, a exemplo do comando find, é extremamente poderoso e flexível. Para mais informações sobre seu uso e sintaxe, consulte as man pages.

O comando find aceita diversas diretivas que lhe instruem sobre o que pesquisar em um sistema de arquivos. Arquivos maiores que um tamanho pré-determinado, que tenham sido modificados ou acessados até determinada data, etc.

Vamos então ver uma forma pouco usada, mas que pode ter sua utilidade.

Eu criei em um diretório um arquivo estranho. Deu um trabalho para criar o arquivo, pois ele continha uma marca de tabulação em seu nome.

```
$ ls
a  ?a  b  c  d  e
$ ls | cat -v
a
    a
b
c
d
e
```

Bom, agora eu quero remover este arquivo. Existem várias maneiras de se fazer isto, mas eu quero demonstrar uma que use o recurso do comando `find` que eu quero demonstrar. Quem sabe um dia alguém ache um exemplo mais inteligente :-)

```
$ ls -i
3358935 a  3358929 ?a  3358930 b  3358931 c  3358932 d  3358933 e
```

O comando `ls` irá exibir os i-nodes (index nodes) dos arquivos que são informações. i-nodes são estruturas de dados que indicam ao sistema operacional a localização física dos arquivos.

De posse desta informação, eu posso fazer:

```
find . -inum 3358929 -ok rm '{}' \;
```

O parâmetro `-ok` é uma verificação de segurança para garantir que nada saia errado. Ele vai buscar o arquivo cujo i-node seja igual a 3358929 e removê-lo.

Caso eu tenha certeza absoluta do que estou fazendo uma alternativa é usar a diretiva `-exec`, que executa o comando diretamente:

```
find . -inum 3358929 -exec rm '{}' \;
```

Para descobrir links simbólicos que apontam para arquivos que não existem mais:

```
find . -type l -follow 2>/dev/stdout | cut - d: -f 2 | xargs rm -f
```

Isso dá conta do recado!

Repare que o `2>/dev/stdout` pode ser substituido por `2>&1`, mas o primeiro é mais legivel.

Isso se faz necessario pois o `find`, ao seguir um link quebrado escreve a mensagem da saida padrão de erro (stderr), por isso temos que redirecionar, para que o `cut` consiga pegar e passar pra frente.

7. chmod, chown, chgrp, find e xargs

Para alterar recursivamente a propriedade de uma árvore de diretórios para uma determinada pessoa e grupo, usamos o comando `chown`:

```
chown -R queiroz:queiroz ~queiroz
```

O argumento `queiroz:queiroz` identifica o nome do usuário e o grupo ao qual pertence. Fornecer os dois argumentos é opcional, podemos especificar apenas o nome do usuário, como abaixo:

```
chown -R queiroz ~queiroz
```

Caso queiramos trocar apenas a propriedade do grupo, o comando é outro, `chgrp`:

```
chgrp -R queiroz ~queiroz
```

Até aqui tudo bem. Entretanto, frequentemente usamos os comandos `chgrp` e `chown` em conjunto com o comando `xargs` e `find`:

```
find . -type d | xargs chown -R queiroz
```

Com o comando acima, pretendo alterar apenas a propriedade dos diretórios a partir de um determinado ponto. Se existir algum diretório com um nome que contenha espaços em branco, teremos problema.

```
# find . -type d | xargs chown -R queiroz
chown: impossível acessar `./Rubens': Arquivo ou diretório não encontrado
chown: impossível acessar `Queiroz': Arquivo ou diretório não encontrado
chown: impossível acessar `de': Arquivo ou diretório não encontrado
chown: impossível acessar `Almeida': Arquivo ou diretório não encontrado
chown: impossível acessar `./Rubens': Arquivo ou diretório não encontrado
chown: impossível acessar `Queiroz': Arquivo ou diretório não encontrado
chown: impossível acessar `de': Arquivo ou diretório não encontrado
chown: impossível acessar `Almeida/Dia': Arquivo ou diretório não encontrado
chown: impossível acessar `de': Arquivo ou diretório não encontrado
chown: impossível acessar `Faxina': Arquivo ou diretório não encontrado
```

No exemplo acima eu criei dois diretórios com espaços em branco em seu nome: `Rubens Queiroz de Almeida` e `Dia de Faxina`. O comando `xargs` forneceu como entrada ao comando `chown`, as partes individuais dos nomes, levando em consideração o espaço em branco como delimitador do nome.

Para resolver este problema, precisamos informar ao comando `xargs` que o delimitador é outro:

```
find . -type d -print0 | xargs -0 chown -R queiroz
```

A opção `-print0` termina os nomes dos arquivos com um zero, de forma que os nomes de arquivos com espaços em branco sejam corretamente tratados. Da mesma forma, no comando `xargs` o argumento `-0` indica que o separador do nome dos arquivos é o zero e não o espaço em branco.

O argumento `-print` é opcional. Versões antigas de sistemas Unix exigiam que fosse especificado no comando `find`. As versões modernas de sistemas GNU/Linux e FreeBSD não exigem sua especificação, a não ser em casos como acima, em que atribuimos um valor diferente do padrão ao

delimitador do nome de arquivos.

8. O comando split

Muitas vezes precisamos dividir um arquivo em vários outros menores, seguindo alguma convenção. Para isto podemos usar tanto o comando `split`.

O comando `split` nos permite dividir um arquivo baseando-se no número de linhas ou número de bytes que cada arquivo novo deve conter.

Por exemplo:

```
% split -l 10 /etc/passwd
```

Este comando criará vários arquivos denominados `xaa`, `xab`, `xac`, etc. Nem sempre estes nomes são os mais convenientes. Neste caso podemos, com o acréscimo de mais um parâmetro, determinar o sufixo do nome dos arquivos que serão criados:

```
% split -l 10 /etc/passwd pas-
% ls
pas-aa  pas-ab  pas-ac  pas-ad  pas-ae  pas-af  pas-ag  pas-ah
```

Os arquivos criados passaram a conter o prefixo `pas-`, permitindo identificar mais claramente os contadores dos arquivos (aa, ab, ac, etc.)

Além do particionamento em linhas, o comando `split`, quando invocado com a opção `b`, irá efetuar a divisão do arquivo baseando-se no número de bytes:

```
% split -b 32k /etc/passwd pas-
```

ou então

```
% split -b 32 /etc/passwd pas-
```

ou ainda

```
% split -b 32m /etc/passwd pas-
```

No primeiro exemplo, o arquivo `/etc/passwd` será dividido em vários arquivos de 32kbytes cada um, ao passo que no segundo exemplo, o arquivo será dividido em arquivos de 32 bytes cada. No terceiro exemplo, o arquivo `/etc/passwd` é dividido em arquivos de 32MB cada (pouco provável :-)

Um outro exemplo, para dividir um arquivo em vários para gravação em disquetes de 1.44, o comando `split` é uma das opções possíveis:

```
# split -b 1400000
```

O comando `split` irá criar vários arquivos chamados xaa, xab, xac, etc.

Para restaurá-los basta usar o comando `cat`, como abaixo:

```
# cat x* > original_filename
```

O próximo exemplo mostra uma shell em que o comando split é usado para gerar o arquivo de índice de um website, chamado Contando Histórias. Neste site existe uma página onde é relacionado todo o conteúdo do site. Esta página é gerada através de um shell script que conta o número de mensagens existentes, divide este número por dois, e monta uma tabela com duas colunas. Para entender melhor o que é feito, nada melhor do que visitar a página de arquivo do site.

Vamos então ao script e à explicação de seu funcionamento.

```
#!/bin/bash

homedir=/html/contandohistorias

cd $homedir/html

# O laço que se segue trabalha
# sobre todos os arquivos do diretório
# /html/contandohistorias/inc que
# tenham a terminação "inc". Estes são arquivos
# no formato html, gerados pelo software txt2tags
# (txt2tags.sourceforge.net).

for file in *.inc
do

# O arquivo php final é formado a partir do nome do
# arquivo terminado em "inc". Este nome é atribuído
# à variável $php, definida no próximo comando

  php=`echo $file | sed 's/inc/php/'`

# No arquivo html a primeira linha contém o título
# da mensagem, formatada como título de nível 1
# (H1). O título é extraído desta linha com o
# comando sed e em seguida convertido em uma
# referência html, para ser usada mais tarde na
# montagem do arquivo geral.

  sed -n 1p $file | sed 's:<H1>::;s:</H1>:</A>:' | sed "s:^:<BR><A HREF=/historias/$php>:" >> /t

# Usamos o comando tac para inverter a ordem das
# mensagens, deixando as mais recentes em primeiro
# lugar na listagem.

  tac /tmp/idx.tmp > /tmp/idx.$$ && mv /tmp/idx.$$ /tmp/idx.tmp
done

cp /tmp/idx.tmp $homedir/inc/listagem.inc

# Fazemos a seguir a contagem de linhas do arquivo
# idx.tmp, que representa o total de mensagens já
# enviadas. A variável $half é obtida dividindo
# por 2 o número total de linhas do arquivo

lines=`wc -l /tmp/idx.tmp | awk '{print $1}'`
half=`expr $lines / 2`

# Usamos agora o comando split para partir o
# arquivo em dois. A diretiva "-l" sinaliza que
# a divisão do arquivo deve ser feita levando-se
# em conta o número de linhas (lines).
```

```
split -l $half /tmp/idx.tmp

# o comando split gera dois arquivos "xaa" e
# "xbb". Estes dois arquivos formarão as duas
# colunas da tabela.

mv xaa $homedir/inc/coluna1.inc
mv xab $homedir/inc/coluna2.inc

# A seguir, fazemos a construção do arquivo
# php final, através da inclusão dos diversos
# elementos da página: cabeçalho (Head.inc), barra
# de navegação (navbar.inc), barra da esquerda
# (esquerda.inc), e as duas colunas da tabela
# (coluna1.inc e coluna2.inc).


echo "<?PHP include(\"/html/contandohistorias/inc/Head.inc\"); ?>
<div id=top>
<H1>Contando Histórias</H1>
<?PHP include(\"/html/contandohistorias/inc/navbar.inc\"); ?>
</div>

<div id=mainleft>
<?PHP include(\"/html/contandohistorias/inc/esquerda.inc\"); ?>
</div>

<div id=maincenter>
<h1>Arquivo Lista Contando Histórias</h1>
<table>
<tr valign=top>
<td>
<?PHP include(\"/html/contandohistorias/inc/coluna1.inc\"); ?>
</td>
<td>
<?PHP include(\"/html/contandohistorias/inc/coluna2.inc\"); ?>
</td>
</tr>
</table>
</html>
</body>" > $homedir/arquivo.php

rm /tmp/idx.*
```

9. O comando seq

O comando `seq` realiza uma tarefa bastante útil. Veja um exemplo:

```
$ seq 1 5
1
2
3
4
5
```

Neste caso, o comando `seq` simplesmente imprimiu uma seqüência de números de 1 a 5.

Podemos também fazer assim:

```
$ seq 0 2 10
0
2
4
6
8
10
```

Neste caso, a seqüência de números foi impressa com um incremento de dois a cada interação. Em outras palavras, o comando `seq` foi instruído a imprimir os números de 0 a 10, somando 2 a cada interação.

Agora, um exemplo mais prático:

```
for n in ``seq 0 5 1000``
 do
  sed -n ${n}p testfile
 done
```

Este pequeno laço irá fazer a impressão, por meio do comando `sed`, de cada quinta linha do arquivo chamado `testfile`.

Ou ainda, eu posso criar um arquivo obedecendo a algumas normas:

```
for n in ``seq 1 100``
 do
   echo "Linha $n" >> testfile
 done
```

Teremos então o arquivo `testfile`, com o seguinte conteúdo:

```
Linha 1
Linha 2
Linha 3
Linha 4
Linha 5
Linha 6
Linha 7
Linha 8
Linha 9
```

```
Linha 10
 ....
```

Mais um exemplo de uso do comando seq:

```
$ wc -l arqtxt
712 arqtxt
$ seq 1 712 | paste - arqtxt > arqtxt.num
```

O que isto faz ?

O primeiro comando (`wc -l arqtxt`) conta o número de linhas do arquivo arqtxt, no caso são 712 linhas. O segundo comando gera uma versão do arquivo arqtxt com cada linha numerada de 1 a 712.

O comando paste é parte da suíte padrão de comandos para acesso a "bancos de dados de texto puro" (plain text database) do *nix, logo também está presente no Linux.

O comando seq pode ser usado para processar um grande lote de arquivos de log, por exemplo.

Muitas vezes, os arquivos tem o formato W3030621.log, que é composto por ano-mes-dia, e assim o comando seq pode gerar a sequencia de dias para o processamento.

Utiliza-se então o comando seq com a opção `-w`, que gera sempre a sequencia com o mesmo numero de caracteres, sempre colocando um zero no inicio:

```
seq -w 1 10
01
02
03
04
05
...
09
10
```

Depois, basta concatenar o inicio do nome do arquivo com a sua sequencia.

10. Os comandos basename/dirname

Dois comandos que permitem a manipulação de nomes e caminhos de arquivos dentro de sistemas Unix são os comandos `basename` e `dirname`.

O comando `basename`, quando recebe como argumento o caminho de um arquivo, remove a porção de diretórios, deixando o nome do arquivo.

```
% basename /usr/local/bin/gzip
gzip
```

Ou seja, este comando pode ser utilizado para extrair apenas o nome de um arquivo a partir do caminho completo, neste caso, o nome gzip.

Em outras palavras, o que este comando faz é retirar a última parte da cadeia de caracteres que não contenha o caractere `/`. Se a cadeia de caracteres terminar em `/`, este caracter também é removido:

Outra aplicação interessante é o uso deste comando para remoção da extensão. Veja só:

```
$ basename /usr/local/doc/propost.txt .txt
proposta
```

Agora para que serve isto? O mais óbvio é quando se quer renomear múltiplos arquivos:

```
#!/bin/sh
for file in `find . -name \*.txt -print`
do
    mv $file `dirname $file`/`basename $file .txt`.doc
done
```

Este script renomeia todos os arquivos que tiverem a extensão .txt, a partir de um certo ponto na árvore de diretórios para um arquivo com o mesmo nome base e extensão .doc.

Já o comando dirname retorna como resultado o caminho inteiro fornecido à exceção do último nome, como exemplificado abaixo:

```
% dirname /usr/local/bin/gzip
/usr/local/bin
```

Apareceu um comando novo neste script, o comando `dirname`. Ele faz o contrário do que faz o comando basename:

```
$ dirname /usr/local/doc/proposta.txt
/usr/local/doc
```

11. O comando fmt

O comando `fmt` é usado para fazer a formatação das linhas de um arquivo:

```
fmt -w 80 arquivo.txt > novoarquivo.txt
```

No comando acima, a largura máxima da linha é definida em 80 caracteres.

O comando `fmt` não faz alinhamento, ele tenta deixar as linhas com um tamanho próximo ao especificado.

O comando `fmt` pode ser invocado para fazer esta formatação tanto diretamente como a partir de um editor de texto como o vi.

Para invocá-lo a partir do vi, entre em modo de comando, pressionando a tecla <ESC> e digite:

```
:%!fmt -w 60
```

O caractere `%` indica que a formatação deve ser aplicada a todo o texto. O caractere `!` invoca um comando externo ao editor. O que se segue, `fmt -w 60`, indica que as linhas devem ser formatadas com largura máxima de 60 caracteres. Isto nem sempre é possível, pois o `fmt` não realiza separação silábica.

12. O comando uniq

O comando `uniq` pode ser usado para gerar um relatório ou para omitir as linhas repetidas em um arquivo. As principais opções são:

-c, --count	inclua antes de cada linha o número de ocorrências
-d, --repeated	imprima apenas as linhas duplicadas
-f, --skip-fields=N	não realize a comparação nos primeiros N campos
-i, --ignore-case	ignorar as diferenças de caixa (maíusculas ou minúsculas
-s, --skip-chars=N	evite comparar os primeiros N caracteres
-u, --unique	imprima apenas as linhas sem duplicidade
-w, --check-chars=N	inão compare mais do que N caracteres em cada linha
--help	exibe a ajuda e sai
--version	exibe a versão do programa e sai

Por exemplo, suponhamos que temos um arquivo com todas as palavras encontradas em um livro, uma por linha. Para imprimir a contagem de ocorrência no texto de cada palavra, podemos usar o comando:

```
uniq -c arquivopalavras.txt
```

Pegando a transcrição do diálogo, no filme *Matrix Revolutions*, de Neo com o Arquiteto (*The Architect Transcript*), podemos gerar uma lista das palavras mais usadas:

```
#!/bin/bash

cat thearchitecttranscript.txt |\
  sed 's/ /\n/g;s/://g;s/\.//g;s/[0-9]//g;s/?//g' |\
  sort | cat -s | uniq -c | sort -nr | head
```

O resultado:

```
82 the
45 of
40 to
29 The
28 is
22 a
21 Architect
15 you
15 and
14 Neo
```

Exercício

Explique o significado dos comandos utilizados neste pequeno shell script.

13. O comando du

O comando `du` significa *display usage*, e indica a quantidade de espaço ocupada por arquivos e diretórios. Para explicar o significado do comando `du` nada melhor do que um exemplo. Para obter uma listagem de diretórios em sistemas GNU/Linux, ordenados do maior para o menor, use o comando:

```
$ du -S | sort -nr | more
```

A opção `-k` é bastante útil, pois permite a visualização do espaço ocupado em unidades de 1024 bytes (kbytes, Megabytes, Gigabytes).

OPÇÕES POSIX

-a	Exibe a contagem para todos os arquivos encontrados, não somente diretórios.
-k	Usa unidades de 1024 bytes ao invês do padrão de unidades de 512 bytes.
-s	Somente devolve o espaço usado para o atual argumento fornecido, e não para seus sub-diretórios.
-x	Somente conta o espaço no mesmo dispositivo como argumento fornecido.

OPÇÕES GNU

-a, --all	Exibe a contagem para todos os arquivos, não somente diretórios.
-b, --bytes	Imprime o tamanho em bytes, ao invés de kilobytes.
--block-size=tamanho	Imprime o tamanho em blocos de tamanho bytes. (Novo no Utilitários de Arquivo 4.0.)
-c, --total	Imprime um total geral para todo argumento depois que todos os argumentos tenham sido processados. Isto pode ser usado para descobrir o total do disco usado de um conjunto de arquivos ou diretórios.
-D, --dereference-args	Diferencia ligações simbólicas que são argumentos de comando de linha. Não afeta outras ligações simbólicas. Isto ajuda a descobrir a utilização dos diretórios do disco, assim como /usr/tmp, os quais freqüentemente são ligações simbólicas.
--exclude=modelo	Quando recursivo, salta sub-diretórios ou arquivos concordantes com modelo. O modelo pode ser um arquivo de modelo global padrão do interpretador Bourne. (Novo no Utilitários de Arquivo 4.0.)
-h, --human-readable	Anexa o rótulo de tamanho, como por exemplo M para binários de megabytes ('mebibytes'), para cada tamanho.
-H, --si	Faça igual à opção -h, mas use a unidade oficial do SI (com potência de mil ao invês de 1024, de modo que M representa 1000000 ao invés de 1048576). (Novo no Utilitários de Arquivo 4.0.)
-k, --kilobytes	Imprime o tamanho em kilobytes.

-l, --count-links	Conta o tamanho de todos arquivos, mesmo se eles já aparecem (como ligação forte).
-L, --dereference	Diferencia ligações simbólicas (mostra o espaço de disco usado por um arquivo ou diretório apontado pela ligação no lugar do espaço usado pela ligação).
-m, --megabytes	Imprime o tamanho em blocos de megabytes (1,048,576 bytes).
--max-depth=n	Imprime o total para um diretório (ou arquivo, com o sinalizador -a) só se for n ou menos níveis abaixo do argumento da linha de comando; --max-depth=0 é igual ao sinalizador -s (Novo no Utilitários de Arquivo 4.0.)
-s, --summarize	Exibe somente um total para cada argumento.
-S, --separate-dirs	Relata o tamanho para cada diretório separadamente, não incluindo o tamanho dos sub-diretórios.
-x, --one-file-system	Salta diretórios que estão em diferentes sistemas de arquivos daquele no qual o agumento está sendo processado.
-X arquivo, --exclude-from=arquivo	Como --exclude, exceto que pega o modelo para excluir a partir do arquivo determinado. Modelos são listados um por linha. Se o arquivo é fornecido como '-', modelos são lidos da entrada padrão.

Como podemos ver, o comando `du` possui uma grande variedade de opções. Estas facilidades simplificam em muito a codificação de scripts para obter informações sobre uso de arquivos em sistemas.

Vejamos a seguir, um pequeno tutorial escrito por Domingos Antonio Pereira Creado, sobre como monitorar o uso de espaço em disco usando o comando `du`. Além do comando `du`, vários outros comandos são utilizados. Se você não conhecer algum deles, não se preocupe, tente se concentrar no uso do programa `du`.

Quando um volume enche a situação normalmente pega fogo. Se for servidor ainda de arquivos então... E a situação é complicada pois não tem outro jeito se não ser a liberação de espaço, ou removendo arquivos ou movendo para outros volumes. E para ser rápida a operação tem que atacar logo quem está ocupando grandes partes do volume. Costumo normalmente utilizar o seguinte comando:

```
du -s * | sort -rn | head -10
```

O `du -s *` cria uma lista com os tamanhos e nomes dos objetos no diretório corrente - no caso de diretórios o tamanho dos objetos dentro dele serão somados (parametro `-s`) , o `sort -rn` pega a lista gerada pelo du e ordena a primeira coluna na forma numérica (o `-n`) e de forma inversa (o parametro `-r`) e o `head -10` mostra somente as 10 primeiras linhas.

Assim serão retornados os 10 maiores vilões da ocupação do volume.

Pode-se incrementar um pouco mais trocando a simples expansão `*` por uma mais seletiva, como por exemplo `/home/d*` para saber qual das contas de usuários que iniciam com `d` estão ocupando mais espaço, ou ainda `/home/d*/*` para saber quais os diretórios das contas dos usuários `d*` estão ocupando maior espaço.

Se você vai procurar quem está ocupando mais espaço em sua conta, lembre-se que as expansões acima não incluem os objetos "escondidos" (ou iniciados com `.`), até dá para utilizar a expansão `.*` mas ela inclui também o `..` o que não é uma boa.

Assim se você vai caçar comedores de disco em uma conta, talvez seja melhor utilizar a construção

```
du -s `ls -A` | sort -rn | head -10
```

Nessa linha de trocar a expansão por uma lista, o céu torna-se o limite (quer dizer a quantidade de memória).

A seguir, temos um exemplo de uso do comando `du`, para monitorar a taxa de ocupação de espaço em disco.

```
#!/bin/bash
# Script para mandar um aviso por e-mail para RESPONSAVEL
# quando PARTICAO estiver acima de MAX % de uso

MAX=95
RESPONSAVEL=rubens.queiroz@gmail.com

for Particao in /
do

SIZE=`df $Particao|sed -e '1d;s/[[:space:]]\+/ /g'|cut -d' ' -f5|sed -e 's/^\([0-9]\+\)%/\1/g'`

   if [ $SIZE -gt $MAX ] ; then
    /root/bin/cleanlog.sh
    echo "Partição \"$Particao\": $SIZE % de uso" >> /tmp/df.$$
   fi
done

if [ -e /tmp/df.$$ ] ; then
mail -s "Aviso: Partições com alta taxa de ocupação - `date`" $RESPONSAVEL < /tmp/df.$$
fi
```

14. O comando paste

Colaboração: Júlio Neves

O paste é um comando pouco usado por sua sintaxe ser pouco conhecida. Vamos brincar com 2 arquivos criados da seguinte forma:

```
$ seq 10 > inteiros
$ seq 2 2 10 > pares
```

Para ver o conteúdo dos arquivos criados, vamos usar o paste na sua forma careta:

```
$ paste inteiros pares
1     2
2     4
3     6
4     8
5     10
6
7
8
9
10
```

Agora vamos transformar a coluna do pares em linha:

```
$ paste -s pares
2     4     6     8     10
```

O separador default é <TAB>, mas isso pode ser alterado com a opção -d. Então para calcular a soma do conteúdo de pares primeiramente faríamos:

```
$ paste -s -d'+' pares # também poderia ser -sd'+'
2+4+6+8+10
```

e depois passaríamos esta linha para a calculadora (bc) e então ficaria:

```
$ paste -s -d'+' pares | bc
30
```

Assim sendo, para calcular o fatorial do número contido em $Num, basta:

```
$ seq $Num | paste -sd'*' | bc
```

15. O comando csplit

Outro comando também utilizado para se dividir um arquivo em vários outros é o comando csplit (Content Split).

Ao contrário do comando split, abordado na dica anterior, o comando csplit permite que se especifique uma string que irá indicar o delimitador de cada um dos novos arquivos.

Tomemos como exemplo o arquivo abaixo, chamado arq1:

arq1

```
Capítulo 1

Era uma vez, era uma vez três porquinhos, Palhaço, Palito e Pedrito.

Capítulo 2
E o Lobo Mau, ...
Capítulo 3
E o caçador, matou o Lobo Mau, casou-se com a Chapeuzinho Vermelho,
e viveram felizes para sempre.

        The End
@@@ Fim arq1
```

O autor, colocou todos os capítulos do livro em apenas um arquivo e depois se arrependeu. Agora ele quer criar vários arquivos contendo um capítulo cada. O comando abaixo pode resolver este problema:

```
% csplit -f Capit arq1 "/Capitulo/" {2}
% ls -l
total 4
-rw-r--r--   1 queiroz  supsof     0 Jun 17 18:31 Capit00
-rw-r--r--   1 queiroz  supsof    85 Jun 17 18:31 Capit01
-rw-r--r--   1 queiroz  supsof    29 Jun 17 18:31 Capit02
-rw-r--r--   1 queiroz  supsof   136 Jun 17 18:31 Capit03
-rw-r--r--   1 queiroz  supsof   250 Jun 17 18:11 arq1
```

Traduzindo, o comando csplit irá criar vários arquivos iniciados em `Capit`, até um máximo de 3 arquivos (parâmetro {2}, computa-se o número entre colchetes + 1).

Este valor indica o número de vezes que o comando será repetido.

No nosso exemplo, foi especificado exatamente o número de capítulos contidos no arquivo original (3). Caso não conheçamos este valor, podemos especificar um número que sabemos maior que o número de arquivos existentes. O comando csplit irá reclamar, e apagar todos os arquivos já criados. Para evitarmos que isto aconteça, basta especificar a diretiva `-k`, ou seja, a reclamação continuará sendo feita, mas o trabalho já feito não será removido.

O que não pode é se especificar um número inferior ao desejado. Neste caso, o comando ficaria como:

```
% csplit -k -f Capit arq1 "/Capitulo/" {9}
0
85
29
```

```
csplit: {9} - out of range
136
```

A quebra será feita, tomando-se por base o nosso exemplo, antes da string Capitulo, exclusive. Devido a isto, o primeiro arquivo, Capit00, está vazio. Os arquivos criados, à exceção do arquivo Capit00 que está vazio, contêm:

Capit01

```
Capitulo 1

Era uma vez, era uma vez três porquinhos, Palhaço, Palito e Pedrito.
  ...
```

Capit02

```
Capitulo 2
E o Lobo Mau, ...
```

Capit03

```
Capitulo 3
E o caçador, matou o Lobo Mau, casou-se com a Chapeuzinho Vermelho,
e viveram felizes para sempre.

        The End
```

16. Os comandos more e less

O comando `more` é utilizado para visualizar, página a página, o conteúdo de um arquivo. É semelhante a um editor de textos em recursos como localizar caracteres, navegar para a frente e para trás no documento, e algumas outras facilidades. Entretanto, a sua forma mais comum de uso é sua invocação sem nenhum parâmetro além, é claro, do nome do arquivo que se deseja visualizar.

Na mesma linha do comando `more`, temos o comando `less`. Existe até um ditado que diz, `less is more`, para brincar com o fato de que o comando `less` possui mais recursos do que o comando `more`. A sintaxe de uso do comando `less` é similar em muitos aspectos à sintaxe do editor `vi`, o que lhe dá muito mais flexibilidade.

Mesmo com todos os recursos adicionais que possui, o `less` ainda é menos usado do que o comando `more`, talvez devido ao seu nome pouco intuitivo. Em termos práticos, do que realmente é utilizado, tanto faz usar um ou outro.

17. O comando sed

O SED é um dos comandos que merece um estudo mais aprofundado, pois é o canivete suiço da programação shell. Dá para fazer o impensável com ele quando se trata de manipulação de cadeias de caracteres. Para aprofundar o conhecimento de sed leia o SED HOWTO, de autoria de Aurélio Marinho Jargas.

O sed já foi usado abusivamente nos exemplos anteriores, e é provável que você já tenha chegado aqui com um bom conhecimento de seu uso. Apesar disto, chegou a hora de apresentá-lo corretamente. De forma bem simples, o SED é um editor de textos não interativo. Mostraremos a seguir alguns exemplos de uso, que podem ser usados de diversas formas diferentes em seus programas shell.

A seguir, incluo uma tradução de partes do documento HANDY ONE-LINERS FOR SED.

Inserção de linhas em branco em um arquivo

```
sed G
```

Inserir uma linha em branco acima de cada linha que contenha a expressão "regex".

```
sed '/regex/{x;p;x;}'
```

Inserir uma linha em branco abaixo de todas as linhas que contenham a expressão "regex":

```
sed '/regex/G'
```

Inserir uma linha em branco acima e abaixo de todas as linhas que contenham a expressão "regex".

```
sed '/regex/{x;p;x;G;}'
```

17.1. Conversão de Texto e Substituição

Apagar espaços em branco e marcas de tabulação que estejam no início de cada linha

```
sed 's/^[ \t]*//'
```

Apagar espaços em branco (espaços, tabulação) ao final de cada linha

```
sed 's/[ \t]*$//'
```

Apagar espaços em branco (espaços, tabulação) no início e fim de cada linha

```
sed 's/^[ \t]*//;s/[ \t]*$//'
```

Inserir cinco espaços em branco no início de cada linha

```
sed 's/^/   /'
```

Substituir todas as ocorrências da palavra "foo" pela palavra "bar"

```
sed 's/foo/bar/'       # Substitui apenas a primeira ocorrência
                       # da palavra bar em cada linha
sed 's/foo/bar/4'      # substitui apenas a quarta ocorrência na linha
sed 's/foo/bar/g'      # substitui TODAS as palavras na linha (g)
```

Substitui a palavra "foo" pela palavra "bar" apenas nas linhas que contenham a sequência "baz"

sed '/baz/s/foo/bar/g'

Substitui a palavra "foo" pela palavra "bar" exceto nas linhas que contenham a sequência "baz"

```
sed '/baz/!s/foo/bar/g'
```

Substitui as palavras "scarlet" ou "ruby" ou "puce" por "red"

```
sed 's/scarlet/red/g;s/ruby/red/g;s/puce/red/g'
```

Se uma linha termina em "\", junte-a com a próxima linha

```
sed -e :a -e '/\\$/N; s/\\\n//; ta'
```

Insira uma linha em branco a cada cinco linhas

```
gsed '0~5G'
sed 'n;n;n;n;G;'
```

Impressão Seletiva de Algumas Linhas

Imprimir as dez primeiras linhas de um arquivo

```
sed 10q
```

Imprimir a primeira linha de um arquivo

```
sed q
```

Imprimir apenas as linhas que combinem com uma expressão regular

```
sed -n '/regexp/p'
sed '/regexp/!d'
```

Imprimir apenas as linhas que NÃO combinem com uma expressão regular

```
sed -n '/regexp/!p'
sed '/regexp/d'
```

Imprimir apenas as linhas que tiverem 65 ou mais caracteres

```
sed -n '/^.\{65\}/p'
```

Imprimir apenas as linhas que tiverem 65 caracteres ou menos

```
sed -n '/^.\{65\}/!p'
```

```
sed '/^.\{65\}/d'
```

Imprimir a parte do arquivo que contiver uma expressão regular até o final do arquivo

```
sed -n '/regexp/,$p'
```

Imprimir linhas selecionadas de um arquivo

```
sed -n '8,12p'
sed '8,12!d'
```

Imprimir a linha 52

```
sed -n '52p'
sed '52!d'
sed '52q;d'
```

Imprimir a parte do arquivo que fica entre duas expressões regulares (regexp)

```
sed -n '/Belo Horizonte/,/Campinas/p'
```

17.2. Deleção Seletiva de Linhas

Imprime todo o arquivo, EXCETO a parte entre duas expressões regulares

```
sed '/Belo Horizonte/,/Campinas/d'
```

Apagar as dez primeiras linhas de um arquivo

```
sed '1,10d'
```

Apagar a última linha de um arquivo

```
sed '$d'
```

Use de `\t` em scripts sed: Para maior clareza na documentação, nós usamos a expressão `\t` para indicar uma marca de tabulação (0x09) nos scripts. Entretanto, a maior parte das versões do sed não reconhece esta abreviação, então, ao invés de digitar estes caracteres, digite diretamente a tecla <TAB>. `\t` é reconhecido diretamente como um metacaractere em expressões regulares no awk, perl, HHsed, sedmod e GNU sed v3.02.80. Usuários de distribuições modernas de GNU/Linux contam com a versão 4 ou superior do SED.

18. O comando sleep

O comando `sleep` serve para suspender a execução de uma shell por um determinado período de tempo. O comando sleep aceita, como diretiva, um número em segundos.

O comando

```
sleep 300
```

suspende a execução do script (dorme) por trezentos segundos ou cinco minutos.

19. O comando echo

O comando `echo` simplesmente ecoa para a tela o argumento que lhe for fornecido:

```
echo "Hello World"
```

O resultado pode ser redirecionado para um arquivo:

```
echo "Hello World" > helloworld.txt
```

O `echo` pode incluir diversas linhas:

```
echo "Adeus
Mundo
Cruel" > mundocruel.txt
```

Podemos também suprimir, com a diretiva `-n`, a quebra de linha.

```
echo -n "Hello World" > helloworld.txt
echo "Hello World" >> helloworld.txt
```

Vejamos o resultado:

```
% cat helloworld.txt
Hello WorldHello World
```

Como podemos ver, os dois comandos `echo` foram combinados em uma única linha.

20. O comando touch

O comando `touch` tem várias finalidades. Ao emitirmos o comando `touch` se o arquivo que especificarmos não existir, será criado um arquivo vazio com o nome que fornecermos

```
touch arquivonovo.txt
```

A seguir reproduzimos um trecho da documentação do comando `touch` com a explicação das diretivas suportadas.

Opção	Descrição
-a	Altera a data de acesso do arquivo.
-c	Não cria o arquivo.
-m	Altera a data de modificação do arquivo.
-r arquivo_de_referência.	Usa o rótulo de tempo correspondente do arquivo_de_referência como o novo valor para o(s) rótulo(s) de tempo alterado.
-t time	Usa a data especificada como novo valor para o(s) rótulo(s) de tempo alterado. O argumento é um número decimal na forma [[CC]YY]MMDDhhmm[.SS] com significado óbvio. Se CC não é especificado, o ano CCYY é considerado no intervalo de 1969-2068. SE SS não é especificado, ele é considerado como 0. Ele pode ser especificado no intervalo 0-61 de forma que isto é possível se referir a saltos de segundos. A data resultante é considerada como a data para o fuso horário especificado pela variável de ambiente TZ. É errado se a data resultante antecede 1 de Janeiro de 1970.

21. O Comando rsync

Nos dias de hoje, com os dados em sistemas computadorizados sendo atualizados constantemente ao longo das 24 horas do dia, surgiu a necessidade de se fazer a replicação de conteúdo de forma mais ágil e que permita a recuperação quase imediata de informações.

Eu uso com frequência o rsync para fazer estas tarefas. Desde a replicação de dados pessoais em outros computadores até o backup de sistemas inteiros.

A sintaxe é bastante simples. Alguns exemplos:

```
rsync -avz -e ssh acme.com.br:/home/queiroz .
```

O comando acima irá copiar, no diretório corrente, todo o diretório chamado /home/queiroz. Já o comando

```
rsync -avz -e ssh acme.com.br:/home/queiroz/ .
```

irá copiar apenas o conteúdo do diretório /home/queiroz

As diretivas usadas significam:

a	archive, basicamente indica que voce quer que a cópia seja recursiva e que tudo seja preservado (links simbólicos, por exemplo).
v	verbose, escreva tudo que estiver fazendo
z	compactar os arquivos transferidos
e	especifica a shell remota a ser usada, no nosso caso, ssh, o que garante que os dados serão transmitidos usando criptografia

O comando rsync possui uma grande vantagem: ele copia apenas o que mudou na árvore de diretórios. De um arquivo modificado ele irá transferir apenas o blocos novos ou alterados.

Antes de transferir os dados, faz uma comparação do arquivo na origem e no destino. Os arquivos são quebrados em segmentos e os seus checksums são comparados. Os pedaços cujos checksums forem diferentes são transmitidos.

Em um mundo em que os computadores estão ficando cada vez mais baratos, o rsync pode ser uma alternativa, entre as muitas existentes, de se manter um espelho de uma máquina de produção. Em caso de falhas, a máquina espelho assume rapidamente o lugar da máquina principal.

A seguir, temos um exemplo simples de sincronização de conteúdo de um servidor inteiro, com a exclusão de alguns diretórios (diretiva `--exclude`). Da mesma forma, este script remove do backup os arquivos que já tenham sido apagados do servidor original (diretiva `--exclude`). O script também toma especial cuidado para registrar em um log tudo o que ocorre durante o backup.

```
#!/bin/sh

cd /work/backup/jvds

rsync -avz -e ssh                       \
```

```
--delete                        \
--exclude "/proc"               \
--exclude "/var/spool/mail"     \
--exclude "/var/spool/postfix" \
--exclude "/tmp"                \
www.acme.com:/ . > /root/logs/acme_`date +%Y%m%d`.log
```

O rsync foi escrito pelo mesmo criador do Samba, Andrew Tridgell.

22. O comando wget

22.1. Download seletivo de arquivos com wget

Autor: Gentil de Bortoli Júnior

Algumas pessoas têm necessidade de fazer o download de apenas determinados tipos de arquivos como, por exemplo, PDFs ou imagens.

Isso pode ser feito de maneira muito simples, utilizando o wget. Uma pessoa que deseja baixar todos os PDFs do endereço http://ldp.conectiva.com.br pode fazer algo como:

```
$ mkdir LDP_PDFs ; cd LDP_PDFs
$ wget -A .pdf -r -nd http://ldp.conectiva.com.br
```

Explicando:

A	Baixe somente os arquivos com a extensão fornecida
r	Siga os links recursivamente
nd	Não crie hierarquia de diretórios

Como resultado desse comando, dentro do diretório LDP_PDFs você terá todos os PDFs que podem ser encontrados seguindo os links no site em questão.

22.2. Download de Páginas ou Arquivos na Web

O comando wget nos permite realizar o download de páginas Web e nos oferece diversas facilidades interessantes.

A mais útil é a possibilidade de retomar um download interrompido. Para buscar arquivos grandes, como por exemplo o OpenOffice, que tem mais de 60MB, este recurso é realmente fundamental.

Para isto, basta invocar o comando wget com a opção `-t0`:

```
wget -t0 ftp://ftp.qualquer-lugar.com/openofice.tgz
```

Um outro recurso interessante nos permite realizar o download de documentos web. Muitos sites não oferecem a possibilidade de se baixar o documento inteiro e nos forçam a navegar por uma infinidade de páginas para obter a informação que queremos. Com o wget podemos baixar o documento inteiro da seguinte forma:

```
wget -t0 --no-parent http://www.qualquer-lugar.com/index.html
```

A opção --no-parent impede que durante o download o programa saia da página que se pretende baixar, por meio de um link para, por exemplo, a página principal do site, e vá para outros locais.

23. O comando file

Em sistemas Linux, encontramos diversos tipos de arquivos, geralmente identificados por suas terminações:

Arquivos compactados com o programa compress	.Z
Arquivos criados com o comando `tar` (tape archive)	.tar
Arquivos compactados com o programa gzip	.gz
Arquivos criados com o comando `tar` e compactados com o programa `gzip`	.tgz
Arquivos texto	.txt
Arquivos html	.html/.htm
Arquivos PostScript	.ps
Arquivos de áudio	.au/.wav
Arquivos de imagens	.xpm/.jpg/.gif/.png
Arquivos de Distribuição de Software (Red Hat Package Manager)	.rpm
Arquivos de configuração	.conf
Código fonte de programas C	.c
Arquivos de Cabeçalho	.h
Código Objeto	.o
Programas Perl	.pl
Programas TCL	.tcl
Código Compartilhado	.so

Esta é a convenção, que nem sempre é usada. Pode-se perfeitamente criar um arquivo texto que não tenha a terminação .txt. O comando `file` nos permite identificar a que categoria um arquivo pertence. Vejamos alguns exemplos:

```
$ file /etc/passwd n*
/etc/passwd:   ASCII text
networks:      empty
nscd.conf:     ASCII text
nsswitch.conf: English text
ntp:           directory
ntp.conf:      English text
```

O comando `file` baseia suas decisões consultando o arquivo `/usr/share/file/magic`, onde estão registradas as características dos principais tipos de arquivos do sistema. Para mais informações consultar a documentação do arquivo `magic`.