

Curso de LPIC-1

Tópico 103.1 - Trabalhando na Linha de Comando Shell, Bash, Echo, Type, Path

Douglas Luna - @dougluna 

SHELL	3
Funções do Shell	3
Tipos de Shell (os principais)	3
Bourne Shell	3
Korn Shell	3
C Shell	3
Bash - Bourne Again Shell (Foco da LPIC)	3
Verificar qual é seu Shell	4
echo	4
type	5
PATH	6
Caminho absoluto	6
Caminho Relativo ou Parcial	7
Dentro do diretório	7
pwd	8
Introdução sobre variáveis de ambiente	9
Como declarar uma variável	10
Variáveis Locais e Globais	11
export	12
Verificar as variáveis	14
set	14
env	15
Como remover variáveis criadas	17
unset	17
Variáveis Pré-definidas do ambiente Linux	18
Variáveis dinâmicas	19
Sequência de Comandos	21
O separador “;” (Ponto e virgula)	21
O comando && (E comercial)	22

O comando (Pipe)	23
O comando !! (Exclamação)	24
Repetição de comandos	25
history	25
Como limpar o arquivo de Histórico	27
.bash_history	27
Pesquisando comandos digitados	29
O Auto Completar (com TAB)	30
Comandos de ajuda	32
man	32
info	34
whatis e apropos	35
uname	36
alias	37
unalias	38
which	39
Quoting	40
Aspas Simples - Single Quotes	40
Aspas Duplas - Double Quotes	41
Barra Invertida - Escape Character	42

SHELL

O shell é a ligação entre o usuário e o sistema. É ele quem interpreta os comandos entrados para outros aplicativos ou diretamente em chamadas de sistema. Além disso, os recursos do shell são indispensáveis para lidar com muitos arquivos ao mesmo tempo, para realizar uma tarefa repetidamente ou para programar uma ação para determinada ocasião, entre outros recursos. Começamos apresentando os tipos de shell mais difundidos e depois definindo alguns conceitos que serão úteis na sua utilização, para então tratarmos de exemplos práticos.

Funções do Shell

- Analisar dados a partir do prompt (dados de entrada);
- Interpretar comandos;
- Controlar ambiente Unix-like (console);
- Fazer redirecionamento de entrada e saída;
- Ambiente de programação e Execução de programas;
- Linguagem de programação interpretada.

Tipos de Shell (os principais)

Bourne Shell

É o shell padrão para Unix, ou seja, a matriz dos outros shells, portanto é um dos mais utilizados. É representado por "**sh**". Foi desenvolvido por Stephen Bourne, por isso Bourne Shell.

Korn Shell

Este shell é o Bourne Shell evoluído, portando todos os comandos que funcionavam no Bourne Shell funcionarão neste com a vantagem de ter mais opções. É representado por "**ksh**".

C Shell

É o shell mais utilizado em BSD, e possui uma sintaxe muito parecida com a linguagem C. Este tipo de shell já se distancia mais do Bourne Shell, portanto quem programa para ele terá problemas quanto a portabilidade em outros tipos. É representado por "**csh**".

Bash - Bourne Again Shell (Foco da LPIC)

É o shell desenvolvido para o projeto GNU usado pelo GNU/Linux, é muito usado pois o sistema portador dele evolui e é adotado rapidamente. Possui uma boa portabilidade, pois possui características do Korn Shell e C Shell. É representado por "**bash**".

O FOCO DE ESTUDO É ESTE SHELL: BASH.

Verificar qual é seu Shell

Para verificar qual SHELL estamos usando, basta dar o comando abaixo, ele irá imprimir na tela o caminho do shell bash “/bin/bash”:

- Comando: echo \$SHELL
- Resultado: /bin/bash
- Ou seja, o Shell que estamos usando é o BASH.

echo

```
$ echo $SHELL
```

```
dougluna@dip:~$ echo $SHELL  
/bin/bash
```

echo – O comando serve para imprimir informações na tela o resultado daquilo passado na frente, ou seja, ele joga o resultado para sua saída padrão. Em conjunto com o símbolo de redirecionamento de saída >>, estudado mais à frente, também é utilizado para **concatenar** (ou seja, adiciona a informação no arquivo, sem retirar as já existentes) informações para dentro de um arquivo. Exemplo:

```
$ echo "Este é o curso de LPIC-1" >> /home/dougluna/teste.txt
```

```
dougluna@dip:~$ echo "Este é o curso de LPIC-1" >> /home/dougluna/teste.txt  
dougluna@dip:~$ cat /home/dougluna/teste.txt  
Este é o curso de LPIC-1
```

No exemplo acima, introduzimos a frase no arquivo teste.txt, mesmo o arquivo não existindo no diretório, ele é criado automaticamente ao executarmos o comando.

type

No Linux existem comandos internos, que já são embutidos no programa SHELL e comandos que são externos ou migrados de outras aplicações, que estão em nosso sistema. Para sabermos se um comando é de origem SHELL, usamos o comando **type**, exemplo:

```
$ type echo
```

1. Após o comando, teremos o retorno de uma mensagem nos dizendo se é ou não um comando interno do shell, ou seja, já está em sua compilação de origem.

```
dougluna@dip:~$ type echo  
echo is a shell builtin
```

O comando é interno conforme mensagem “**echo** is a shell builtin”

2. Quando fazemos o type apontando para outro comando que é externo, ele nos mostra a localização do comando:

```
dougluna@dip:~$ type tar  
tar is /usr/bin/tar
```

O **tar** se encontra no path **/usr/bin/tar**

3. Um comando mesmo que externo após digitado muitas vezes o Linux nos cria um cache interno como é o caso do comando “clear”. O comando “clear” é um comando externo. Após algumas vezes digitado ele passa a ser hashed, “cacheado” pelo sistema Linux:

```
dougluna@dip:~$ type clear  
clear is hashed (/usr/bin/clear)
```

O **clear** se encontra em cache no path **/usr/bin/clear**

PATH

A variável de ambiente **PATH** indica para o sistema o caminho dos comandos externos dentro do Linux. Quando damos o comando abaixo ele nos mostra os diretórios em que os programas e comandos estão localizados:

```
$ echo $PATH
```

Quando um comando é acionado ele procura em cada diretório onde está o programa correspondente. Os diretórios estão separados por ":".

```
dougluna@dip:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

- 1º - /usr/local/sbin
- 2º - /usr/local/bin
- 3º - /usr/sbin
- 4º - /usr/bin
- 5º - /sbin
- 6º - /usr/games
- 7º - /usr/local/games

Há também formas de chamar um comando quando ele não é interno e nem está presente dentro dos diretórios dos programas externos acima. Para fazer isso podemos fazer das seguintes formas:

Usaremos o arquivo do exercício disponibilizado no curso como é mostrado na aula:

Caminho absoluto

Dessa forma ele executará normalmente, pois indicamos o path exato de onde está o programa que queremos executar.

```
$ /home/dougluna/lpi/Exercicios/Script_Exemplo.sh
```

```
dougluna@dip:~$ /home/dougluna/lpi/Exercicios/Script_Exemplo.sh
Este é um Script de Teste

Sat Jun 6 22:00:13 -03 2020

Fim do Script
```

Caminho Relativo ou Parcial

Neste caso, se você já estiver em uma pasta que está no mesmo caminho que o script que deseja executar, basta continuar o comando até o caminho do script. Perceba que, estamos dentro da pasta do usuário **/home/kali**, podemos conferir com o comando “**pwd**”, o script se encontra dentro de **Exercicios/Script_Exemplo.sh** que já é parte do caminho do script:

```
$ pwd
```

```
$ lpi/Exercicios/Script_Exemplo.sh
```

```
dougluna@dip:~$ pwd
/home/dougluna
dougluna@dip:~$ lpi/Exercicios/Script_Exemplo.sh
Este é um Script de Teste

Sat Jun  6 22:04:38 -03 2020

Fim do Script
```

A partir deste diretório execute o **lpi/Exercicios/Script_Exemplo.sh**

Dentro do diretório

Neste caso, usaremos “./” **que significa: neste diretório**. Para executar o comando dessa forma temos que estar dentro do diretório em que o script se encontra, e a partir daí fazemos da seguinte forma para executarmos o comando com sucesso:

```
$ ./Script_Exemplo.sh
```

```
dougluna@dip:~/lpi/Exercicios$ ./Script_Exemplo.sh
Este é um Script de Teste

Sat Jun  6 22:06:01 -03 2020

Fim do Script
```

Neste diretório execute o **Script_Exemplo.sh**

pwd

O comando `pwd` tem como principal função mostrar o nome completo do diretório atual de trabalho.

```
$ pwd
```

```
dougLuna@dip:~/lpi/Exercicios$ pwd --help
pwd: pwd [-LP]
    Print the name of the current working directory.

Options:
  -L      print the value of $PWD if it names the current working
          directory
  -P      print the physical directory, without any symbolic links

By default, 'pwd' behaves as if '-L' were specified.

Exit Status:
Returns 0 unless an invalid option is given or the current directory
cannot be read.
dougLuna@dip:~/lpi/Exercicios$ pwd
/home/dougLuna/lpi/Exercicios
dougLuna@dip:~/lpi/Exercicios$ |
```


Introdução sobre variáveis de ambiente

Variável de ambiente é uma variável de um sistema operacional que geralmente contém informações sobre o sistema, caminhos de diretórios específicos no sistema de arquivos e as preferências do utilizador. Ela pode afetar a forma como um processo se comporta, e cada processo pode ler e escrever variáveis de ambiente.

Em todos os sistemas Unix, cada processo possui seu conjunto privado de variáveis de ambiente. **Por padrão, quando um processo é criado ele herda uma cópia das variáveis que foram exportadas no ambiente do processo pai.** Todos os tipos de Unix assim como o DOS e o Microsoft Windows possuem variáveis de ambiente; entretanto, variáveis para funções parecidas entre os sistemas possuem nomes distintos. Programas podem acessar os valores das variáveis de ambiente para efeitos de configuração.

Shell scripts e arquivos de lote usam variáveis para armazenar dados temporários e para comunicar dados e preferências a processos filhos.

No Unix, as variáveis de ambiente são normalmente inicializadas durante a inicialização do sistema, e no início de cada sessão. Esse assunto será estudado melhor no tópico 5.

As variáveis podem ser usadas tanto por scripts quanto pela linha de comando.

São geralmente referenciadas usando-se símbolos especiais na frente ou nas extremidades no nome da variável. Por exemplo, **Unix usa-se o \$**. O **cifrão**, quando iniciamos o comando identifica que é uma variável de ambiente, se não colocarmos o cifrão ele apenas imprime na tela o conteúdo, como vimos na aula anterior:

Se executarmos o comando abaixo, ele nos retornará o caminho de todos os paths da variável no sistema:

```
$ echo $PATH
```

```
dougluna@dip:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

Sem o cifrão ele apenas imprime a palavra na tela:

```
$ echo PATH
```

```
dougluna@dip:~$ echo PATH
PATH
dougluna@dip:~$
```

Como declarar uma variável

Para declarar (criar) uma variável no ambiente Linux seguimos o princípio de qualquer linguagem de programação, **primeiros damos o nome e depois o valor**. Exemplo no terminal:

```
$ NOME_VARIAVEL=valor
```

Acima dizemos que o nome da variável é **NOME_VARIAVEL** e depois declaramos o seu valor, ou seja, NOME_VARIAVEL é o mesmo que **valor**.

```
dougluna@dip:~$ NOME_VARIAVEL=valor
dougluna@dip:~$ echo $NOME_VARIAVEL
valor
dougluna@dip:~$
```

Para ficar mais claro, vamos declarar mais uma variável:

```
$ CURSOLINUX=lp1
```

Se digitarmos no terminal o comando abaixo ele nos mostrar o valor de atribuímos a variável **CURSOLINUX** que é *lp1*:

```
$ echo $CURSOLINUX
```

```
dougluna@dip:~$ CURSOLINUX=lp1
dougluna@dip:~$ echo $CURSOLINUX
lp1
dougluna@dip:~$
```

Note que, primeiro declaramos a variável e logo depois consultamos qual o valor dela.

Variáveis Locais e Globais

Importante frisar que, quando declaramos uma variável da forma que é explicado acima, ele fica disponível e visível somente em âmbito local, ou seja, a variável só é visível na sessão do shell que está aberta, quaisquer sessões abertas posteriormente não encontrarão as variáveis criadas.

Vejamos, como já criamos as variáveis **NOME_VARIAVEL** e **CURSOLINUX** vamos iniciar um novo bash dentro do próprio bash que está aberto, lembrando que estas variáveis ainda estão em nível local:

```
$ bash
```

```
dougluna@dip:~$ bash  
dougluna@dip:~$
```

Note que estamos em outra seção, agora faça o teste e tente consultar o valor das variáveis **NOME_VARIAVEL** e **CURSOLINUX**:

```
$ echo $NOME_VARIAVEL
```

```
$ echo $CURSOLINUX
```

```
dougluna@dip:~$ echo $NOME_VARIAVEL  
  
dougluna@dip:~$ echo $CURSOLINUX  
  
dougluna@dip:~$
```

Não nos voltará nenhum valor, pois estas variáveis estão armazenadas apenas localmente para aquele usuário e aquela seção anterior. Para voltar ao **bash** anterior digite “**exit**”. Isso vale também caso o **bash** seja fechado, ou seja, se fecharmos o terminal as variáveis também serão apagadas.

export

Para que os processos filhos consigam ver essas variáveis (globais) usamos o comando **export** (export NOME_VARIAVEL), pois só após a exportação dessas variáveis é que os processos filhos poderão enxergar as mesmas, seguindo a hierarquia. Toda seção que for aberta durante o bash atual, será “filha” dela, assim herdando as variáveis. **O comando export só funciona para processos que são originados a partir do bash atual.** Cuidado para não confundir com um novo terminal, pois é totalmente independente.

Vamos declarar a variável TESTE=Linux:

```
$ TESTE=Linux
```

```
dougluna@dip:~$ TESTE=Linux
dougluna@dip:~$ echo $TESTE
Linux
dougluna@dip:~$
```

Agora, vamos usar o arquivo da aula para testar a variável, o arquivo se chama Script_Variavel.sh, vamos ver o que tem dentro dele:

```
$ cat Script_Variavel.sh
```

```
dougluna@dip:~$ cat lpi/Exercicios/Script_Variavel.sh
#!/bin/bash
echo "O script le e imprime o valor da variavel TESTE"
echo " "
echo "O valor da variavel TESTE é:" $TESTE


dougluna@dip:~$
```

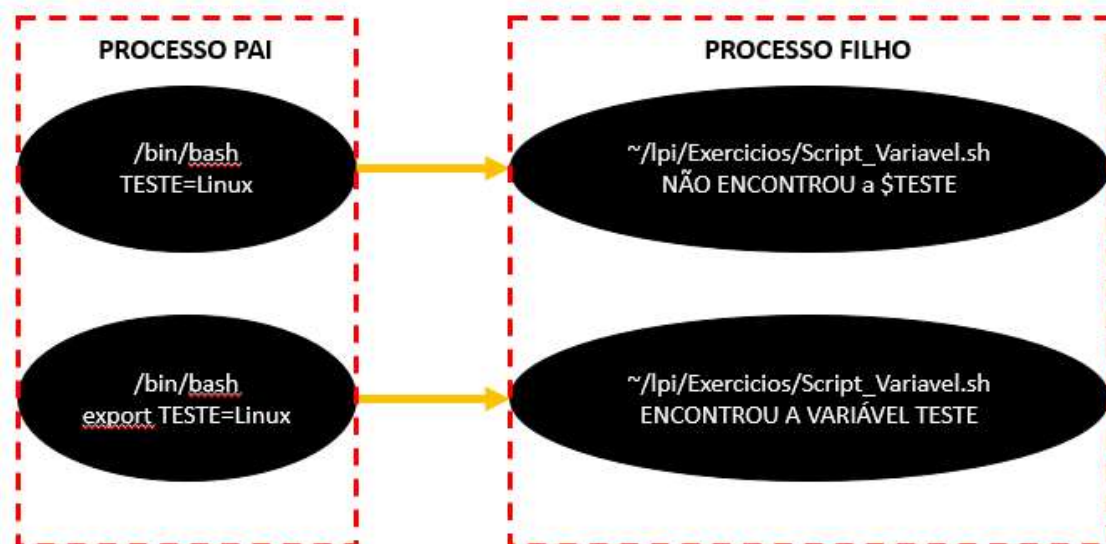
Podemos ver que durante o script há um campo chamando o valor da variável TESTE, vamos lembrar que quando tem o **cifrão “\$”** quer dizer que é uma variável.

Se fizermos o comando, vamos ver que ficará um campo sem o valor da variável.

```
$ ./Script_Variavel.sh
```

```
dougluna@dip:~$ lpi/Exercicios/Script_Variavel.sh
O script le e imprime o valor da variavel TESTE

O valor da variavel TESTE é:  
dougluna@dip:~$
```



Isso acontece porque a variável está local, então somente o bash atual consegue encontrá-lo. Para que toda seção a partir do atual passe a enxergar a variável **TESTE** temos que fazer um **export**.

```
$ export TESTE
```

```
dougluna@dip:~$ export TESTE
```

Agora, se rodarmos novamente o script do exercício temos o seguinte resultado:

```
dougluna@dip:~$ lpi/Exercicios/Script_Variavel.sh
O script le e imprime o valor da variavel TESTE

O valor da variavel TESTE é: Linux ✓
dougluna@dip:~$
```

Podemos criar a variável e depois fazer o export dela, ou podemos criar e exportar diretamente uma variável com o comando abaixo.

```
$ export NOME_VARIAVEL=valor
```

```
dougluna@dip:~$ export SO=Ubuntu
dougluna@dip:~$ bash
dougluna@dip:~$ echo $SO
Ubuntu
dougluna@dip:~$
```

Verificar as variáveis

set

O comando **set** **exibirá todas as variáveis, as locais e as globais declaradas no bash atual**. Ou seja, todas que foram iniciadas automaticamente, que foram declaradas manualmente, e que são ou não exportadas.

```
$ set | less
```

```
doug luna@dip:~$ set | less
```

```
PWD=/home/doug luna
SHELL=/bin/bash
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
SHLVL=1
SO=Ubuntu
TERM=xterm-256color
TESTE=Linux
UID=1000
USER=doug luna
WSENv=WT_SESSION: :WT_PROFILE_ID
```

Ainda não estudamos o comando parâmetro **less**, virá nas próximas aulas. Mas ele é usado para minimizar a quantidade de informações na tela. Após o comando, veremos várias variáveis de ambiente inclusive aquelas que declaramos.

env

O comando **env** mostrará apenas as variáveis globais.

Diferenças os comandos entre set e env

A grande diferença entra o **set** e o **env** é que o **set** é um comando de origem do **bash**, já o **env** é um comando de uma aplicação externa, por isso ele enxerga somente as globais.

```
dougluna@dip:~$ type env
env is hashed (/usr/bin/env)
dougluna@dip:~$ type set
set is a shell builtin
dougluna@dip:~$
```

Vamos declarar uma variável ABC com o valor cde (ABC=cde).

```
dougluna@dip:~$ ABC=cde
```

Agora, vamos dar o comando para visualizar as variáveis locais e globais, o **set**.

```
$ set | less
```

De cara, já conseguimos ver a variável que declaramos:

```
ABC=cde ←
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdh
iiranges:histappend:intera
BASH_ALIASES=()
```

Agora, vamos fazer a mesma coisa, porém com o comando **env**:

```
$ env | less
```

```
SHELL=/bin/bash
WSL_DISTRO_NAME=Ubuntu-20.04
WT_SESSION=10d2a817-8937-4dda-816a-1d939501c603
NAME=dip
PWD=/home/dougluna
LOGNAME=dougluna
HOME=/home/dougluna
```

Note que já não conseguimos ver a variável **ABC**, pois o comando **env** mostra somente as variáveis que são globais. Para que o comando **env** a encontre temos que fazer o **export**, veja:

```
$ export ABC=cde
```

```
doug luna@dip:~$ export ABC
doug luna@dip:~$ ou^C
doug luna@dip:~$ export ABC=cde
```

Agora vamos repetir o comando **env**:

```
$ env | grep ABC
```

```
doug luna@dip:~$ env | grep ABC
ABC=cde
doug luna@dip:~$
```

Isso acontece porque agora a variável está disponível globalmente.

Uma outra opção do **env** é que, ele pode alterar o valor uma variável de forma temporária, por exemplo, com o script do exercício, o valor da variável **TESTE** é Linux, vamos alterá-lo apenas na execução. (*Lembrando que estamos dentro do diretório **/lpi/Exercícios***):

```
$ env TESTE=Windows ./Script_Variavel.sh
```

```
doug luna@dip:~/lpi/Exercicios$ env TESTE=Windows ./Script_Variavel.sh
O script le e imprime o valor da variavel TESTE

O valor da variavel TESTE é: Windows
doug luna@dip:~/lpi/Exercicios$
```

Agora se executarmos novamente o comando de forma normal, ele apresentará o valor verdadeiro da variável.

```
$ ./Script_Variavel.sh
```

```
doug luna@dip:~/lpi/Exercicios$ ./Script_Variavel.sh
O script le e imprime o valor da variavel TESTE

O valor da variavel TESTE é: Linux
doug luna@dip:~/lpi/Exercicios$
```


Como remover variáveis criadas

Da mesma forma que temos como criar uma variável, também temos como removê-las do sistema. Para isso usamos o comando “**unset**”.

unset

Sempre que quisermos excluir uma variável usamos o comando “**unset**”. Para remover o valor da variável TESTE usaremos o comando:

```
$ unset TESTE
```

Agora, vamos consultar a variável com o comando echo e veremos que não temos mais o valor da variável TESTES disponível.

```
$ echo $TESTE
```

```
dougluna@dip:~$ echo $TESTE
Linux
dougluna@dip:~$ unset TESTE
dougluna@dip:~$ echo $TESTE

dougluna@dip:~$
```

Variáveis Pré-definidas do ambiente Linux

Ainda sobre variáveis vale frisar que existem variáveis que já são predefinidas no sistema Linux, as quais, são carregadas durante o sistema, na aula citamos algumas que inclusive podem ser cobradas no exame.

```
$ set | more
```

Aqui vão as Principais Variáveis de Ambiente:

- **DISPLAY**: Indica às aplicações gráficas onde as janelas deverão ser exibidas. Será estudado no Tópico 106.
- **HISTFILE**: Arquivo do histórico de comandos
- **HISTFILESIZE**: Quantidade de linhas/comandos armazenados no arquivo de histórico
- **HOME**: indica o diretório do usuário atual
- **LOGNAME e USER**: Nome do usuário atual
- **PATH**: Diretórios em que o Linux irá procurar por arquivos executáveis
- **PS1**: Aparência do prompt do shell.
- **PWD**: Diretório atual
- **OLDPWD**: Diretório anterior
- **SHELL**: Qual shell está sendo utilizado
- **TERM**: xterm - Mostra qual terminal estamos usando, no caso estamos usando interface gráfica. Caso optemos por logar sem passar por interface gráfica, aparecerá **TERM=tty**.

Podemos conferir todas as variáveis com o comando echo.

```
$ echo $TERM
```

```
$ echo $HISTFILE
```

Variáveis dinâmicas

Existem algumas variáveis de ambiente que são definidas dinamicamente pelo SHELL, é importante que conheçamos elas. Elas são identificadas pelo cifrão no início, por exemplo:

Este comando mostra o PID do processo atual.

```
$ echo $$
```

```
dougluna@dip:~$ echo $$  
9  
dougluna@dip:~$
```

Este comando mostra o PID do último processo que executamos em background. Vamos executar o comando top em background com o & e depois chamados o variável dinâmica \$!.

```
$ top &
```

```
$ echo $!
```

```
dougluna@dip:~$ top &  
[1] 411  
dougluna@dip:~$  
  
[1]+  Stopped                  top  
dougluna@dip:~$  
dougluna@dip:~$ echo $!  
411  
dougluna@dip:~$
```

O **PID do comando top** que foi o último a ser executado é o **mesmo** que o retorno da **variável dinâmica \$!**.

Este comando mostra o código de saída (exit code) do último do processo executado. Temos o retorno de 0, quando o último comando foi executado com sucesso e 1 ou 2 com foi com falha (qualquer número diferente de 0 significa falha).

```
$ echo $?
```

```
dougluna@dip:~$ type env
env is hashed (/usr/bin/env)
dougluna@dip:~$ echo $?
0 ✓
dougluna@dip:~$ type envp
-bash: type: envp: not found
dougluna@dip:~$ echo $?
1 ✗
dougluna@dip:~$
```

Ainda temos o “~”, que contém o /home do usuário, seja ele qual for.

```
$ echo ~
```

```
dougluna@dip:~$ echo ~
/home/dougluna
dougluna@dip:~$
```

Ainda podemos apontar outro usuário com o mesmo comando, e assim obtemos o home deste usuário:

```
$ echo ~root
```

```
dougluna@dip:~$ echo ~root
/root
dougluna@dip:~$
```

Quando executamos o comando **cd ~**, vamos direto para o /home do usuário atual. Material adicional:

```
$ cd ~
```

```
dougluna@dip:~/lpi$ pwd
/home/dougluna/lpi
dougluna@dip:~/lpi$ cd ~
dougluna@dip:~$ pwd
/home/dougluna
dougluna@dip:~$
```

Sequência de Comandos

Habitualmente no Linux alguns usuários geralmente digitam comando por comando, porém é possível fazer com que o sistema execute os comandos sequencialmente, e há várias formas de fazer isso, vejamos:

O separador “;” (Ponto e virgula)

A primeira forma que é explicada na aula é o separador “;”, através deles podemos executar um comando atrás do outro, ele tem uma particularidade que é, independente se o comando está certo ou errado ele é executado.

\$ clear ; date ; ls

```
dougluna@dip:~$ echo " "  
dougluna@dip:~$ clear ; date ; ls
```

```
Sun Jun  7 14:39:38 -03 2020  
PLP  lpi  teste.txt  
dougluna@dip:~$
```

Ele executou o **clear**, o **date** e depois o **ls**.

Sendo assim, ele executa um comando por vez não importando se o comando anterior, no caso **clear** esteja certo ou não, ele executará em os demais comandos.

O comando && (E comercial)

Diferente do “;” o && acusa erro e para a execução se o primeiro não comando não funcionar.

Por exemplo: Se o primeiro comando for executado com sucesso ele parte para o segundo comando, caso contrário ele para a execução acusando o erro.

\$ ls tmp/teste && echo Linux

Note que **primeiro** executamos o ls apontando o arquivo /teste dentro do diretório /tmp e em seguida ele deve executar o comando echo, caso não exista o arquivo dentro do diretório /tmp ele acusa erro e para a execução.

```
dougluna@dip:~$ ls /tmp/teste && echo Linux
ls: cannot access '/tmp/teste': No such file or directory
dougluna@dip:~$
```

O segundo comando não foi executado pois o primeiro deu erro.

IMPORTANTE: O comando && entende que:

Faça isso **E** isso= **faça ls && echo**

Quando o arquivo existe, ele apenas executa normalmente.

```
dougluna@dip:~$ touch /tmp/teste
dougluna@dip:~$ ls /tmp/teste && echo Linux
/tmp/teste
Linux
dougluna@dip:~$
```

O primeiro comando foi executado, então o valor do exit code será 0, assim o segundo também é executado com sucesso.

O comando || (Pipe)

O separador || (pipe pipe) faz o inverso do &&, ele executa o segundo comando caso o primeiro comando falhe. Vejam:

```
dougluna@dip:~$ ls /tmp/seg || echo Linux
ls: cannot access '/tmp/seg': No such file or directory
Linux
dougluna@dip:~$
```

Se o primeiro comando foi executado com erro, então o valor do exit code será 1 ou 2, assim o segundo é executado com sucesso.

Se o primeiro comando der certo ele ignora a regra do pipe e apenas executa o primeiro comando existente.

IMPORTANTE: O comando || entende:

Faça isso **OU** isso= **faça ls || echo**

```
dougluna@dip:~$ ls /tmp/teste || echo Linux
/tmp/teste
dougluna@dip:~$
```

Caso o primeiro foi executado com sucesso ele para e o segundo não é executado.

O comando !! (Exclamação)

O comando !! repete o último comando executado no bash.

```
$ !!
```

```
dougluna@dip:~$ date
Sun Jun  7 14:52:49 -03 2020
dougluna@dip:~$
dougluna@dip:~$ !!
date
Sun Jun  7 14:52:51 -03 2020
dougluna@dip:~$
```

Nesse caso digitamos o comando **date** e depois digitamos o **!!** **que executa novamente o nosso último comando que foi o date.**

```
dougluna@dip:~$ cat /tmp/teste
Comando !!
dougluna@dip:~$
dougluna@dip:~$ !!
cat /tmp/teste
Comando !!
dougluna@dip:~$
```


Repetição de comandos

history

O comando `history` lista todos os últimos comandos digitados no `bash`, cada usuário tem o seu arquivo de histórico.

```
dougluna@dip:~$ history
1  clear
2  ls
3  cd lpi/
4  cd ..
5  ls -la lpi/
6  cat /tmp/teste
7  history
```

Como vemos na imagem acima, o comando lista os últimos comando digitados, e na antes de cada comando há um número.

Outra forma que podemos executar novamente o comando é da seguinte forma:

```
$ !2
```

Sendo assim ele repetirá o comando 2, da lista acima.

```
dougluna@dip:~$ history
1  clear
2  ls
3  cd lpi/
4  cd ..
5  ls -la lpi/
6  cat /tmp/teste
7  history
dougluna@dip:~$ !2
ls
PLP lpi teste.txt
dougluna@dip:~$
```

Podemos utilizar o “!” seguido da string, por exemplo:

```
$ !ls
```

Dessa forma ele buscará o comando no history e o executará da o último comando que ele encontrou aquela string, nesse caso o “ls”.

```
dougluna@dip:~$ history
 1 clear
 2 ls
 3 cd lpi/
 4 cd ..
 5 ls -la lpi/
 6 cat /tmp/teste
 7 history
 8 ls
 9 ls -la lpi/exame-101/topico-103/
10 history
dougluna@dip:~$ !ls
ls -la lpi/exame-101/topico-103/
total 12
drwxr-xr-x 2 dougluna dougluna 4096 Jun  6 20:15 .
drwxr-xr-x 6 dougluna dougluna 4096 Jun  6 20:14 ..
-rw-r--r-- 1 dougluna dougluna  468 Jun  6 21:59 topico-103.txt
dougluna@dip:~$ |
```

Nesse caso o último comando que tinha a string **ls** era o número 9 `ls -la lpi/exame-101/topico-103`

Como limpar o arquivo de Histórico

Para limparmos o arquivo de histórico do nosso usuário executamos o comando abaixo:

```
$ history -c
```

```
dougluna@dip:~$ history -c
dougluna@dip:~$ history
 1  history
dougluna@dip:~$
```

Após executarmos a limpeza, vemos que só tem o último comando.

Podemos verificar onde estão armazenados os comandos digitados usando o comando `set` e filtrando pela variável de ambiente `HISTFILE`.

```
dougluna@dip:~$ set | grep HISTFILE
HISTFILE=/home/dougluna/.bash_history
HISTFILESIZE=2000
dougluna@dip:~$
```

Temos o path do arquivo onde o history é armazenado e tamanho máximo dele.

.bash_history

Os comandos digitados ficam armazenados no arquivo **.bash_history**, este arquivo é encontrado dentro da pasta do usuário, no meu caso em `/home/dougluna` como no exemplo:

```
dougluna@dip:~$ cat .bash_history
sudo apt-get update && sudo apt-get upgrade -y
which fp
ls
which gcc
sudo apt-get update
sudo apt-get install build-essential
```

Mas, não havíamos zerado o arquivo history?

Quando fazemos o **login**, o que está no `.bash_history` é carregado em memória, e o comando `history` funciona com o que está na memória, quando fazemos o **logout** é feito um `append` do que há de novo na memória para o arquivo `.bash_history`.

Ao usarmos o `history -c`, os comandos do `history` em memória é limpo, e quando você faz o `logout`, é feito um `append` de nada no `.bash_history`, ou seja, ele não é alterado. Para limpar definitivamente há 2 opções, executar os comandos abaixo, forçando que o `history` em memória, em branco, seja refletido no `.bash_history`:

```
$ history -c && history -w
```

Ou simplesmente limpar manualmente o arquivo:

```
$ cat /dev/null > ~/.bash_history
```

```
dougluna@dip:~$ cat /dev/null > ~/.bash_history
dougluna@dip:~$ cat .bash_history
dougluna@dip:~$
```

Pesquisando comandos digitados

No bash ainda temos a possibilidade de buscar comandos que já digitamos anteriormente, para abrir a caixa de pesquisa pressionamos o **ctrl+R**. Este procura os comandos dentro do seu histórico de comandos.

Conforme digitamos o comando desejado ele já mostra os comandos iniciados com aquela string. Encontrando o comando, basta teclar o enter.

```
dougluna@dip:~$  
(reverse-i-search)`c': |cat .bash_history
```

Conforme você vai digitando, ele vai completando com os comandos do seu history que possui aquela string.

Outro exemplo:

```
(reverse-i-search)`ls': |ls -la lpi/
```

Pesquisamos novamente com a string iniciando com ls, depois apertamos Enter e ele executa.

```
dougluna@dip:~$ ls -la lpi/  
total 32  
drwxr-xr-x 6 dougluna dougluna 4096 Jun  6 20:13 .  
drwxr-xr-x 9 dougluna dougluna 4096 Jun  6 23:51 ..  
drwxr-xr-x 2 dougluna dougluna 4096 May 18  2017 Exemplos  
drwxr-xr-x 2 dougluna dougluna 4096 Nov 27  2017 Exercicios  
drwxr-xr-x 6 dougluna dougluna 4096 Jun  6 20:14 exam-101  
drwxr-xr-x 2 dougluna dougluna 4096 Jun  6 20:13 exam-102  
-rw-r--r-- 1 dougluna dougluna 1171 Jun  6 20:44 infos.txt  
-rw-r--r-- 1 dougluna dougluna 3131 Jun  6 14:03 original.tgz  
dougluna@dip:~$ |
```

O Auto Completar (com TAB)

Essa função se dá quando digitamos um comando, seja ele para arquivo ou diretórios, ele completa o comando quando pressionamos a tecla tab. O auto completar tem suas particularidades.

Exemplo para verificar todos os comandos que iniciam com “ls”:

\$ ls + TAB + TAB

```
dougluna@dip:~$ ls
ls          lsb_release  lsipc       lsmod       lspgpot
lsasrv.dll  lsblk        lslocks     lsmproxy.dll lstelemetry.dll
lsasrv.mof  lscpu        lslogins    lsns        lsusb
lsass.exe   lshw         lsm.dll     lsof
lsattr      lsinitramfs  lsmem       lspci
dougluna@dip:~$ ls|
```

Com o comando ls + Tab + Tab ele encontra todos esses comandos.

```
dougluna@dip:~$ lsi
lsinitramfs  lsipc
dougluna@dip:~$ lsi|
```

Agora com o lsi + Tab + Tab ele encontra dois comandos

Isso também funciona com arquivos. Exemplo para verificar todos os arquivos que começam com “Script” no diretório lpi/Exercícios:

\$./S + TAB -> Completa para ./Script_

\$./Script_ + TAB + TAB

```
dougluna@dip:~$ cd lpi/Exercicios/
dougluna@dip:~/lpi/Exercicios$ ./Script_
Script_Exemplo.sh  Script_Variavel.sh
dougluna@dip:~/lpi/Exercicios$ ./Script_|
```

Primeiro digitamos **./S** e depois Tab, assim ele vai completar até onde começa a diferença, porque os dois arquivos começam com **Script_**. Se apertar Tab + Tab ele mostra as opções.

Se digitarmos E e depois Tab, ele vai autocompletar para o arquivo Script_Exemplo.sh

```
dougluna@dip:~/lpi/Exercicios$ ./Script_E
dougluna@dip:~/lpi/Exercicios$ ./Script_Exemplo.sh |
```

Outro exemplo se digitarmos “to” e existir apenas um comando que começa com essas letras, quando dermos um Tab ele vai autocompletar tudo, caso contrário daremos 2 Tab.

```
dougluna@dip:~$ to
toe          tokenbinding.dll  top          touch
dougluna@dip:~$ tou^C
dougluna@dip:~$ touch |
```

Agora se digitarmos tou e depois Tab, ele vai autocompletar para o comando touch, pois só existe esse comando com as iniciais “t” “o” “u”.

Comandos de ajuda

Um dos recursos de extrema importância do Shell são os comandos para obter ajuda. Entre eles temos o **man**, **info**, **whatis** e o **apropos**. Veremos mais sobre eles abaixo.

man

O comando **man** é o principal comando de ajuda. Ele mostra o manual de ajuda do comando, basicamente todos os comandos têm o seu manual de referência, e eles são acessados pelo man exemplo:

```
$ man ls
```

```
LS(1)                                User Commands                                LS(1)
NAME
  ls - list directory contents
SYNOPSIS
  ls [OPTION]... [FILE]...
DESCRIPTION
  List information about the FILES (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

  Mandatory arguments to long options are mandatory for short options too.

  -a, --all
      do not ignore entries starting with .

  -A, --almost-all
      do not list implied . and ..

  --author
      with -l, print the author of each file

  -b, --escape
      print C-style escapes for nongraphic characters

  --block-size=SIZE
      with -l, scale sizes by SIZE when printing them; e.g., '--block-size=M'; see SIZE format below
```

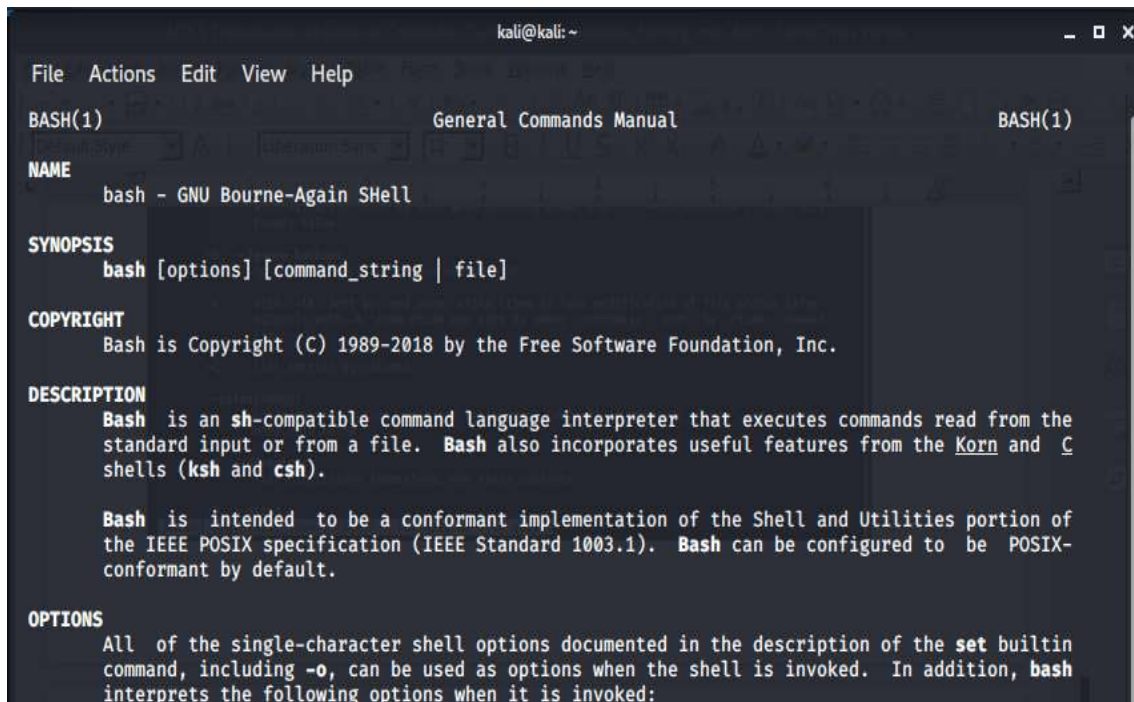
Pressione "q" para sair
e
/String para pesquisar

Abriu o manual de referência do comando ls.

Após o comando ser executado é aberto o manual de referência completo do comando ls.

Uma observação é que, quando o comando é interno, ou seja, faz parte do bash ele não possui o man, sendo assim temos que consultar o manual do bash:


```
$ man bash
```



```

kali@kali: ~
File Actions Edit View Help
BASH(1) General Commands Manual BASH(1)
NAME
  bash - GNU Bourne-Again SHell
SYNOPSIS
  bash [options] [command_string | file]
COPYRIGHT
  Bash is Copyright (C) 1989-2018 by the Free Software Foundation, Inc.
DESCRIPTION
  Bash is an sh-compatible command language interpreter that executes commands read from the
  standard input or from a file. Bash also incorporates useful features from the Korn and C
  shells (ksh and csh).

  Bash is intended to be a conformant implementation of the Shell and Utilities portion of
  the IEEE POSIX specification (IEEE Standard 1003.1). Bash can be configured to be POSIX-
  conformant by default.
OPTIONS
  All of the single-character shell options documented in the description of the set builtin
  command, including -o, can be used as options when the shell is invoked. In addition, bash
  interprets the following options when it is invoked:

```

Quando o comando é interno ele não tem um “man”, como o comando “cd”. Ele faz parte do bash, então você tem que consultar no manual do bash.

```

doug luna@dip:~$ type cd
cd is a shell builtin
doug luna@dip:~$ man cd
No manual entry for cd
doug luna@dip:~$

```

Outra forma de usar o man é com o parâmetro **-k**:

```
$ man -k "system information"
```

Dessa forma o comando traz qualquer referência que contenha o conteúdo “system information” e o comando do conteúdo informado. Veja:

```

doug luna@dip:~$ man -k "system information"
dumpe2fs (8)      - dump ext2/ext3/ext4 filesystem information
sysinfo (2)      - return system information
uname (1)        - print system information
doug luna@dip:~$

```

Novamente buscando a palavra touch:

```
$ man -k "touch"
```

```
dougluna@dip:~$ man -k "touch"
git-merge-tree (1)  - Show three-way merge without touching index
touch (1)           - change file timestamps
dougluna@dip:~$
```

info

Um pouco diferente do **man** é o comando **info**, ele basicamente é um man de forma reduzida. Tanto o **man** quanto o **info** são comandos para buscar ajuda sobre determinados comandos:

```
$ info vim
```

```
|VIM(1)                                     General Commands Manual      VIM(1)
NAME
    vim - Vi IMproved, a programmer's text editor
SYNOPSIS
    vim [options] [file ..]
    vim [options] -
    vim [options] -t tag
    vim [options] -q [errorfile]
    ex
    view
    gvim gview evim eview
    rvim rview rgvim rgview
DESCRIPTION
-----Info: (*manpages*)vim, 365 lines --Top-----
No menu item 'vim' in node '(dir)Top'
```

Info do comando vim, podemos notar abaixo a palavra “Info”

whatis e apropos

O comando **whatis** é baseado no comando. Ele te informa a descrição do comando que desejar:

```
$ whatis tcpdump
```

```
dougluna@dip:~$ whatis tcpdump
tcpdump (8)          - dump traffic on a network
dougluna@dip:~$
```

Sua sintaxe é simples: **whatis** *COMANDO*

Já o comando **apropos** faz a busca baseado na descrição e traz o comando referente a ela, assim como o **man -k**:

```
$ apropos "ssh"
```

```
dougluna@dip:~$ apropos "ssh"
authorized_keys (5) - OpenSSH daemon
git-shell (1)      - Restricted login shell for Git-only SSH access
rcp (1)            - OpenSSH secure file copy
rlogin (1)         - OpenSSH remote login client
rsh (1)            - OpenSSH remote login client
scp (1)            - OpenSSH secure file copy
sftp (1)           - OpenSSH secure file transfer
sftp-server (8)    - OpenSSH SFTP server subsystem
slogin (1)         - OpenSSH remote login client
ssh (1)            - OpenSSH remote login client
ssh-add (1)        - adds private key identities to the OpenSSH authentication agent
ssh-agent (1)      - OpenSSH authentication agent
ssh-argv0 (1)      - replaces the old ssh command-name as hostname handling
ssh-copy-id (1)    - use locally available keys to authorise logins on a remote machine
ssh-import-id (1)  - retrieve one or more public keys from a public keyserver and append them ...
ssh-import-id-gh (1) - retrieve one or more public keys from a public keyserver and append them ...
ssh-import-id-lp (1) - retrieve one or more public keys from a public keyserver and append them ...
ssh-keygen (1)     - OpenSSH authentication key utility
ssh-keyscan (1)    - gather SSH public keys from servers
ssh-keysign (8)    - OpenSSH helper for host-based authentication
ssh-pkcs11-helper (8) - OpenSSH helper for PKCS#11 support
ssh-sk-helper (8)  - OpenSSH helper for FIDO authenticator support
ssh_config (5)     - OpenSSH client configuration file
sshd (5)           - OpenSSH daemon
sshd (8)           - OpenSSH daemon
sshd_config (5)    - OpenSSH daemon configuration file
dougluna@dip:~$ |
```

Todos os comandos que possuem a palavra ssh na sua descrição.

uname

O comando **uname** imprime na tela certas informações do sistema. Podemos consultar o manual do comando com o **man**:

```
$ man uname
```

E ainda podemos consultar a ajuda referente ao comando, sendo assim, ele retornará todos os parâmetros que podem ser usados com o comando **uname**:

```
$ uname --help
```

```
dougluna@dip:~$ uname --help
Usage: uname [OPTION]...
Print certain system information.  With no OPTION, same as -s.

  -a, --all                print all information, in the following order,
                           except omit -p and -i if unknown:
  -s, --kernel-name        print the kernel name
  -n, --nodename            print the network node hostname
  -r, --kernel-release     print the kernel release
  -v, --kernel-version     print the kernel version
  -m, --machine            print the machine hardware name
  -p, --processor          print the processor type (non-portable)
  -i, --hardware-platform  print the hardware platform (non-portable)
  -o, --operating-system   print the operating system
      --help              display this help and exit
      --version            output version information and exit

GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Report uname translation bugs to <https://translationproject.org/team/>
Full documentation at: <https://www.gnu.org/software/coreutils/uname>
or available locally via: info '(coreutils) uname invocation'
dougluna@dip:~$ |
```

Quando usamos o comando mais as opções é impresso na tela a informação de acordo com o parâmetro usado:

```
$ uname -a
```

```
dougluna@dip:~$ uname -a
Linux dip 4.19.104-microsoft-standard #1 SMP Wed Feb 19 06:37:35 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
dougluna@dip:~$ |
```

Cada parâmetro traz uma informação, no caso do `-a` ele nos mostra todas as informações.

alias

Um alias, é uma forma de criar atalho para algum comando, quando digitamos alias no terminal, ele nos mostra alguns que já estão criados:

```
$ alias
```

```
dougluna@dip:~$ alias
alias alert='notify-send --urgency=low -i "${[ $? = 0 ]}&& echo terminal || echo error" "$(history|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*//;s/[\;]&]\s*alert$/'\''$)''"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
dougluna@dip:~$
```

Se observarmos, veremos que o alias é um atalho para comandos, na imagem acima vemos alguns exemplos como `ls= 'ls --color=auto'`

Para criar um alias de algum comando, basta digitamos no bash da seguinte forma para criar uma alias do comando `ls /tmp`:

```
$ alias lt='ls /tmp'
```

```
dougluna@dip:~$ alias lt='ls /tmp'
dougluna@dip:~$ alias
alias alert='notify-send --urgency=low -i "${[ $? = 0 ]} && echo terminal || echo error)" "$(history|tail -n1|sed -e '\''
s/^\s*[0-9]\+\s*//;s/[:;&]\s*alert$/\s*alert$/'\''")'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
alias lt='ls /tmp'
dougluna@dip:~$
```

Agora se digitamos no bash o comando alias criado “lt” ele listará o que tem no diretório /tmp.

```
dougluna@dip:~$ lt
remote-wsl-loc.txt          vscode-ipc-20cf611e-a0ce-4385-b617-383cb3203e0b.sock
teste                      vscode-ipc-87a510dd-357d-411f-b793-5c0300f4471c.sock
tmux-1000                  vscode-ipc-975ab080-6ce8-45f0-97c2-8adc5bcb4606.sock
vscode-git-04b6537808.sock vscode-ipc-b16d023c-9c24-48e1-957a-1d6b6ed4ee0e.sock
vscode-git-9a8507dac4.sock vscode-ipc-d357e272-0953-4267-bd5c-5583ff6e3f85.sock
vscode-git-a121c466ec.sock vscode-ipc-e6015748-06e8-4578-9768-9bdf882047e5.sock
vscode-git-ae4dc5266b.sock vscode-typescript1000
dougluna@dip:~$
```

Vale dizer que, os comandos criados são temporários, se desligarmos o sistema ou fecharmos a sessão eles são perdidos, em aulas posteriores veremos como tornar isso permanente.

unalias

O comando **unalias** pode ser utilizado para remover um alias ativo. Por exemplo:

\$ unalias lt

```
dougluna@dip:~$ alias | grep lt
alias lt='ls /tmp'
dougluna@dip:~$ unalias lt
dougluna@dip:~$ alias | grep lt
dougluna@dip:~$
```

- Primeiro executamos o comando alias usando o grep para filtrar o lt
- Depois usamos o unalias para remover o alias lt
- Verificamos o comando alias usando o grep para filtrar o lt e não temos mais retorno, pois ele foi removido.

which

O comando `which` tem ligação com o `PATH`, local onde o `bash` vai procurar os diretórios de um binário, **o `which` é uma ferramenta de busca que serve para localizar um comando, ele vai nos diretórios configurados na variável de ambiente `PATH` e encontra o valor que foi passado como parâmetro para ele.** Podemos usar o `whatis` para ter uma breve descrição do comando `which`:

```
dougluna@dip:~$ whatis which
which (1)          - locate a command
dougluna@dip:~$
```

Por exemplo queremos localizar o comando `echo`, executamos:

```
$ which echo
```

```
dougluna@dip:~$ which echo
/usr/bin/echo
dougluna@dip:~$
```

Ele pode localizar qualquer arquivo que estiver dentro desses diretórios do `PATH`. Em breve veremos mais comandos de localização.

Outro exemplo:

```
dougluna@dip:~$ which python3
/usr/bin/python3
dougluna@dip:~$ which mkdir
/usr/bin/mkdir
dougluna@dip:~$
```

Quoting

No bash temos uma série de caracteres especiais, por exemplo o “\$” que usamos com a variável, como no comando “echo \$PATH” para obtermos os diretórios do bash.

A **primeira coisa que o shell faz é interpretar as variáveis** e depois ele executa o comando com o resultado dessa interpretação de variáveis.

As **aspas duplas**, **aspas simples** e a **barra invertida** existem justamente para proteger, impedir e controlar a interpretação dessas variáveis.

Se executarmos o comando “**echo ***” a primeira coisa que o shell fará é interpretar o asterisco, o asterisco significa todos os arquivos do diretório local. Ao executar o comando veremos todos os arquivos, conforme abaixo:

```
$ echo *
```

```
dougluna@dip:~$ echo *  
PLP lpi teste.txt  
dougluna@dip:~$
```

Aspas Simples - Single Quotes

Incluir caracteres entre aspas simples (‘ ’) preserva o valor literal de cada caractere entre aspas. Uma aspas simples pode não ocorrer entre aspas simples, mesmo quando precedida por uma barra invertida. Exemplo:

```
$ echo ‘*’
```

```
dougluna@dip:~$ echo ‘*’  
*  
dougluna@dip:~$
```


Exemplo, criação a variável `TESTE=Curso` e vamos usar as aspas simples para proteger:

```
doug luna@dip:~$ TESTE=Curso
doug luna@dip:~$ echo $TESTE
Curso
doug luna@dip:~$ echo '$TESTE'
$TESTE
doug luna@dip:~$
```

Note que o CIFRÃO FOI IGNORADO e não temos o retorno da variável `TESTE`.

Aspas Duplas - Double Quotes

A inclusão de caracteres entre aspas duplas (" ") preserva o valor literal de todos os caracteres entre aspas, com exceção de:

'\$', - Cifrão

' ', - Crase

'\ ' - Barra invertida

e, quando a expansão do histórico estiver ativada, '!' . o shell está no modo POSIX (consulte Modo Bash POSIX), o '!' não tem significado especial entre aspas duplas, mesmo quando a expansão do histórico está ativada. Os caracteres '\$' e " 'mantêm seu significado especial entre aspas duplas (consulte Expansões do Shell). A barra invertida mantém seu significado especial somente quando seguido por um dos seguintes caracteres: '\$', "'", '"', '\', ou newline. Nas aspas duplas, as barras invertidas que são seguidas por um desses caracteres são removidas. As barras invertidas que precedem os caracteres sem um significado especial não são modificadas. Uma citação dupla pode ser citada entre aspas duplas precedendo-a com uma barra invertida. Se ativada, a expansão do histórico será realizada, a menos que um '!' Que apareça entre aspas duplas seja escapado usando uma barra invertida. A barra invertida que precede o '!' Não é removida.

Os parâmetros especiais '*' e '@' têm um significado especial entre aspas duplas (consulte Expansão dos parâmetros do shell).

Se eu fizer o comando `echo *` com o asterisco protegido por aspas duplas, o shell impede a execução do asterisco e imprime o asterisco conforme o comando `echo`. Exemplo:

```
$ echo "*"
"
```

```
dougluna@dip:~$ echo "*"
*
dougluna@dip:~$
```

Exemplo, criação a variável TESTE=Curso e vamos usar as aspas duplas para proteger:

```
dougluna@dip:~$ TESTE=Curso
dougluna@dip:~$ echo $TESTE
Curso
dougluna@dip:~$ echo "$TESTE"
Curso
```

Note que o CÍFRÃO NÃO FOI IGNORADO e temos o retorno do valor da variável TESTE.

Barra Invertida - Escape Character

Uma barra invertida não citada '`\`' é o caractere de escape do Bash. Ele preserva o valor literal do próximo caractere a seguir, com exceção da nova linha. Se um par `\newline` aparecer e a barra invertida em si não estiver entre aspas, a `\newline` será tratada como uma continuação de linha (ou seja, será removida do fluxo de entrada e efetivamente ignorada).

```
$ echo \*AJAJA
```

```
dougluna@dip:~$ echo \*AJAJA
*AJAJA
dougluna@dip:~$
```

Com vários caracteres, asterisco, cifrão e espaço, por exemplo, para ele proteger cada um desses caracteres:

```
$ echo \*\ \$
```

```
dougluna@dip:~$ echo \*\ \$
* $
dougluna@dip:~$
```

Protegemos o `*` asterisco, o espaço e o `$` cifrão, usando a barra invertida.