Laboratório 1.7: Bash - Shell, Programação em Scripts, Inicialização

Sobre o Bash

Conforme mencionado, o Bash é o interpretador de comando padrão da maioria das distribuições Linux. Você vem utilizando ele em todos os laboratórios anteriores e, neste momento, já deve estar bem expert nele, correto? *Provavelmente não*.

O Bash vai muito além de um simples interpretador de comandos. Ele é uma linguagem de programação completa que implementa e aprimora o que ficou conhecido como *Shell Command Language*, conhecido simplesmente como `sh`. Assim como o Bash, existem outros interpretadores de comando no Linux que também implementam o *sh* e outros que implementam outras linguagens. A seguir, citamos alguns:

Shell	Compatível com o `sh`?	Descrição
`Bourne Shell`	Sim	Foi, durante muito tempo, o shell padrão de todos os sistemas Unix. Uma das primeiras implementações do que ficou conhecido como *Shell Command Language* (`sh`). Diversos outros shells futuros (a seguir) foram diretamente baseados no código-fonte dele ou na linguagem criada.
`Bash`	Sim	*Bourne Again Shell*. Um dos shells mais populares, padrão na maioria das distribuições. Ele implementa o `sh` completo, mas possui uma série de outros aprimoramentos na linguagem. Foi criado para substituir o *Bourne Shell*.
`Ash`	Sim	*Almquist shell*. Criado com base no código-fonte do *Bourne Shell*. Ele implementa a linguagem `sh`, mas deixa de fora os aprimoramentos presentes em outros shells mais complexos, objetivando ser mais leve. Muito usado em sistemas Unix BSD, seu código-fonte serviu de base para o Dash, a seguir.
`Dash`	Sim	*Debian Almquist shell*. Um shell mais leve que implementa o `sh` clássico, sem melhorias, baseado no código-fonte do `ash`. Por ser mais leve e padrão, é muito usado em scripts no lugar do Bash, em especial na

		inicialização de sistemas baseados no Debian (e.g., Ubuntu, Mint).
`Ksh`	Sim	*KornShell*. Usado em sistemas Unix. Baseado originalmente no código do *Bourne Shell*, implementa o `sh` com algumas melhorias em cima da linguagem.
`Zsh`	Sim	Mais um shell baseado no *Bourne Shell*, com algumas melhorias. É o shell atualmente em uso no MacOS, substituindo o Bash em 2019.
`Csh`, `Tcsh`	Não	Um shell cuja linguagem de programação se parece mais com a linguagem C, considerado mais fácil de ser lido que o Bash. Foi um dos primeiros shells a terem recurso de autocompletar comandos.

Na maioria dos scripts, o interpretador utilizado é o `/bin/sh`. Dependendo da distribuição Linux, este comando pode ser um link simbólico para algum outro interpretador compatível. Veja qual é o caso do seu sistema:

```
$ ls -lh /bin/sh
```

```
lrwxrwxrwx 1 root root 4 Aug 19  2021 /bin/sh -> dash
```

Apesar da maioria dos scripts usarem o `/bin/sh`, o shell realmente usado pelos usuários é o `bash`, como veremos na próxima seção. Devido a isso, neste laboratório, focaremos mais no Bash, por ser também o interpretador de comandos padrão na maioria das outras distribuições e por possuir uma linguagem de programação com recursos mais completos.

1. Interpretadores de Comando

Conforme mencionado, o shell realmente usado pelos usuários é o `bash`. Você pode verificar isso através do comando abaixo, que lista e conta os diferentes shells observados no `/etc/passwd`:

```
$ cat /etc/passwd | cut -f 7 -d: | sort | uniq -c
```

```
 3 /bin/bash
   2 /bin/false
   1 /bin/sync
  26 /usr/sbin/nologin
```

O comando anterior usa o comando `cut` para selecionar apenas o sétimo campo do `passwd` (o shell dos usuários). Em seguida, o `sort` é usado para ordenar as linhas e, por

fim, o comando `uniq` seleciona apenas as linhas únicas, mostrando a quantidade de repetições de cada uma.

Além do `bash`, você pode ver que o comando `false` é usado como "shell" por alguns usuários do sistema. Ele não faz nada, apenas retorna um erro para o sistema operacional:

```
$ /bin/false                # Executa o comando
$ echo $?                   # Imprime o retorno do último comando para
o sistema operacional
$ /bin/true                 # Executa outro comando
$ echo $?                   # Compare com a saída to comando true
```

```
echo $?                  # Imprime o retorno do último comando
para o sistema operacional
1
```

No sistema operacional, quando um comando retorna 0 (zero), isso é um sinal de sucesso. Qualquer coisa diferente de zero indica um erro e o erro específico é identificado através do valor retornado.

Já o "shell" `/usr/sbin/nologin` é usado pela maioria dos usuários de sistema como mais um mecanismo de segurança. Ele retorna *false* também, mas imprime uma mensagem de erro tanto na saída padrão quanto nos logs de segurança do sistema. Você pode executá-lo para ver o que acontece:

```
$ /usr/sbin/nologin         # Executa o comando
$ echo $?                   # Imprime o retorno do último comando para
o sistema operacional
```

```
This account is currently not available.
```

Veja o erro de segurança sendo reportado nos logs do sistema:

```
$ cat /var/log/auth.log | grep nologin | tail -1
```

```
May  9 23:58:33 tardis nologin: Attempted login by fbd (UID:
1014) on /dev/pts/24
```

2. Atalhos no Terminal

Navegação

Ao usar um terminal, você já deve ter notado que você pode usar as setas do teclado (direita e esquerda) para navegar e editar o comando atual. Se você pressionar a tecla `Ctrl` enquanto pressiona as setas, o cursos irá navegar por *palavras*, ao invés de caracteres. Vá no terminal e teste isso em alguns dos comandos anteriores. Além disso, no terminal, você pode usar as setas para cima e para baixo para navegar entre os comandos anteriores (histórico).

Por fim, as teclas de `home` e `end` também podem ser usadas para ir para o início e fim da linha.

Autocompletar Comandos

O Bash possui um dos sistemas de autocompletar comandos mais avançados, o que é um dos fatores que o fazem tão popular. Autocompletar um comando é basicamente começar a digitar algo e, em seguida, digitar a tecla `TAB` para solicitar ao shell que ele deduza e escreva o restante do comando. Isso funciona não só para nome de comandos, mas também para nome de arquivos, diretórios e até mesmo opções de comandos!

Vá no terminal, digite apenas `add` e pressione a tecla `TAB` duas vezes seguidas. Cole abaixo o resultado que saiu no terminal.

```
addgnupghome addr2line
addgroup add-shell
```

Note como ao pressionar `TAB`, se existirem vários comandos com o mesmo início, o Bash não faz nada. Se você pressionar `TAB` novamente, ele mostra a lista de comandos possíveis. Se você continuar digitando mais letras, por exemplo, a letra `u`, e pressionar `TAB` novamente, o Bash saberá que único comando possível é o `adduser` e irá autocompletar o seu comando. Neste caso, você deixou de digitar 2 caracteres, uma redução em quase 30% do tempo!

O recurso de autocompletar pode ser usado também para nomes de arquivos.

Vá no terminal, digite apenas `cat /etc/add` e pressione a tecla `TAB`. Cole abaixo o resultado do recurso de autocompletar.

```
cat /etc/adduser.conf    # Uma redução de quase 40% do tempo!
```

Busca de Comandos

Outro atalho muito útil é o `Ctrl+r`. Ele permite pesquisar entre os comandos anteriores, facilitando encontrar comandos digitados há vários dias mais rapidamente. Basta digitar `Ctrl+r` e começar a digitar as partes do comando que você lembra. Por exemplo, no laboratório anterior, usamos o comando `adduser`, tente encontrar o comando completo digitando apenas o seu início.

Atalhos de Edição

Outro atalho interessante é o `Ctrl+k`. Ele apaga todos os caracteres após o atual. Por exemplo, escolha um dos comandos anteriores (seta para cima), vá até o meio do comando (seta para a esquerda) e pressione `Ctrl+k`. Se você quiser fazer o contrário, ou seja, apagar tudo que está antes do cursor, pressione `Ctrl+u`.

3. Scripts Bash

Conforme mencionado, o Bash é um interpretador da linguagem de programação *Shell Command Language*, com algumas melhorias próprias. Nesta seção, mostraremos rapidamente a sintaxe dessa linguagem e como criar um script Bash.

Shebang

Todo script no Linux/Unix, começa com uma linha especial que diz qual é o interpretador que deve ser usado para executar o script. Essa linha é conhecida como *shebang* e possui a sintaxe abaixo:

```
#!interpretador [opções]
```

Você pode ver exemplos do *shebang* em qualquer script do sistema:

```
$ head -1 /etc/init.d/cron
```

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

Hello World

Vamos criar o nosso (quase) primeiro script bash:

```
$ nano HelloWorld.sh            # Inclua nele o conteúdo abaixo
```

```
#!/bin/bash

echo "Hello World!"
```

Note o *shebang* na primeira linha indicando o `bash` como interpretador. Note também que o comando usado para imprimir uma mensagem na tela é o `echo`. Uma vez criado o script, antes de executá-lo pela primeira vez, precisamos dar a permissão de execução para o arquivo:

```
$ chmod 700 HelloWorld.sh
```

Aí sim, podemos executá-lo:

```
$ ./HelloWorld.sh
```

```
Hello World!
```

Variáveis

Uma convenção usada nos scripts (apesar de não ser obrigatório) é que os nomes das variáveis são escritos em maiúsculas. Assim como em outras linguagens, para declarar uma variável é só usar o operador de atribuição igual (=). Entretanto, diferentemente das outras linguagens, não pode haver espaços antes ou depois do igual. Para acessar o valor da variável, usamos o símbolo de dólar na frente dele (e.g., `$PERSON`).

```
$ nano HelloWorld.sh            # Substitua nele o conteúdo abaixo
```

```
#!/bin/bash

PERSON="Barney Calhoun"

echo "Hello, $PERSON!"
```

```
$ ./HelloWorld.sh
```

```
Hello, Barney Calhoun!#!/bin/bash
Hello World!
```

Estrutura Condicional

A sintaxe das estruturas da linguagem (condicionais, repetições) é talvez uma das mais diferentes em relação às linguagens tradicionais. O exemplo abaixo permite deduzir a sintaxe de uma estrutura condicional (`if`):

```
$ nano HelloWorld.sh            # Substitua nele o conteúdo abaixo
```

```
#!/bin/bash

echo -n "Digite um número: "
read NUMBER

if [ $NUMBER -gt 0 ]; then
    echo "O número $NUMBER é maior que zero!"
elif [ $NUMBER -eq 0 ]; then
    echo "O número $NUMBER é zero!"
```

```
else
    echo "O número $NUMBER é menor que zero!"
fi
```

Note como a comparação de números é feita através dos operadores `-gt` (*grater than*), `-lt` (*less than*) e `-eq` (*equal*). O operador == é usado apenas para comparar strings.

```
$ ./HelloWorld.sh
```

O script anterior usou o comando do bash `read` para obter um valor da entrada padrão (teclado). Você pode usar o *pipe* também para enviar dados para o seu script:

```
$ echo "5"  | ./HelloWorld.sh
$ echo "0"  | ./HelloWorld.sh
$ echo "-8" | ./HelloWorld.sh
```

```
Digite um número: O número 5 é maior que zero!
Digite um número: O número 0 é zero!
Digite um número: O número -8 é menor que zero!
```

Estrutura de Repetição

A sintaxe da estrutura de repetição também é um pouco diferente. Mas, desta vez, não usaremos o script, mas sim a própria linha de comando. A ideia é mostrar que o interpretador de comando do seu terminal é exatamente o mesmo que interpreta os scripts executáveis.

A sequência de comandos abaixo é um loop infinito que fica imprimindo uma mensagem a cada segundo. Copie todas as linhas de uma vez e cole no seu terminal. Pressione `Ctrl+C` para cancelar a execução:

```
$ while [ 1 ] ; do
      echo "Loop infinito ..."
      sleep 1
  done
```

```
Loop infinito ...
Loop infinito ...
Loop infinito ...
--- Continua ---
```

Conclusão

Essa seção mostrou muito superficialmente como scripts em bash podem ser desenvolvidos. Caso tenha se interessado, você pode buscar mais informações na Internet ou analisar os vários exemplos de scripts disponíveis no seu sistema:

```
$ grep -r "^#\!/bin/sh" /etc 2> /dev/null
```

```
/etc/dhcp/dhclient-exit-hooks.d/hook-dhclient:#!/bin/sh
/etc/NetworkManager/dispatcher.d/hook-network-manager:#!/bin/sh
/etc/cron.daily/popularity-contest:#!/bin/sh
/etc/cron.daily/bsdmainutils:#!/bin/sh
/etc/cron.daily/apport:#!/bin/sh -e
--- Continua ---
```

4. Inicialização do Bash

Sempre que você abre um terminal e o bash é executado, ele lê um arquivo especial de configuração localizado no seu *home* chamado `.bashrc`.

Este arquivo é, na verdade, um script bash que é chamado para configurar a sua sessão. Edite-o e veja o seu conteúdo:

```
$ nano ~/.bashrc
```

Entretanto, como você pode ver, esse script não começa com o *shebang* (`#!/bin/bash`). O motivo disso é que o *shebang* resulta um novo shell sendo executado, não mudando, portanto, as configurações do terminal atual.

Para executar um script que queremos que ele mude as configurações do nosso terminal, usamos o comando `source`:

```
$ source ~/.bashrc
```

É exatamente isso que o bash faz quando ele é iniciado.

Para verificar que o arquivo é realmente executado, inclua o comando abaixo na última linha do seu `.bashrc`:

```
$ echo "Bash iniciado!"
```

Em seguida, abra um novo terminal para ver a mensagem.

Para algo mais interessante, instale os programas `fortune` e `cowsay`:

```
$ sudo apt install fortune cowsay
```

Em seguida, substitua a última linha (a do `echo` anterior) do seu `.bashrc` para as duas abaixo:

```
fortune | cowsay -n
echo  # Imprime uma linha em branco
```

Abra um novo terminal para ver o resultado.

Cole aqui a mensagem que aparece quando você abre um terminal novo.

```
< You will be advanced socially, without any special effort on your
part. >
 ------------------------------------------------------------------
------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

True happiness comes from closing 100 chrome tabs after solving an obscure programming bug. – Christina Zhu