DESCRIÇÃO

Ferramentas para automatização de tarefas em um sistema operacional Linux, empregando a ferramenta CRON e programação SHELL SCRIPT.

PROPÓSITO

Conhecer ferramentas de automatização do Linux essenciais para tarefas do dia a dia em servidores e estações de trabalho permitirá que tarefas rotineiras de administração e gerência de sistemas Linux sejam realizadas de forma mais eficiente e eficaz.

PREPARAÇÃO

Para melhor aproveitamento do curso, recomendamos dispor de uma máquina Linux para treinar com os exemplos apresentados. Caso não disponha de um computador com o Linux instalado, há duas formas simples de obtê-lo, sem precisar reinstalar seu computador:

> Mais simples e rápido: No Windows 10, instale o “Linux Subsystem” (da própria Microsoft). Em seguida, na loja Microsoft Store, instale uma distribuição Linux. Sugestão: Ubuntu Linux.
>
> Mais avançado: Instale uma máquina virtual utilizando um Hypervisor como o Virtualbox ou o Microsoft Hyper-V. Em seguida, baixe o CD de instalação de uma distribuição, à sua escolha, e instale-o na máquina virtual.

OBJETIVOS

MÓDULO 1

Definir agendamentos, por meio da ferramenta
CRON, para a execução automática de tarefas

MÓDULO 2

Definir SCRIPTS para a automatização de tarefas no terminal Linux, sem a necessidade de interação do usuário

MÓDULO 3

Empregar variáveis de ambiente e estruturas de decisão em SCRIPTS

MÓDULO 4

Esquematizar tarefas complexas em SCRIPTS com o uso de estruturas de repetição

INTRODUÇÃO

Imagine que você é o administrador de um servidor Linux, responsável por um importante serviço da empresa e, ao final de cada dia, um relatório sobre esse serviço deve ser enviado para os gestores. Para emitir esse relatório diário, são necessários alguns comandos e a contabilização de resultados. Em seguida, os números obtidos precisam ser digitados e enviados por e-mail.

A situação acima é muito comum e faz parte do dia a dia de muitos administradores, com tarefas rotineiras e repetitivas. Mas o cenário também descreve uma condição de baixa eficiência do trabalho, já que exige um esforço manual para realizar uma tarefa recorrente.

Pense em todo o tempo acumulado, ao longo de meses ou anos, para a realização de uma única e específica tarefa, repetidamente. Por fim, a pessoa responsável por essa tarefa deverá estar disponível nos dias e horários determinados, e sua ausência precisará ser suprida por outra pessoa, ou a tarefa não será realizada.

Para cenários assim, existem ferramentas capazes de automatizar tarefas em sistemas operacionais, como o Linux. Neste curso, aprenderemos como programar SCRIPTS para realizar tarefas concatenando sequências de comandos. Aprenderemos, também, a utilizar o serviço CRON para agendar a execução automática de processos.

MÓDULO 1

⦿ Definir agendamentos, por meio da ferramenta CRON, para a execução automática de tarefas

CONCEITOS

A necessidade de automatizar a execução de processos não é nova, e ainda nos anos 1970, nas primeiras versões do sistema operacional UNIX, foi criada uma ferramenta específica para essa função, que recebeu o nome de CRON.

De simples configuração, o CRON permite determinar dias e horários em que comandos serão executados, de forma automática e sem a intervenção de usuários.

FERRAMENTA MULTIUSUÁRIO

É importante entender que o CRON é uma ferramenta multiusuário. Na prática, isso significa que cada usuário no sistema operacional poderá ter sua configuração própria e independente dos demais usuários.

No CRON, os processos são sempre executados pelo usuário a quem pertence a configuração.

Por exemplo, considere duas configurações de CRON, aqui representadas com palavras:

★ EXEMPLO

– Usuário ‘root’

Executar todos os dias às 23h o comando: /usr/limpeza

- Usuário ‘bob’

Executar todos os domingos às 9h o comando: /home/bob/relatorio_semanal

O CRON se encarregará de executar, todos os dias, às 23h o comando ‘/usr/limpeza’.

Esse comando será executado pelo usuário ‘root’ no sistema operacional.

Da mesma forma, todos os domingos, às 9h, o comando ‘/home/bob/relatorio_semanal’ será executado pelo usuário ‘bob’, automaticamente, através do CRON.

📢 ATENÇÃO

Lembre-se: Como o CRON é um serviço multiusuário, sempre considere qual usuário executará os comandos configurados. Esteja atento para as permissões que serão necessárias para cada comando ser executado com sucesso.

COMO CONFIGURAR O CRON DE USUÁRIO

A configuração que cada usuário possui no CRON é chamada de **CRONTAB**. **Para manipular uma CRONTAB, usaremos o comando com esse mesmo nome.**

CRONTAB

O ‘TAB’ se refere a tabelas.

Para editar o CRONTAB, é usado o comando:

$ crontab -e

Observação:

Ao usar o comando ‘crontab’ pela primeira vez, poderá ser apresentada uma opção para escolha do editor de texto.

Assim como em quase tudo no Linux, o CRONTAB também é um arquivo texto.

💡 DICA

Escolha o editor com que se sinta mais confortável. Se não tiver certeza, prefira um editor mais simples como o ‘nano’.

$ crontab -e

no crontab for bob - using an empty one

Select an editor. To change later, run 'select-editor'.

1. /bin/nano <---- easiest
2. /usr/bin/vim.basic
3. /usr/bin/vim.tiny
4. /bin/ed

Quando terminar de editar o CRONTAB, você deverá salvá-lo. Se for o editor ‘nano’, use a combinação de teclas Crtl-X.

Ao abrir o editor, o CRONTAB poderá estar preenchido com diversas informações e instruções que as distribuições Linux incluem para auxiliar os usuários.

REGRAS E FORMATOS DO CRONTAB

Todas as linhas sem conteúdo são ignoradas pelo CRON.

Todas as linhas começadas com ‘#’ são tratadas como comentários e, portanto, ignoradas pelo CRON. Em geral, os editores mostrarão essas linhas com cores destacadas para facilitar a visualização.

Cada tarefa será configurada em uma (e somente uma) linha, que deverá conter os campos requeridos pelo CRON.

Cada linha possui 6 ou mais campos de configuração, separados por espaços.

Os cinco primeiros campos de cada linha representarão as configurações de tempo, ou seja, quando cada tarefa deverá ser executada.

O sexto campo e os demais representam o comando que será executado pelo CRON, no dia e horário estabelecido.

MINUTO

Número de 0 a 59, representando o minuto em que a tarefa será executada.

HORA

Número de 0 a 23, representando a hora em que a tarefa será executada.

DIA DO MÊS

Número de 1 a 31, representando o dia do mês.

MÊS

Número de 1 a 12 (1 – janeiro, 2 – fevereiro etc). Também são admitidas abreviações, em inglês, ‘jan’, ‘feb’, ‘mar’...

DIA DA SEMANA

0 → domingo

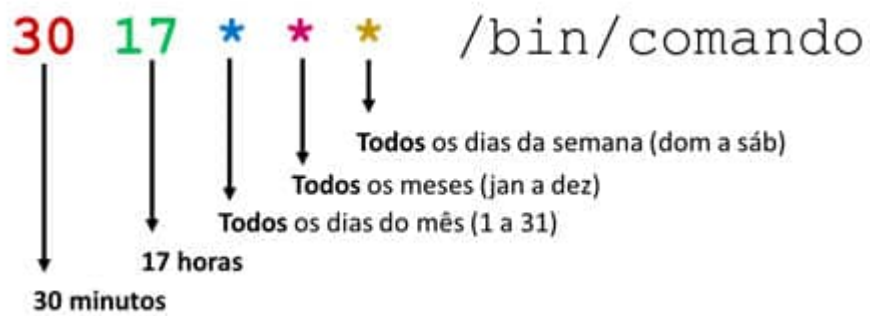
1 → segunda-feira

2 → terça-feira...

O dia da semana costuma causar confusão entre nativos da língua portuguesa, pois nesse idioma alguns dias da semana são numerais (segunda, terça, quarta...). Esteja atento para isso.

O CRON também admite o uso da máscara ‘*’ para representar ‘qualquer valor’.

Vejamos alguns exemplos:



O CRON executará ‘/bin/comando’ todos os dias, sempre às 17h30.

30 17 10 * * /bin/comando

O CRON executará ‘/bin/comando’ no dia 10 de todos os meses, às 17h30.

30 17 * 12 * /bin/comando

O CRON executará ‘/bin/comando’ todos os dias do mês de dezembro, às 17h30.

30 17 * * 5 /bin/comando

O CRON executará ‘/bin/comando’ nas sextas-feiras, às 17h30.

30 17 10 3 * /bin/comando

O CRON executará ‘/bin/comando’ no dia 10 de março, às 17h30.

30 17 * 3 1 /bin/comando

O CRON executará ‘/bin/comando’ todas as segundas-feiras do mês de março, às 17h30.

O CRON também permite incluir mais de um valor por campo e criar agendamentos sofisticados.

Valores separados por vírgula: A execução será disparada quando qualquer um dos valores for atendido.

Valores separados por traço: Indica intervalo, e o comando será executado quando qualquer um dos valores no intervalo for atendido.

0,15,30,45 * * * * /bin/comando

Nesse exemplo, o comando será executado nos minutos 0, 15, 30 e 45, todas as horas do dia, todos os dias.

0,15,30,45 0,12 * * * /bin/comando

Ao incluir as horas 0 e 12, o comando será executado todos os dias em 8 horários:

0h, 0h15, 0h30, 0h45, 12h, 12h15, 12h30 e 12h45.

0,15,30,45 0,12 * * 1-5 /bin/comando

O comando será executado nos horários do exemplo acima, porém somente de segunda a sexta-feira (1-5).

0,30 8-17 * * 1-5 /bin/comando

O comando será executado nos minutos 0 e 30, das horas 8 a 17 e somente de segunda a sexta-feira. Ex: 8h, 8h30, 9h, 9h30 .... 16h30, 17h e 17h30.

Também é possível definir o período entre intervalos usando o caractere ‘/’. Por exemplo:

0 */2 * * * /bin/comando

O comando será executado no minuto 0 a cada duas horas (/2): 0h, 2h, 4h, 6h ... 22h.

0/10 * * * * /bin/comando

O comando será executado a cada 10 minutos: 0h, 0h10, 0h20....

EXEMPLO DE UMA TAREFA AUTOMÁTICA DE BACKUP

Como vimos, os cinco primeiros campos de cada linha correspondem ao agendamento da execução. A partir do 6º campo você deve incluir o comando que será executado, com seus parâmetros, exatamente como faria em um terminal.

O comando ‘tar -cfz /tmp/backup.tar.gz /home/bob’ cria um arquivo TAR comprimido, com todo o conteúdo do diretório /home/bob. O nome desse arquivo será backup.tar.gz e estará dento do diretório /tmp. É um tipo de comando muito usado para fazer o backup de diretórios de usuários ou de aplicações em geral.

Se quisermos automatizar essa tarefa para que seja executada todos os dias às 20h, devemos incluir a linha abaixo no CRONTAB:

0 20 * * * tar -cfz /tmp/backup.tar.gz /home/bob

Lembre-se: o CRON é um serviço multiusuário. O usuário que for executar o comando ‘tar’ deverá ter, ao mesmo tempo, permissão para ler o conteúdo do diretório de origem (/home/bob) e permissão para escrever no diretório de destino (/tmp).

📢 ATENÇÃO

Importante: Não use o comando ‘sudo’ dentro do CRONTAB. O comando ‘sudo’ é interativo e requer a digitação de senha, o que não será possível enquanto estiver sendo executado pelo CRON. Em vez de usar o ‘sudo’, configure o comando diretamente no CRONTAB do usuário ‘root’.

A CONFIGURAÇÃO DE CRON PARA O SISTEMA

Além das configurações individuais por usuário, o CRON também pode ter uma configuração global, para o todo o sistema. Em geral, esse tipo de configuração é usado para rotinas de

serviços e processos do sistema.

★ EXEMPLO

Alguns exemplos:

Rotação periódica de arquivos de logs, apagando arquivos antigos.

Restart automático de processos que necessitem desse tipo de operação.

Verificação de atualizações disponíveis e notificação.

Verificação de dispositivos de hardware e notificação dos problemas encontrados.

O arquivo usado é o **/etc/crontab**. Como ele define uma configuração para todo o sistema, seu formato exige um campo a mais em cada linha, que é a indicação do usuário que executará cada comando.

30 10 * * * root /bin/comando

No exemplo acima, a indicação de que o CRON deverá executar o comando como o usuário ‘root’.

As principais distribuições Linux trazem a configuração global com uma organização prática, já que rotinas poderão ser incluídas e excluídas à medida que serviços são instalados e desinstalados no servidor.

Para que o arquivo /etc/crontab não precise ser alterado, é costume criar diretórios para as rotinas diárias, semanais e mensais. Em cada diretório serão criados programas (scripts) para cada uma das tarefas definidas.

E no arquivo principal, /etc/crontab, é configurada a execução de todos os SCRIPTS em cada um dos diretórios, nos horários agendados.

Esses diretórios costumam ter os nomes como abaixo:

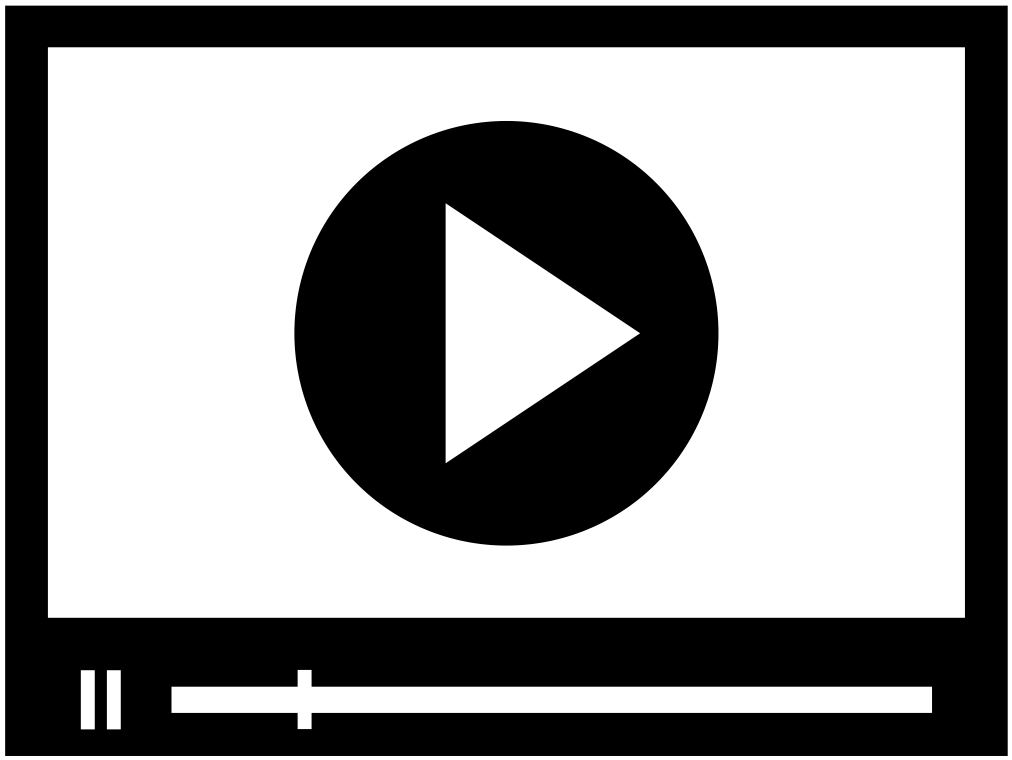
/etc/cron.hourly → Rotinas de hora em hora.

/etc/cron.daily → Rotinas diárias.

/etc/cron.weekly → Rotinas semanais.

/etc/cron.monthly → Rotinas mensais.

Para exercitar: Procure esses diretórios na sua máquina Linux e veja os seus conteúdos.

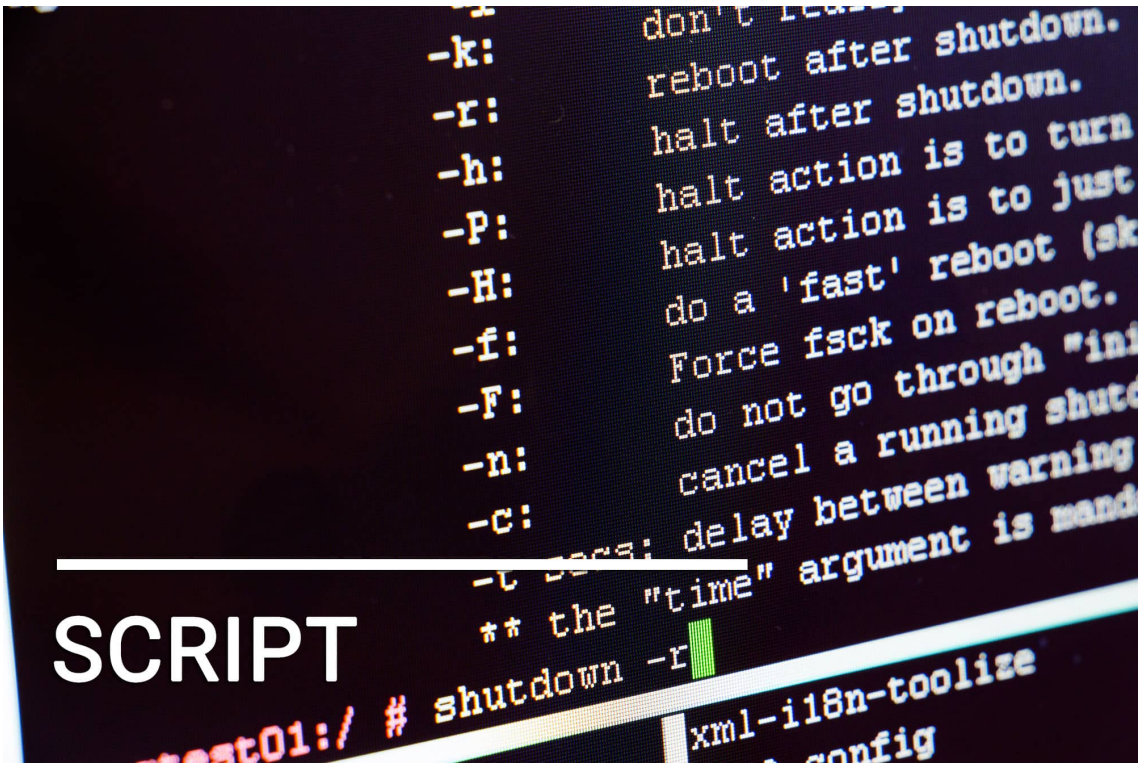


Assista ao vídeo abaixo e descubra, na prática, como agendar tarefas no Linux:

VERIFICANDO O APRENDIZADO

MÓDULO 2

⦿ Definir SCRIPTS para a automatização de tarefas no terminal Linux, sem a necessidade de interação do usuário



CONCEITOS

Uma das características do terminal (SHELL) de comandos dos sistemas baseados no Unix, incluindo o Linux, é a possibilidade de desenvolvimento de lógicas avançadas por meio de SCRIPTS.

Os SCRIPTS são arquivos-texto com sequências de comandos a serem executados pelo SHELL.

💬 COMENTÁRIO

Por serem lidos e interpretados diretamente pelo SHELL, não há necessidade de um processo de compilação como em diversas linguagens de programação. Após escritos, os arquivos contendo os SCRIPTS podem ser executados imediatamente.

O SHELL do Linux permite a criação de variáveis de ambiente e possui comandos para comparação e operação de valores, além de estruturas de repetição. Combinando esses recursos, podemos automatizar tarefas complexas.

Os SCRIPTS mais simples são aqueles que simplesmente executam diversos comandos em sequência, economizando nosso tempo. Os mais complexos podem analisar os resultados dos comandos e tomar decisões para os comandos seguintes, executando a tarefa com segurança, integridade e eficiência.

O INTERPRETADOR DE SHELL

Existem diversos interpretadores disponíveis para o Linux, e as sintaxes para SCRIPTS variam entre eles. Atualmente, as principais distribuições adotam o interpretador BASH como o padrão, portanto vamos utilizá-lo como referência nesse curso.

A PALAVRA MÁGICA QUE IDENTIFICA UM SCRIPT

O Linux nos permite executar um SCRIPT da mesma forma que executamos um comando ou programa qualquer, mas isso tem uma implicação: Precisamos informar que o conteúdo daquele arquivo é um SCRIPT e não um arquivo qualquer.

Para isso, todo SCRIPT deve ser iniciado com uma palavra mágica, composta por dois caracteres ‘#!’ seguidos do caminho para o interpretador de SHELL.

#!/bin/bash

Neste exemplo estamos declarando que o arquivo é um SCRIPT e deverá ser interpretado pelo BASH, sendo /bin/bash o caminho completo para esse programa interpretador.

UM SCRIPT MUITO SIMPLES

Vamos fazer um SCRIPT muito simples, contendo apenas um comando. O nome desse SCRIPT será ‘script1’ e criaremos em nosso diretório HOME.

1

2

3

4

1

Para começar, abrimos um editor de texto comum, como o ‘vi’ ou ‘nano’.

$ nano script1

2

Dentro do arquivo vamos incluir apenas duas linhas:

A palavra mágica.

O comando ‘date’, que apenas retornará a data e hora atual.

```
#!/bin/bash
date
```

Já percebemos que esse exemplo é um SCRIPT muito simples, sem grande utilidade, já que terá o mesmo efeito de simplesmente digitarmos o comando ‘date’. Mas o objetivo agora é rodar o nosso primeiro SCRIPT.

3

Salve o arquivo e saia do editor. Se estiver usando o ‘nano’, use Ctrl-X, confirme que deseja salvar o arquivo e escolha o nome (‘script1’).

4

Se agora usarmos o comando ‘ls’ para ver o conteúdo do diretório, encontraremos o nosso ‘script1’ entre os arquivos. Porém, se tentarmos digitar ‘script1’ no PROMPT, receberemos um erro do tipo “comando não encontrado”.

Há duas questões que ainda precisam ser tratadas:

1

A primeira delas é que criamos um arquivo texto, mas em momento algum autorizamos que ele seja executado, e essa é uma característica de segurança importante do Linux.

Não basta criar um arquivo texto para que ele se comporte como um programa, é necessário dar permissão de execução para ele.

Para dar essa permissão, usaremos o comando ‘chmod’.

$ chmod u+x script1

Nesse exemplo, concederemos a permissão de execução para o proprietário (OWNER) do arquivo. Caso deseje-se que qualquer usuário possa executá-lo, podemos usar o parâmetro ‘a+x’.

Ao usar o comando ‘ls -l’, veremos que a permissão foi concedida:

-rwxr--r-- 1 bob bob 17 Sep 14 14:50 script1

Se tentarmos executar o comando agora, ainda receberemos um erro, o que nos leva à segunda questão, outro mecanismo de proteção do Linux: **Apenas programas em alguns diretórios do sistema podem ser executados sem a necessidade de indicar o seu caminho.**

2

Como o SCRIPT foi criado no HOME e esse não é um dos diretórios permitidos, precisaremos indicar o caminho, que pode ser utilizando o caminho absoluto ou relativo.

Para executar o ‘script1’ que está no diretório atual, vamos empregar o caminho relativo, com o comando abaixo:

./script1

Onde ./ é um caminho que significa ‘o diretório atual’.

Ao executar o SCRIPT teremos no terminal a saída do comando ‘date’, como esperávamos.

$ ./script1
Mon Sep 14 16:29:45 -03 2020

LINHAS EM BRANCO E COMENTÁRIOS

❓ VOCÊ SABIA

As linhas sem conteúdo e as iniciadas com ‘#’ são ignoradas pelo interpretador.

As linhas iniciadas com ‘#’ podem ser usadas para comentários, observações e instruções sobre o SCRIPT.

Não confunda com a palavra mágica na primeira linha: Ela começa com ‘#’ e é importante.

INCLUINDO TEXTOS NA RESPOSTA DO SCRIPT

Nosso primeiro objetivo foi atingido: Criar um SCRIPT simples e executá-lo. O próximo passo é evoluir esse SCRIPT e mudar seu comportamento.

Nesse exemplo, vamos mudar a saída para indicar somente a hora e com um texto mais amigável. Desejamos que a saída seja assim quando o SCRIPT for executado:

Hora certa 12:30

Para fazer essa mudança, precisamos:

Inserir um comando para exibir o texto “Hora certa: “

Modificar o comando ‘date’ para mostrar a hora no formato HH:MM (onde HH é a hora com dois dígitos e MM o minuto, também com dois dígitos).

A mudança no ‘date’ é fácil. Consultando a MANPAGE do comando, vemos que é possível determinar a saída passando o formato desejado como argumento:

$ date +%H:%M

SAIBA MAIS

(%H será substituído pela hora e %M pelo minuto. Para mais formatos, consulte a MANPAGE do comando ‘date’).

O texto pode ser inserido usando-se o comando ‘echo’. Vamos mudar o SCRIPT para:

```
#!/bin/bash
echo "Hora certa"
date +%H:%M
```

O comando ‘echo’ envia para a saída padrão (stdout) o texto passado como parâmetro.

Ao executar, teremos como saída:

```
$ ./script1
Hora certa
12:30
```

As informações que desejamos já estão presentes, porém em linhas diferentes. Isso ocorre porque o comando ‘echo’ insere uma quebra de linha após o texto. É possível inibir a quebra de linha passando o parâmetro ‘-n’ para o comando ‘echo’. Porém, vamos adaptar o SCRIPT colocando tudo no mesmo ‘echo’.

```
#!/bin/bash
echo "Hora certa $(date +%H:%M)”
```

Quando colocamos o comando entre ‘$(‘ e ‘)’ estamos dizendo ao ‘echo’: Execute esse comando e envie o resultado junto ao texto para a saída. Então, ao executarmos o script, teremos:

```
$ ./script1
Hora certa 12:30
```

Evoluindo o nosso SCRIPT para informar a data e hora:

```
#!/bin/bash
echo "Data: $(date +%d/%m/%Y) Hora: $(date +%H:%M)"
```

Ao executar, teremos a resposta:

$ ./script1

Data: 15/09/2020 Hora: 16:44

Repare nesse último exemplo que o comando ‘date’ foi executado duas vezes dentro do ‘echo’.

INSERINDO CARACTERES ESPECIAIS NO COMANDO ‘ECHO’

O comando ‘echo -e’ permite o uso de caracteres especiais. Por exemplo, para adicionar uma quebra de linha, pode ser usada a combinação ‘\n’.

Ainda no nosso SCRIPT anterior:

```
#!/bin/bash
echo -e "Data: $(date +%d/%m/%Y)\nHora: $(date +%H:%M)"
```

Ao executar, teremos a resposta:

$ ./script1

Data: 15/09/2020

Hora: 16:44

CRIANDO PAUSAS NA EXECUÇÃO DE UM SCRIPT

Em muitas situações, pode ser interessante, ou necessário, incluir pausas na execução de um SCRIPT. O comando ‘sleep’ permite realizar uma pausa com tempo configurável.

Vamos criar o arquivo ‘script2’ para essa demonstração:

```
#!/bin/bash
date +%T
sleep 1
date +%T
sleep 2
```

```
date +%T
sleep 3
date +%T
```

No ‘script2’ o comando ‘date’ é executado 4 vezes. Entre eles foi usado o comando ‘sleep’ com diferentes tempos. O número passado como parâmetro ao ‘sleep’ corresponde ao tempo, em segundos, de pausa.

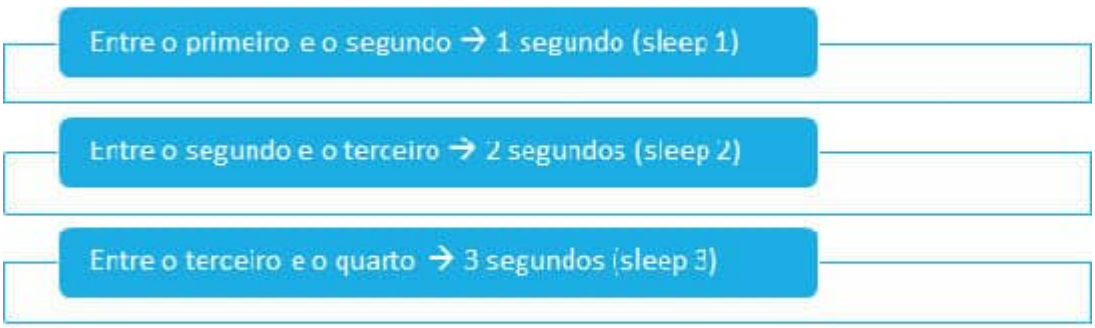
Lembrando: Para executar o ‘script2’, é necessário conceder permissão de execução.

```
$ chmod u+x script2
```

Ao executá-lo, obtemos:

```
$ ./script2
13:05:29
13:05:30
13:05:32
13:05:35
```

Repare, na indicação dos segundos, os intervalos do comando ‘sleep’, como programado.



Em muitos casos, pode ser interessante aguardar uma interação do usuário. O comando ‘read’ suspende a execução do SCRIPT até que o usuário tecle ENTER.

Substituindo no SCRIPT os comandos ‘sleep’ por ‘read’.

```
#!/bin/bash
date +%T
read
```

```
date +%T
read
date +%T
read
date +%T
```

O SCRIPT aguardará que o usuário digite ENTER após executar cada comando ‘date’. Comparando-se as horas, será possível saber quanto tempo o SCRIPT aguardou por cada interação do usuário.

```
$ ./script2
13:09:12

13:09:14

13:09:16

13:09:18
```

As linhas em branco foram inseridas pela resposta do comando ‘read’ ao ENTER do usuário.

💡 DICA

Experimente incluir o parâmetro ‘-s’ nos comandos ‘read’ para que as linhas em branco não apareçam.

APAGANDO TODO O CONTEÚDO DO TERMINAL

Às vezes, pode ser interessante apagar todo o conteúdo do terminal no início ou durante a execução de um SCRIPT, facilitando a visualização das respostas. Para isso, podemos usar o comando ‘clear’.

Modificando o SCRIPT anterior, foi incluído um comando ‘clear’ no início do arquivo:

```
#!/bin/bash
clear
date +%T
read
date +%T
read
date +%T
read
date +%T
```

O resultado será igual ao anterior, porém o terminal estará limpo e a resposta começará na primeira linha.

TERMINANDO UM SCRIPT COM VALOR DE RETORNO

Uma característica de processos no Linux é que eles terminam retornando um valor numérico para o sistema operacional, chamado de “return value” e que serve para indicar o sucesso da execução ou a ocorrência de erros.

💬 COMENTÁRIO

O valor de retorno pode ser de 0 a 255, e a regra geral é de que um valor de retorno 0 indica execução com sucesso. Os demais valores, de 1 a 255, podem ser usados para indicar o tipo de erro durante a execução.

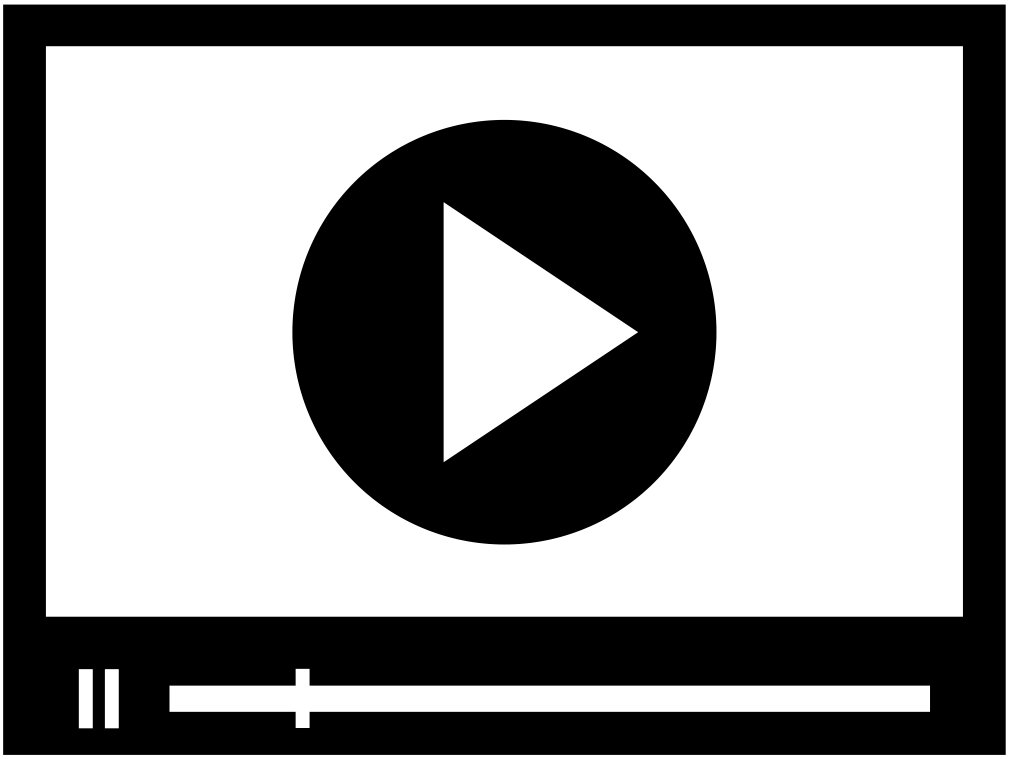
O comando ‘exit’ permite terminar o SCRIPT retornando um valor.

Por exemplo, se um SCRIPT contiver a linha abaixo:

```
exit 1
```

Quando o ‘exit’ for executado, o SCRIPT será imediatamente encerrado e com valor de retorno igual a ‘1’.

Se o comando ‘exit’ for executado sem o parâmetro de valor, ele retornará o resultado do último comando no SCRIPT. Isso também vale para um SCRIPT que não termina através do comando ‘exit’.

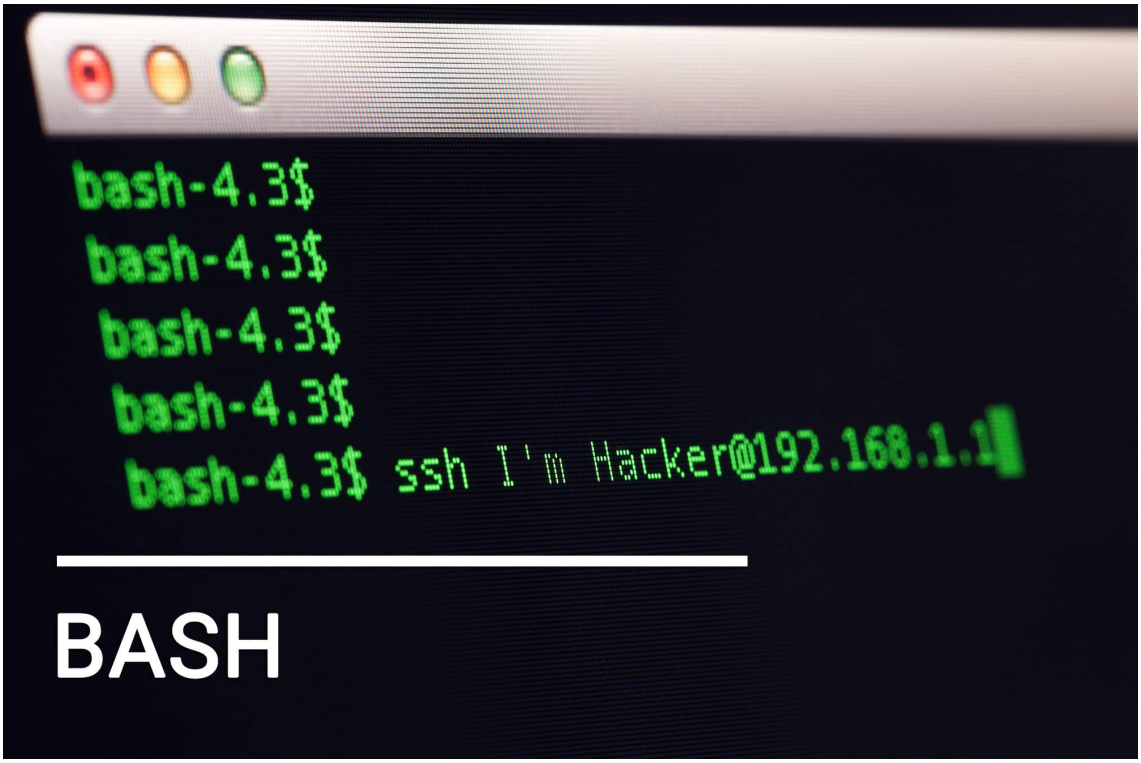


No vídeo a seguir, você encontrará comandos do Linux diferentes dos apresentados no tema, passando por todos os passos, desde a criação do script, alteração das permissões e a execução do script.

VERIFICANDO O APRENDIZADO

MÓDULO 3

⦿ Empregar variáveis de ambiente e estruturas de decisão em SCRIPTS



CONCEITOS

Assim como nas linguagens de programação, o terminal BASH também permite a criação de variáveis em seu ambiente.

💬 COMENTÁRIO

As variáveis são, de modo simplificado, um espaço para o armazenamento de informações.

Utilizando variáveis podemos compartilhar dados entre vários comandos e tomar decisões para desviar o fluxo de execução do SCRIPT.

Toda variável possui:

NOME

O nome de uma variável é a chave pela qual faremos referência a ela.

VALOR

O valor é a informação armazenada.

ATRIBUIÇÃO DE VALORES A UMA VARIÁVEL

Para declarar uma variável, atribuindo valor a ela, usamos a sintaxe abaixo:

VAR=1

Nesse exemplo, declaramos uma variável com o nome “VAR” e a ela foi atribuído o valor “1”. Se a variável VAR já existir, o valor anterior será perdido e substituído pelo novo.

📢 ATENÇÃO

Na atribuição acima não deve haver espaços antes ou depois do sinal de igualdade, o que faria o terminal interpretar como a tentativa de executar o comando ‘VAR’ ou ‘VAR=’

Para atribuir textos com espaços, é necessário o uso de aspas:

VAR=”Terminal do Linux”

O nome da variável é sempre case-sensitive, ou seja, o terminal distingue caracteres maiúsculos e minúsculos no nome das variáveis.

REFERENCIANDO VARIÁVEIS

Para fazer referência a uma variável, e obter o seu conteúdo, é necessário usar o símbolo $ à esquerda do nome. Se a variável se chama VAR, faremos referência a ela como $VAR.

Por exemplo, experimente usar esses comandos no seu terminal:

VAR=”Terminal do Linux”
echo “$VAR”

A saída do comando ‘echo’ será a frase “Terminal do Linux”. O ‘echo’ substituiu a referência $VAR pelo conteúdo da variável.

ATENÇÃO 1

ATENÇÃO 1

A variável só é utilizada sem o prefixo $ quando estamos atribuindo valor a ela. Para fazer referência, o $ sempre é necessário.

ATENÇÃO 2

ATENÇÃO 2

No terminal BASH, ao contrário da maioria das linguagens de programação, as variáveis não possuem tipo. Portanto, a interpretação do conteúdo como número ou texto vai depender do contexto da operação que está sendo realizada.

ATRIBUIÇÃO DA SAÍDA DE UM COMANDO A UMA VARIÁVEL

Suponha que você está desenvolvendo um SCRIPT no qual a informação retornada por um comando será utilizada posteriormente. Uma das formas de você armazenar essa informação temporariamente é por meio de variáveis.

No exemplo abaixo, simulamos um SCRIPT cuja execução é demorada. Para que o usuário saiba quanto tempo a execução demorou, esse SCRIPT apresentará no final os horários em que começou e o atual.

No início da execução, o horário será salvo em uma variável de nome “INICIO”.

```
#!/bin/bash
INICIO=$(date +%T)
```

```
echo "Executando uma tarefa demorada"
sleep 5

echo "Início: $INICIO / Término: $(date +%T)"
exit 0
```

💬 COMENTÁRIO

Repare na segunda linha que o comando ‘date’ é executado, retornando a hora naquele momento, que será armazenada na variável INICIO. O comando ‘sleep 5’ simula uma tarefa demorada, aqui com 5 segundos de duração.

Ao término da execução teremos os dois horários, o que nos permitirá saber quanto tempo o SCRIPT levou para executar:

./script3>
Executando uma tarefa demorada
Início: 17:47:00 / Término: 17:47:05

📢 ATENÇÃO

O conteúdo de variáveis é armazenado em memória, e serve muito bem para quantidades pequenas de dados. Se precisar guardar um volume grande, prefira utilizar arquivos temporários.

PASSANDO PARÂMETROS PARA O SCRIPT

O terminal do Linux permite que um SCRIPT seja executado com parâmetros. Assim, podemos passar valores para o SCRIPT a cada execução, sem precisar modificar o seu conteúdo.

Suponha um SCRIPT sendo executado com o comando abaixo, onde foram passados 4 parâmetros (lembre-se: Os parâmetros são separados pelo espaço).

$ **./ script4** valor1 valor2 valor3 valor4

Cada um dos parâmetros pode ser lido pelo SCRIPT consultando as variáveis $1, $2, $3 e $4, respectivamente. A variável $0 é o nome do próprio SCRIPT que foi executado.

Considere o ‘script4’:

```
#!/bin/bash
echo "Nome do script: $0"
echo "Parametro 1: $1"
echo "Parametro 2: $2"
echo "Parametro 3: $3"
echo "Parametro 4: $4"
```

Esse SCRIPT irá buscar os 4 primeiros parâmetros e os exibirá. Os parâmetros que não forem preenchidos serão vistos como vazio pelo SCRIPT.

Ao executar o SCRIPT, teremos a saída (não esqueça de conceder permissão de execução):

```
$ ./script4 valor1 valor2 valor3 valor4
Nome do script: ./script4
Parametro 1: valor1
Parametro 2: valor2
Parametro 3: valor3
Parametro 4: valor4
```

📢 ATENÇÃO

Para parâmetros acima do número 10 a referência deve ser feita assim:

${10}, ${11}, ${12} ...

O OPERADOR LÓGICO IF

Assim como quase todas as linguagens de programação, o BASH também permite comparações por meio do comando ‘if’.

O comando recebe uma expressão para ser avaliada, e seu retorno dependerá do resultado da avaliação.

A sintaxe básica é a seguinte:

if [CONDIÇÃO]
then
Alguma tarefa
fi

O comando ‘if’ avalia a condição apresentada.

VERDADEIRO

Se o resultado for verdadeiro, ele executará o código entre ‘then’ e ‘fi’, nesse exemplo representado pela frase “# Alguma tarefa #”.

FALSO

Se o resultado for falso, o fluxo da execução é desviado para depois do ‘fi’, portanto os comandos em “# Alguma tarefa #” **não** são executados.

O operador ‘if’ também permite o uso da declaração ‘else’, que será executada sempre que o resultado da condição for falso.

if [CONDIÇÃO]
then
# Alguma tarefa executada se a condição for verdadeira #
else
# Alguma tarefa executada se a condição for falsa #
fi

COMPARADORES NUMÉRICOS

Observe os comparadores numéricos

IGUALDADE (-EQ → IS EQUAL)

if [[“$A” -eq “$B”]]

Resulta verdadeiro se o valor da variável $A for igual ao da variável $B.

DIFERENTE (-NE → IS NOT EQUAL)

if [[“$A” -ne “$B”]]

Resulta verdadeiro se o valor da variável $A for diferente do da variável $B.

MAIOR QUE (-GT → IS GREATER THAN)

if [[“$A” -gt “$B”]]

Resulta verdadeiro se o valor da variável $A for maior que o valor da variável $B.

MAIOR OU IGUAL (-GE → IS GREATER THAN OR EQUAL)

if [[“$A” -ge “$B”]]

Resulta verdadeiro se o valor da variável $A for maior que o valor da variável $B ou igual.

MENOR QUE (-LT → LESS THAN)

if [[“$A” -lt “$B”]]

Resulta verdadeiro se o valor da variável $A for menor que o valor da variável $B.

MENOR OU IGUAL (-LE → IS LESS THAN OR EQUAL)

if [[“$A” -le “$B”]]

Resulta verdadeiro se o valor da variável $A for menor que o valor da variável $B ou igual.

Para exemplificar, considere esse ‘script5’ que recebe dois valores como parâmetros e realiza uma série de comparações:

```
#!/bin/bash

A=$1
B=$2

if [[ "$A" -eq "$B" ]]
then
echo "A e B são iguais"
fi

if [[ "$A" -ne "$B" ]]
then
```

```
echo "A e B são diferentes"
fi

if [[ "$A" -gt "$B" ]]
then
echo "A é maior que B"
fi

if [[ "$A" -lt "$B" ]]
then
echo "A é menor que B"
fi
```

O primeiro parâmetro ($1) é atribuído como o valor da variável $A, e o segundo como o valor da variável $B.

Ao executarmos, a saída dependerá dos valores passados:

$ ./script5 10 5
A e B são diferentes
A é maior que B

$ ./script5 10 20
A e B são diferentes
A é menor que B

$ ./script5 10 10
A e B são iguais

COMPARADORES DE CADEIAS DE CARACTERES (STRINGS)

Observe agora os comparadores de cadeias de caracteres:

IGUALDADE

if [[“$A” = “$B”]]

Resulta verdadeiro se o valor (texto) da variável $A for igual ao da variável $B.

DIFERENTE

if [[“$A” != “$B”]]

Resulta verdadeiro se o valor (texto) da variável $A for diferente do da variável $B.

CADEIA NULA (NULL) OU VAZIA

if [[-z “$A”]]

Resulta verdadeiro se o valor (texto) da variável $A tiver zero caracteres (tamanho zero).

CADEIA NÃO NULA (NOT NULL)

if [[-n “$A”]]

Resulta verdadeiro se o valor (texto) da variável $A tiver tamanho maior do que zero.

💬 COMENTÁRIO

O SCRIPT que desenvolvemos no exemplo anterior possui uma falha: Se for executado sem parâmetros vai retornar erro, pois tentará comparar valores ($1 e $2) que não existem.

Esse problema pode ser facilmente resolvido incluindo-se um teste de validade dos parâmetros no início da execução.

```
#!/bin/bash

# Atribuir os parâmetros como as variáveis A e B
A=$1
B=$2

# Teste de validade dos parâmetros. Se algum deles for nulo
# o SCRIPT aborta a execução e retorna valor 1 para indicar
# que ocorreu uma falha.
# Testa se o primeiro valor é válido.
```

```
if [[ -n "$A" ]]
then
echo "O primeiro valor é válido: $A"
else
echo "O primeiro valor é nulo"
exit 1
fi

# Testa se o segundo valor é válido.
if [[ -n "$B" ]]
then
echo "O segundo valor é válido: $B"
else
echo "O segundo valor é nulo"
exit 1
fi

# Inicia as comparações entre $A e $B
if [[ "$A" -eq "$B" ]]
then
echo "A e B são iguais"
fi

if [[ "$A" -ne "$B" ]]
then
echo "A e B são diferentes"
f1

if [[ "$A" -gt "$B" ]]
then
echo "A é maior que B"
fi

if [[ "$A" -lt "$B" ]]
then
echo "A é menor que B"
```

```
fi

exit 0
```

Se tentarmos executar o SCRIPT sem preencher ambos os parâmetros, o erro do usuário será detectado e indicado.

$ ./script5
O primeiro valor é nulo

$ ./script5 22
O primeiro valor é valido: 22
O segundo valor é nulo

EXECUTANDO O SCRIPT SEM PARÂMETROS

No último exemplo criamos um SCRIPT para comparar dois valores passados por meio de parâmetros na linha de comando. Em algumas situações, porém, pode ser mais prático e intuitivo pedir que o usuário digite os valores durante a execução.

Esse tipo de SCRIPT é chamado de INTERATIVO, pois exige a interação de um usuário.

O comando ‘read’, que usamos no módulo anterior para aguardar uma interação do usuário, também pode ser usado para receber um valor e atribuí-lo a uma variável.

$ read A

Irá aguardar o usuário digitar alguma informação, e terminar com ENTER. A informação digitada será atribuída à variável A.

$ read A
Mensagem → Texto digitado pelo usuário

$ echo “$A”
Mensagem

Vamos criar uma versão interativa do exemplo que executamos anteriormente. Nessa versão, apenas o começo do SCRIPT será modificado – em vez de receber A e B por parâmetros, eles

serão obtidos do usuário pelo comando ‘read’.

```
#!/bin/bash
# Pedir ao usuário a digitação dos valores
# O parâmetro '-n' no echo faz com que ele não insira uma quebra de linha apos o texto
echo -n "Digite o primeiro valor: "
read A
echo -n "Digite o segundo valor: "
read B

### O RESTANTE DO SCRIPT É IGUAL AO script5
# Teste de validade dos parâmetros. Se algum deles for nulo
# o SCRIPT aborta a execução e retorna valor 1 para indicar
# que ocorreu uma falha.
# Testa se o primeiro valor é nulo.
if [[ -n "$A" ]]
then
echo "O primeiro valor é valido: $A"
else
echo "O primeiro valor é nulo"
exit 1
fi

# Testa se o segundo valor é nulo.
if [[ -n "$B" ]]
then
echo "O segundo valor é válido: $B"
else
echo "O segundo valor é nulo"
exit 1
fi

# Inicia as comparações entre $A e $B
if [[ "$A" -eq "$B" ]]
then
echo "A e B são iguais"
```

```
fi

if [[ "$A" -ne "$B" ]]
then
echo "A e B são diferentes"
fi

if [[ "$A" -gt "$B" ]]
then
echo "A é maior que B"
fi

if [[ "$A" -lt "$B" ]]
then
echo "A é menor que B"
fi

exit 0
```

REALIZANDO OPERAÇÕES ARITMÉTICAS EM SCRIPT

Operações aritméticas com variáveis são muito úteis em SCRIPTS. Podem ser usadas para cálculos comuns e até controle em estruturas de repetição.

Como um primeiro exemplo, observe o ‘script6’ que solicitará dois valores e retornará a soma deles:

```
#!/bin/bash
# Pede a digitação dos valores e os armazena nas
# variáveis A e B


echo -n "Digite o primeiro valor (A): "
read A
```

```
echo -n "Digite o segundo valor (B): "
read B


# Calcula A+B e atribui o resultado na variável C
(( C=A+B ))

# Exibe o resultado
echo "O resultado de A+B = $C"

# Exibe o mesmo resultado, porém indicando o cálculo
# no echo, sem a necessidade da variável C.
echo "O resultado de A+B = $(( A+B ))"

exit 0
```

Ao executar o comando (informando A=12 e B=15), temos:

```
$ ./script6
Digite o primeiro valor (A): 12
Digite o segundo valor (B): 15
O resultado de A+B = 27
O resultado de A+B = 27
```

📢 ATENÇÃO

O cálculo foi realizado de duas formas. Na primeira, o resultado foi atribuído a uma variável (C). Na segunda forma, o cálculo foi feito diretamente a partir do comando ‘echo’.

INCREMENTANDO VARIÁVEIS

Em estruturas de repetição, precisamos de variáveis auxiliares, que são constantemente incrementadas. O incremento de uma variável X pode ser feito com:

((X++))

Exemplo:

$ X=1

$ echo $X
1

$ ((X++))

$ echo $X
2

📢 ATENÇÃO

No exemplo acima, com comandos digitados diretamente no SHELL, o primeiro declara a variável X e atribui o valor ‘1’ a ela. Confirmamos que o valor foi atribuído no segundo comando (‘echo’), mostrando o valor de X. No terceiro comando, incrementamos o valor de X em uma unidade, o que nos é comprovado pelo último comando (‘echo’).

OPERAÇÕES COM PONTO FLUTUANTE

Uma desvantagem do SHELL é que ele não realiza operações com ponto flutuante. Se tentarmos uma divisão, por exemplo, o resultado será sempre um inteiro, perdendo-se as casas decimais.

Um comando que pode nos auxiliar é o ‘bc’ (consulte a MANPAGE para maiores informações). No exemplo abaixo, o comando é invocado recebendo de um ‘echo’ a expressão para ser calculada.

$ echo "scale=5; 29/3" | bc -l
9.66666

A expressão tem duas partes, separadas por ponto e vírgula. A primeira parte (scale=5) define quantas casas decimais deverá ter a resposta. A segunda parte é o cálculo, nesse caso 29

dividido por 3.

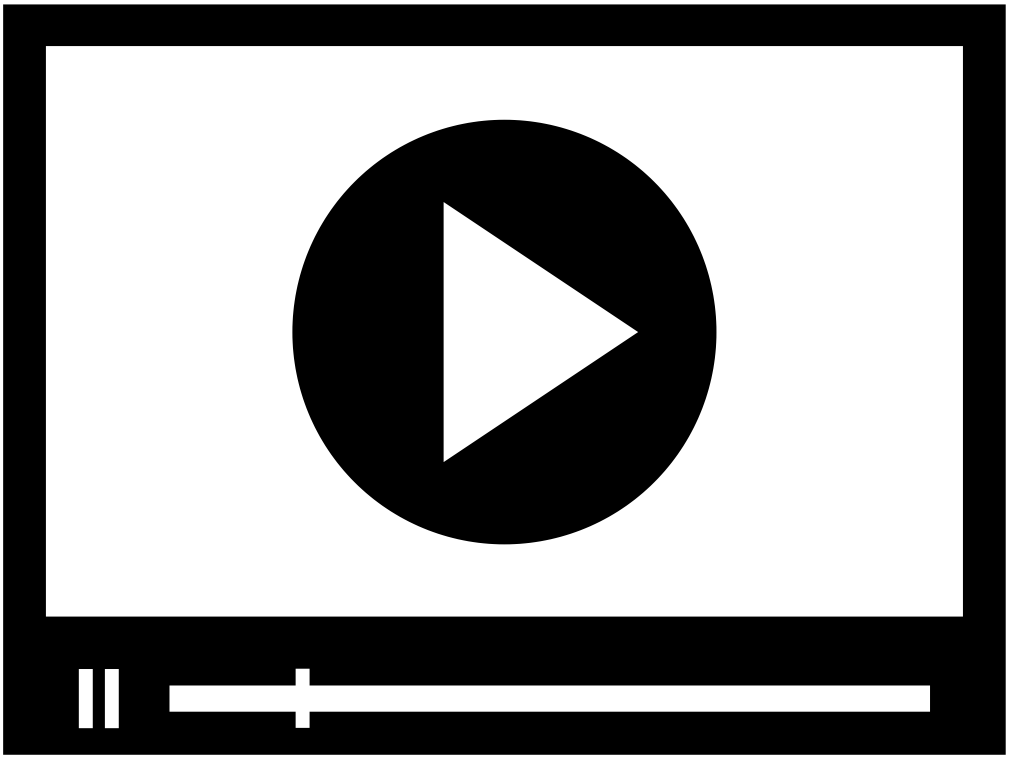
Considere o ‘script7’ abaixo, que pede ao usuário três números (A, B e C) e calcula sua média.

```
#!/bin/bash

echo -n "Digite o primeiro valor (A): "
read A
echo -n "Digite o segundo valor (B): "
read B
echo -n "Digite o terceiro valor (C): "
read C

MEDIA=$(echo "scale=2;($A+$B+$C)/3" | bc)

echo “A média dos três valores é: $MEDIA”
exit 0
```

Ao executar o SCRIPT:

```
$ ./script7
Digite o primeiro valor (A): 10
Digite o segundo valor (B): 9
Digite o terceiro valor (C): 8.5
A média dos três valores é: 9.16
```

Repare que o comando ‘bc’ realizou o cálculo utilizando ponto flutuante, sem perder as casas decimais no resultado.

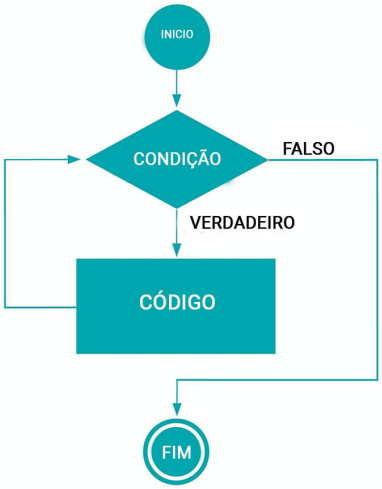


Ao assistir ao vídeo a seguir, você observará scripts com empregos de variáveis e estruturas de decisão, além do uso do switch case como alternativa ao uso do IF.

VERIFICANDO O APRENDIZADO

MÓDULO 4

- Esquematizar tarefas complexas em SCRIPTS com o uso de estruturas de repetição



CONCEITOS

Estruturas de repetição, também chamadas de LOOPS, são recursos fundamentais em qualquer modelo de programação. Através dessas estruturas, podemos implementar lógicas baseadas na execução repetitiva de códigos até que o nosso objetivo seja atingido.

Toda programação em LOOP deve definir uma condição para saída, ou seja, o critério para determinar o encerramento da execução, do contrário o LOOP será executado indefinidamente. Essa condição é chamada de LOOP infinito e, quase sempre, é indesejada.

Neste módulo, veremos duas estruturas para realizar LOOPS: **‘while’** e **‘for’**.

A ESTRUTURA DE REPETIÇÃO WHILE

A construção do LOOP com ‘while’ é simples:

```
while [ CONDIÇÃO ]
do
comando1
comando2
...
done
```

1

Inicialmente, a condição é avaliada. Se for atendida, todos os comandos entre ‘do’ e ‘done’ são executados.

2

Então, a condição é avaliada novamente e, se ainda for atendida, os comandos serão novamente executados.

3

Quando a condição não for atendida, o LOOP é interrompido e o SCRIPT segue executando os comandos após o ‘done’.

Cada execução no LOOP é chamada de **iteração**.

Observe o SCRIPT de exemplo abaixo. Nele, o usuário é convidado a digitar um número e o SCRIPT contará até esse número, começando em 1. O LOOP é realizado com ‘while’:

```
#!/bin/bash

# Apaga todo o conteúdo do terminal para facilitar
# a visualização
clear

# Pede a digitação de um número
echo -n "Digite um número: "
read num

# Define a variável auxiliar $i
i=1

# O while vai executar enquanto a condição for atendida
# A condição é $i MENOR OU IGUAL A $num
while [[ $i -le $num ]]
do
# Exibe o contador (variável $i)
echo "Contando... $i"

# Incrementa a variável $i em uma unidade
```

((i++))

Aguarda um segundo

sleep 1

done

A execução chegará a esse ponto depois que o LOOP

estiver encerrado.

exit 0

Ao executar o SCRIPT:

$ ./script8

Digite um número: **10**

Contando... 1

Contando... 2

Contando... 3

Contando... 4

Contando... 5

Contando... 6

Contando... 7

Contando... 8

Contando... 9

Contando... 10

INTERFERINDO NA EXECUÇÃO DE UM LOOP

Os comandos ‘**break**’ e ‘**continue**’ permitem interferir na execução de um LOOP:

BREAK

O comando ‘break’ força a interrupção imediata do LOOP, mesmo que a condição tenha sido satisfeita.

CONTINUE

Já o comando ‘continue’ força uma nova iteração do LOOP, ou seja, retorna a execução para o começo do LOOP.

Adaptando o SCRIPT anterior, será incluído um critério simples de interrupção, a partir do relógio do sistema:

Se o ponteiro dos segundos no relógio estiver em zero, o LOOP é interrompido imediatamente.

Para tal, usaremos um ‘if’ a cada iteração comparando o valor do segundo atual com o “00”.

Se for igual, o comando ‘break’ é executado, interrompendo o LOOP.

```
#!/bin/bash
# Apaga todo o conteúdo do terminal para facilitar
# a visualização
clear

# Pede a digitação de um número
echo -n "Digite um número: "
read num

# Define a variável auxiliar $i
i=1

# Exibe a hora de início
echo "Hora de início: $(date +%H:%M:%S)"

# O while vai executar enquanto a condição for atendida
# A condição é $i MENOR OU IGUAL A $num
while [[ $i -le $num ]]
do

# Exibe o contador $i
```

```
echo "Contando... $i"

# Incrementa a variável $i em uma unidade
(( i++ ))

# Testa se o segundo do relógio é igual a zero.
# Se for, interrompe o LOOP por meio do comando break
if [[ $(date +%S) -eq "00" ]]
then
echo "LOOP interrompido: $(date +%H:%M:%S)"
break
fi

# Aguarda um segundo
sleep 1
done

# A execução chegará a esse ponto depois que o LOOP
# estiver encerrado.
exit 0
```

As alterações em relação ao exemplo anterior estão destacadas.

Ao executar o SCRIPT, podemos ver a interrupção forçada pela condição dos segundos no relógio do computador.

```
$ ./script9
Digite um número: 60
Hora de início: 15:09:50
Contando... 1
Contando... 2
Contando... 3
Contando... 4
Contando... 5
Contando... 6
Contando... 7
Contando... 8
Contando... 9
```

Contando... 10

LOOP interrompido: 15:10:00

Se fosse digitado um valor abaixo de 60, haveria uma chance do SCRIPT encerrar sua execução normalmente, antes do ponteiro dos segundos passar pelo ‘0’.

💡 DICA

Experimente trocar o comando ‘break’ por ‘continue’. Nesse caso, quando o ponteiro dos segundos for zero, uma nova iteração do LOOP será iniciada, sem a execução da espera pelo comando ‘sleep’.

A ESTRUTURA DE REPETIÇÃO FOR

O ‘for’ permite realizar um LOOP a partir de uma lista de variáveis.

Considere o SCRIPT de exemplo abaixo:

```
#!/bin/bash

for numero in 10 11 15 20 29 36 41 45
do
echo "Número: $numero"
done
exit 0
```

Ao executarmos, temos o seguinte resultado:

$ ./script10

Número: 10

Número: 11

Número: 15

Número: 20

Número: 29

Número: 36

Número: 41

Número: 45

Para cada número na lista o ‘for’ realizou uma iteração.

O ‘for’ também pode receber máscara de arquivos. Observe o exemplo abaixo:

```
#!/bin/bash

contador=1

for arquivo in *
do
echo "Arquivo #$contador: $arquivo"
(( contador++ ))
done
```

Nesse exemplo, o ‘for’ recebe um ‘*’ como lista. Esse caractere é uma máscara que será substituída por uma lista com todas as entradas (arquivos e diretórios) existentes no diretório atual. Adicionalmente, foi incluída uma variável ‘contador’ para contar a quantidade de arquivos à medida que são exibidos.

Ao executar o SCRIPT, teremos:

$ ./script11

Arquivo #1: loop1

Arquivo #2: loop2

Arquivo #3: loop3

Arquivo #4: loop4

Arquivo #5: script1

Arquivo #6: script2

Arquivo #7: script3

Arquivo #8: script4

Arquivo #9: script5

Arquivo #10: script5i

Arquivo #11: script6

Arquivo #12: script7

SAIBA MAIS

Utilizando os conceitos deste curso, podemos criar SCRIPTS com lógicas avançadas para atender às nossas necessidades.

Vamos propor um exemplo de SCRIPT para aplicarmos os conceitos que aprendemos até aqui: Exemplo de SCRIPT: Levantamento de espaço ocupado

OBTENDO UMA VARIÁVEL A PARTIR DE UM ARQUIVO TEXTO

O SCRIPT de Levantamento de espaço ocupado, visto anteriormente, tem uma característica que pode ser considerada como inconveniente por muitos administradores:

Para mudar a relação de diretórios, é necessário alterar o SCRIPT.

Isso pode gerar alguns problemas:

1

Um usuário inexperiente poderia danificar o SCRIPT.

2

Será necessário que um usuário possua permissão de escrita para modificar o SCRIPT.

3

Se a relação de diretórios for grande, o SCRIPT inevitavelmente ficará grande.

Uma forma de resolver o problema é separar essa informação, a lista de diretórios, do SCRIPT.

💬 COMENTÁRIO

A ideia é que o arquivo do SCRIPT contenha somente o nosso “programa” e a lista de diretórios seja colocada em um arquivo à parte.

Para fazer isso, primeiro devemos criar um arquivo para a lista de diretórios. Cada diretório deve estar em uma linha do arquivo. Por exemplo, foi o criado o arquivo ‘lista_diretorios’ com o conteúdo abaixo:

/etc
/home
/lib
/usr/bin
/usr/lib
/usr/local
/usr/sbin
/var/lib
/var/log
/var/spool
/tmp

Em seguida, vamos substituir a declaração da variável DIRETORIOS em nosso exemplo de SCRIPT de Levantamento de espaço ocupado, para que seu conteúdo seja obtido no arquivo que acabamos de criar:

```
# A relação de diretórios a serem avaliados será obtida a
# partir do arquivo ‘diretorios’ e atribuída como conteúdo
# partir do arquivo ‘diretorios’ e atribuída como conteúdo
DIRETORIOS=$(< lista_diretorios)
```

Repare que foi utilizado o redirecionador de entrada ‘<’, para leitura do arquivo ‘lista_diretórios’.

Desse ponto em diante, o SCRIPT seguirá sua execução como antes.

CONSIDERAÇÕES AO EXECUTAR UM SCRIPT A PARTIR DO CRON

💬 COMENTÁRIO

É muito provável que você desenvolva SCRIPTS para execução automática, através de serviços como o CRON. Como se trata de uma execução não assistida, o seu SCRIPT não poderá depender de um terminal.

Sempre considere:

PARA A ENTRADA DE DADOS

> Seu SCRIPT não deverá interromper a execução esperando qualquer interação de usuário. Se isso acontecer, seu SCRIPT poderá permanecer parado indefinidamente.
>
> Comandos e dados que mudam a cada execução devem ser passados por meio de arquivos, ainda que temporários, e parâmetros de execução.

PARA A SAÍDA DE DADOS

> Prefira salvar as mensagens geradas no SCRIPT em um arquivo texto de LOG.
>
> Caso seu SCRIPT não envie diretamente para um arquivo, e não seja desejável adaptá-lo, você pode utilizar um redirecionador (> ou >>) na execução, diretamente no CRONTAB.

Por exemplo:

```
30 19 * * 0 /home/bob/script1 >> /home/bob/script1.log
```

Toda a saída do script1 será escrita no final do arquivo /home/bob/script1.log.

O CRON pode ser configurado para enviar as saídas de um comando por mensagem para a caixa postal local do usuário. Porém, se considerar que essas informações são importantes e não podem ser perdidas, prefira o arquivo texto de LOG.

SAIBA MAIS

Um recurso muito útil para quem escreve SCRIPTS são as expressões regulares (regular expressions, ou simplesmente, regex). Com elas podemos realizar comparações complexas e transformações de dados.

Em diversos exemplos, desenvolvemos SCRIPTS que pediam ao usuário para digitar valores ou textos, e vimos como fazer uma validação simples dessas variáveis.

E se precisarmos de uma validação mais rigorosa?

E se quisermos, por exemplo, garantir que o usuário digitou um número de, exatamente, 3 dígitos? Observe o SCRIPT abaixo:

```
#!/bin/bash

echo -n "Digite um número de 3 dígitos: "
read numero
if [[ $numero =~ ^[0-9]{3}$ ]]
then
echo "OK"
else
echo "O número não possui 3 dígitos."
fi
```

O comparador =~ utiliza expressão regular na comparação.

E qual é o significado dessa expressão regular?

Significa início da cadeia de caracteres. Ou seja, a validação começa verificando desde o 1º caractere da sequência.

[0-9]

Procura um caractere compreendido entre 0 e 9.

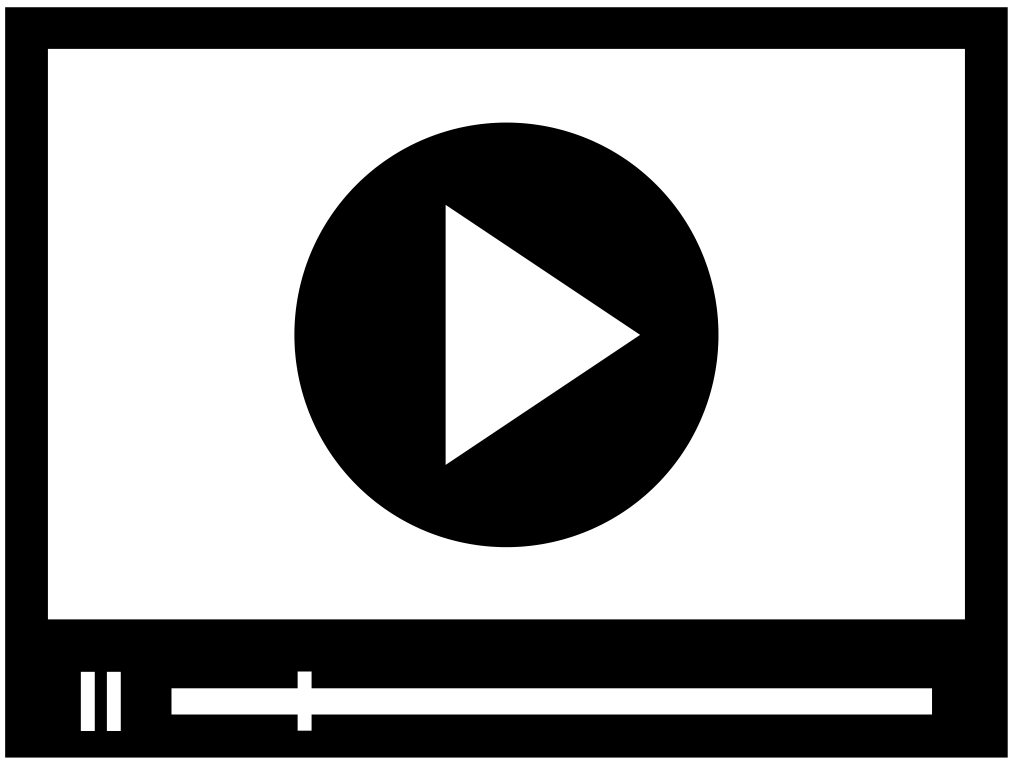
{3}

Repetição por 3x da regra imediatamente anterior [0-9].

$

Fim da cadeia de caracteres.

Ou seja, a comparação só será válida se a cadeia for composta por, exatamente, 3 caracteres de 0 a 9.



Ao assistir ao vídeo a seguir, você observará scripts com emprego das estruturas de repetição, combinadas com os recursos apresentados nos módulos anteriores e exemplos de casos práticos para a administração de servidores Linux.

VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Ao longo deste tema, conhecemos ferramentas muito importantes para a administração de servidores e estações baseadas no sistema operacional Linux.

A possibilidade de criar SCRIPTS para a execução de tarefas nos permite definir algoritmos sofisticados para tratar praticamente qualquer cenário que nosso ambiente nos apresente.

Com o CRON, podemos agendar a execução dessas tarefas e tornar nossa administração automatizada e independente da interação de usuários.

Apesar de não serem verdadeiramente uma linguagem de programação, os SCRIPTS de BASH podem ser estruturados com a mesma base lógica de qualquer linguagem simples, uma grande vantagem para aqueles que já conhecem programação e podem aproveitar seus conhecimentos.

Para aqueles que ainda não têm grande intimidade com programação, deixo como sugestão investir um pouco de tempo no estudo de algoritmos. A capacidade de abstrair problemas reais em passos lógicos e programá-los nos oferece grande versatilidade e eficiência, essenciais para qualquer administrador de sistemas.

⚠ PODCAST

AVALIAÇÃO DO TEMA:

REFERÊNCIAS

MAXWELL, S. **Administração de sistemas Unix**. 1. ed. Rio de Janeiro: Ciência Moderna, 2003.

MOTA FILHO, J. **Descobrindo o Linux**: Entenda o Sistema Operacional GNU/Linux. 3. ed. São Paulo: Novatec, 2012.

PETERSEN, R. **Ubuntu 18.04 LTS Server**: Administration and Reference. 1. ed. Surfing Turtle Press, 2018.

THE LINUS DOCUMENTATION PROJECT. **Advanced Bash-Scripting Guide.** *In*: TLDP. Publicado em meio eletrônico em: 10 mar. 2014.

UBUNTU. **Ubuntu Server Guide.** Consultado em meio eletrônico em: 19 out. 2020.

EXPLORE+

Para saber mais sobre os assuntos tratados neste tema, leia:

Capítulo 18 de Advanced Bash-Scripting Guide, TLDP.

CONTEUDISTA

Fernando Diniz Hämmerli

🔗 CURRÍCULO LATTES