

Convertendo imagem e montando script

Transcrição

Nós fomos contratados pela *Multillidae*, para ajudá-los em algumas tarefas que eles terão nas semanas seguintes.

A *Multillidae* está com um projeto de abrir uma **loja virtual** e um dos setores dessa loja será o setor de livros de tecnologia. A *Multillidae* comprou alguns livros da *Casa do Código* para colocar na plataforma online de vendas.

Entretanto, a plataforma da *Multillidae* só aceita os arquivos no formato `.png` . Mas, os arquivos que foram passados da *Casa do Código* estão na extensão `.jpg` .

A missão que nos foi dada é justamente **encontrar uma forma de converter esses arquivos** de extensão `.jpg` para `.png` .

Os diretores da *Multillidae* nos passaram o [link \(`https://drive.google.com/open?id=0BzmYQVmW4W7nUW40M2dfQWxlTm8`\)](https://drive.google.com/open?id=0BzmYQVmW4W7nUW40M2dfQWxlTm8), para o download das imagens.

Muito bem! Após ter feito o download das imagens, vamos abrir o terminal e verificar o diretório, para ver se de fato, o arquivo está lá na pasta "Downloads".

Como foi visto no curso de Linux, (se você ainda não fez o curso, clique [aqui \(`https://cursos.alura.com.br/course/linux-ubuntu-processos?preRequirementFrom=shells scripting`\)](https://cursos.alura.com.br/course/linux-ubuntu-processos?preRequirementFrom=shells scripting)), para mudar de diretório basta utilizar o comando seguido da pasta `cd Downloads/` e depois utilizar o comando `ls` que **listará** o conteúdo do diretório em que estamos.

Perceba que assim acessamos o arquivo com as imagens em .jpg , mas ele está compactado em zip. Vamos descompactá-lo utilizando o comando unzip `imagens-livros.zip`

```
rafael@rafael-VirtualBox:~/Downloads$ ls
imagens-livros.zip
rafael@rafael-VirtualBox:~/Downloads$ unzip imagens-livros.zip
Archive: imagens-livros.zip
  creating: imagens-livros/
  inflating: imagens-livros/algoritmos.jpg
  inflating: imagens-livros/amazon_aws.jpg
  inflating: imagens-livros/arduino_pratico.jpg
  inflating: imagens-livros/asp_net.jpeg
  inflating: imagens-livros/big_data.jpg
  inflating: imagens-livros/codeigniter.jpeg
  inflating: imagens-livros/docker.jpg
```

Vamos listar o diretório com o comando `ls` . O resultado será esse:

imagens-livros imagens-livros.zip

[COPIAR CÓDIGO](#)

O primeiro diretório é a pasta que contém as imagens a serem convertidas e o segundo diretório é o arquivo compactado. Podemos removê-lo já que ele não é mais necessário. Utilizando o comando `rm imagens-livros.zip` conseguimos deixar somente o que nos interessa.

Legal, vamos entrar nesse diretório para ver o conteúdo dele:

```
$ cd imagens-livros/
$ ls
```

[COPIAR CÓDIGO](#)

```
rafael@rafael-VirtualBox:~/Downloads$ cd imagens-livros/
rafael@rafael-VirtualBox:~/Downloads/imagens-livros$ ls
algoritmos.jpg      codeigniter.jpg    java_ee.jpg        node.jpg          scala.jpg       windows_server.jpg
amazon_aws.jpg       cordova.jpg     jenkins.jpg       nosql.jpg        scratch.jpg    xamarin_forms.jpg
arduino_pratico.jpg dsl.jpg         jquery.jpg       orientacao_objetos.jpg  seguranca.jpg  zend.jpg
asp_net.jpg         elasticsearch.jpg mantra_produtividade.jpg postgres.jpg   sass.jpg        turbine_css.jpg
big_data.jpg        es6.jpg        metricas_agiles.jpg    sass.jpg        vue.jpg
```

Temos aqui todos esse livros que precisamos realizar a conversão.

Depois que soubemos de nossa missão dentro da *Multillidae*, começamos a fazer algumas pesquisas. Vimos que no próprio Ubuntu existe uma ferramenta capaz de fazer essa conversão: o *ImageMagick*!!!

Para realizar essa conversão, basta utilizar o comando `convert` e dizer qual arquivo queremos converter. De início, daremos um foco maior no arquivo `algoritmos.jpg`, para ter a certeza de que realmente o ImageMagick irá realizar essa conversão. Feito isso, diremos o nome e a extensão do arquivo para o qual iremos converter:

```
$ convert algoritmos.jpg algoritmos.png
```

[COPIAR CÓDIGO](#)

Na sequência, usamos o comando para listar, o `ls`. Vimos que o arquivo `algoritmos.png` se encontra nesse mesmo diretório. Mas, será verdade? Vamos confirmar se realmente esse arquivo tem a extensão `.png`.



Maravilha! Ao observar verificamos que o **ImageMagick** conseguiu converter esse arquivo! Ele parece ser a solução ideal para nós. Mas, repare que há cerca de 25 livros que precisam sofrer a conversão de extensão e utilizar o comando `convert livro.jpg livro.png` várias vezes para cada um dos livros não é uma solução muito elegante. Imagine que amanhã, os diretores tenham mais 100 imagens para converter, então, nesse caso, teríamos que colocar 100 vezes o mesmo comando para cada livro. Não seria nada prático, não é mesmo?

Justamente em situações como essas que o **Shell Scripting** consegue ajudar a *automatizar essas tarefas*.

SHELL SCRIPTING

Podemos interpretar o **Shell** como uma *interface* que nós, usuários, acessamos os recursos no Sistema Operacional. Já a palavra **Scripting**, significa *roteiro* e é uma lista de comandos que serão interpretados pelo Sistema Operacional.

Montaremos um script capaz de realizar essa conversão para nós, de uma forma mais eficiente do que o usuário simplesmente colocar o comando no terminal para cada imagem. Automatizaremos essa tarefa.

Para criar o arquivo, precisamos de um **editor de texto**. Fique à vontade para escolher o editor de seu gosto: [Nano](https://www.nano-editor.org/) (<https://www.nano-editor.org/>), [Vi/Vim](http://ex-vi.sourceforge.net/) (<http://ex-vi.sourceforge.net/>), [gEdit](https://wiki.gnome.org/Apps/Gedit) (<https://wiki.gnome.org/Apps/Gedit>) ou algum outro de sua preferência. Ao longo do curso, usaremos o **Nano** e o **gEdit**.

Vamos voltar a nossa "home" com o comando `cd` para criar um diretório no qual vamos guardar todos os scripts feitos durante o curso.

```
$ cd  
$ mkdir Scripts  
$ cd Scripts/
```

[COPIAR CÓDIGO](#)

Após ter mudado para dentro da nova pasta "Scripts", criaremos o primeiro Script para converter a imagem do formato `.jpg` para `.png`. Com o comando `nano nome_de_um_arquivo.sh` criaremos e editaremos esse arquivo utilizando o editor **Nano**. Como estamos trabalhando com Shell Scripting colocaremos a extensão `.sh`:

```
$ nano conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

Com o editor aberto, temos que dizer como esses comandos serão interpretados. Na primeira linha, vamos dizer qual vai ser o interpretador do script. Para isso, diremos por meio do comando:

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

[COPIAR CÓDIGO](#)

Após especificar o interpretador dos comandos que serão digitados a seguir, nos resta somente digitá-los para checar se o script funcionará. Para realizar a conversão, utilizamos o comando `convert` e converteremos o arquivo `amazon_aws.jpg` para o formato `.png`:

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

```
convert amazon_aws.jpg amazon_aws.png
```

[COPIAR CÓDIGO](#)

Repare que o arquivo `amazon_aws.jpg` está em um diretório diferente do nosso script. As imagens estão no diretório "imagens-livros", que está dentro de "Downloads". Os scripts, por sua vez, estão no diretório "Scripts". Por essa razão, é necessário colocar o caminho de onde esses arquivos estão, para que possamos ter a referência de onde eles estão localizados.

Tanto o diretório "Scripts" quanto o diretório "Downloads" estão dentro da **home**. Com isso, podemos usar o `~` pois eles estão dentro da home!

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

```
convert ~/Downloads/imagens-livros/amazon_aws.jpg ~/Downloads/:
```

[COPIAR CÓDIGO](#)

Queremos salvar a conversão `amazon_aws.png` no mesmo diretório dos arquivos `.jpg`, para que tudo fique no mesmo lugar.

Se tudo deu certo, vamos salvar o nosso script com o comando `Ctrl + X` para sair e `Y` de (yes) para salvar as alterações.

Como falamos que o interpretador do script é o **bash**, para rodar esse script, colocamos `bash` e o nome do script e depois teclamos o "Enter":

```
bash conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

Já que estamos no diretório "Scripts", vamos acessar o diretório "Downloads/imagens-livros":

```
$ bash conversao-jpg-png.sh  
$ cd ~/Downloads/imagens-livros.  
$ ls
```

[COPIAR CÓDIGO](#)

Observe, agora temos o `algoritmos.png` e temos o `amazon_aws.png`, que é o resultado da conversão que nosso script fez. Mas perceba que o script só está fazendo a conversão do arquivo `amazon_aws`, ou seja, nós tiramos o comando do terminal e colocamos no script e isso fez com que voltassemos ao início do problema, precisando digitar uma linha de comando para cada imagem a ser convertida.

Vamos ver como melhorar esse script nas próximas aulas!

Instalando o sistema operacional

Durante o curso nós utilizaremos o Ubuntu para montar nossos Scripts. Para instalar o Ubuntu para acompanhar com os exemplos do curso, você poderá realizar o download nesse link: http://releases.ubuntu.com/16.04.2/ubuntu-16.04.2-desktop-amd64.iso?_ga=2.172336035.1044129451.1500912557-237643101.1496840147 (http://releases.ubuntu.com/16.04.2/ubuntu-16.04.2-desktop-amd64.iso?_ga=2.172336035.1044129451.1500912557-237643101.1496840147).

Se a sua máquina principal não é Linux, recomendamos que você use uma máquina virtual para instalar o Ubuntu. Uma opção é instalar o VirtualBox, onde poderá ser feito o download nesse link:

<https://www.virtualbox.org/wiki/Downloads>
(<https://www.virtualbox.org/wiki/Downloads>).

Uma vez que o download foi concluído, clique no botão **Install Ubuntu**

 [Install \(as superuser\)](#)

Welcome

English

Español
Esperanto
Euskara
Français
Gaeilge
Galego
Hrvatski
íslenska
Italiano
Kurdî
Latviski
Lietuviškai
Magyar
Nederlands
Norsk bokmål
Norsk nynorsk
Polski
Português



[Try Ubuntu](#)



[Install Ubuntu](#)

You can try Ubuntu without making any changes to your computer, directly from this CD.

Or if you're ready, you can install Ubuntu alongside (or instead of) your current operating system. This shouldn't take too long.

You may wish to read the [release notes](#).

Feito isso, clique no botão **Continue** para continuar a instalação

 [Install \(as superuser\)](#)

Preparing to install Ubuntu

[Download updates while installing Ubuntu](#)

This saves time after installation.

[Install third-party software for graphics and Wi-Fi hardware, Flash, MP3 and other media](#)

This software is subject to license terms included with its documentation. Some is proprietary.

Fluendo MP3 plugin includes MPEG Layer-3 audio decoding technology licensed from Fraunhofer IIS and Technicolor SA.

[Quit](#)

[Back](#)

[Continue](#)

Posteriormente, clique na opção **Install now**

 **Install (as superuser)**

Installation type

This computer currently has no detected operating systems. What would you like to do?

Erase disk and install Ubuntu

Warning: This will delete all your programs, documents, photos, music, and any other files in all operating systems.

Encrypt the new Ubuntu installation for security

You will choose a security key in the next step.

Use LVM with the new Ubuntu installation

This will set up Logical Volume Management. It allows taking snapshots and easier partition resizing.

Something else

You can create or resize partitions yourself, or choose multiple partitions for Ubuntu.

Quit

Back

Install Now

Por fim, clique em **Continue** até chegar no ponto para inserir seu nome, coloque seus dados e termine a instalação

Install (as superuser)

Who are you?

Your name: 

Your computer's name: 

The name it uses when it talks to other computers.

Pick a username: 

Choose a password: 

Confirm your password: 

Log in automatically

Require my password to log in

Encrypt my home folder

Back

Continue

Ao terminar esse processo deveremos ter o Ubuntu instalado em nossa máquina!

Mãos à obra: Criando o primeiro script

Uma vez que preparamos nosso ambiente, vamos até a nossa home com o comando `cd ~` e criaremos um diretório onde teremos todos os nossos scripts. Vamos criar um diretório chamado de Scripts com o comando `mkdir Scripts`.

Feito isso, faça o download dos arquivos das imagens passadas pelos diretores da Mutillidae, que pode ser encontrado nesse link: [\(https://caelum-online-public.s3.amazonaws.com/shell-script/aula_1/imagens-livros.zip\)](https://caelum-online-public.s3.amazonaws.com/shell-script/aula_1/imagens-livros.zip). Uma vez que o download foi realizado, vá até o diretório **Downloads** com o comando `cd ~/Downloads` para certificar que o download foi de fato concluído.

Uma vez que certificamos que o download foi concluído, devemos descompactar os arquivos com o comando `unzip imagens-livros.zip`

```
rafael@rafael-VirtualBox:~/Downloads$ ls
imagens-livros.zip
rafael@rafael-VirtualBox:~/Downloads$ unzip imagens-livros.zip
Archive:  imagens-livros.zip
  creating:  imagens-livros/
  inflating:  imagens-livros/algoritmos.jpg
  inflating:  imagens-livros/amazon_aws.jpg
  inflating:  imagens-livros/arduino_pratico.jpg
  inflating:  imagens-livros/asp_net.jpeg
  inflating:  imagens-livros/big_data.jpg
  inflating:  imagens-livros/codeigniter.jpeg
```

Ao descompactar os arquivos, nós podemos remover o arquivo `.zip` com o comando `rm *.zip`. Feito isso, vamos voltar ao nosso diretório de scripts com o comando `cd ~/Scripts` e vamos criar nosso primeiro script chamado de `conversao-jpg-png.sh`. Para isso, você pode utilizar o editor de texto de sua preferência, por exemplo com o nano teremos `nano conversao-jpg-png.sh`

Na primeira linha de nosso script coloque o interpretador `#!/bin/bash` e depois coloque o comando `convert ~/Downloads/imagens-livros/algoritmos.jpg ~/Downloads/imagens-livros/algoritmos.png`. Salve as alterações realizadas e execute o seu script com o comando `bash conversao-jpg-png.sh`. Depois vá até o diretório `imagens-livros` e procure pelo livro `algoritmos`. Qual foi o resultado?

Opinião do instrutor

Uma vez que que nosso script está dentro do diretório `Scripts` e o diretório dos livros está dentro do diretório `imagens-livros` que está dentro do diretório `Downloads`, torna-se necessário passar o caminho o qual a imagem do livro `algoritmos.jpg` está localizada. Feito isso, devemos ser capazes de ver que a conversão foi realizada e devemos ter o arquivo `algoritmos.png`

```
#!/bin/bash
convert ~/Downloads/imagens-livros/algoritmos.jpg ~/Downloads/imagens-livros/algoritmos.png
```



Passando parâmetros para o script

Transcrição

Na etapa anterior começamos a montar o script que era capaz **somente** de realizar a conversão do arquivo `amazon_aws.jpg` para o arquivo `amazon_aws.png`. Entretanto, ainda não conseguimos oferecer uma grande interação com o usuário final.

A ideia é, quando o usuário final executar o script, que ele seja capaz de passar parâmetros para serem interpretados.

Por exemplo, dentro da pasta "imagens-livros", temos o arquivo `arduino_pratico.jpg`, então o usuário poderia passar o nome desse arquivo como parâmetro para ser interpretado, assim o script se encarrega de fazer a conversão desse livro:

```
$ bash conversao-jpg-png.sh arduino_pratico
```

[COPIAR CÓDIGO](#)

Perceba que dessa forma, o script converterá apenas a imagem de um único livro, pois o arquivo está estático e isso não é interessante para nós. Precisamos fazer com que o script pegue o **conteúdo** do primeiro parâmetro.

Para isso, acessaremos novamente o script com `nano conversao-jpg-png.sh` e implementaremos a tarefa de pegar o conteúdo do **primeiro** parâmetro passado pelo usuário. Para pegar o conteúdo, utilizamos o símbolo `$`.

E já que estamos falando do *primeiro* parâmetro, podemos referenciá-lo pelo número 1.

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

```
convert ~/Downloads/imagens-livros/$1.jpg ~/Downloads/imagens-
```

[COPIAR CÓDIGO](#)



Após essas alterações, vamos fechar o arquivo com "Ctrl + x" e "y" para salvar. Vamos rodar esse script passando o nome do livro:

```
$ bash conversao-jpg-png.sh arduino_pratico
```

[COPIAR CÓDIGO](#)

Aparentemente, foi feita a conversão. Vamos dar uma olhada na pasta.

Dentro da pasta "imagens-livros" encontramos o livro `arduino_pratico.png`! Ou seja, o script funcionou corretamente! Só que, temos **vários** arquivos dentro dessa pasta prontos para serem convertidos. Será que conseguimos passar mais de 1 parâmetro ao mesmo tempo? Vamos tentar!

Voltando ao script com o comando `nano conversao-jpg-png.sh`, queremos colocar o **segundo** parâmetro a ser interpretado. Pela lógica, assim como o parâmetro 1 é referenciado pelo primeiro, o segundo parâmetro será referenciado pelo número 2!

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

```
convert ~/Downloads/imagens-livros/$1.jpg ~/Downloads/imagens-  
convert ~/Downloads/imagens-livros/$2.jpg ~/Downloads/imagens-
```

[COPIAR CÓDIGO](#)

Perceba que esse é um código repetitivo e estamos sujeitos a cometer erros mais facilmente. A partir do momento que o código apresenta sinais de repetição de um determinado comando, podemos **isolá-lo**s em uma constante e fazer a **referência** ao conteúdo da mesma.

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

```
CAMINHO_IMAGENS=~/Downloads/imagens-livros
```

```
convert $CAMINHO_IMAGENS/$1.jpg $CAMINHO_IMAGENS/$1.png  
convert $CAMINHO_IMAGENS/$2.jpg $CAMINHO_IMAGENS/$2.png
```

[COPIAR CÓDIGO](#)

Legal! Vamos rodar esse script novamente para ver o resultado. Se tudo estiver certo, o usuário será capaz de passar 2 parâmetros simultaneamente.

```
$ bash conversao-jpg-png.sh asp_net big_data
```

[COPIAR CÓDIGO](#)

Após a execução, veremos se o arquivo convertido realmente está lá.

```
$ cd ~/Downloads/imagens-livros/
```

```
$ ls
```

[COPIAR CÓDIGO](#)

```
rafael@rafael-VirtualBox:~/Scripts$ cd ~/Downloads/imagens-livros/  
rafael@rafael-VirtualBox:~/Downloads/imagens-livros$ ls  
algoritmos.jpg      asp_net.png      es6.jpg          nosql.jpg        turbine_css.jpg  
algoritmos.png     big_data.jpg      java_ee.jpg      orientacao_objetos.jpg vue.jpg  
amazon_aws.jpg      big_data.png      jenkins.jpg      postgres.jpg    windows_server.jpg  
amazon_aws.png     codeigniter.jpg   jquery.jpg       sass.jpg        xamarin_forms.jpg  
arduino_pratico.jpg cordova.jpg      manteca_producitividade.jpg scala.jpg  zend.jpg
```

Agora, o script consegue suportar 2 parâmetros que são passados simultaneamente pelo usuário!

Voltemos ao script:

```
$ cd ~/Scripts/  
$ nano conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

Após observar atentamente o script, chegamos a conclusão que ele não está muito legal, pois temos duas linhas de comando que estão fazendo a mesma coisa! E como podemos melhorar o nosso script?

Veremos a seguir.

Utilizando laço de repetição

Transcrição

Até a etapa anterior, o script era capaz de receber dois parâmetros passados pelo usuário. Para pegar o conteúdo desses parâmetros, usamos o símbolo `$`.

E se o usuário pudesse passar um **terceiro**, ou até mesmo um **quarto** parâmetro? Seria necessário repetir o código, e com certeza, essa não é a solução mais elegante.

Então, *para cada parâmetro* passado pelo usuário, é executada a conversão. Com isso, é necessário implementar uma **estrutura de repetição** falada no curso de [Lógica de Programação](https://cursos.alura.com.br/course/logica-programacao-javascript-html?preRequirementFrom=shellscripting) (<https://cursos.alura.com.br/course/logica-programacao-javascript-html?preRequirementFrom=shellscripting>).

"Para cada variável `imagem` dentro do conjunto de parâmetros passados pelo usuário, englobamos todos eles usando `$@`".

Com a afirmação acima, conseguimos criar o nosso **laço** e definir o que ele irá executar!

```
#!/bin/bash

CAMINHO_IMAGENS=~/Downloads/imagens-livros

for imagem in $@
do
```

```
convert $CAMINHO_IMAGENS/$imagem.jpg $CAMINHO_IMAGENS/$imagem  
done
```

[COPIAR CÓDIGO](#)

Como estamos criando um laço de repetição, não tem necessidade de ter duas linhas de comando realizando a mesma tarefa. Retiramos uma.

Queremos converter o conteúdo da variável `imagem` que vai receber cada parâmetro passado pelo nosso usuário, por isso utilizamos `$imagem` no lugar de `$1`.

Uma vez que esse laço `for` foi concluído, precisamos dizer que a tarefa dele foi finalizada e com o `done`, conseguimos esse resultado.

Para sair e salvar, utilizamos "Ctrl + x" e "y".

Vamos executar esse script passando 4 parâmetros:

```
$ bash conversao-jpg-png.sh codeigniter cordova dsl elasticsearch
```

[COPIAR CÓDIGO](#)

Aparentemente, o script rodou sem nenhum problema. Vamos ver se os arquivos de conversão estão no diretório "imagens-livros".

```
$ cd ~/Downloads/imagens-livros/  
$ ls
```

[COPIAR CÓDIGO](#)

Observe a imagem abaixo:

```
rafaelgrafeael-VirtualBox:~/Scripts$ nano conversao.jpg.png.sh
rafaelgrafeael-VirtualBox:~/Scripts$ bash conversao.jpg.png.sh codeigniter cordova dsl elasticsearch
rafaelgrafeael-VirtualBox:~/Scripts$ cd ~/Downloads/imagens-livros/
rafaelgrafeael-VirtualBox:~/Downloads/imagens-livros$ ls
algoritmos.jpg      asp_net.png      dsl.jpg        jquery.jpg      sass.jpg       xamarin_forms.jpg
algoritmos.png      big_data.jpg    dsl.png        mantra_produtoividade.jpg  scala.jpg     zend.jpg
amazon_aws.jpg      big_data.png   elasticsearch.jpg metricas_agets.jpg scratch.jpg
amazon_aws.png      codeigniter.jpg elasticsearch.png node.jpg       segurança.jpg
arduino_pratico.jpg codeigniter.png es6.jpg       nosql.jpg      turbine_css.jpg
arduino_pratico.png cordova.jpg    java_ee.jpg    orientacao_objetos.jpg vue.jpg
asp_net.jpg         cordova.png    jenkins.jpg   postgres.jpg  windows_server.jpg
rafaelgrafeael-VirtualBox:~/Downloads/imagens-livros$
```

Como podemos ver, o script foi executado com sucesso!

Entendemos que com o `$@` pegamos todos os parâmetros que o usuário passar, independente da quantidade.

Os diretores da *Multillidae*, nos passaram a missão de realizar a **conversão automática** de todo o conteúdo do diretório "imagens-livros".

Na próxima aula, veremos como podemos melhorar o script.

Mãos à obra: Utilizando o laço de repetição

Perceba que nosso diretório **imagens-livros** possui uma grande quantidade de imagens para serem convertidas. Dessa forma, nós podemos melhorar nosso script para que possamos passar mais parâmetros para serem processados. Só que ficar repetindo código, não é a solução mais elegante, então usaremos a estrutura de repetição **for**

- Abra novamente o script que estamos trabalhando com o comando **nano conversao-jpg-png.sh**.
- Guarde o caminho **~/Downloads/imagens-livros** em uma constante, chamada por exemplo de **CAMINHO_IMAGENS**.

Obs: Não pode haver espaços entre o nome da constante, o símbolo do = e o conteúdo da constante, tem que ser por exemplo,
NOME_CONSTANTE=conteúdo

Feito isso, digite o laço de repetição:

```
for [variável] in $@
do
    convert $CAMINHO_IMAGENS/${variável}.jpg $CAMINHO_IMAGENS/!
done
```

[COPIAR CÓDIGO](#)

Execute seu script passando como parâmetro os quatro próximos livros:

- **arduino_pratico**
- **asp_net**
- **big_data**

- codeigniter

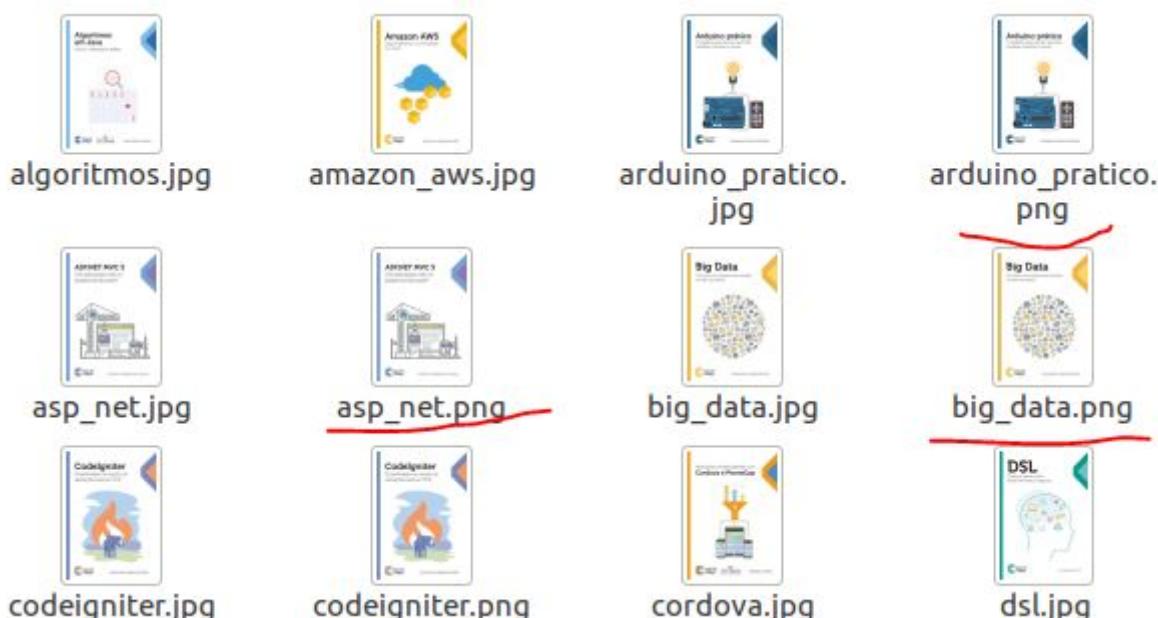
Qual é o resultado?

Opinião do instrutor

Ao criar um laço de repetição estamos pegando cada parâmetro passado pelo usuário e armazenando na variável, enquanto tivermos um parâmetro na variável a linha de conversão dos arquivos será executada e esse processo será repetido até que todos os parâmetros passados pelo usuário tenham sido convertidos. No final, nosso script deverá estar parecido como o abaixo:

```
#!/bin/bash  
  
CAMINHO_IMAGENS=~/Downloads/imagens-livros  
for imagem in $@  
do  
    convert $CAMINHO_IMAGENS/$imagem.jpg $CAMINHO_IMAGENS/$imagem.png  
done
```

Resultado ao rodar o script:



Convertendo automaticamente todos os arquivos

Transcrição

Na aula anterior, montamos um script de conversão de imagens, capaz de receber vários parâmetros passados de uma vez pelo usuário e por meio do `$@`, conseguimos englobar todos eles.

Entretanto, os diretores da *Multillidae* nos pediram para encontrarmos uma forma de montar nosso próprio script, para justamente fazer a verificação de **todos** os arquivos `.jpg` que estão dentro do diretório "imagens-livros" e que façamos a conversão deles para o formato `.png`.

Existem vários comandos que utilizamos no terminal, para começar e concluir o processo de conversão. O que ainda não havíamos comentado, é que podemos colocar todos eles dentro do script!

Vamos fazer isso, assim, quando esse script for executado, ele já entrará no diretório certo, o "imagens-livros", para fazer a verificação dos arquivos `.jpg` que já existem. Assim, é permitido fazer a conversão do restante dos arquivos!

Acessando o script...

```
$ cd ~/Scripts/  
$ nano conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

...mudaremos algumas coisas. A primeira delas é remover a constante `CAMINHO_IMAGENS`. Vamos entrar direto nesse diretório, colocando o comando `cd`.

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

```
cd ~/Downloads/imagens-livros
```

[COPIAR CÓDIGO](#)

Quando essa linha for executada, já estaremos dentro da pasta. Então, o que queremos fazer quando estivermos dentro desse diretório?

Uma varredura! Isso mesmo, uma varredura de todos os arquivos que tenham a extensão .jpg . Não pegaremos todos os arquivos que serão passados pelo usuário. Queremos fazer uma verificação dos arquivos que tem a extensão .jpg .

Sabemos que todos eles terminam com .jpg , mas antes da extensão, eles podem ter qualquer nome ou caractere. Com o * antes do .jpg podemos englobar todos os arquivos, independente do caractere que antecede a extensão:

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

```
cd ~/Downloads/imagens-livros
```

```
for imagem in *.jpg
do
    convert $CAMILHO_IMAGENS/$imagem.jpg $CAMILHO_IMAGENS/$imagem
done
```

[COPIAR CÓDIGO](#)

Com a linha do for , conseguimos fazer toda a verificação do conteúdo com a extensão .jpg que existe dentro do diretório "imagens-livros".

Uma vez que já estamos dentro do diretório "imagens-livros", não precisamos nos preocupar em passar o caminho completo, pois já estamos dentro do diretório onde os arquivos existem e onde queremos salvá-los.

```
#!/bin/bash

cd ~/Downloads/imagens-livros

for imagem in *.jpg
do
    convert $imagem.jpg $imagem.png
done
```

[COPIAR CÓDIGO](#)

Veremos se o script irá funcionar com essas alterações. Para sair, usamos "Ctrl + x" e "y" para salvar.

Se o script está funcionando corretamente, esperamos que ele faça a conversão de todas as imagens .jpg para o formato .png de forma automática.

```
convert: unable to open image `algoritmos.jpg.jpg': No such file
convert: no images defined `algoritmos.jpg.png' @ error/convert.c
convert: unable to open image `amazon_aws.jpg.jpg': No such file
convert: no images defined `amazon_aws.jpg.png' @ error/convert.c
convert: unable to open image `arduino_pratico.jpg.jpg': No such
convert: no images defined `arduino_pratico.jpg.png' @ error/conv
.
.
.
.
.
```

[COPIAR CÓDIGO](#)

Ops! Houve algumas mensagens de erro no terminal! Ele não conseguiu abrir o arquivo `algoritmos.jpg.jpg`. Por quê isso está acontecendo? Bom, vamos voltar ao script.

```
$ nano conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

Quando fazemos a **varredura** pesquisando TODOS os arquivos de extensão `.jpg` , serão passados para a variável `imagem` . Acontece que será passado o nome completo do arquivo, inclusive a sua extensão.

Dentro do laço de repetição, `convert $imagem.jpg $imagem.png` , perceba que estamos introduzindo mais uma extensão, além da extensão que virá na variável `imagem` .

Por isso, vamos remover a extensão `.jpg` que está depois da variável, para que esse problema não ocorra mais!

```
#!/bin/bash

cd ~/Downloads/imagens-livros

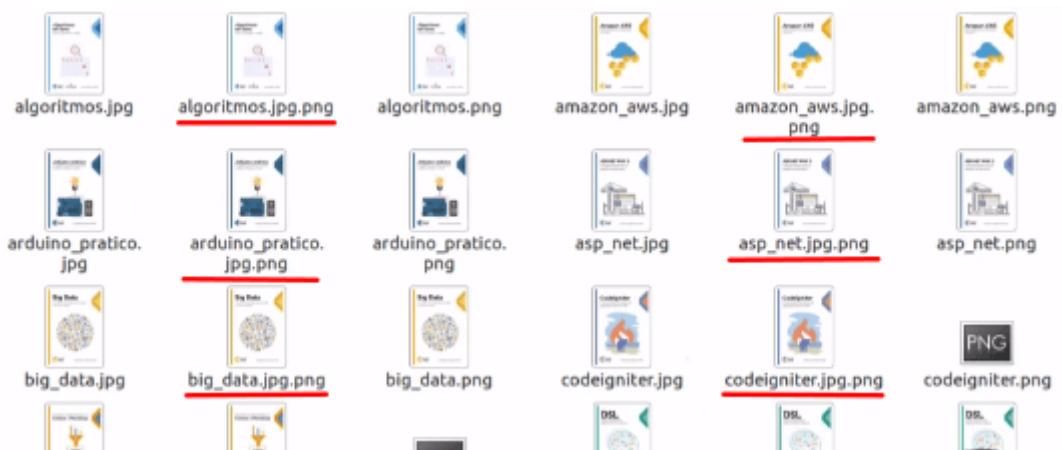
for imagem in *.jpg
do
    convert $imagem ${imagem%.jpg}.png
done
```

[COPIAR CÓDIGO](#)

Salvamos e rodamos novamente o script:

```
$ bash conversao-jpg.png.sh
```

[COPIAR CÓDIGO](#)



Na pasta "imagens-livros", encontramos os arquivos .jpg com o nome correto, porém, os arquivos de extensão .png , estão com o nome diferente, por exemplo: `algoritmos.jpg.png` .

Queremos somente o nome do arquivo, pois assim ficará mais fácil de visualizar. Então, temos esse problema. O nosso script funcionou, mas temos nomes pouco elegantes. A seguir, veremos como podemos solucionar esse problema.

Removendo extensão e salvando em outro diretório

Transcrição

Criamos um script capaz de fazer a varredura em todos os arquivos .jpg que existiam dentro de `imagens-livros`. Vimos que o teste funcionou parcialmente, pois a conversão do arquivo .jpg para .png, não teria ".jpg" no nome. Por exemplo: "algoritmos.jpg.png". Queremos manter apenas o nome do arquivo original.

Faremos alguns testes no terminal e depois acessaremos o script.

Acessaremos a pasta `imagens-livros` e listaremos o conteúdo do diretório com o comando `ls`.

```
$ cd ~/Downloads/imagens-livros/  
$ ls
```

[COPIAR CÓDIGO](#)

Podemos também listar um arquivo acrescentando seu nome após o `ls`:

```
$ ls algoritmos.jpg
```

[COPIAR CÓDIGO](#)

Foram feitas algumas pesquisas e vimos que alguns comandos conseguem **manipular** textos. Um deles é o `awk`!

É necessário redirecionar a saída, para que o `awk` possa tratá-la. Para redirecionar o comando para a saída, usamos o `|` (lê-se *pipe*). Especificamos para o `awk` qual será o **campo delimitador** e qual será o local onde faremos o corte na mensagem exibida.

Onde seria esse corte? Justamente no **ponto** da mensagem, em `algoritmos.jpg`, faremos o corte entre a palavra "algoritmos" e a extensão `.jpg`.

```
$ ls algoritmos.jpg | awk -F. '{ print $1 }'
```

[COPIAR CÓDIGO](#)

O `awk` irá separar a mensagem em duas partes. O comando `-F` define o local de corte. E com o `'{ print $1 }'` conseguimos imprimir a *primeira* parte da mensagem.

Como o comando funcionou no terminal, ele **deve** funcionar no script.

Copiaremos o comando testado e o colaremos dentro do script. Mas, antes, vamos acessá-lo:

```
$ cd ~/Scripts/  
$ nano conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

Dentro do laço `for` pegaremos o conteúdo da `imagem`, que está com a extensão `.jpg`, e usaremos o comando que criamos para usar somente o resultado sem a extensão.

```
#!/bin/bash  
  
cd ~/Downloads/imagens-livros  
  
for imagem in *.jpg
```

```
do
  ls algoritmos.jpg | awk -F. '{ print $1 }'
  convert $imagem $imagem.png
done
```

[COPIAR CÓDIGO](#)

Todavia, pegaremos o **conteúdo** que virá de `imagem`. Vamos aproveitar para *armazenar* somente o resultado em uma outra variável, que chamaremos de `imagem_sem_extensao`.

O armazenamento ocorre quando colocamos `$()` no comando.

```
#!/bin/bash

cd ~/Downloads/imagens-livros

for imagem in *.jpg
do
  imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')
  convert $imagem_sem_extensao.jpg $imagem_sem_extensao.png
done
```

[COPIAR CÓDIGO](#)

Salvamos o script e antes de executá-lo removeremos todo o conteúdo `.png` que está no diretório `imagens-livros`.

```
$ cd ~/Downloads/imagens-livros/
$ rm *.png
```

[COPIAR CÓDIGO](#)

Nesse ponto temos somente o conteúdo `.jpg`. Vamos executar o nosso script.

```
$ ~/Scripts/  
$ bash conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

Certo! Vamos acessar a pasta para ver o resultado. Veremos que foram mantidos somente o nome dos arquivos originais e a eles foi acrescentado a extensão .png

Os diretores da *Multillidae*, nos pediram algumas coisa a mais. A primeira delas é para que guardemos os arquivos convertidos em um diretório próprio chamado de png . Eles também pediram que seja mostrado uma mensagem de **sucesso** ou de **falha** uma vez que o script for executado.

Guardaremos as conversões no diretório png :

```
$ nano conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

Será que dentro do diretório `imagens-livros` , existe o diretório `png` ? Vamos usar o `if` para fazer a verificação.

```
#!/bin/bash  
  
cd ~/Downloads/imagens-livros  
if [ -d png ]  
  
for imagem in *.jpg  
do  
    imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')  
    convert $imagem_sem_extensao.jpg $imagem_sem_extensao.png  
done
```

[COPIAR CÓDIGO](#)

Essa verificação `if [-d png]` verifica somente se o diretório existe. Se o diretório não existe, ele vai ser criado. O símbolo de negação é `!`.

```
if [ ! -d png ]
```

[COPIAR CÓDIGO](#)

Isso significa que, se não existir o diretório `png`, então ele será criado!

```
if [ ! -d png ]
then
    mkdir png
```

[COPIAR CÓDIGO](#)

Uma vez que já passamos a condição, temos que finalizar o comando `if` com `fi`:

```
if [ ! -d png ]
then
    mkdir png
fi
```

[COPIAR CÓDIGO](#)

tudo o que fizemos salvamos com "Ctrl + x" e "yes".

Antes de executar o script, vamos ao diretório `imagens-livros` e removemos todos os arquivos `.png`, pois eles serão salvos dentro do diretório `png`.

```
$ cd ~/Downloads/imagens-livros/
$ rm *.png
$ ls
```

[COPIAR CÓDIGO](#)

Voltemos ao script.

```
$ bash conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

Vamos ver o resultado. Dentro de `imagens-livros` temos o diretório `png`, onde estão armazenado todos os arquivos convertidos nesse formato.

Tivemos um avanço, mas nosso trabalho não para por aqui. Assim que o script for executado, temos que saber se a conversão ocorreu com sucesso ou se houveram falhas.

Mãos à obra: Convertendo arquivos do diretório

Agora nós temos a missão de converter todos os arquivos dentro do diretório **imagens-livros** para o formato .png. Como primeiro passo, vá até o diretório **imagens-livros** e remova todos os arquivos com a extensão **.png**, você poderá fazer isso indo no terminal e digitando `cd ~/Downloads/imagens-livros` e depois digitando `rm *.png`. Uma vez que não temos mais os arquivos **.png** nesse diretório, abra seu script com seu editor de texto de preferência, por exemplo **nano conversao-jpg-png.sh**.

O primeiro passo será "entrar" no diretório **imagens-livros**, para isso, coloque na primeira linha do seu script o comando

```
cd ~/Downloads/imagens-livros
```

Uma vez que estamos "dentro" do diretório **imagens-livros**, vamos consolidar todos os arquivos convertidos em um diretório chamado de **png**. Não sabemos se existe um diretório **png** dentro do diretório **imagens-livros**, dessa forma, nós podemos pedir para o nosso script que se não existir um diretório **png**, que o script crie tal diretório. Logo após o trecho do código `cd ~/Downloads/imagens-livros`, faça essa verificação com o seguinte código

```
if [ ! -d png ]
then
    mkdir png
fi
```

[COPIAR CÓDIGO](#)

Obs: É necessário separar o if, os colchetes e o conteúdo, com espaço

Uma vez que essa verificação foi feita, temos certeza que agora teremos um diretório **png** dentro do nosso diretório **imagens-livros**. Feito isso, devemos agora fazer a varredura dos arquivos presentes no diretório **imagens_livros** que possuem o formato **.jpg**. Não podemos nos esquecer que queremos apenas o nome, sem a extensão **.jpg**, então usaremos o **awk** para nos ajudar nessa tarefa. Devemos ter o seguinte trecho de código para realizar essa etapa:

```
for imagem in *.jpg
do
    imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')
    convert $imagem_sem_extensao.jpg png/$imagem_sem_extensao.|done
```

[COPIAR CÓDIGO](#)

Obs: Não podemos ter espaço entre o nome da variável, o símbolo **do =** e o conteúdo da variável

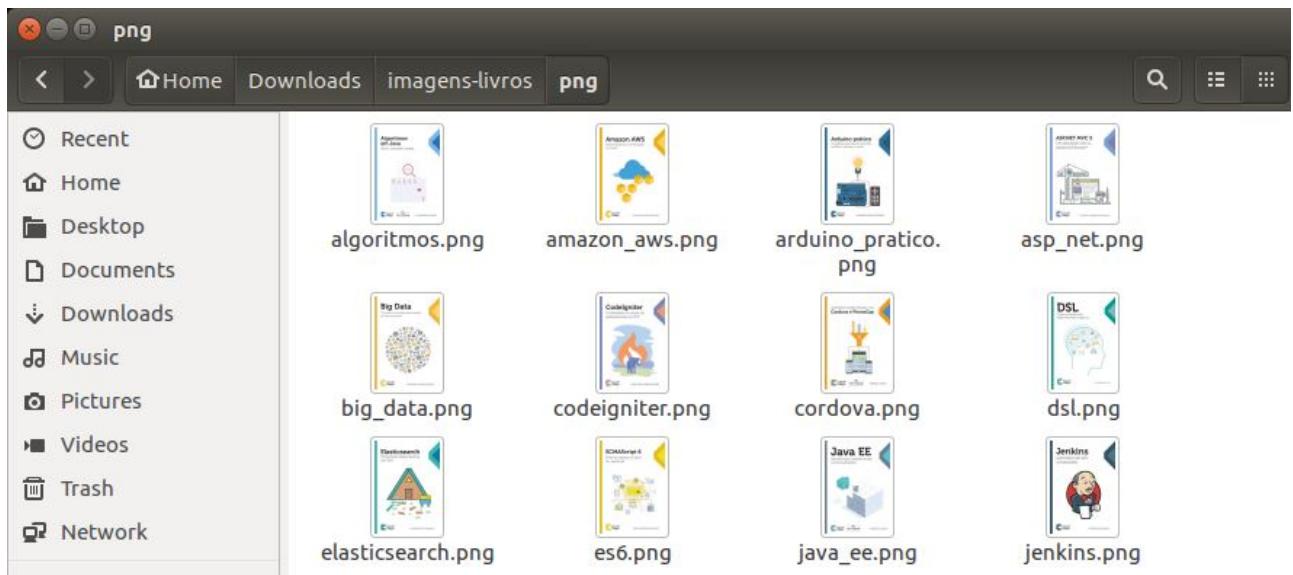
Tente executar seu script. Qual é o resultado?

Opinião do instrutor

Nosso script está verificando no diretório **imagens-livros** todos os arquivos que possuem a extensão ***.jpg**. Através do **awk**, nós estamos especificando como sendo o campo delimitador o **.** e estamos pedindo para imprimir somente o primeiro valor (**\$1**) que irá conter o nome do arquivo sem a extensão.

Uma vez que essa etapa foi concluída, nós pegamos o conteúdo da variável **imagem_sem_extensao** e realizamos a conversão do formato **jpg** para o formato **png** e estamos salvando os resultados no diretório **png**

Ao executarmos o script, deveremos ver todos os arquivos no formato png presentes no diretório **png**



O script deverá estar parecido com o abaixo:

```
#!/bin/bash

cd ~/Downloads/imagens-livros
if [ ! -d png ]
then
    mkdir png
fi

for imagem in *.jpg
do
    imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')
    convert $imagem_sem_extensao.jpg png/$imagem_sem_extensao.png
done
```

Status de saída

Transcrição

Conseguimos montar um script, capaz de remover a extensão .jpg que está presente nos arquivos originais, além de verificar se o diretório png existia dentro da pasta imagens-livros .

Dentro do laço for , estamos passando todo o conteúdo das imagens .png para o diretório png . Entretanto, há outra etapa a ser feita. Foi pedido para passar uma mensagem de sucesso ou de falha quando ocorresse o processo de conversão das imagens.

Analisaremos o código a seguir:

```
#!/bin/bash

cd ~/Downloads/imagens-livros
if [ ! -d png ]
then
    mkdir png
fi

for imagem in *.jpg
do
    imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')
    convert $imagem_sem_extensao.jpg png/$imagem_sem_extensao.png
done
```

[COPIAR CÓDIGO](#)

Basicamente, esse código está fazendo a conversão de imagens. Assim como foi visto no curso de [Lógica de Programação](https://cursos.alura.com.br/course/logica-programacao-javascript-html?preRequirementFrom=shells scripting) (<https://cursos.alura.com.br/course/logica-programacao-javascript-html?preRequirementFrom=shells scripting>), conseguimos isolar o código em uma função!

No começo do script, criaremos uma função que terá o nome de `converte_imagem()` e englobará todo o código. Portanto, é necessário "abrir e fechar chaves".

```
#!/bin/bash

converte_imagem(){
cd ~/Downloads/imagens-livros
if [ ! -d png ]
then
    mkdir png
fi

for imagem in *.jpg
do
    imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')
    convert $imagem_sem_extensao.jpg png/$imagem_sem_extensao.png
done
}
```

COPIAR CÓDIGO

Uma vez criada essa função, precisamos **invocá-la** apenas colocando o nome dela. É muito importante colocar a invocação da criação depois que ela for criada. Devido à execução dos comandos que são executados em ordem sequencial.

Repare que dentro da função `converte_imagem()` há variáveis que foram criadas, por exemplo, a variável `imagem_sem_extensão`. Uma vez que essa variável foi criada,

dentro da função `converte_imagem()` o conteúdo dela só deveria ser acessado dentro do escopo da função em que ela está.

Faremos um teste. Vamos ver o que acontece se pedirmos para imprimir o conteúdo da variável, estamos fora da função.

```
#!/bin/bash

converte_imagem(){
cd ~/Downloads/imagens-livros
if [ ! -d png ]
then
    mkdir png
fi

for imagem in *.jpg
do
    imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')
    convert $imagem_sem_extensao.jpg png/$imagem_sem_extensao.png
done
}

converte_imagem

echo $imagem_sem_extensao
```

[COPIAR CÓDIGO](#)

Salvamos e executamos esse arquivo.

```
$ bash conversao-jpg-png.sh
```

[COPIAR CÓDIGO](#)

Podemos ver, que foi impresso o nome "zend". Zend é o último arquivo que tem ^ em nosso diretório `imagens-livros`. E quando pedimos para imprimir a variável

`imagem_sem_extenso`, o último valor que ela tinha era o "zend".

Isso significa que conseguimos acessar o conteúdo da variável `imagem_sem_extenso` estando fora da função em que ela foi criada. Obviamente, não queremos que isso aconteça. E sim, que essa variável **somente** exista dentro da função `converte_imagem()`, afim de evitar o vazamento de escopo.

Então, para dizer que essa variável só pode ser acessada dentro dessa função convertemos a imagem e colocamos o termo **local**. **Local** indica que o conteúdo dessa variável, só pode ser acessado dentro do `converte_imagem()`.

```
for imagem in *.jpg
do
    local imagem_sem_extenso=$(ls $imagem | awk -F. '{ print $1 }
    convert $imagem_sem_extenso.jpg png/$imagem_sem_extenso.png
done
```

[COPIAR CÓDIGO](#)

Vamos salvar e rodar novamente esse script, para ver se conseguimos acessar o conteúdo dessa variável fora do escopo do método.

Observando o resultado vemos que nenhum dado foi impresso! Isto ocorreu devido ao uso da palavra **local** antes da variável, e assim o conteúdo só é acessado dentro da função.

Voltemos ao script. Vamos focar na tarefa que os diretores nos pediram: Se o script for executado com sucesso ou se ocorre algum problema na conversão das imagens.

Quando executamos um comando no Linux, ele emite um **status de saída**. Quando o comando é executado *com sucesso*, o status de saída é zero (0). Já quando ocorre falhas, esse status de saída pode variar de 1 a 255.

Nossa estratégia é verificar o status de saída da função `converte_imagem()`. Se ele for zero, ela obteve sucesso na conversão. Se for diferente de zero, ocorre alguma falha. Para isso, usaremos um `if`:

```
converte_imagem
if [ $? -eq 0 ]
then
    echo "Conversão realizada com sucesso"
else
    echo "Houve uma falha no processo"
fi
```

[COPIAR CÓDIGO](#)

* Vamos entender:

Usamos `$?` para pegar o **status de saída** da função `converte_imagem()`. Com o `-eq`, dizemos que alguma coisa é igual a outra, no caso, comparamos o valor do status de saída com o número zero. Se for igual a zero, imprime-se "Conversão realizada com sucesso". Se *não* for zero, é porque houve algum problema, então imprime "Houve uma falha no processo". Não podemos nos esquecer que, sempre que abrimos um `if`, temos que fechá-lo também com `fi`.

Salvamos e executamos o script para verificar os resultados com as novas alterações.

Observe, a mensagem "Conversão realizada com sucesso" apareceu para nós! Então, além de conseguir a conversão, colocamos uma mensagem para o usuário dizendo que a conversão foi realizada com sucesso.

Agora, forçaremos um erro em nosso script, para ver se de fato ele está funcionando. Para simular um erro, colocamos um diretório que **não existe**.

```
#!/bin/bash

converte_imagem(){
cd ~/Downloads/IMAGENS-LIVROS
if [ ! -d png ]
```

[COPIAR CÓDIGO](#)

Sabemos que não existe esse diretório, então esperamos que ele acuse um erro, esperamos que apareça a mensagem "Houve uma falha no processo", assim saberemos que o script está funcionando corretamente. Vamos salvar e rodar o script e o resultado é exatamente como esperávamos: "Houve uma falha no processo". Mas, também há uma mensagem no log na qual está sendo descrito o erro. Essa informação não é relevante para o usuário, basta ele saber que o processo deu certo ou não.

Então, vamos redirecionar essa mensagem de erro para um arquivo, assim, um administrador de sistemas poderá entender porque a conversão não foi feita. Será necessário utilizar **descritores de arquivos** que são indicadores de acesso de arquivos e recursos de entrada e saída.

Os fluxos padrões existentes em nosso programa são os de entrada, o de saída e os das mensagens de erro.

Um exemplo para o fluxo de entrada é quando digitamos no teclado. Um exemplo para o fluxo de saída é o resultado que temos. Já o fluxo das mensagens de erro é como o erro apresentado acima.

Esses fluxos padrões são **referenciados por números**.

- 0 é referenciado para entrada padrão
- 1 é referenciado para saída padrão
- 2 é referenciado para mensagens de erro

Queremos redirecionar as mensagens de erro padrão para um arquivo, para que o administrador de sistemas possa analisar com mais detalhes o que exatamente aconteceu.

Voltemos ao script utilizando o comando `nano conversao-jpg-png.sh`. Queremos redirecionar as mensagens de erro, caso ocorra alguma durante o processo, para outro arquivo.

```
#!/bin/bash

converte_imagem(){
cd ~/Downloads/imagens-livros
if [ ! -d png ]
then
    mkdir png
fi

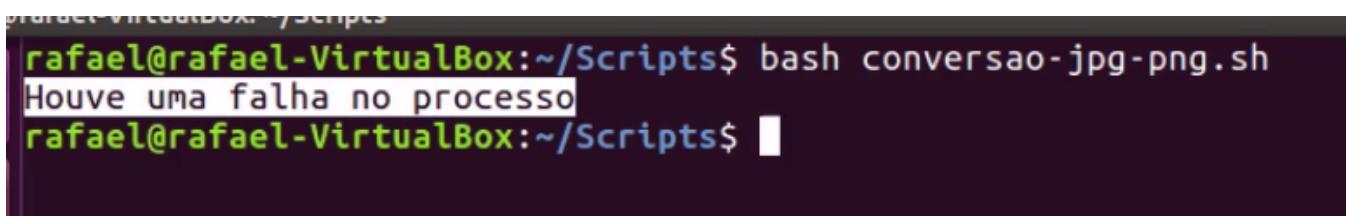
for imagem in *.jpg
do
    imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')
    convert $imagem_sem_extensao.jpg png/$imagem_sem_extensao.png
done
}

converte_imagem 2>erros_conversao.txt
if [ $? -eq 0 ]
then
    echo "Conversão realizada com sucesso"
else
    echo "Houve uma falha no processo"
fi
```

[COPIAR CÓDIGO](#)

Esperamos que todas as mensagens de erros, passem para o arquivo `erros_conversao.txt` e que só seja exibido a mensagem "Houve uma falha no

processo". Vamos salvar e ver se de fato está tudo funcionando.

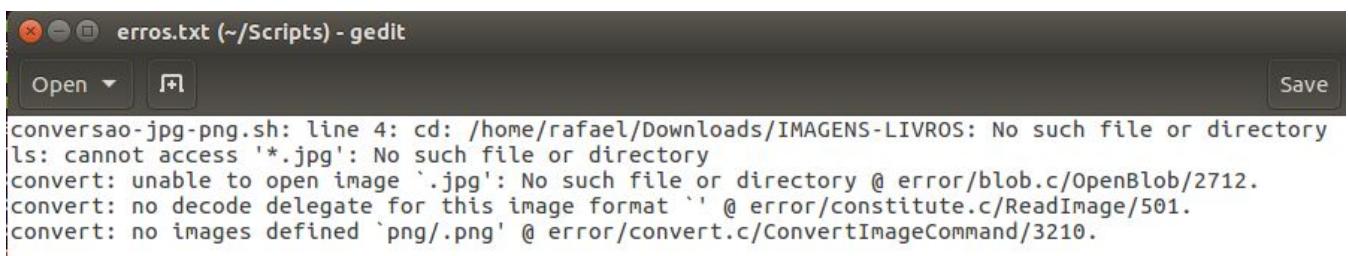


```
rafael@rafael-VirtualBox:~/Scripts$ bash conversao-jpg-png.sh
Houve uma falha no processo
rafael@rafael-VirtualBox:~/Scripts$
```

Nos é retornada a mensagem "Houve uma falha no processo". Será que temos o arquivo de texto contendo as mensagens de erro?

Estamos executando o script dentro do diretório `/Scripts` e não tínhamos referenciado nenhum outro local para salvar o arquivo `erros_conversao.txt`. Logo, ele **deve** estar presente dentro do diretório `/Scripts`. Usando o comando `ls`, conseguimos, de fato, encontrá-lo lá.

E o que encontramos é...



```
erros.txt (~/Scripts) - gedit
Open Save
conversao-jpg-png.sh: line 4: cd: /home/rafael/Downloads/IMAGENS-LIVROS: No such file or directory
ls: cannot access '*.jpg': No such file or directory
convert: unable to open image `*.jpg': No such file or directory @ error/blob.c/OpenBlob/2712.
convert: no decode delegate for this image format `*' @ error/constitute.c/ReadImage/501.
convert: no images defined `png/.png' @ error/convert.c/ConvertImageCommand/3210.
```

...O conteúdo da mensagem de erro!

Conseguimos, dessa forma, fazer as duas requisições dos diretores:

1º - fazer a conversão automática de todos os arquivos dentro do diretório `imagens-livros`, colocando todos os arquivos `.png` dentro do diretório `png` ;

2º - por meio do status de saída, verificar se a função `converte_imagem()` está sendo executada com sucesso, tendo o status de saída igual a zero ou caso contrário, terá o status diferente de zero e com isso conseguimos redirecionar a mensagem de erro para o arquivo `erros_conversao.txt`.

Vamos seguir em frente para mais uma aula.

Mãos à obra: Verificando o status de saída

Uma vez que realizamos a conversão dos arquivos, é importante que nós façamos uma validação para saber se de fato a execução dos comandos foi realizada com sucesso ou se houve algum problema, para que possamos assim notificar o usuário.

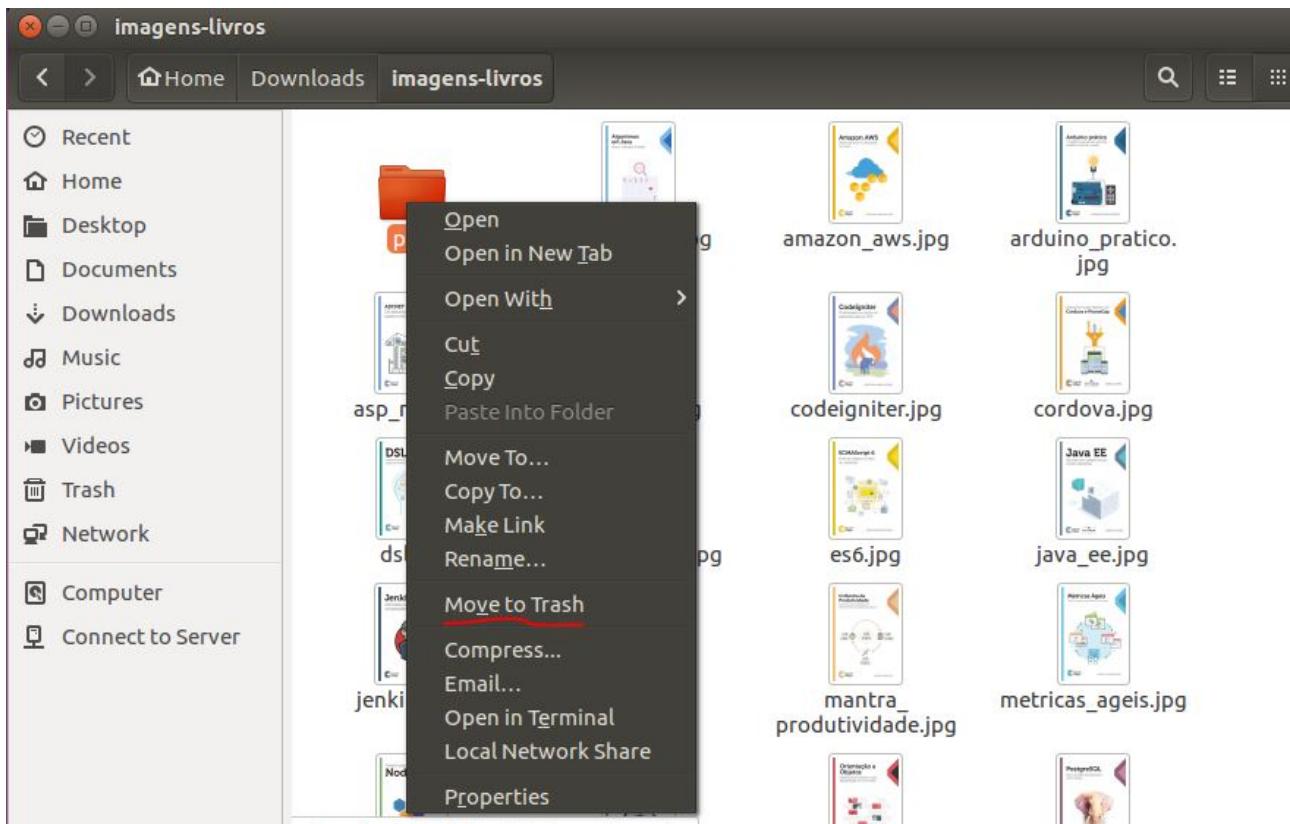
Para isso, abra seu script com seu editor de texto preferido, por exemplo **nano conversao-jpg-png.sh** e envolva todo o código criado até então, dentro de uma função, por exemplo uma função chamada **converte_imagem**

Uma vez que a função foi declarada, será necessário chamá-la, utilizando para isso o nome da função criada. Ao chamarmos a função, vamos verificar o status de saída da execução dessa função, se for 0 é porque a mesma foi executada com sucesso e a conversão dos arquivos foi realizada, se for diferente de 0 é porque tivemos algum problema. Faça essa verificação com o seguinte trecho de código:

```
if [ $? -eq 0 ]
then
    echo "Conversão realizada com sucesso"
else
    echo "Houve uma falha no processo de conversão"
fi
```

[COPIAR CÓDIGO](#)

Uma vez que fizemos essas alterações no nosso script, vamos remover o diretório **png** e realizar novamente a conversão dos arquivos para ver se a conversão é realizada e se teremos a mensagem mostrada no terminal. Para remover o diretório **png** clique com o botão direito do mouse e escolha a opção **Move to trash**



Execute o script. Qual é o resultado?

Opinião do instrutor

Ao executar o script, nós vemos que a conversão dos arquivos foi realizada com sucesso e agora somos capazes de ver a mensagem que havíamos colocado no script.

```
rafael@rafael-VirtualBox:~/Scripts$ bash conversao-jpg-png.sh
Conversão realizada com sucesso
rafael@rafael-VirtualBox:~/Scripts$
```

Nosso script deverá estar parecido com o cenário abaixo:

```
#!/bin/bash

converte_imagem(){
cd ~/Downloads/imagens-livros
if [ ! -d png ]
then
    mkdir png
fi

for imagem in *.jpg
do
    local imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')
    convert $imagem_sem_extensao.jpg png/$imagem_sem_extensao.png
done
}

converte_imagem
if [ $? -eq 0 ]
then
    echo "Conversão realizada com sucesso"
else
    echo "Houve erro na conversão"
fi
```

Opcional: Mão à obra - Redirecionando mensagens de erro

Os descritores de arquivos são indicadores utilizados para acessar um arquivo ou fluxos padrões como entrada, saída, e mensagens de erros. Esses fluxos padrões de entrada, saída e mensagens de erro podem ser acessados pelos descritores 0, 1 e 2 respectivamente.

No script que montamos, redirecione as mensagens de erro da função **converte_imagem** para um arquivo chamado de erros.txt. Uma vez que queremos redirecionar as mensagens de erro, devemos utilizar o descritor de arquivo 2. Para redirecionar essa saída para um arquivo, nós colocamos o símbolo de >. O trecho do código deverá ficar conforme abaixo:

```
converte_imagem 2>erros.txt
```

[COPIAR CÓDIGO](#)

Agora, para verificar se o script está funcionando como esperado. Troque o caminho de acesso do diretório **imagens-livros** para um nome que não existe, por exemplo **IMAGENS-LIVROS**, execute o script e depois tente abrir o arquivo **erros.txt**. Qual é o resultado?

Obs: Depois do exercício, para voltar o funcionamento do script, coloque o nome correto do seu diretório

Opinião do instrutor



Nosso script ao verificar a linha para alterar o diretório para **IMAGENS-LIVROS** irá enviar uma mensagem de erro dizendo que tal diretório não pode ser acessado, com isso, nós teríamos essa mensagem de erro sendo mostrada no terminal, mas quando estamos utilizando o trecho **2>erros.txt**, estamos redirecionando essa saída para o arquivo **erros.txt**. Dessa forma, ao abrirmos o arquivo, devemos visualizar as mensagens de erros



The screenshot shows a Gedit window titled "erros.txt (~/Scripts) - gedit". The window contains the following text:

```
conversao-jpg-png.sh: line 4: cd: /home/rafael/Downloads/IMAGENS-LIVROS: No such file or directory
ls: cannot access '*.jpg': No such file or directory
convert: unable to open image `*.jpg': No such file or directory @ error/blob.c/OpenBlob/2712.
convert: no decode delegate for this image format '' @ error/constitute.c/ReadImage/501.
convert: no images defined 'png/*.png' @ error/convert.c/ConvertImageCommand/3210.
```

Nosso script deverá estar parecido com o cenário abaixo:

```
#!/bin/bash

converte_imagem(){
cd ~/Downloads/IMAGENS-LIVROS
if [ ! -d png ]
then
    mkdir png
fi

for imagem in *.jpg
do
    local imagem_sem_extensao=$(ls $imagem | awk -F. '{ print $1 }')
    convert ${imagem_sem_extensao}.jpg png/${imagem_sem_extensao}.png
done
}

converte_imagem 2>erros.txt
if [ $? -eq 0 ]
then
    echo "Conversão realizada com sucesso"
else
    echo "Houve erro na conversão"
fi
```

Imagens espalhadas em vários diretórios

Transcrição

Na etapa anterior, os diretores da *Multillidae* haviam nos passado o diretório `imagens-livros` para realizarmos a conversão dos arquivos da extensão `.jpg` para a extensão `.png`.

Com o passar dos meses, a *Multillidae* cresceu muito, e eles repararam que a divisão de colocar todos os arquivos `.jpg` dentro de somente um diretório, não estava ajudando muito, pois por causa do volume de dados, acabava dificultando o manuseio com as imagens.

Então, eles decidiram dividir os arquivos `.jpg` em diretórios respectivos a sua área de conhecimento. Recebemos um novo diretório chamado de `imagens-novos-livros`, onde devemos também fazer a conversão dos arquivos `.jpg` para `.png`.

Esse diretório está estruturado da seguinte maneira:



Como podemos ver, ela está dividida por áreas de conhecimento. Quando clicamos em um diretório, encontramos outros diretórios e outras imagens, e assim por diante.

Acontece que o script que criamos nas aulas anteriores, não pode nos ajudar nessa tarefa, pois o script anterior considerava que todas as imagens estariam dentro do mesmo diretório.

Agora temos vários diretórios, que contêm vários diretórios, e que estes contêm vários outros diretórios, e também, várias imagens espalhadas nesses diretórios.

Criaremos um **novo script** para atender as requisições dos diretores da *Multillidae*.

Vamos ao terminal, e atualmente estamos no diretório `/Scripts`, onde criaremos esse novo script para atender essa requisição.

```
$ nano conversao-novos-livros.sh
```

[COPIAR CÓDIGO](#)

Como já foi comentado, a primeira linha do script deve conter o **interpretador**.

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

[COPIAR CÓDIGO](#)

O primeiro passo a partir de agora, é sair do diretório `/Scripts`, e chegar no diretório `/imagens-novos-livros`, onde estão os arquivos que sofrerão a conversão:

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

```
cd ~/Downloads/imagens-novos-livros
```

[COPIAR CÓDIGO](#)

Uma vez que essa linha for executada, estaremos dentro do diretório `/imagens-novos-livros`. Uma vez estando nessa pasta, é necessário realizar uma varredura de todo o conteúdo dela. Para isso, utilizaremos a estrutura de repetição **for!!**

Junto com o `for`, colocaremos o nome de uma variável, e depois um `*`, dizendo que será feito uma varredura de todo o conteúdo dentro de `/imagens-novos-livros`.

```
#!/bin/bash

cd ~/Downloads/imagens-novos-livros
for arquivo in *
```

[COPIAR CÓDIGO](#)

Após a linha do `for`, implementaremos o comando de **execução** do `.`. Dentro de `/imagens-novos-livros`, podemos encontrar arquivos que sejam **diretórios**, ou até mesmo **imagens** que estão no formato `.jpg`. Com isso, temos que analisar esses dois casos. (*Se é um diretório, ou se é uma imagem*).

Dentro do `if`, usaremos o comando `-d` para verificar se o conteúdo da variável `arquivo` é um diretório. Se for um *diretório*, temos que acessá-lo e fazer uma varredura do conteúdo.

Se o conteúdo dessa variável não é um diretório, então é uma imagem. Portanto, faremos a conversão do arquivo `.jpg` para `.png`.

```
for arquivo in *
do
  if [ -d $arquivo ]
  then
    #Entrar no diretório e fazer varredura do conteúdo
  else
    #Conversao jpg para png
  fi
done
```

[COPIAR CÓDIGO](#)

Imagine que o conteúdo de `arquivo` é um diretório, por exemplo o `/backend`. Como foi falado, temos que entrar nesse diretório, e realizar uma varredura do conteúdo. E é isso que faremos agora no código:

```
for arquivo in *
do
    if [ -d $arquivo ]
    then
        cd $arquivo
        for conteudo_arquivo in *
        do
            if [ -d $conteudo_arquivo ]
            then
                #Entrar no diretorio e fazer a varredura
            else
                #Conversao jpg para png
            fi
        done
    else
        #Conversao jpg para png
    fi
done
```

[COPIAR CÓDIGO](#)

Após acessar o diretório `cd $arquivo`, fizemos a varredura utilizando o laço de repetição `for`.

Perceba que esse código é exatamente igual ao código externo. E vimos que ficar repetindo código não é a solução mais elegante que podemos ter. Com isso, na sequência veremos como podemos solucionar esse problema.

Chamando a função dentro dela mesma

Transcrição

Montamos um script que procura por um arquivo e faz a comparação para saber se esse arquivo é um diretório ou não. Caso ele seja um diretório, temos que acessá-lo, fazer a varredura novamente para poder verificar se o arquivo do próximo diretório é outro diretório ou um arquivo de imagem.

Perceba que, conforme acessamos os diretórios, temos que fazer esse mesmo processo várias e várias vezes!

A parte interna do bloco `if`, é exatamente igual a parte externa. E como já vimos, ficar repetindo código não é uma boa prática e nem a solução mais elegante que podemos adotar.

Vamos alterar a parte interna do código para tentar resolver esse problema. Abriremos o mesmo script, porém, utilizando o *Gedit* para ficar mais fácil de editar as informações.

```
$ gedit conversao-novos-livros.sh
```

COPIAR CÓDIGO

Apagaremos toda a parte interna no bloco do `if` que está se repetindo.

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

```
cd ~/Downloads/imagens-novos-livros  
for arquivo in *
```

```
do
  if [ -d $arquivo ]
  then

  else
    #Conversao jpg para png
  fi
done
```

[COPIAR CÓDIGO](#)

Se nessa parte do código, conseguíssemos de alguma forma **chamar** o bloco externo, nosso problema estaria resolvido!

Como faremos? Colocaremos o código dentro de uma função e chamaremos ela no `then` do `if`. Ou seja, estaremos chamando a função **dentro dela mesma** e, assim, criando uma **estrutura de recursão**!

Envolveremos esse bloco de código em uma função chamada `varrer_diretorio()`:

```
#!/bin/bash
varrer_diretorio(){
  cd ~/Downloads/imagens-novos-livros
  for arquivo in *
  do
    if [ -d $arquivo ]
    then

    else
      #Conversao jpg para png
    fi
  done
}
```

[COPIAR CÓDIGO](#)

Dentro do `if`, como os passos se repetem, chamaremos a função `varrer_diretorio`. Entretanto, precisamos prestar atenção em um ponto: quando chegar a hora de chamar a função `varrer_diretorio()`, temos que passar para ela o **conteúdo da variável** `arquivo`. Para passar um parâmetro para a função é só colocá-lo após a chamada do método.

```
do
  if [ -d $arquivo ]
  then
    varrer_diretorio $arquivo
  else
    #Conversao jpg para png
  fi
done
```

[COPIAR CÓDIGO](#)

Uma vez que passarmos esse parâmetro para a função `varrer_diretorio()`, faremos a varredura. Como fazemos para pegar o parâmetro assim como fizemos anteriormente? Utilizamos `$1` + [numero que se refere à posição do parâmetro]!

```
#!/bin/bash
varrer_diretorio(){
$1
cd ~/Downloads/imagens-novos-livros
for arquivo in *
do
  if [ -d $arquivo ]
  then
    varrer_diretorio $arquivo
  else
    #Conversao jpg para png
  fi
done
}
```

[COPIAR CÓDIGO](#)

Então, quando a linha `varrer_diretorio $arquivo` for executada a função `varrer_diretorio()` será chamada e passará o conteúdo da variável `arquivo` como **parâmetro** para essa função.

Pegamos o conteúdo dessa variável pelo **\$1**. Mas, queremos entrar nesse diretório e fazer a varredura. Na linha `cd ~/Downloads/imagens-novos-livros` já estamos alterando o diretório para `/imagens-novos-livros` e, assim, estaremos sempre entrando no diretório que vai ser passado como parâmetro no bloco do `if`.

Ao invés de colocar um caminho estático na linha do comando `cd`, deixaremos claro para sempre estar entrando no diretório que será passado pelo bloco dentro do `if`. Com essa ideia, já conseguimos resolver um grande problema.

Vamos dar uma olhada de como está a estrutura do diretório `/imagens-novos-livros`.

Logo que entramos na pasta, vemos alguns diretórios e algumas imagens. E quando acessamos alguns desses diretórios, encontramos outros diretórios, um dentro do outro.

Para não perder a referência dos locais desses arquivos e diretórios é importante que tenhamos o caminho completo deles!

Voltemos ao terminal para realizar alguns testes.

Então, queremos *encontrar o caminho completo* de um diretório ou de um arquivo que está dentro do diretório `/imagens-novos-livros`. Usaremos o comando `find` passando uma dica de onde queremos que ele procure o arquivo ou diretório.

Sabemos que, as imagens dos livros estão espalhadas em vários diretórios, mas todos eles tem `/imagens-novos-livros` como **diretório pai**!

Então, vamos falar para o comando `find`, para que ele encontre essa informação dentro da home (`~/Downloads/imagens-novos-livros`).

Dessa forma, estamos falando pra ele encontrar esse diretório ou esse arquivo de imagem que vai estar dentro de `/imagens-novos-livros`. E qual é o nome desse arquivo? Usamos o comando `-name` para passar o nome do arquivo ou diretório. Pegaremos como exemplo o diretório `/java_basico`, que está dentro de `/java`, que por sua vez está dentro de `/backend`, que está dentro de `/imagens-novos-livros`, que está dentro de `/Downloads`, que está, por fim, dentro de `/home`.

Veremos se esse comando mostrará o caminho completo da pasta `java_basico`

```
$ find ~/Downloads/imagens-novos-livros -name java_basico
```

[COPIAR CÓDIGO](#)

O que temos é justamente a impressão desse caminho completo! Veja o que foi exibido:

```
/home/rafael/Downloads/imagens-novos-livros/backend/java/java_basico
```

Com isso, não perderemos mais a referência de onde o diretório está localizado. Vamos copiar esse comando, para colar em nosso script.

Quando o laço `for` do script estiver fazendo a varredura, podemos pegar o caminho completo da variável `arquivo`.

```
#!/bin/bash

varrer_diretorio(){
    cd $1
    for arquivo in *
    do
```

```
find ~/Downloads/imagens-novos-livros -name $arquivo
if [ -d $arquivo ]
then
    varrer_diretorio $arquivo
else
    #Conversao jpg para png
fi
done
}
```

[COPIAR CÓDIGO](#)

Trocamos a variável estática `java_basico` pelo conteúdo de `arquivo`, assim podemos pegar o caminho **completo** do `arquivo`. Armazenaremos o **resultado** dentro da variável `caminho_arquivo`. Não podemos nos esquecer de colocar todo esse comando entre `$()`, assim estamos dizendo que este é um comando que deve ser executado e que o seu resultado será armazenado na variável `caminho_diretorio`.

```
#!/bin/bash

varrer_diretorio(){
    cd $1
    for arquivo in *
    do
        caminho_arquivo=$(find ~/Downloads/imagens-novos-livros -
            if [ -d $arquivo ]
            then
                varrer_diretorio $arquivo
            else
                #Conversao jpg para png
            fi
        done
    }
```

[COPIAR CÓDIGO](#)

Como a variável está dentro da função `varrer_diretorio()` dizemos que ela é uma **variável local**, assim, evitamos o vazamento de escopo, como já visto. Agora, vamos fazer toda referência dos diretórios e dos arquivos pelo caminho completo deles, já que temos a variável local `caminho_arquivo`.

```
if [ -d $caminho_arquivo ]
then
    varrer_diretorio $caminho_arquivo
else
    #Conversao jpg para png
fi
```

[COPIAR CÓDIGO](#)

Dessa forma, estamos passando o caminho completo do diretório ou arquivo, *sem perder a referência*.

O que nos resta agora é tratar da parte de converter as imagens de `.jpg` para `.png`. Criamos uma função que será responsável somente por converter as imagens. Vamos chamá-la de `converte_imagem`.

Como boa prática, sempre temos que colocar a função **antes** dela ser invocada.

```
#!/bin/bash

converte_imagem(){

}

varrer_diretorio(){
    cd $1
    for arquivo in *
```

```
do
    caminho_arquivo=$(find ~/Downloads/imagens-novos-livros -
    if [ -d $caminho_arquivo ]
    then
        varrer_diretorio $caminho_arquivo
    else
        converte_imagem
    fi
done
}
```

[COPIAR CÓDIGO](#)

Quando o bloco do `else` for executado é porque o conteúdo da variável `$caminho_arquivo` **não é um diretório!** Se ele não for um diretório, então ela só pode ser uma imagem! Precisamos passar essa imagem para a função `converte_imagem()`.

```
if [ -d $caminho_arquivo ]
then
    varrer_diretorio $caminho_arquivo
else
    converte_imagem $caminho_arquivo
fi
```

[COPIAR CÓDIGO](#)

Uma vez que o parâmetro é passado para o `converte_imagem()`, temos que pegá-lo, da mesma forma que fizemos com `varrer_diretorio()`.

Para pegar o parâmetro, utilizamos `$1`, mas, antes, colocaremos o `$` em uma variável local, prezando pela legibilidade do código:

```
#!/bin/bash

converte_imagem(){
```

```
local caminho_imagem=$1
}

varrer_diretorio(){
    // laço de repetição
}
```

[COPIAR CÓDIGO](#)

Agora que temos o caminho completo da imagem, precisamos **remover** a extensão **.jpg** do caminho da imagem. Faremos essa tarefa com a ajuda do **awk** ! Adicionaremos outra variável local chamada **imagem_sem_extensao**, para que seja armazenado nessa variável o resultado do comando a seguir:

```
converte_imagem(){
    local caminho_imagem=$1
    local imagem_sem_extensao=$(ls $caminho_imagem | awk -F. '{ p
}
```

[COPIAR CÓDIGO](#)

Nos resta realizar a conversão das imagens.

Queremos converter o conteúdo da variável **\$imagem_sem_extensao** utilizando o comando **convert**. Vamos converter para o formato **.png**.

```
converte_imagem(){
    local caminho_imagem=$1
    local imagem_sem_extensao=$(ls $caminho_imagem | awk -F. '{ p
    convert $imagem_sem_extensao.jpg $imagem_sem_extensao.png
}
```

[COPIAR CÓDIGO](#)

O script está quase completo. Só nos resta chamar a função **varrer_diretorio()** passando o diretório pai, para iniciar todo o processo de varredura.

```

#!/bin/bash

converte_imagem(){
    local caminho_imagem=$1
    local imagem_sem_extensao=$(ls $caminho_imagem | awk -F. '{ p
convert $imagem_sem_extensao.jpg $imagem_sem_extensao.png
}

varrer_diretorio(){
    cd $1
    for arquivo in *
    do
        local caminho_arquivo=$(find ~/Downloads/imagens-novos-li
        if [ -d $caminho_arquivo ]
        then
            varrer_diretorio $caminho_arquivo
        else
            converte_imagem $caminho_arquivo
        fi
    done
}

varrer_diretorio ~/Downloads/imagens-novos-livros

```

[COPIAR CÓDIGO](#)



É necessário conferir se o script realmente conseguiu fazer a conversão ou se teve algum problema. Verificaremos o **status de saída** da execução! Se ele for **zero**, é porque tudo foi realizado com sucesso e se for **diferente de zero**, houve algum problema, Vamos colocar essa mensagem no terminal:

```

varrer_diretorio ~/Downloads/imagens-novos-livros
if [ $? -eq 0 ]
then
    echo "Conversão realizada com sucesso"

```

```
else
    echo "Houve um problema na conversão"
fi
```

[COPIAR CÓDIGO](#)

Vamos testar esse script, esperamos que ele funcione como esperado. Utilizamos o "Ctrl + x" para sair e "y" para salvar as alterações e executamos o script!

```
$ bash conversao-novos-livros.sh
```

[COPIAR CÓDIGO](#)

Após instantes, obtemos o seguinte resultado:

Conversao realizada com sucesso

Quando abrimos o diretório, nos deparamos com os arquivos .jpg e seus respectivos arquivos .png . Em todas as pastas, vamos encontrar esses arquivos.

A tarefa de montar um script que realizasse a conversão das imagens espalhadas em vários diretórios foi concluída com sucesso!

Mãos à obra: Fazendo a varredura em vários diretórios

Nossa missão agora será converter as imagens de vários arquivos passados pelos diretores da Mutillidae que se encontram em vários diretórios. Faça o download desse novo diretório nesse link: https://caelum-online-public.s3.amazonaws.com/shell-script/aula_3/imagens-novos-livros.zip

O próximo passo será entrar no diretório de **Scripts** e criar um novo script, o chamaremos de **conversao-diferentes-diretorios.sh**. Uma vez que o script foi criado, vamos colocar na primeira linha o interpretador do nosso script que será **#!/bin/bash**. Feito isso, vamos começar a montar nosso script, nossa atenção será agora montar a função para "varrer" os diretórios.

O primeiro passo de nossa atenção será passar o caminho completo do arquivo para que não percamos a referência de sua localização e guardaremos seu valor em uma variável chamada de **caminho_arquivo**, para isso digite o seguinte comando:

```
for arquivo in *
do
    local caminho_arquivo=$(find ~/Downloads/imagens-novos-livi
done
```

COPIAR CÓDIGO

Conseguimos assim, obter o caminho completo dos arquivos, agora falta verificar se tal arquivo é um diretório ou uma imagem, logo abaixo da linha onde a variável **caminho_arquivo** está inserida, coloque:

```
if [ -d $caminho_arquivo ]
then
    varrer_diretorio $caminho_arquivo
else
    #Conversao jpg para png
fi
```

[COPIAR CÓDIGO](#)

Por fim, quando a variável **caminho_arquivo** for um diretório, nós estaremos passando esse parâmetro para a função **varrer_diretorio**. Dessa forma, nós precisamos entrar nesse parâmetro para fazer a varredura. Para pegarmos o conteúdo do parâmetro passado utilizamos o **\$1** e para mudar o diretório usamos **cd**. Coloque em cima da linha do laço for o comando

```
cd $1
```

Agora, envolva todo esse código dentro de uma função chamada de **varrer_diretorio**, na sequência iremos montar a função para converter as imagens encontradas do formato **jpg** para **png**



Opinião do instrutor

Veja que na função **varrer_diretorio**, estamos fazendo a varredura dos arquivos, caso o arquivo encontrado seja um diretório, nós chamamos internamente a função **varrer_diretorio** e passamos o caminho do diretório como parâmetro para que possamos assim entrar nesse diretório e repetir todo o processo de varredura. No final dessa etapa, nosso script deverá estar parecido com o código abaixo

```
varrer_diretorio(){
cd $1
for arquivo in *
do
    local caminho_arquivo=$(find ~/Downloads/imagens-novos-livros -name $arquivo)
    if [ -d $caminho_arquivo ]
    then
        varrer_diretorio $caminho_arquivo
    else
        #Conversao jpg para png
    fi
done
}
```

Mãos à obra: Montando a função `converte_imagem`

Uma vez que criamos a função para varrer os diretórios, precisamos montar uma função que seja capaz de fazer as conversões das imagens. Abra o script que estamos trabalhando e logo acima da função `varrer_diretorio` crie uma função chamada de `converte_imagem`.

Essa função deverá como o próprio nome diz, converter as imagens passadas pela função `varrer_diretorio`, dessa forma, teremos que pegar esse parâmetro passado pela função `varrer_diretorio`. Para não ficarmos com vários `$1` espalhados no nosso código, vamos guardá-lo dentro de uma variável local chamada de `caminho_imagem`. Dentro da função `converte_imagem` coloque o seguinte trecho

```
local caminho_imagem=$1
```

[COPIAR CÓDIGO](#)

Feito isso, deveremos remover a extensão `.jpg` desse arquivo de imagem para podermos fazer a conversão, vamos pedir ajuda do `awk` como fizemos nas etapas anteriores. Logo abaixo do trecho da atribuição da variável `caminho_imagem`, coloque o seguinte trecho na função `converte_imagem`

```
local imagem_sem_extensao=$(ls $caminho_imagem | awk -F. '{ pr:  
convert $imagem_sem_extensao.jpg $imagem_sem_extensao.png
```

[COPIAR CÓDIGO](#)

Agora que montamos a função `converte_imagem`, devemos passar o parâmetro. Volte para a função `varrer_diretorio` e altere o comentário que

havíamos feito "*Conversao jpg para png*" para o seguinte comando:

```
converte_imagem $caminho_arquivo
```

[COPIAR CÓDIGO](#)

Com isso estamos passando o caminho completo do arquivo, que sabemos que é uma imagem porque está no bloco do **else**, para a função **converte_imagem**. Agora só falta nós chamarmos a função **varrer_diretorio** para iniciar o processo de varredura, faremos isso no próximo exercício.

Opinião do instrutor



Nós criamos agora a função **converte_imagem** que como o próprio nome diz, tem como finalidade converter as imagens que serão passadas como parâmetro pela função **varrer_diretorio**

No fim, nosso código deverá estar parecido com a imagem abaixo:

```
#!/bin/bash

converte_imagem(){
local caminho_imagem=$1
local imagem_sem_extensao=$(ls $caminho_imagem | awk -F. '{ print $1 }')
convert $imagem_sem_extensao.jpg $imagem_sem_extensao.png
}

varrer_diretorio(){
cd $1
for arquivo in *
do
    local caminho_arquivo=$(find ~/Downloads/imagens-novos-livros -name $arquivo)
    if [ -d $caminho_arquivo ]
    then
        varrer_diretorio $caminho_arquivo
    else
        converte_imagem $caminho_arquivo
    fi
done
}
```


Mãos à obra: Finalizando nosso script

Agora que já criamos as funções `converte_imagem` e `varrer_diretorio`, falta nós chamarmos a função `varrer_diretorio` para que nós iniciemos o processo de varredura. Depois da função `varrer_diretorio` faça a chamada para essa função, passando como parâmetro o diretório `imagens-novos-livros`

```
varrer_diretorio ~/Downloads/imagens-novos-livros
```

[COPIAR CÓDIGO](#)

Vamos colocar uma mensagem no terminal caso o processo tenha sido realizado com sucesso. Logo abaixo dessa linha, faça a verificação:

```
if [ $? -eq 0 ]
then
    echo "Conversão realizada com sucesso"
else
    echo "Houve uma falha no processo de conversão"
fi
```

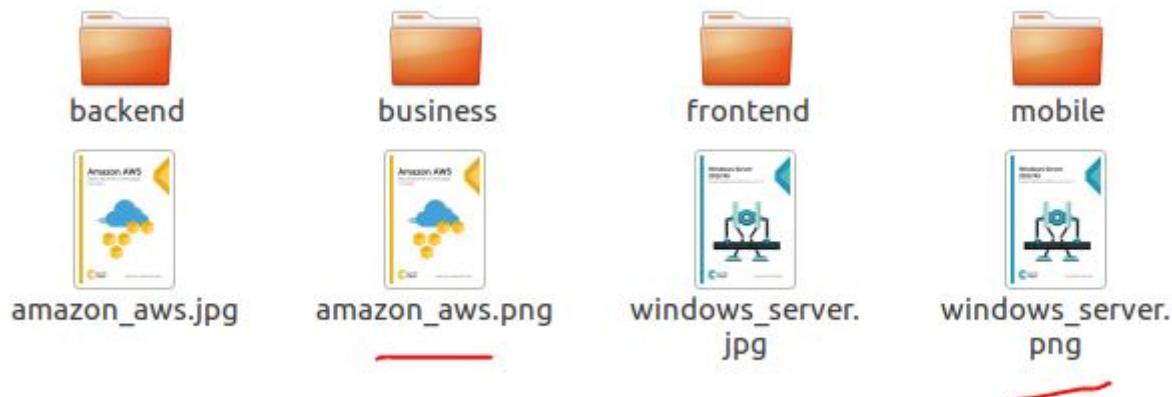
[COPIAR CÓDIGO](#)

Feito isso, execute o script. Qual é o resultado?

Opinião do instrutor



Ao executarmos nosso script, devemos ter a mensagem de que a conversão foi realizada com sucesso. Se formos no diretório **imagens-novos-livros**, devemos ver de fato todos os arquivos convertidos.



Nosso script no final deverá estar dessa forma:

```
#!/bin/bash

converte_imagem(){
local caminho_imagem=$1
local imagem_sem_extensoao=$(ls $caminho_imagem | awk -F. '{ print $1 }')
convert ${imagem_sem_extensoao}.jpg ${imagem_sem_extensoao}.png
}

varrer_diretorio(){
cd $1
for arquivo in *
do
    local caminho_arquivo=$(find ~/Downloads/imagens-novos-livros -name $arquivo)
    if [ -d $caminho_arquivo ]
    then
        varrer_diretorio $caminho_arquivo
    else
        converte_imagem $caminho_arquivo
    fi
done
}
varrer_diretorio ~/Downloads/imagens-novos-livros
if [ $? -eq 0 ]
then
    echo "Conversão realizada com sucesso"
else
    echo "Houve uma falha no processo de conversão"
fi
```


Listando os processos do sistema

Transcrição

Em um certo dia de reunião, os diretores da *Multillidae* reclamaram sobre a existência de alguns processos com grande quantidade de memória alocada e que estariam prejudicando os serviços que a empresa oferece.

Nossa missão é localizar os **dez** processos que estão com uma quantidade maior de memória alocada. Devemos salvar cada um dos processos em um arquivo com o respectivo **nome do processo** e extensão **.log**.

Por exemplo, vamos considerar o navegador **Mozilla Firefox** sendo um destes processos. Encontramos um arquivo chamado **Firefox.log** e dentro dele devem constar as seguintes informações:

- Data
- Horário
- Alocação em Megabytes

Estas informações precisam de uma formatação específica. A **data** deve conter primeiro o **ano**, logo após o **mês** e em seguida o **dia**. Separado por **vírgula**, temos o **horário**, que deve conter primeiro a **hora**, em seguida o **minuto** e por fim o **segundo**. Por último, vem a configuração em MB da alocação de memória para esse respectivo processo.

2017-07-21, 15:09:30, 150 MB

[COPIAR CÓDIGO](#)

Vamos localizar esses dez processos, salvá-los com seu nome e sua extensão .log e cada um desses arquivos deve conter essas informações.

Testaremos essas saídas no terminal antes de montar o script.

O primeiro passo da estratégia é **buscar os dez processos**. Podemos listar todos os que existem no sistema com o comando `ps -e`. Obtemos um resultado parecido com esse:

PID	TTY	TIME	CMD
1	?	00:00:04	systemd
2	?	00:00:00	kthreadd
3	?	00:00:03	ksoftirqd/0
5	?	00:00:00	kworker/0:0H
7	?	00:00:04	rcu_sched
8	?	00:00:00	rcu_bh
9	?	00:00:00	migration/0
10	?	00:00:00	lru-add-drain
11	?	00:00:00	watchdog/0
12	?	00:00:00	cpuhp/0
13	?	00:00:00	kdevtmpfs
14	?	00:00:00	netns
15	?	00:00:00	khungtaskd
...			

[COPIAR CÓDIGO](#)

Na primeira coluna consta o número de **identificação do processo**. Com ele, seremos capazes de filtrar todas as informações que são necessárias para, como o *nome do processo* e o *tamanho de memória alocada*.

Quando executamos o comando `ps -e`, podemos especificar dizendo quais informações de **saída** são respectivas ao número do processo.

```
$ ps -e -o pid
```

[COPIAR CÓDIGO](#)

Com esse comando listamos somente o `PID` no *output*, que são os números de identificação do processo. Não necessariamente, os processos que estão listados em ordem são de fato os processos que tem uma quantidade maior de memória alocada. É preciso rearranjar os processos para que sejam mostrados primeiro os de maior quantidade de memória utilizada.

Pelo fato de precisarmos fazer uma nova **ordenação**, precisamos pedir para o comando `ps`.

```
$ ps -e -o pid --sort -size
```

[COPIAR CÓDIGO](#)

Quando tecemos o "Enter", serão listados os processos com maior quantidade de memória alocada. Fizemos uma ordenação dos processos, baseando-se no parâmetro `-size`, que é justamente o tamanho de memória alocada para cada processo.

Observe que não precisamos de todas as informações, e sim, somente dos **dez** primeiros processos dessa lista. Vamos redirecionar a saída do comando que testamos para o **head** que foi visto no curso de *Linux*.

O **head** filtrará automaticamente os dez primeiros processos. Mas para isso, é preciso que o **head** traga **onze** linhas, pois a primeira será para o cabeçalho e as outras dez serão para os processos.

Vamos redirecionar novamente:

```
$ ps -e -o pid --sort -size | head -n 11
```

[COPIAR CÓDIGO](#)

Esse foi o resultado que tivemos.

```
PID  
14009  
27362  
1445  
1584  
13107  
1594  
1633  
1501  
826  
1376
```

[COPIAR CÓDIGO](#)

Para melhorar o resultado, seria interessante eliminar a linha que contém o PID , assim teremos somente as linhas que serão utilizadas. Faremos isso redirecionando toda a saída para o comando grep assim ele pegará somente as linhas que contém números. Como podemos fazer isso?

```
$ ps -e -o pid --sort -size | head -n 11 | grep [0-9]
```

[COPIAR CÓDIGO](#)

Foram filtradas somente as linhas que contém números! Então, com o número de identificação do processo, conseguimos ter várias informações, inclusive com relação ao nome do processo, ao tamanho de memória alocada do processo, etc.

Primeiro, veremos se conseguimos ter o nome dos processos, pois precisamos salvar os arquivos com o nome + .log . Faremos esse teste com o processo 14009 para obter seu nome.

```
$ ps -p 14009 -o comm=
```

[COPIAR CÓDIGO](#)

E temos como resultado o nome do processo: firefox !

Vamos para o diretório /Scripts , para montar o script que atenderá a requisição dos diretores da *Multillidae*.

```
$ cd Scripts/  
$ nano processos-memoria.sh
```

[COPIAR CÓDIGO](#)

Como já sabemos, a primeira linha deve conter o interpretador e depois vamos trazer os dois comandos que testamos no terminal:

Se são comandos temos que envolvê-los entre \$() e armazenar o resultado do comando em uma variável.

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

```
processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9])
```

```
$(ps -p 14009 -o comm=)
```

[COPIAR CÓDIGO](#)

Certo, assim, teremos os números dos dez processos, mas para não ficar repetindo o código várias vezes utilizaremos o laço de repetição `for` para pegar o número de identificação PID para que tenhamos as informações de cada processo.

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

```
processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9])  
for pid in $processos
```

```
do
```

```
$(ps -p $pid -o comm=)
```

[COPIAR CÓDIGO](#)

Vamos pegar o conteúdo da variável `pid` e imprimimos o valor.

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

```
processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9]
for pid in $processos
do
    echo $(ps -p $pid -o comm=)
done
```

[COPIAR CÓDIGO](#)

Para salvar, usamos "Ctrl + X" e "Y".

Vamos verificar se, de fato, ele é capaz de imprimir os nomes desses dez processos. Vamos rodar o script com `bash processos-memoria.sh`.

O resultado será este:

```
firefox
mysqld
compiz
gnome-software
soffice.bin
nautilus
fwupd
indicator-datetime
Xorg
hud-service
```

[COPIAR CÓDIGO](#)

Temos a impressão dos dez processos que estão com maior quantidade de memória alocada! A primeira parte da nossa missão foi concluída. Agora, precisamos nos preocupar com o conteúdo interno dos arquivos: a data, a hora, e a alocação em megabytes.

Mãos à obra: Listando os nomes dos processos

Vamos começar nossa tarefa passada pelos diretores da *Mutillidae* para salvar os 10 processos com maior quantidade de memória alocada em arquivos separados com o respectivo nome do processo e a extensão `.log`.

Para isso, vá até o diretório de *Scripts* e crie um novo script chamado `processos-memoria-alocada.sh`. Como fizemos nas etapas anteriores, a primeira linha do nosso script irá conter o interpretador, então digitamos

```
#!/bin/bash .
```

Na sequência, devemos pegar os números de identificação desses 10 processos e vamos armazenar em uma variável chamada `processos`.

```
processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9]
```

[COPIAR CÓDIGO](#)

Como temos 10 processos, não vamos ficar repetindo código, não é mesmo? Iremos utilizar a estrutura de repetição `for`, como fizemos anteriormente e vamos pedir para que seja impresso o nome de cada número de processo listado

```
for pid in $processos
do
    echo $(ps -p $pid -o comm=)
done`
```

[COPIAR CÓDIGO](#)

Feito isso, execute o script. Qual é o resultado?



Opinião do instrutor

Veja que com o comando `ps -e -o pid --sort -size`, estamos filtrando todos os números dos processos com ordenação de alocação de memória decrescente, ou seja, os processos com maior alocação de memória são listados antes.

O que nós precisamos são os 10 primeiros processos com maior quantidade de memória alocada, isso faz com que redirecionemos a saída para o `head`, porém o `head` traz por padrão as 10 primeiras linhas e a primeira linha é o cabeçalho `PID`, o que nós precisamos então é trazer 11 linhas, totalizando assim a primeira linha com o nome do cabeçalho `PID` e as outras 10 linhas com os números dos processos.

Para finalizar, devemos ter somente os números dos processos, então redirecionamos essa saída para o `grep` para que tenhamos somente os números sem o cabeçalho `PID`.

Ao executar o Script, nós teremos os nomes dos processos listados

```
rafael@rafael-VirtualBox:~/Scripts$ bash processos-memoria-alocada.sh
firefox
compiz
gnome-software
evolution-calen
evolution-calen
evolution-calen
fwupd
indicator-datetime
evolution-addressee
unity-files-dae
```

Nosso script deverá estar parecido com o abaixo:

```
#!/bin/bash

processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9])
for pid in $processos
do
    echo $(ps -p $pid -o comm=)
done
```

Criando arquivos e salvando data e hora

Transcrição

Até agora o script criado é capaz de descobrir os nomes dos 10 processos com maior quantidade de alocação de memória. Entretanto, daremos foco a parte interna de cada um dos arquivos.

Descobrimos um comando que nos ajudará nessa tarefa: `date`.

O comando `date` nos mostra o dia e o horário de execução. Mas, se lembarmos da requisição dos diretores da *Multillidae*, eles pediram que a data estivesse formatada da seguinte maneira:

ano-mês-dia, horas:minutos:segundos

[COPIAR CÓDIGO](#)

Precisamos formatar a saída para atender ao formato solicitado pelos diretores. Focaremos a nossa atenção na primeira parte da formatação, que se refere à **data**.

Vamos utilizar o comando `$F` que significa **data completa** em inglês.

`$ date +%F`

[COPIAR CÓDIGO](#)

O resultado é o seguinte:

2027-07-21

[COPIAR CÓDIGO](#)

Podemos ir mais além desse resultado, concatenando-o com as horas, minutos e segundos.

```
$ date +%F,%H:%M:%S
```

[COPIAR CÓDIGO](#)

E temos o seguinte:

2027-07-21,15:31:58

[COPIAR CÓDIGO](#)

Então, com esse comando, conseguimos resolver esse problema. Vamos copiá-lo para colar no script `processos-memoria.sh`.

```
do
    echo $(ps -p $pid -o comm=)
    date +%F,%H:%M:%S
done
```

[COPIAR CÓDIGO](#)

Para descobrir o nome do processo, vamos utilizar o comando anterior e salvamos o resultado do comando em uma variável.

```
do
    nome_processo=$(ps -p $pid -o comm=)
    echo $(date +%F,%H:%M:%S)
done
```

[COPIAR CÓDIGO](#)

Para imprimir *utilizamos o echo* e para redirecionar a saída para um arquivo acrescentamos o `>` e o `nome_do_arquivo.log`

```
echo $(date +%F,%H:%M:%S) > $nome_processo.log
```

[COPIAR CÓDIGO](#)

Só que sempre que o script for executado, a informação que estiver presente no arquivo `nome_do_processo.log`, vai ser sobreescrita. E não queremos isso! Nosso desejo, na verdade, é ter um histórico das informações.

É importante juntar a nova informação que estamos executando para o arquivo do `nome_do_processo.log`. Para resolver esse problema, basta acrescentar mais um `>>`, assim ele não irá sobre escrever.

Já conseguimos ter a data e o horário, mas, acontece que ainda temos que colocar a memória em megabytes.

Se deixarmos o comando desta forma, o `echo` vai imprimir na mesma linha a data e a hora, só que por padrão, quando ele termina de ser executado, ele vai para a próxima linha. Não queremos isso, pois ainda precisamos colocar a locação em MB após o campo do horário. Para isso, precisamos falar para o `echo` **não** fazer uma quebra de linha.

```
echo -n $(date +%F,%H:%M:%S) >> $nome_processo.log
```

[COPIAR CÓDIGO](#)

Vamos aproveitar para alterar a formatação para *vírgulas*, para depois poder acomodar os megabytes da memória alocada.

```
echo -n $(date +%F,%H:%M:%S,) >> $nome_processo.log
```

[COPIAR CÓDIGO](#)

Vamos salvar as alterações com "Ctrl + X" e "Y".

Assim como descobrimos o nome do processo pelo comando, também podemos descobrir o tamanho da memória alocada por ele. É uma forma bem parecida ao que já fizemos. Queremos listar o processo que tenha o número de identificação do PID, imprimindo o tamanho no *output*.

```
$ ps -p 14009 -o size
```

[COPIAR CÓDIGO](#)

O que foi retornado?

```
SIZE
```

```
947732
```

[COPIAR CÓDIGO](#)

É mostrado o tamanho da memória alocada para o processo "14009".

Mas, repare que interessante! O número "947732" está em **kilobytes**, e a requisição dos nossos diretores é de que a informação esteja em **megabytes**. Vamos ver como podemos solucionar esse problema.

Mãos à obra: Passando data e hora para o arquivo

Uma vez que conseguimos obter os nomes dos processos, nós precisamos nos atentar ao conteúdo desses arquivos. Vamos portanto abrir o script que trabalhamos na aula anterior e fazer esses ajustes.

Dentro do laço for, guarde o resultado dos nomes dos processos em uma variável chamada de `nome_processo`

```
nome_processo=$(ps -p $pid -o comm=)
```

[COPIAR CÓDIGO](#)

Logo em seguida, vamos salvar a data e hora no arquivo que terá o nome do processo.

```
echo $(date +%F,%H:%M:%S,) >> $nome_processo.log
```

[COPIAR CÓDIGO](#)

Salve as alterações, iremos finalizar nosso script na sequência.

Opinião do instrutor



Uma vez que temos os nomes dos processos, nós devemos salvá-los em arquivos separados com a extensão `.log`. Esses arquivos devem conter a data, hora e a alocação em MB, nesse momento, estamos salvando a data e a hora, na sequência iremos verificar o tamanho da memória alocada em MB.

Até o momento, nosso script deve estar parecido com o abaixo:

```
#!/bin/bash

processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9])
for pid in $processos
do
    nome_processo=$(ps -p $pid -o comm=)
    echo $(date +%F,%H:%M:%S) >> ${nome_processo}.log
done
```

Criando arquivos de log completo

Transcrição

Com o comando `ps -p 14009 -o size` é impresso o tamanho da memória alocada do processo informado. Entretanto, o resultado que temos é um tamanho em **kilobytes** mostrado a seguir:

SIZE

931348

[COPIAR CÓDIGO](#)

Ao dividir esse valor em kilobytes por **1024**, conseguimos ter o valor em **megabytes**!

O primeiro passo é eliminar a linha de cabeçalho `SIZE`, pois queremos somente o número. Como podemos solucionar esse problema?

Como já visto, utilizamos o `grep` para redirecionar a saída desse resultado, mostrando somente o valor numérico.

\$ `ps -p 14009 -o size | grep [0-9]`

[COPIAR CÓDIGO](#)

E temos o seguinte resultado:

931348

[COPIAR CÓDIGO](#)

Agora que temos o resultado com números, podemos pensar em fazer a divisão com o comando `echo 931348/1024`. Porém, quando fazemos isso, o echo imprime literalmente o valor `931348/1024`, mas esperamos que uma operação de divisão seja feita.

Para que possamos ter essa operação de divisão, utilizaremos a ferramenta `bc` que vai realizar a divisão entre os valores e passaremos os parâmetros da divisão, através de `>>>`:

```
$ bc <<< 931348/1024
```

[COPIAR CÓDIGO](#)

E temos o resultado da divisão:

```
909
```

[COPIAR CÓDIGO](#)

Se calcularmos essa operação na calculadora, veremos que o valor será de `909,51953125`. Podemos recalcular de uma forma que tenha uma precisão maior. Suponhamos que valor tenha duas casas após a vírgula.

Para isso, trabalharemos com **escala de 2**, e envolveremos a operação entre aspas.

```
$ bc <<< "scale=2;931348/1024"
```

[COPIAR CÓDIGO](#)

E como resultado, obtemos esse valor `909.51`, com duas casas após a vírgula. Vamos copiar os dois comandos criados, para que nós coloquemos no script `processos-memoria.sh`.

```
for pid in $processos
do
    nome_processo=$(ps -p $pid -o comm=)
    echo -n $(date +%F,%H:%M:%S,) >> $nome_processo.log
    tamanho_processo=$(ps -p $pid -o size | grep [0-9])
done
```

[COPIAR CÓDIGO](#)

Então, colamos o comando testado no terminal, o envolvemos no `$()`, apagamos o valor do processo Firefox 14009 para que seja possível pegar o número do processo respectivo da variável `$pid` e ,assim, armazenamos o resultado na variável `tamanho_processo` .

O próximo passo é colar o segundo comando no Script:

```
for pid in $processos
do
    nome_processo=$(ps -p $pid -o comm=)
    echo -n $(date +%F,%H:%M:%S,) >> $nome_processo.log
    tamanho_processo=$(ps -p $pid -o size | grep [0-9])
    echo $(bc <<< "scale=2;$tamanho_processo/1024") >> $nome_proc
done
```

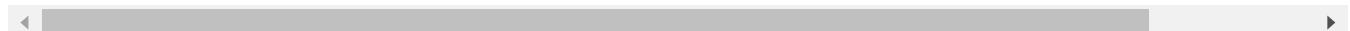
[COPIAR CÓDIGO](#)

Redirecionamos a impressão para o arquivo `.log` . Como toda a linha é um comando, o envolvemos em `$()` . O valor numérico **931348** é o valor estático do processo do Firefox. Por essa razão, pegamos o conteúdo do `$tamanho_processo` que será dividido por 1024.

Para melhorar o script, sabemos que o resultado será em megabytes, por isso, colocamos o comando entre aspas e dizemos que seu resultado é um valor em megabytes:

```
echo "$(bc <<< "scale=2;$tamanho_processo/1024") MB">>> $nome_proc
```

[COPIAR CÓDIGO](#)



Ao chegar até aqui, fizemos um grande avanço. Mas, vamos fazer uma melhoria. Para que os arquivos não fiquem espalhados por todos os lugares, vamos colocá-los dentro de um diretório. Aqui mesmo dentro do diretório `/Scripts` e verificaremos se existe um diretório chamado de `/log`, onde salvaremos todos esses arquivos. Caso ele não exista, vamos criá-lo.

Logo no início do script, faremos essa verificação.

```
#!/bin/bash

if [ ! -d log ]
then
    mkdir log
fi
```

[COPIAR CÓDIGO](#)

Agora com a certeza de que o diretório `/log` vai existir, podemos salvar os arquivos dentro do `/log` que foi criado.

```
#!/bin/bash

if [ ! -d log ]
then
    mkdir log
fi
```

```
processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9])
for pid in $processos
do
```

```
nome_processo=$(ps -p $pid -o comm=)
echo -n $(date +%F,%H:%M:%S,) >> log/$nome_processo.log
tamanho_processo=$(ps -p $pid -o size | grep [0-9])
echo $(bc <<< "scale=2;$tamanho_processo/1024") >> log/$nome_
done
```

[COPIAR CÓDIGO](#)

É importante saber se os arquivos foram salvos com sucesso ou se houve algum problema. Então, podemos fazer essa verificação por meio do **status de saída**. Caso ele seja `0` é porque está tudo "ok". Se for diferente de `0`, é porque um problema ocorreu.

Como já vimos, podemos envolver todo o código em uma função e, assim, verificar qual será o status de saída dessa função.

Chamaremos essa função de `processos_memoria()`:

```
processos_memoria(){
processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9])
for pid in $processos
do
    nome_processo=$(ps -p $pid -o comm=)
    echo -n $(date +%F,%H:%M:%S,) >> log/$nome_processo.log
    tamanho_processo=$(ps -p $pid -o size | grep [0-9])
    echo $(bc <<< "scale=2;$tamanho_processo/1024") >> log/$nome_
done
}
```

[COPIAR CÓDIGO](#)

Após criar a função vamos invocá-la no final do código.

```
processos_memoria(){
processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9] ..
```

```
for pid in $processos
do
    nome_processo=$(ps -p $pid -o comm=)
    echo -n $(date +%F,%H:%M:%S,) >> log/$nome_processo.log
    tamanho_processo=$(ps -p $pid -o size | grep [0-9])
    echo $(bc <<< "scale=2;$tamanho_processo/1024") >> log/$nome_
done
}
```

processos_memoria

[COPIAR CÓDIGO](#)



Uma vez chamada a função, vamos verificar seu status de saída:

```
processos_memoria
if [ $? -eq 0 ]
then
    echo "Os arquivos foram salvos com sucesso"
else
    echo "Houve um problema na hora de salvar os arquivos"
fi
```

[COPIAR CÓDIGO](#)

Hora de testar o script!

Não esqueça de sair e salvar: "Ctrl + X" "Y"

Ao executar o script `bash processos-memoria.sh`, temos o status de saída: os arquivos foram salvos sem nenhum problema. Ao acessar o diretório `/log`, encontramos arquivos com os nomes dos processos junto de suas respectivas extensões `.log`.



Clicando em um desses arquivos, por exemplo, vamos encontrar informações parecidas com essa:

2017-07-21,16:57:47,909.51 MB

[COPIAR CÓDIGO](#)

Temos a **data**, o **horário** e a **locação em MB**. Comentamos em um momento anterior sobre ter colocado o símbolo `>>` antes de salvar o processo na pasta log. Isso quer dizer que agora eles têm o conteúdo que acabamos de executar. Se executarmos novamente, as informações serão mantidas e, ainda, serão acrescentadas as novas informações. Vamos ver se realmente ocorre isso! Vamos rodar novamente o script.

Clicando em algum dos arquivos, vamos ter uma linha a mais de informações e vemos que a informação anterior se manteve, assim como havíamos planejado.

2017-07-21,16:57:47,909.51 MB

2017-07-21,16:59:05,909.51 MB

[COPIAR CÓDIGO](#)

Tarefa concluída com sucesso!

Mãos à obra: Salvando os arquivos com extensão .log

No nosso script, nós fomos capazes de obter a data e horário e salvamos dentro do arquivo com o nome do processo **.log**. Agora iremos realizar a última etapa que seria de salvar o tamanho da memória alocada para cada processo dentro desse arquivo.

Abra novamente o script que estamos trabalhando com o editor de texto de sua preferência. Feito isso, iremos para cada número do processo obter o tamanho da memória alocada e iremos guardar o resultado em uma variável chamada de **tamanho_processo**

```
tamanho_processo=$(ps -p $pid -o size | grep [0-9])
```

[COPIAR CÓDIGO](#)

Esse valor apresentado é em kB, vamos dividir esse valor por 1.024 para que possamos obter assim o valor em MB e salvar o resultado no arquivo com a extensão **.log**

```
echo "$(bc <<< "scale=2;$tamanho_processo/1024") MB" >> $nome_
```

[COPIAR CÓDIGO](#)

Para finalizar, vamos guardar todos os arquivos dentro de um diretório chamado **log** que estará dentro do diretório **Scripts**. Será que esse diretório existe? Não sabemos, vamos pedir para que nosso script faça essa verificação. Logo abaixo da linha com o interpretador coloque:

```
if [ ! -d log ]
then
    mkdir log
fi
```

[COPIAR CÓDIGO](#)

Feito isso, vá até as linhas com os nomes dos arquivos com a extensão .log e coloque que esses arquivos serão salvos nesse diretório log, ficando:

```
echo -n $(date +%F,%H:%M:%S,) >> log/$nome_processo.log
...
echo "$(bc <<< "scale=2;$tamanho_processo/1024") MB" >> log/$n
```

[COPIAR CÓDIGO](#)

Por fim, envolva todo o código em uma função chamada de **processos_memoria** e faça a chamada dessa função. Ao chamar a função, verifique o status de saída e imprima uma mensagem no terminal.

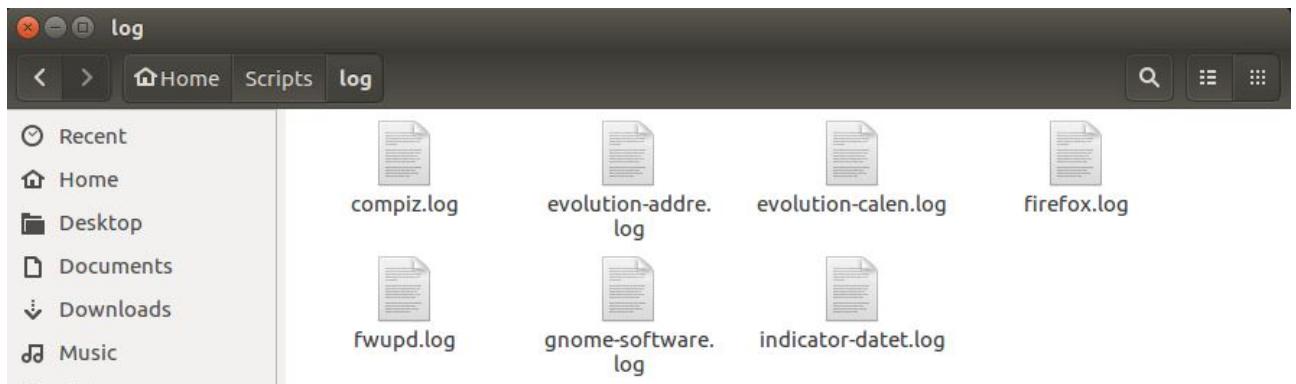
```
processos_memoria
if [ $? -eq 0 ]
then
    echo "Os arquivos foram salvos com sucesso"
else
    echo "Houve um problema na hora de salvar os arquivos"
fi
```

[COPIAR CÓDIGO](#)

Agora execute o seu script, qual o resultado?

Opinião do instrutor

Ao executar nosso script, nós devemos visualizar que o diretório **log** foi criado e dentro teremos os arquivos com os nomes dos processo com a data, horário e alocação da memória em MB



Nosso script deverá estar parecido com o listado abaixo:

```
#!/bin/bash

if [ ! -d log ]
then
    mkdir log
fi

processos_memoria(){
processos=$(ps -e -o pid --sort -size | head -n 11 | grep [0-9])
for pid in $processos
do
    nome_processo=$(ps -p $pid -o comm=)
    echo -n $(date +%F,%H:%M:%S,) >> log/$nome_processo.log
    tamanho_processo=$(ps -p $pid -o size | grep [0-9])
    echo "$(bc <<< "scale=2;$tamanho_processo/1024") MB" >> log/$nome_processo.log
done
}

processos_memoria
if [ $? -eq 0 ]
then
    echo "Os arquivos foram salvos com sucesso"
else
    echo "Houve um problema na hora de salvar os arquivos"
fi
```

Obs: Os arquivos salvos podem estar um pouco diferentes do mostrado no vídeo por depender dos processos que estão na sua máquina

