Junho de 2021

Gabriel de Melo Cruz
gabrieldemelocruz@gmail.com | github.com/gmelodie | gmcruz.me

Ninjaterm: vim, bash e mais umas coisinhas
Notas de Aula

Sobre este documento

Os comandos `em verde e nesta fonte` devem ser digitados no seu terminal (independente de estar no Ubuntu nativo ou no WSL). Exemplo (deve ser digitado **incluindo as aspas**):

```
echo "isto eh um comando para ser digitado no terminal"
```

Quando letras maiúsculas aparecerem em comandos, copie exatamente como está. Isso:

```
echo $SHELL
```

É diferente disso:

```
echo $shell
```

Obs: respostas dos exercícios no final do documento

Olá, mundo!

Para "esquentarmos" um pouquinho vamos usar o exemplo clássico do `hello world`. Baixe o arquivo `hello.txt`:

```
wget gmcruz.me/courses/ninjaterm/hello.txt
```

Obs: o comando acima vai salvar o arquivo com o mesmo nome que tem no servidor, isto é, `hello.txt`. Para salvá-lo com outro nome use a opção (ou *flag*) `-O`:

```
wget -O outro-nome.txt gmcruz.me/courses/ninjaterm/hello.txt
```

Agora leia o arquivo com `cat`:

```
cat hello.txt
```

O comando `cat` significa *concatenate* (“concatenar” em inglês), pega o conteúdo dos arquivos que foram passados para ele, concatena e imprime na tela. Se passarmos apenas um arquivo, como fizemos acima, isso equivale a ler os conteúdos de apenas aquele arquivo.

Exercício 1: baixe o arquivo `there.txt` e use `cat` para concatenar os conteúdos de `hello.txt` e `there.txt`

Arquivos e diretórios

Um diretório, ou uma pasta, é onde os arquivos (ou até outras pastas) são armazenadas. Para vermos o diretório atual podemos usar o comando `pwd` (“present working directory” em inglês). Para listarmos os conteúdos de um diretório no terminal usamos `ls`, e para mudar de diretório usamos `cd`. Vamos explorar um pouco mais esses comandos.

CD, LS, PWD

Listar os conteúdos do diretório atual

```
ls
```

Obs: no Linux o ponto (`.`) significa diretório atual, então `ls .` tem o mesmo efeito que `ls`

Ir para o diretório Documents

```
cd Documents
```

Atenção: o diretório teste deve existir e estar dentro do mesmo diretório que estamos

Ir um diretório acima (assim como o ponto, dois pontos `..` significa diretório acima)

```
cd ..
```

Imprimir o caminho do diretório atual

```
pwd
```

Exercício 2:

a. Use o comando `man touch` para descobrir o que faz o comando `touch`
b. Crie um arquivo vazio usando `touch`

ECHO

O comando `echo` ecoa uma *string* (um pedaço de texto). Ele é muito útil para compor outros comandos mais complexos.

```
echo ecoe isto

echo "aspas tambem funcionam"
```

Podemos "escapar" qualquer caracter especial, como as aspas, com uma barra invertida (\) para que elas sejam ecoadas também

```
echo \"imprima as aspas por favor\"
```

Até agora só criamos arquivos vazios com `touch`. Vamos usar o `echo` para criar o arquivo `ola.txt` com o texto `ola mundo!`:

```
echo ola mundo! > ola.txt
```

O operador `>` é muito útil para redirecionar a saída de programas para arquivos. Nesse caso, a saída, `ola mundo!`, é redirecionada para o arquivo `ola.txt`.

Exercício 3:

a. Use echo para imprimir exatamente `"hello world!";` (com aspas e o ponto e vírgula).
b. Use echo para criar um arquivo `tchau.txt` com o conteúdo `até mais`.
c. Use o operador `>` para salvar a saída do comando `ls` em um arquivo chamado `ls.txt`.

MKDIR, MV, RM

Até agora nossos arquivos estão todos jogados! O comando `mkdir` vai nos ajudar a arrumar a casa. É com ele que criamos novos diretórios. Depois, vamos mover arquivos usando `mv` e finalmente apagar toda essa tranqueira com `rm`.

Vamos criar um novo diretório para os arquivos do nosso workshop

```
mkdir ninjaterm
```

Obs: a figura mostra que estávamos no diretório `/tmp`, criamos `ninjaterm` dentro de `/tmp`, entramos em `ninjaterm` e agora estamos em `/tmp/ninjaterm`

Vamos agora mover todos os arquivos .txt que criamos.

```
mv ../*.txt .
```

Aqui usamos o operador `*`, que significa "qualquer coisa". Então o comando move todos os arquivos do diretório acima (que é o `/tmp`, onde estávamos anteriormente) que atendem à regra "qualquer coisa"`.txt`, ou seja, todos os arquivos que terminam em `.txt` são movidos para o diretório atual (`.`), que é o `ninjaterm`.

GREP

Grep significa **Global Regular Expression Print**, ele usa expressões regulares para encontrar e imprimir partes de um certo texto. Vamos usar ele com o operador pipe (`|`), muito semelhante ao `<`, para filtrar quase tudo que usarmos daqui em diante. Mas vamos com calma, primeiro vamos ver como se usa o grep.

```
wget gmcruz.me/courses/ninjaterm/gabriel.txt

grep gabriel.txt gabriel
```

Esse último comando procura por todas as linhas que contém "gabriel" no arquivo `gabriel.txt`. Mas se olharmos o arquivo `gabriel.txt` com o `cat`, vamos ver que existem muitas outras ocorrências de "gabriel"

A diferença é que no arquivo temos ocorrências em maiúsculo (“Gabriel”) e em minúsculo (“gabriel”). Vemos então que o `grep` é *case sensitive*, ou seja, faz diferença entre letras maiúsculas e minúsculas. Podemos fazer o `grep` ignorar isso usando a opção `-i` (que significa *ignore case*):

```
grep -i gabriel.txt gabriel
```

Boa! Agora que sabemos como usar o `grep` de maneira geral, vamos dar uma olhada no *pipe* (`|`). Pipe é essa barrinha vertical (cuidado para não confundir com a barra normal `/` ou com a barra invertida `\`) que joga a saída de um comando como entrada de outro. Podemos usar o pipe com o `grep` para filtrar a saída do comando `ls` por exemplo. Aqui ele vai mostrar apenas os arquivos listados que contém “txt” em alguma parte deles:

```
ls  | grep txt
```

Para inverter a busca, ou seja, encontrar tudo menos o que você passa para o `grep`, use a opção `-v`:

```
grep -v gabriel gabriel.txt
```

Você também pode usar opções juntas:

```
grep -vi gabriel gabriel.txt
```

Exercício 4:

a. Baixe o arquivo `gmcruz.me/courses/ninjaterm/big.txt`
b. Encontre a linha que contém “OPA” no arquivo `big.txt`
c. Encontre a linha que tem “resposta” e não tem “seja” no arquivo big.txt
d. Faça a mesma coisa em **b** e **c**, mas encontre o número das linhas do arquivo (veja o manual do grep: `man grep`)

Vim

Essa seção é mais uma colinha de comandos, visto que Vim só se aprende fazendo. Use essa colinha quando não se lembrar de algum comando, mas os comandos listados aqui estão longe de ser uma lista exaustiva das funcionalidades do Vim. Não hesite em pesquisar em outras fontes caso necessário.

Modos

- Insert: i
- Normal: ESC
- Visual: v
    - visual block: Ctrl + V
    - visual line: Shift + V

Verbos

- Copiar (yank): y
- Colar (paste): p
- Deletar: d
- Mudar (change): c
- Desfazer (undo): u
- Refazer (redo): r
- Encontrar na linha (find): f
- Repetir o último comando (dot): .

Movimentos

- Próxima palavra (word): w
- Fim (end) da próxima palavra: e
- Voltar (back) uma palavra: b
- Até (till): t
- Dentro (inner): i
- Em volta (around): a

Exemplos de comandos

- Deletar as três próximas palavras: d3w
- Mudar dentro dos parêntesis: ci) ou ci(
- Substituir “gabriel” por “jolas”: :%s/gabriel/jolas/cg
- Pesquisar “teste” no arquivo: /teste
    - Ir para a próxima ocorrência: n
    - Ir para a última ocorrência: N (Shift + n)

Exercício 5:

a. Baixe o arquivo gmcruz.me/courses/ninjaterm/bullshit.txt
b. Reordene as linhas de acordo com a numeração no início de cada linha
c. Mude a linha 7 para conter o seu próprio nome
d. Delete os números no início das linhas
e. Delete todos os os números do arquivo

Bash

O que é um(a) *shell*?

Shell em inglês significa "casca". No contexto do Linux e outros sistemas operacionais, significa um programa que serve como uma interface entre alguns programas do sistema operacional e o usuário. Assim, o shell é como uma casca para o "motor" do seu computador. Ah... não foi uma analogia tão ruim assim vai.

O shell então é um programa! Manja programa? Tipo aqueles ~~horríveis~~ que a gente faz pra ICC1. O shell é responsável por ler o que você passa pra ele, chamar outros programas (tipo o ls e o cat) e te mostrar a saída. Nesse sentido, ele é só uma interface para facilitar a nossa vida. Isso também quer dizer que podemos ter vários tipos de shell, não só um -- da mesma forma que temos vários programas para editar texto. Na verdade o Bash é um deles (talvez o mais famoso). Outros shells são, por exemplo, `zsh` e `fish`.

Para saber qual shell você está usando use o seguinte comando:

```
echo $SHELL
```

Shell vs. Terminal

Essa é outra dúvida muito comum. Terminal na verdade é algo mais recente. Antigamente, os computadores não eram pequenininhos que nem hoje. Havia poucos computadores e eram enormes -- do tamanho de um campo de futebol às vezes. Nesses computadores você tinha algumas estações (terminais) que tinham uma tela e um teclado. Assim as pessoas podiam usar o mesmo computador ao mesmo tempo. Hoje os terminais são virtuais, chamados de *terminal emulators* (emuladores de terminais em inglês). Eles são responsáveis por mostrar a saída do shell na tela, cores e essas coisas. Exemplos de terminal emulators são `gnome-terminal` e `konsole`.

Para simplificar, pense no terminal como a interface gráfica e o shell como o programa que roda nessa interface.

Variáveis de Ambiente (*Environment Variables*)

Lembra quando usamos `echo $SHELL` para descobrir qual shell estávamos usando? O que fizemos foi basicamente olhar o conteúdo da variável `SHELL`. Essa variável é uma variável de ambiente, ou seja, uma variável que vive no ambiente do seu shell e ajuda com várias coisinhas. Vamos ver outras variáveis e como elas são usadas.

Para listar todas as variáveis de ambiente usamos o comando `printevn`

`USER` contém o username do usuário logado

```
echo $USER
```

Podemos também criar nossas próprias variáveis. Para criar uma variável não usamos $, mas quando referenciamos uma variável já criada usamos. Vamos criar uma variável para conter a primeira parte da URL do curso (`gmcruz.me/courses/ninjaterm/`)

```
ninjacourse=gmcruz.me/courses/ninjaterm
```

E agora podemos usá-la sem precisar repetir toda vez a URL

```
wget $ninjacourse/hello.txt
```

Processos e Binários

Um processo é uma unidade de execução que o sistema operacional gerencia. Todo programa quando está rodando vira um processo. Isso é útil porque você pode rodar o mesmo programa várias vezes, nesse caso você teria vários processos rodando o mesmo programa. Podemos listar processos usando o comando `ps`.

Listar todos os processos que foram iniciados pelo shell atual

```
ps a
```

Listar todos os processos deste usuário

```
ps u
```

Listar todos os processos não iniciados pelo shell atual

```
ps x
```

Podemos também combinar essas opções para ter todos os processos do usuário, associados ou não ao shell atual

```
ps aux
```

Process ID (PID)

PID é um número que representa o processo. Faz muito sentido ter esse número visto que podemos ter vários processos rodando o mesmo programa (como quando você abre várias janelas do Browser). O PID serve para identificar mais facilmente os processos do ponto de vista do sistema operacional. O PID é listado na segunda coluna quando rodamos o comando `ps`.

Exercício 6:

a. Use `ps` e `grep` para encontrar o PID do seu shell atual
b. Use o `man` para descobrir como usar e use `pgrep` para fazer a mesma coisa do item anterior
c. Descubra o que é a variável de ambiente cujo nome é `$`

Apêndice: Outros materiais

Cut

Cut faz isso mesmo: corta. Ele separa uma *string*, uma linha de texto, de acordo com um certo delimitador (o espaço, por exemplo) e pega um dos *fields* (campos) da separação. Por exemplo, vamos separar a string "olá para o mundo" (no arquivo `ola-cut.txt`) usando como `d`elimitador o espaço e pegando o segundo `f`ield ("para"):

```
cut -d " " -f 2 ola-cut.txt
```

`-d " "`: usa o delimitador espaço
`-f 2`: escolhe o segundo campo (field)

O que eu amo sobre os comandos do Linux é que eles se encaixam e viram muito poderosos juntos. Vamos combinar `ps`, `grep` e `cut` para pegar apenas o PID do Bash:

```
ps aux | grep bash | grep -v grep | cut -d " " -f 8
```

Obs: o segundo `grep` serve para que o processo do próprio `grep bash` não apareça na saída (porque aí teríamos duas linhas para tratar: uma do `bash` e outra do `grep bash`). Ficou confuso? Rode o comando sem o segundo `grep` e veja o que acontece!

Obs 2: Pegamos o oitavo campo (`-f 8`) porque na saída há mais de um espaço entre campos, o que cria vários campos vazios quando separamos por espaço. Quer fritar mais nisso? Procure sobre o comando `sed` e veja como usar ele para unir os espaços na linha.

Exercício Extra A:

a. Baixe os arquivos `second.txt` e `passwd`
b. Use `for` e `cut` para imprimir cada segunda palavra do arquivo `second.txt`
c. Use `for` e `cut` para imprimir todos os usuários do arquivo `passwd`
d. Use o mesmo código do item anterior mas dessa vez liste os usuários do arquivo `/etc/passwd` presente na sua próprima máquina em vez do `passwd` baixado

For

For é uma estrutura de repetição, também chamado do laço de repetição, presente em quase todas as linguagens de programação modernas -- nas mais antigas não tinha e a galera fazia

gambiarras para simular esses laços, então imagina a importância desse treco. No shell podemos criar laços também. A sintaxe do for é a seguinte:

```
for VARIAVEL in LISTA; do COMANDO; done
```

Para ficar mais claro, vamos usar um exemplo concreto. Suponha que recebemos um arquivo com uma lista de diretórios que temos que listar (esse arquivo é o `dirs.txt`, baixe ele). Vamos usar for para listar o conteúdo de todos os diretórios listados no arquivo dirs.txt:

```
for $linha in $(cat dirs.txt); do ls $linha; done
```

Nesse exemplo `$(cat dirs.txt)` é a lista de todas as linhas do arquivo dirs.txt, `$linha` é a variável que recebe o valor de cada elemento da lista, nesse caso cada linha do arquivo dirs.txt, e `ls $linha` é o comando que é executado a cada repetição do for.

Vamos ver um outro exemplo: baixar todos os arquivos listados em `cats.txt`. Primeiro vamos baixar cats.txt:

```
wget gmcruz.me/courses/ninjaterm/cats.txt
```

Agora que temos ele no nosso computador, vamos usar como lista para baixar os arquivos `um.txt`, `dois.txt`, `tres.txt` e `quatro.txt`

```
for $linha in $(cat cats.txt);do wget gmcruz.me/courses/ninjaterm/$linha;done
```

Esse último comando é equivalente a fazer:

```
wget gmcruz.me/courses/ninjaterm/for-cat/um.txt

wget gmcruz.me/courses/ninjaterm/for-cat/dois.txt

wget gmcruz.me/courses/ninjaterm/for-cat/tres.txt

wget gmcruz.me/courses/ninjaterm/for-cat/quatro.txt
```

Exercício Extra B:

a. Baixe o arquivo `pessoas.txt`
b. Para cada pessoa, imprima na tela "iae PESSOA" (onde PESSOA é o nome da pessoa)

Exercício Extra C: *network scan*

Vamos usar um exemplo real para ilustrar o que aprendemos até agora. O arquivo `scan.txt` contém o resultado de um scan feito em uma rede wireless.

a. Baixe o arquivo `scan.txt`

b. Imprima na tela os IPs que foram encontrados no scan
c. Extenda o item anterior e salve os IPs em um arquivo chamado `ips.txt`
d. Elimine linhas com IPs duplicados do arquivo `ips.txt` (dica: use `uniq` e `sort`)
e. Use o comando `ping -c 1` para enviar um ping para cada IP da lista salva em `ips.txt`

Não se preocupe se tiver dificuldades nesse exercício, é para ser mais difícil mesmo!

Operadores lógicos

Todo comando retorna algo. Se tudo deu certo com o comando, ele retorna uma "mensagem" que avisa o shell que tudo deu certo. Nós podemos usar esses retornos juntamente com operadores lógicos para rodar outros programas. Mas calma, vamos ver um exemplo:

```
sleep 3 && echo IAEE
```

Se tudo deu certo, esse comando deve ter "dormido" por três segundos (`sleep 3`) e depois imprimido IAEE (`echo IAEE`). Mas o que é o `&&` no meio?

Assim como em outras linguagens de programação, aqui também temos operadores como `AND` (`&&`) e `OR` (`||`), chamados de operadores lógicos. Eles são úteis para rodar comandos com alguma condição. No exemplo anterior temos um `AND` lógico, ou seja, se o primeiro comando deu certo ele retorna uma mensagem "Verdadeiro", o que faz com que o segundo comando seja executado. Se houvesse algum erro com o primeiro comando, o segundo não rodaria.

Já quando se trata do `OR`, o Bash roda ambos os comandos independente do retorno. Vamos ver o que acontece com o exemplo anterior quando trocamos o `AND` (`&&`) por um `OR` (`||`):

```
sleep 3 || echo IAEE
```

Ele dorme por 3 segundos e não imprime nada! Como se trata do `OR`, se o primeiro comando retornar "Verdadeiro" o segundo não roda (o resultado do `OR` é verdadeiro independente do segundo comando, já que o primeiro é verdadeiro).

Exercício Extra D:

a. Execute o seguinte comando: `sleep 7 && echo 1 && sleep 2 && echo 2 && sleep 4 && echo 3`. Qual a ordem dos echos? Por que isso acontece?
b. Execute o mesmo comando mas trocando todos os `&&` por `||`. O que muda?

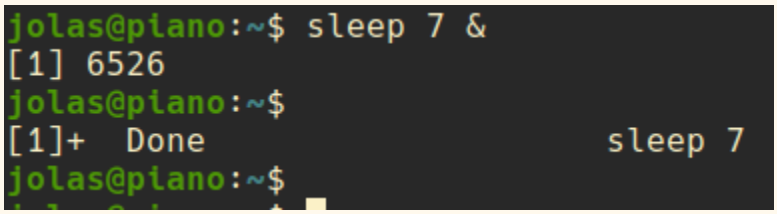
Jobs e processos em *background*

Vamos olhar usar novamente o comando `sleep` como exemplo:

```
sleep 7
```

Esse comando trava o terminal por 7 segundos até que o `sleep` tenha terminado de rodar. No entanto, é possível rodar ele em *background* (ou, numa tradução livre, em "plano de fundo") usando o operador (`&`):

```
sleep 7 &
```



Isso nos permite fazer outras coisas enquanto o sleep roda. Mas o que é esse `[1] 6526`?

Como já vimos sobre processos, é importante falar de jobs e das diferenças entre *jobs* e processos. Jobs, ou "trabalhos" em inglês, são programas em execução atrelados à sessão do shell atual. Assim, todo job é um processo associado ao shell atual, mas há diversos processos que não são jobs de nenhum shell (o processo do seu navegador, por exemplo). No exemplo anterior, quando rodamos `sleep 7 &` em background ele se torna um job do shell e recebe um job ID (no caso o ID dele é 1, denotado pelo `[1]` na saída). Vemos também que ele é de fato um processo (seu PID, `6526`, está logo depois do job ID).

Finalmente, podemos levar processos para o background, trazê-los para o *foreground* ("plano de frente"), e pausá-los com `bg`, `fg` e `Ctrl+z` respectivamente.

```
sleep 100 &
```

Iniciamos o `sleep` em background.

```
fg
```

Trazemos o `sleep` para o foreground, bloqueando novamente o terminal.

```
Ctrl+z
```

Pausamos o processo.

```
bg
```

Finalmente, o levamos de volta para o background.

Exercício Extra E:

a. Rode o seguinte comando: `sleep 300 &; sleep 200 &; sleep 100 &;`. Pesquise sobre o comando `fg` e use-o para trazer cada um dos jobs para o foreground e mate-os com `Ctrl+c`.

Respostas dos exercícios

Exercício 1:

```
wget gmcruz.me/courses/ninjaterm/there.txt

cat hello.txt there.txt
```

Exercício 2:

a. Touch muda tempos de criação de um arquivo existente, mas cria um novo arquivo vazio se não existir
c. `touch arquivo-vazio.txt`

Exercício 3:

a. `echo \"hello world!\"\;`
b. `echo até mais > tchau.txt`
c. `ls > ls.txt`

Exercício 3:

a. `wget gmcruz.me/courses/ninjaterm/big.txt`
b. `grep OPA big.txt`
c. `grep resposta big.txt | grep -i seja`
d. Use `grep -n`

Exercício 5:

a. `wget gmcruz.me/courses/ninjaterm/bullshit.txt`
b. Use `dd` para cortar e `pp` para colar as linhas no lugar correto (usando visual lines também é uma boa opção)
c. Use *change word* (`cw`) e escreva seu nome, ou `:%s/gabriel/seu_nome/cg`
d. Use o visual block (`Shift+v`) para selecionar a primeira coluna e delete (`d`)
e. `:%s/[0-9]\+//cg`

Obs: compare o arquivo resultante com `ans-bullshit.txt`

Exercício 6:

a. `ps | grep bash`
b. `pgrep bash`
c. É a variável que contém o PID do shell atual (use `echo $$` para vê-la)

Exercícios Extras

Exercício Extra A:

a. `wget gmcruz.me/courses/ninjaterm/second.txt`
   `wget gmcruz.me/courses/ninjaterm/passwd`
b. `cat second.txt | cut -d " " -f 2`
c. `cat passwd | cut -d ":" -f 1`
d. `cat /etc/passwd | cut -d ":" -f 1`

Exercício Extra B:

a. `wget gmcruz.me/courses/ninjaterm/pessoas.txt`
b. `for pessoa in $(cat pessoas.txt); do echo "iae $pessoa"; done`

Exercício Extra C (network scan):

a. `wget gmcruz.me/courses/ninjaterm/scan.txt`
b. `cat scan.txt | cut -d " " -f 2`
c. `cat scan.txt | cut -d " " -f 2 > ips.txt`
d. `cat ips.txt | sort | uniq`
e. `for ip in $(cat ips.txt); do ping -c 1 $ip; done`

Exercício Extra D:

a. A ordem de impressão é 1, 2, e 3 pois, como se trata de ANDs e todos os comandos avaliaram para "Verdadeiro", eles serão executados um seguido do outro.
b. Como o primeiro sleep avalia "Verdadeiro", os demais comandos nem são executados e não há nenhuma saída após os 7s de sleep.

Exercício Extra E:

a. `fg %1`, depois `Ctrl+z`