Boas práticas em programação Shell

Boas práticas em programação Shell
(sinais usados pelo kill e pelo trap)

O comando `kill` envia um sinal para o programa, e é esse sinal que vai determinar a ação que deverá ser executada usando para isso o comando `trap`.

O Linux nos disponibiliza 65 sinais, mas o `SIGKILL` (`-9`) deve ser usado somente em casos extremos, pois é impossível monitorá-lo, inviabilizando a possibilidade de tomar qualquer atitude antes do enceramento do *script*, o que pode criar severas falhas para o sistema.

Usando esse sinal, você não dará chance ao programa de restaurar o ambiente ideal de trabalho, isto é:

1) Derrubar as conexões de *socket;*
2) Limpar arquivos temporários;
3) Informar aos seus filhos que ele está saindo;
4) Restaurar as características do terminal;
5) Fechar bancos de dados

Boas práticas em programação Shell

(sinais usados pelo kill e pelo trap)

```
$ trap "echo Recebi CTRL+C" 2
^CRecebi CTRL+C
$ trap "echo recebi kill -9" 9
$ sleep 30&
[1] 47889
$ kill -9 $!
$ trap
trap -- 'echo Recebi CTRL+C' SIGINT
trap -- 'echo recebi kill -9' SIGKILL
$ trap 2 9
#  Se o teu programa for passar muito tempo sem
#+ comunicação com o terminal,use algo assim:
$ cat parafuso.sh
while kill -EXIT $1 2>&-
do
    for i in \| \/ \- \\ \| \/ \-
    do
        echo -en "\b$i"
        sleep 0.2
    done
done
```

Boas práticas em programação Shell

(Prefira builtins!)

Sempre escuto dizer que o *Shell* é lento. Te garanto que não é, e vou mostrar! Essa afirmação normalmente vem de pessoas que não conhecem bem o interpretador.

Um exemplo óbvio disso é que em mais de 90% dos *sites* você verá as pessoas criando arquivos com o comando `touch`, quando poderia fazer o mesmo com o *Shell* puro. Veja isso:

```
$ time for ((i=1; i<2000; i++))
>     { touch lixo$i; }
real    0m1,703s
user    0m1,313s
sys     0m0,525s
```

```
$ time for ((i=1; i<2000; i++))
>     { > lixo$1; }
real    0m0,119s
user    0m0,031s
sys     0m0,088s
```

Ou seja, usando *Shell* puro conseguimos um ganho de **1.331,09%**. Mas ainda assim existem objeções!

Dizem: o `touch` não é destrutivo e o `>` é. Respondo: se não tem certeza, use o `>>`

Dizem: para criar muitos arquivos o `touch` é mais veloz. Respondo: se for gerar mais de 100 arquivos, simultaneamente, o `touch` é mais veloz, mas, na boa, você já criou mais de 100 arquivos simultaneamente em aplicações profissionais???

Boas práticas em programação Shell

(Prefira builtins!)

Use e abuse de coringas (inclusive os ignorados coringas estendidos)

Acabei de criar indevidamente alguns arquivos, veja:

```
$ ls arq*
arq    arq00  arq02  arq04  arq06  arq08  arq1  arq3  arq5  arq7  arq9
arq0   arq01  arq03  arq05  arq07  arq09  arq2  arq4  arq6  arq8  arq51
```

Quero remover todos, exceto o `arq` e o `arq51` que não foram criados nessa lev

```
$ time for D in arq{0..9} arq{00..09}
> do rm $D
> done
real    0m0,026s
user    0m0,024s
sys     0m0,004s
```

```
$ time rm arq?(0)[0-9]
real    0m0,005s
user    0m0,001s
sys     0m0,004s
```

420,00%

Para aceitar uma resposta positiva, seja ela qual for:

```
$ [[ ${Resp^^} == @(S|SIM|Y|YES) ]] &&
>     echo Resposta positiva
```

Boas práticas em programação Shell

(Prefira builtins!)

```
$ time for ((i=1; i<200; i++)); {
>     for arq in $(ls *); do echo $Arq > /dev/null
> }; done
real    0m3,911s
user    0m2,347s
sys     0m1,607s
```

230,60%

```
$ time for ((i=1; i<200; i++)); {
>     for arq in $(echo *); do echo $Arq > /dev/null
> done; done
real    0m1,396s
user    0m0,968s
sys     0m0,462s
```

18,00%

```
$ time for ((i=1; i<200; i++)); {
>     for arq in *; do echo $Arq > /dev/null
> done; done
real    0m1,183s
user    0m0,718s
sys     0m0,450s
```

Boas práticas em programação Shell

(O cat é cat_astrófico...)

cat (1) - *concatenate files and print on the standard output*

```
$ seq 3 > nums
$ echo Nº | cat - nums
Nº
1
2
3
```

```
$ ls lixo* | paste - - -
lixo01   lixo02   lixo03
lixo04   lixo05   lixo06
lixo07   lixo08   lixo09
```

Pelo menos assim, todos usam `cat`:

```
$ cat quebralingua.txt
Eu vi um velho com um fole velho
nas costas.
Tanto fede o fole do velho,
Quanto o velho do fole fede.
real   0m1,851s
```

Mas nada impede de usarmos assim:

```
$ echo "$(<quebralingua.txt)"
Eu vi um velho com um fole velho
nas costas.
Tanto fede o fole do velho,
Quanto o velho do fole fede.
real    0m1,237s
```

49,63%

Boas práticas em programação Shell

(O cat é cat_astrófico...)

O uso de *here documents*, é muito aconselhável em diversos casos:

```
$ Var1="Posso usar o Python"; Var2="com as variáveis do Shell"
> python3 << FimPy
> print ("$Var1 $Var2")
> FimPy
Posso usar o Python com as variáveis do Shell
```

Nesse caso o *here documents*, trouxe o Python para dentro do *Shell*, usando seu ambiente,

Mas o que mais vemos é o `cat` em parceria com *here documents*, onde bastaria um `ec`

```
$ time for ((i=1; i<2000; i++)); {
>    cat > /dev/null <<- FimCat
>       Hoje em $(date '+%d/%m/%Y')
>       a mensagem era: $Msg
>    FimCat
> }
Hoje em 06/01/2023
a mensagem era: cat=CATástrofe
```

```
real    0m5,258s
user    0m4,038s
sys     0m1,443s
```

```
$ time for ((i=1; i<2000; i++)); {
   echo "Hoje em $(date '+%d/%m/%Y')
   a mensagem era: $Msg" > /dev/null
}
```

```
real    0m3,163s
user    0m2,230s
sys     0m1,029s
```

66,23%

Boas práticas em programação Shell

(O cat é cat_astrófico...)

Os tempos para fazer 2000 *loops*:

```
cat /etc/passwd | wc -l >            ev/null
real      0m2,702s
user      0m3,335s
sys       0m1,284s
```

Percent: **43,41%**

```
cat /etc/passwd | grep -q            wd
real      0m3,397s
user      0m3,976s
sys       0m1,672s
```

Percent: **46,80%**

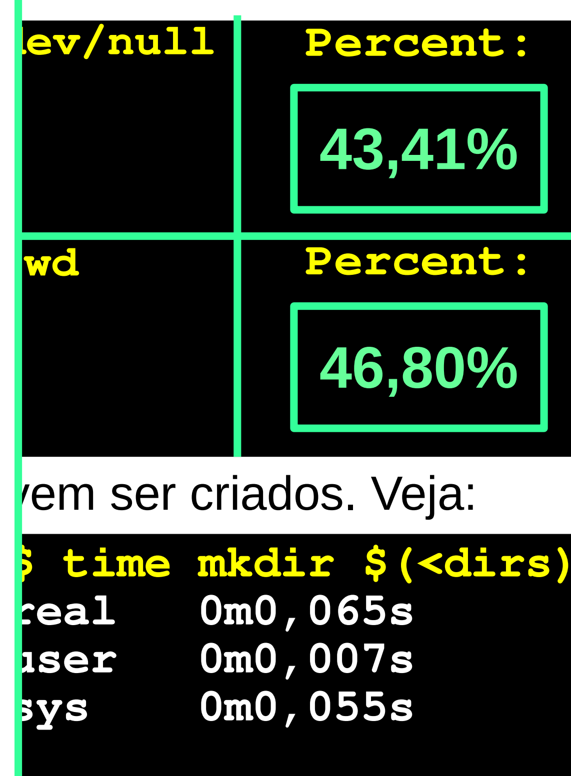
Tenho um arquivo `dirs` q   vem ser criados. Veja:

```
$ time for Dir in $(ca
    do mkdir $Dir; dor
real      0m2,240s
user      0m1,703s
sys       0m0,668s
```

"$(<

3.346,15%

```
$ time mkdir $(<dirs)
real     0m0,065s
user     0m0,007s
sys      0m0,055s
```



Boas práticas em programação Shell

(Quando possível, evite forks)

Os parênteses (exceto na matemática) e o *pipe* (|) criam *forks*, que oneram os tempos:

```
$ echo $BASHPID; : | echo $BASHPID
14096
15501

$ echo $BASHPID -- $(echo $BASHPID)
14230 -- 225153
```

```
$ echo $BASHPID; (echo $BASHPID)
14096
15505

$ echo $BASHPID -- `echo $BASHPID`
14230 -- 225173
```

Os *forks* são *Shells* filhos, que ao seu fim perdem tudo que foi criado/alterado/gerado no seu interior. É frequente vermos construções como as seguintes, mas devemos evitá-las...

Evite	Prefira
`$ cat ARQ | wc -l`	`$ wc -l < ARQ`
`$ echo CADEIA | COMANDO`	`$ COMANDO <<< CADEIA`
`$ V=2; (V=3; echo $V); echo $V`	`$ V=2; { V=3; echo $V; }; echo $V`
`$ cat ARQ | while read ...; done`	`$ while read ...; done < ARQ`

Boas práticas em programação Shell

(Procure soluções inéditas)

Para cada problema sempre existe muitas soluções. **Então pense fora da caixa!!**
Para descobrir o PID do meu *Bash*, fazemos:

```
$ ps au | grep bash
julio   1832   ...   bash
julio   3084   ...   grep --color=auto bash
```

Esta linha é indesejada ?

O que é normal vermos, para evitar a linha indesejada?

```
$ ps au | grep bash | grep -v grep
julio   1832   ...   bash
```

Mas assim é muito mais legal e muito mais eficiente:

```
$ ps au | grep '[b]ash'
julio   1832   ...   bash
```

Só para podermos entender o que houve:

```
$ ps au | grep -F '[b]ash'
julio   1832   ...   grep --color=auto -F [b]ash
```

Boas práticas em programação Shell

(Procure soluções inéditas)

Para cada problema sempre existe muitas soluções. **Então pense fora da caixa!!**

Para pegarmos da 2ª à 4ª linha de um arquivo, 90% dos programadores fariam:

```
$ seq 10 | sed -n '2,4p'
2
3
4
```

```
real    0m2.969s
user    0m0.268s
sys     0m0.544s
```

14,32%

O que não se vê ninguém fazendo é:

```
$ seq 10 | cut -f2-4 -d$'\n'
2
3
4
```

```
real    0m2.620s
user    0m0.356s
sys     0m0.484s
```

Boas práticas em programação Shell

(Expansão de Parâmetros - o Trem Bala)

Não conheço, em linguagens interpretadas, nada tão eficiente quanto as Expansões de Parâmetros. Vamos ver umas comparações para 2.000 ciclos (faremos `Var=Canção`):

Para medir o tamanho de uma variável (esta é a forma mais comum de vermos):

```
$ wc -c <<< $Var
9
```

```
-c, --bytes
        print the byte counts
```

Boas práticas em programação Shell

(Expansão de Parâmetros - o Trem Bala)

Não conheço, em linguagens interpretadas, nada tão eficiente quanto as Expansões de Parâmetros. Vamos ver umas comparações para 2.000 ciclos (faremos `Var=Canção`):

Para medir o tamanho de uma variável (esta é a forma menos ruim):

```
$ wc -m <<< $Var
7
```

-m, --chars
print the character counts

Boas práticas em programação Shell

(Expansão de Parâmetros - o Trem Bala)

Não conheço, em linguagens interpretadas, nada tão eficiente quanto as Expansões de Parâmetros. Vamos ver umas comparações para 2.000 ciclos (faremos `Var=Canção`):

Para medir o tamanho de uma variável (esta é a forma que seria correta):

```
$ echo -n $Var | wc -m
6
```

```
time for ((i=1; i<2000; i++))
  { echo -n $Var | wc -m > /dev/null; }
real   0m3,633s
user   0m3,172s
sys    0m1,409s
```

O turbinado:

```
$ echo ${#Var}
6
```

```
time for ((i=1; i<2000; i++))
   { echo ${#Var} > /dev/null; }
real   0m0,047s
user   0m0,027s
sys    0m0,019s
```

7.629,78%

Boas práticas em programação Shell

(Expansão de Parâmetros - o Trem Bala)

Para acelerar um pouco mais, procurando a variável `$_` no `man bash`, vemos que:

> `$_` *expands to the last argument to the previous command, after expansion.*

```
$ Var=$(ls *com*)
$ echo $Var
acomprar comprados
$ echo $_
comprados
$ echo "$Var"
acomprar
comprados
$ echo $_
acomprar comprados
```

E ainda no `man bash`, procurando pelo *builtin* `:` descobrimos:

> *No effect; the command* ***does nothing*** *beyond* ***expanding arguments*** *and performing any specified redirections. The return status is zero.*

Boas práticas em programação Shell

(Expansão de Parâmetros - o Trem Bala)

Não conheço, em linguagens interpretadas, nada tão eficiente quanto as Expansões de Parâmetros. Vamos ver umas comparações para 2.000 ciclos (faremos `Var=Canção`):

Para medir o tamanho de uma variável (esta é a forma que seria correta):

```
$ echo -n $Var | wc -m
6
```

```
time for ((i=1; i<2000; i++))
   { echo -n $Var | wc -m > /dev/null; }
real   0m3,633s
user   0m3,172s
sys    0m1,409s
```

O turbinado:

```
$ echo ${#Var}
6
```

```
time for ((i=1; i<2000; i++))
     { : ${#Var}; }; echo $_
real   0m0,028s
user   0m0,028s
sys    0m0,000s
6
```

9.344,11%

Boas práticas em programação Shell

(Expansão de Parâmetros - o Trem Bala)

Manipulando um caminho absoluto do arquivo `Prog=/home/julio/tstsh/script.sh`.

O pegar o nome do programa:

```
$ basename $Prog
script.sh
```

Uma forma melhor:

```
$ echo ${Prog##*/}
script.sh
```

Para 2.000 ciclos:

1.829,00% ou **3.407,27%**

Para pegar o diretório:

```
$ dirname $Prog
/home/julio/tstsh
```

Uma forma melhor:

```
$ echo ${Prog%/*}
/home/julio/tstsh
```

Para 2.000 ciclos:

1.997,80% ou **3.795,91%**

Em maiúscula (`V=abacaxi`)

```
$ tr a-z A-Z <<< $V
ABACAXI
```

Uma forma melhor:

```
$ echo ${V^} -- ${V^^}
Abacaxi--ABACAXI
```

Para 2.000 ciclos:

2.350,00% ou **4.925,64%**

Substituindo o `sed`

```
$ sed 's/a./s/g' <<< $V
sssi
```

Uma forma melhor:

```
$ echo ${V/a?/s}-${V//a?/s}
sacaxi--sssi
```

Para 2.000 ciclos:

2.698,87% ou **7.935,48%**

Boas práticas em programação Shell

(Expansão de Parâmetros - o Trem

```
# Também poderia ser:
$ echo ${Var: -6:4}
```

Sabendo que `Var=abacaxi`

Substituindo o `cut -c`:

```
$ cut -c2-5 <<< $Var
baca
```

Uma forma melhor:

```
$ echo ${Var:1:4}
baca
```

Para 2.000 ciclos:

2.245,23% ou 5.372,22%

Substituindo o `if`:

```
$ : ${Usu:=$LOGNAME}
$ echo O nome é $Nom ${Snom=+e o sobrenome é $SNom}
```

Trabalhando a caixa das letras:

```
$ echo ${Var^}
Abacaxi
$ Var=${Var^^}; echo $Var
ABACAXI
$ echo ${Var,}
aBACAXI
$ echo ${Var,,[AI]}
aBaCaXi
```

Boas práticas em programação Shell

(Sempre paralelize)

Aposto como as pessoas que pensam que o *Shell* tem poucos recursos e as que o acham lento, nunca ouviram falar que ele tem 3 instruções para disparar tarefas em paralelo:

1) `xargs`
2) `GNU parallel`
3) `coproc`

Mas antes, veja isso:

```
$ nproc
4
```

Veja uma pequena demo do paralelismo com o `xargs`. Opção `-P INT`:

Uso convencional do `xargs`

```
$ time echo {1..6} |
   xargs -t sleep
sleep 1 2 3 4 5 6
real    0m21,019s
user    0m0,003s
sys     0m0,004s
```

Executando um de cada vez

```
$ time echo {1..6} |
   xargs -tn1 sleep
sleep 1
sleep 2
sleep 3 ...
real    0m21,012s
user    0m0,009s
sys     0m0,004s
```

Paralelizando...

```
$ time echo {1..6} |
   xargs -tn1 -P0 sleep
sleep 1
sleep 2
sleep 3 ...
real    0m6,006s
user    0m0,006s
sys     0m0,005s
```

Boas práticas em programação Shell

(Sempre paralelize)

Aposto como as pessoas que pensam que o *Shell* tem poucos recursos e as que o acham lento, nunca ouviram falar que ele tem 3 instruções para disparar tarefas em paralelo:

1) `xargs`
2) `GNU parallel`
3) `coproc`

Veja uma pequena demo do paralelismo com o `xargs`. Opção `-P INT`:

E como isso é possível?

Paralelizando...

```
$ time echo {1..6} |
    xargs -t sleep
sleep 1 2 3 4 5 6
real    0m21,019s
user    0m0,003s
sys     0m0,004s
```

```
$ time echo {1..6} |
    xargs -tn1 sleep
sleep 1
sleep 2
sleep 3 ...
real    0m21,012s
user    0m0,009s
sys     0m0,004s
```

```
$ time echo {1..6} |
    xargs -tn1 -P0 sleep
sleep 1
sleep 2
sleep 3 ...
real    0m6,006s
user    0m0,006s
sys     0m0,005s
```

Boas práticas em programação Shell

(Finalizando)

Para aqueles que quiserem verificar os dados mostrados, deixo abaixo a função que usei para calcular os percentuais.

Como o `bc` usa ponto decimal e meu `time` usa a notação pt_BR, troquei as vírgulas decimais dos dados da saída do `time` para o sistema en_US e depois desconverti para apresentar a resposta

```
Perc ()
{
        #  $1 = Maior tempo; $2 = menor tempo
        #  O bc usa ponto decimal, então converto
        #+ vírgula em ponto e depois desconverto
    Perc=$(bc <<< "scale=2; (${1/,/.} - ${2/,/.})*100/${2/,/.}")
    echo ${Perc/./,}%
}; export -f Perc
```

Boas práticas em programação Shell

https://www.dicas-l.com.br/educacao/programacao-shell-linux

Julio Neves
+55 21 98112-9988