ESTATÍSTICAS DE UTILIZADORES EM BASH

Licenciatura em Engenharia Informática
Sistemas Operativos

Leandro Silva 93446
Mário Silva 93430
Taxa de contribuição: 50/50

Índice

Introdução

Na unidade curricular de Sistemas Operativos foi-nos pedido para realizarmos dois scripts: userstats.sh e comparestats.sh.

No userstats.sh é feita uma organização das informações fornecidas pelo comando **last** do linux com todos os nomes dos utilizadores, número de sessões, tempo total logged in, tempo máximo e tempo mínimo. Esta informação organizada pode depois ser passada para um ficheiro de texto.

No comparestats.sh é impresso no terminal a diferença das estatísticas entre dois ficheiros de texto obtidos a partir do primeiro script e em datas diferentes.

Ambos os scripts foram realizados em Bash.

Primeiro Script – Userstats.sh

Antes de implementar

Antes de começarmos a escrever o código, pensamos em como iria ser a estrutura deste mesmo e quais os principais objetivos a implementar: obter as estatísticas de todos os utilizadores, tratar desses dados e também conseguir tratar das opções passadas ao programa.

Numa primeira abordagem guardamos o output todo num array da execução do comando **last**, e iteramos o array para encontrar os utilizadores, depois, comparávamos cada utilizador ao argumento passado na opção **-u** (caso este tivesse sido passado), e para fazer os tempos totais, máximo e mínimo logged in, fazíamos numa outra iteração ao array, e por causa das opções **-s** e **-e** víamos as datas e comparávamos aos argumentos passados.

Rapidamente reparamos que era desnecessário muitas funções pois pesquisamos mais sobre as funcionalidades que a Bash possui e faziam exatamente aquilo que pretendíamos em poucas linhas de código.

Tratamento de opções

O nosso código inicia-se com a declaração de algumas variáveis globais que vão ser importantes para o código a seguir.

```
argu=".*"
sort_tp="sort"
only_one="n"
file_ses="last -w"
```

Começamos por fazer o tratamento das opções com utilização do **getopts**, onde colocamos as várias opções que se podem selecionar, sendo que algumas destas recebem argumentos. Neste tipo de opções que recebem argumentos temos:

-u)

Neste caso passamos o argumento dado para uma variável global **argu**, que tem inicialmente com valor ".*", para depois ser usado sempre que for necessário fazer o grep com o valor do utilizador.

-g)

Para esta opção, o argumento dado é passado para uma variável **group** que depois vai ser usada mais tarde no código.

-f)

Aqui fazemos uma verificação se o argumento é na verdade um ficheiro ou não e depois na variável global **file_ses**, que foi inicializada com o valor "last -w", adicionamos " -f $argumento" pois o **last** tem essa opção em que permite selecionar o ficheiro a partir do qual vai-se buscar as sessões.

-s, -e)

Nestas duas opções transformamos a data recebida para um formato que é aceitável pelo comando **last**, e finalmente adicionamos à variável **file_ses**, -s para a data de início e -t para a do fim de sessão mais a $date transformada. O formato do tempo aceitável pelo **last** que escolhemos foi:

YYYY-MM-DD hh:mm (segundos são colocados a 00)

```
    s)
          valArg $OPTARG
          dateCheck
          start_date=$(date    -d    "$OPTARG"    +"%Y-%m-%d
%H:%M")
          file_ses="${file_ses} -s \"$start_date\""
         ;;
    e)
          valArg $OPTARG
          dateCheck
          end_date=$(date -d "$OPTARG" +"%Y-%m-%d %H:%M")
          file_ses="${file_ses} -t \"$end_date\""
         ;;
```

Também verificamos para as opções de ordenação que se podem selecionar apenas uma delas com uma função **valOpt()** que utiliza uma variável global e verifica se valor desta é "y", inicialmente com um valor "n" e depois dá um valor "y" quando uma das opções é selecionada. Para estas opções de ordenação é usada a variável global **sort_tp** inicialmente com valor "sort", e quando uma das opções é selecionada é adicionada a esta variável as opções que o sort dispõe para efetuar a ordenação pretendida.

```
     i)
          sort_tp="$sort_tp -n -k5"
          valOpt
         ;;
     a)
          sort_tp="$sort_tp -n -k4"
          valOpt
         ;;
     t)
          sort_tp="$sort_tp -n -k3"
          valOpt
         ;;
     n)
          sort_tp="$sort_tp -n -k2"
          valOpt
         ;;
     r)
          sort_tp="$sort_tp -r"
         ;;
```

Tratamento dos Dados

Depois de tratados os dados no **getopts** (caso tenham sido passado argumentos ao programa), vamos buscar todos os utilizadores, com recurso ao comando **uniq** vamos buscar os utilizadores que aparecem no output do last pelo menos uma vez e com o **grep** filtramos apenas aqueles que são pretendidos (por default são todos, logo ".*").

```
    users="$(eval $file_ses | awk '{ if (( $1 !~ /reboot/
&& $1 !~ /wtmp/ && $1 !~ /shutdown/  && $10 !~ /in/ && $10
!~ /no/ )) {print $1}}' | sort |uniq | grep "$argu" )"
```

De seguida percorremos o array com todos os utilizadores, e primeiramente verificamos se a variável **group** foi iniciada, e se tiver sido, colocamos num array **user_group** todos os grupos que o utilizador que está a ser percorrido atualmente pertence, e com uma função **includes()** verificamos se esse array contém o grupo do utilizador, se não pertencer, fazemos "continue" para passar para a seguinte iteração do ciclo for, ou seja, o próximo utilizador.

```
  if [ -n "$group" ]; then
      user_group=($(id -G -n $user ))
      if [ $(includes) == "n" ]; then
          continue
      fi
  fi
```

Para obter a contagem do número de sessões que cada utilizador tem, utilizamos o comando **last** e com o awk obtemos apenas a primeira palavra, ou seja, o nome do utilizador e com o grep filtramos apenas o $user que queremos e com o wc -l faz-se a contagem do número de vezes que o utilizador aparece no comando **last** executado.

```
  count=$(eval $file_ses | awk '{ if (( $1 !~ /reboot/ &&
$1 !~ /wtmp/ && $1 !~ /shutdown/  && $10 !~ /in/ && $10 !~
/no/ )) {print $1}}' | grep -o $user | wc -l)
```

De forma a obter todos os tempos que o utilizador esteve logged in, colocamos num array **time_log** utilizando o comando **last** e com o awk vamos buscar apenas a décima parcela de cada linha, ou seja o tempo total de sessão, e com o sed eliminamos os caracteres: '[', ']', '(', ')' e também '-' pois em alguns casos mais específicos encontrei este caracter no inicio dos tempos.

```
  time_log=($(eval $file_ses | awk '{ if (( $1 !~ /reboot/
&& $1 !~ /wtmp/ && $1 !~ /shutdown/  && $10 !~ /in/ && $10
!~ /no/ )) {print}}' | grep "$user" | awk '{ print $10 }' |
tr " " " " | sed 's/[-()]//g'))
```

Depois percorremos o array **time_log**, caso o valor que está no array tenha comprimento superior a 5, ou seja, não vai estar no formato “20:00”, mas sim num formato como “1+20:00”, o que significa que teve dias logged in e portanto temos de multiplicar por 1440 para obter o número de minutos. Para isso, substituímos o ‘+’ usando tr ‘+’ ‘:’ e depois com o awk fornecemos o argumento -F que vai dizer ao awk que tipo de separador deve considerar, sendo que neste caso -F: significa que o separador é ‘:’.

De seguida como $total é inicializada com valor ‘0’, verificamos se $total continua com esse valor, e se tiver, significa que é a primeira iteração ou que o tempo total das sessões até ao momento foi ‘0’ o que não influencia no resto, logo definimos o tempo mínimo e máximo igual ao tempo da sessão atual, e só depois fazemos a soma do tempo total com a sessão atual. Finalmente fazemos duas simples comparações com o tempo da sessão atual para se obter o tempo mínimo e máximo de todas as sessões.

```
     let "total = 0"
     for value in ${time_log[@]}; do
          if (( ${#value} > 5 )); then
                    time_session=$(echo $value | tr '+' ':'
| awk -F: '{ print ($1 * 1440) + ($2 * 60) + $3 }')
          else
                    time_session=$(echo $value | awk -F: '{
print ($1 * 60) + $2 }')
          fi

          if (( $total == 0 )) ; then
                    let "time_max = time_session"
                    let "time_min = time_session"
          fi

          total=$(($total + $time_session))

          if (( time_session < time_min )); then
                    time_min=$time_session
          fi
          if (( time_session > time_max )); then
                    time_max=$time_session
          fi
     done
```

Ao terminar o ciclo for, colocamos num array **userstats** no índice igual ao tamanho do array o conteúdo que pretendemos, que são: o utilizador, nº de sessões, tempo total, tempo máximo e mínimo.

No fim do código fazemos printf do array **userstats** com a devida ordenação.

```
    printf '%s\n' "${userstats[@]}" | $sort_tp
```

Verificações

Quando fazemos a execução do comando **last**, verificamos sempre se a primeira palavra de cada linha do output do **last** não é “reboot”, “wtmp” nem “shutdown” e também se a décima palavra não é “in” e “no” para tratar de casos mais específicos, desta forma:

```
    last | awk '{ if (( $1 !~ /reboot/ && $1 !~ /wtmp/ && $1 !~ /shutdown/ && $10 !~ /in/ && $10 !~ /no/ )) {print}}'
```

Para as opções que recebem argumentos criamos uma função **valArg()** que verifica se o argumento começa por um ‘-’ para evitar casos como por ex: “ $ ./userstats.sh -f -r ”, em que o ‘-f‘ interpreta ‘-r ‘, uma outra opção como argumento dele.

```
    function valArg() {
       if [[ $1 == -* ]]; then
           printf "Invalid Argument.\n"
           how_to
           exit 1;
       fi
    }
```

Também verificamos para as opções de ordenação que se podem selecionar apenas uma delas com uma função **valOpt()** que utiliza uma variável global e verifica se valor desta é “y”, inicialmente com um valor “n” e depois dá um valor “y” quando uma das opções é selecionada.

```
    function valOpt(){
       if [[ $only_one == "y" ]];then
           echo "Can't select these arguments together"
           how_to
           exit 1;
       fi
       only_one="y"
    }
```

Em casos mais específicos, como a opção -f verificamos se o argumento passado é na verdade um ficheiro e se este existe ou não.

```
    f)
        if [ -f "$OPTARG" ] ; then
            file_ses="$file_ses -f $OPTARG";
        else
            printf "File wasn't found.\n";
            how_to
            exit 1;
        fi
        ;;
```

Para a opção -s e -e temos uma função **dateCheck()** que verifica se a data dada como argumento tem o formato desejado, que consideramos que apenas se pode usar formatos do tipo: “Nov 10 00:00”, tivemos que usar esta função que verifica se ao transformar o argumento recebido para a data com o formato desejado ele se mantém igual, mas como em algumas versões da Bash o mês fica em minúsculas e noutras em maiúsculas usados esta sintaxe que encontramos na Internet para colocar ambos em minúsculas.

```
    function dateCheck() {
        if ! [[ $(tr "[:upper:]" "[:lower:]" <<<"$(date
"+%b %d %H:%M" -d "${OPTARG}")") = $(tr "[:upper:]"
"[:lower:]" <<<"${OPTARG}") ]]; then
            echo "The date isn't valid."
            how_to
            exit 1;
        fi
    }
```

Com foi referido anteriormente, usamos a função **includes()** para verificarmos se o array contém o grupo do utilizador, percorrendo o array e devolvendo o valor “y” se for encontrado um valor igual no array ao do grupo pretendido.

```
    function includes() {
        for element in ${user_group[@]}; do
            if [ "${element}" == "${group}" ]; then
                echo "y"
                return
            fi
        done
        echo "n"
        return
    }
```

Uma última verificação é feita mesmo antes do printf final, se o array de utilizadores estiver sem conteúdo, ou seja, o valor do argumento da opção -u não foi encontrado na execução do **last**, e também se o array **userstats**, que é imprimido no final se encontra vazio, pois ao executar o programa normalmente pode ser encontrado utilizadores mas ao percorrer esses utilizadores, e a opção -g tiver sido selecionada, mas nenhum utilizador pertencer ao grupo passado, o array vai encontrar-se vazio.

```
    if [[ ${users[0]} == "" || ${userstats[0]} == "" ]]; then
        echo "No Users Were Found."
        how_to
        exit 1;
    fi
```

Testes de Validação

Foram realizados alguns testes ao programa para verificar se estava a executar como pretendido, e fizemos os seguintes testes:

```
$ ./userstats.sh
sop0101 1 339 339 339
sop0102 1 0 0 0
sop0404 14 1023 343 1
sop0405 4 335 132 5
sop0406 4 76 53 0
sop0407 6 435 238 4
sop0409 1 2 2 2
```

```
$./userstats.sh -f
/var/log/wtmp
sop0101 1 339 339 339
sop0102 1 0 0 0
sop0404 14 1023 343 1
sop0405 4 335 132 5
sop0406 4 76 53 0
sop0407 6 435 238 4
sop0409 1 2 2 2
```

```
$ ./userstats.sh -u "sop.*"
sop0101 1 339 339 339
sop0102 1 0 0 0
sop0404 14 1023 343 1
sop0405 4 335 132 5
sop0406 4 76 53 0
sop0407 6 435 238 4
sop0409 1 2 2 2
```

Estes três testes verificam a execução normal do programa, quando se seleciona o ficheiro que pretende-se utilizar para ir buscar os dados das sessões e também quando se pesquisa por um utilizador com uma expressão regex.

```
$./userstats.sh -g sop
sop0101 1 339 339 339
sop0102 1 0 0 0
sop0404 14 1023 343 1
sop0405 4 335 132 5
sop0406 4 76 53 0
sop0407 6 435 238 4
sop0409 1 2 2 2
```

```
$  ./userstats.sh -s "Nov
22 10:00" -e "Nov 25
18:00"
sop0404 10 666 234 1
sop0405 1 5 5 5
sop0406 1 21 21 21
sop0407 6 435 238 4
```

```
$ ./userstats.sh -t -u
"sop.*"
sop0102 1 0 0 0
sop0409 1 2 2 2
sop0406 5 77 53 0
sop0405 4 335 132 5
sop0101 1 339 339 339
sop0407 6 435 238 4
sop0404 14 1023 343
```

Agora testamos a filtragem de utilizadores inserindo o grupo a que pertencem, a seleção das datas que delimitam a data na qual o utilizador fez log in e, ordenamos por ordem crescente (default) de acordo com o tempo total logged in para os utilizadores pretendidos.

```
$ ./userstats.sh -n -u
"sop.*"
sop0101 1 339 339 339
sop0102 1 0 0 0
sop0409 1 2 2 2
sop0405 4 335 132 5
sop0406 5 77 53 0
sop0407 6 435 238 4
sop0404 14 1023 343 1
```

```
$ ./userstats.sh -t -r -u
"sop.*"
sop0404 14 1023 343 1
sop0407 6 435 238 4
sop0101 1 339 339 339
sop0405 4 335 132 5
sop0406 5 77 53 0
sop0409 1 2 2 2
sop0102 1 0 0 0
```

```
./userstats.sh -a -r -u
"sop.*"
sop0404 14 1023 343 1
sop0101 1 339 339 339
sop0407 6 435 238 4
sop0405 4 335 132 5
sop0406 5 77 53 0
sop0409 1 2 2 2
sop0102 1 0 0 0
```

Validamos agora a ordenação por número de sessões totais de cada utilizador, a mostragem dos dados por ordem decrescente e pela sessão com maior tempo logged in.

```
$ ./userstats.sh -i -r -u "sop.*"
sop0101 1 339 339 339
sop0405 4 335 132 5
sop0407 6 435 238 4
sop0409 1 2 2 2
sop0404 14 1023 343 1
sop0406 5 77 53 0
sop0102 1 0 0 0
```

```
$./userstats.sh -n -u ".*" >
userstats_20191012
```

E finalmente validamos a ordenação pela sessão com menor tempo logged in e a escrita do output do programa para um ficheiro de texto.

Segundo Script - comparestats.sh

No comparestats.sh, para fazer a leitura dos argumentos da linha de comando, também há um **getopts** que recebe o tipo de sort a ser feito na listagem dos utilizadores. A única diferença é que o programa também terá de ler 2 ficheiros com a listagem das informações dos utilizadores preparada no userstats.sh.

Para receber os ficheiros, o programa irá ler o nome dos últimos 2 argumentos da linha de comando e irá validar a sua existência. Se a existência dos ficheiros for confirmada, procede-se à leitura de ambos os ficheiros.

```
# Validates and reads both files
file1="${@:(-2):1}"; file2="${@:(-1):1}"
if [ ! -f $file1 ] || [ ! -f $file2 ]; then
     printf "File wasn't found.\n";
     how_to
     exit 1
fi
IFS=$'\n'
userstats=(`cat $file1`); userstats2=(`cat $file2`)
```

Com o conteúdo dos ficheiros guardado nas variáveis **userstats** e **userstats2**, é feita uma iteração para os utilizadores da segunda variável (que tem as informações mais antigas). Para isso, foi necessário mudar o *internal field separator* para '\n'. Em cada iteração, as informações do utilizador são guardadas num array, stats, de onde depois são separadas e guardadas em variáveis diferentes: **user**, **count**, **total**, **time_max**, **time_min**.

Depois, é feito uma nova iteração, desta vez para os utilizadores de **userstats**, onde também se guardam no array **stats** as informações do utilizador. Com isto, se o stats[0] foi igual à variável **user**, significa que se trata do mesmo utilizador e sendo assim pode-se proceder à substituição das estatísticas de **userstats** (lidas do ficheiro mais recente), pela diferença das estatísticas entre ambos os ficheiros. Note que a variável **found** que inicialmente é false, é alterada para true quando isto acontece.

Se na segunda iteração não for encontrado um utilizador em **userstats** correspondente ao utilizador de **userstats2**, este último é adicionado ao array **userstats** sem nenhuma alteração. Isto ocorre somente quando no fim da segunda iteração found continuar a ser false.

```
     # Puts the userstats2 into the array userstats by
finding the correspondent user and making the difference
     for stats in ${userstats2[@]}; do
          IFS=$' '
          stats=($stats)
          IFS=$'\n'
          user=${stats[0]}; count=${stats[1]};
total=${stats[2]}; time_max=${stats[3]};
time_min=${stats[4]}
          found=false; i=0
          for stats in ${userstats[@]}; do
               IFS=$' '
               stats=($stats)
               IFS=$'\n'
               if [[ ${stats[0]} == $user ]]; then
                    found=true
                    let "count = stats[1] - count"
                    let "total = stats[2] - total"
                    let "time_max = stats[3] - time_max"
                    let "time_min = stats[4] - time_min"
                    userstats[i]=$(echo "$user $count
$total $time_max $time_min")
               fi
               let "i = i + 1"
          done
          if ! $found; then userstats[i]=$(echo "$user
$count $total $time_max $time_min"); fi
     done
```

Por fim, é impresso no terminal a lista de utilizadores do array **userstats** ordenada conforme o valor da variável **sort_tp** determinada no **getopts**, tal como no userstats.sh.

Testes de Validação

Para validar o nosso código criamos dois ficheiros com utilizadores semelhantes em ambos e alguns únicos, e depois de executar alcançamos o output pretendido, a comparação está correta e a ordenação também.

```
$ ./comparestats.sh
ficheiroqql ficheiroqql2
nlau 6 11 4 0
sd0104 4 579 230 78
sop0101 8 880 975 -14
sop0202 16 5487 4410 3
sop0404 14 1023 343 0
sop0406 -5 328 102 0
```

```
$ ./comparestats.sh -r
ficheiroqql ficheiroqql2
sop0406 -5 328 102 0
sop0404 14 1023 343 0
sop0202 16 5487 4410 3
sop0101 8 880 975 -14
sd0104 4 579 230 78
nlau 6 11 4 0
```

```
$ ./comparestats.sh -t
ficheiroqql ficheiroqql2
nlau 6 11 4 0
sop0406 -5 328 102 0
sd0104 4 579 230 78
sop0101 8 880 975 -14
sop0404 14 1023 343 0
sop0202 16 5487 4410 3
```

Conclusões

Com este trabalho, sentimos que conseguimos aprofundar o nosso conhecimento em Bash, quer em modo iterativo (linha de comando), quer em modo não iterativo (scripts). Entender e construir Shell Scripts é indispensável para quem quiser perceber as inúmeras possibilidades do mundo dos interpretadores de comandos.

O número de formas diferentes para executar a mesma tarefa é fascinante e encontrara forma melhor nem sempre é fácil. Contudo, fizemos um esforço para trazer soluções eficientes e otimizadas.

Bibliografia

https://stackoverflow.com/questions/3685970/check-if-a-bash-array-contains-a-value

https://stackoverflow.com/questions/26320553/case-insensitive-comparision-in-if-condition?lq=1

https://www.geeksforgeeks.org/last-command-in-linux-with-examples/