



universidade
de aveiro

Relatório

Trabalho 1

Ferramentas de descoberta do espaço ocupado em disco

Mariana Gameiro, nº 88837, Engenharia Informática
Rui Melo, nº 88889, Engenharia Informática

Índice

1	Introdução
2	<code>totalspace.sh</code>
2	Tratamento de dados
7	Opção -n
8	Opção -d
9	Opção -l
10	Opção -L
11	Combinações e Impressão
12	<code>nespace.sh</code>
13	Conclusão
14	Bibliografia

Introdução

O trabalho foi desenvolvido no âmbito da disciplina de S.O. (Sistemas Operativos) no ano letivo de 2018/19, no primeiro semestre.

O trabalho está composto por três partes:

- totalspace.sh;
- nespace.sh;
- Relatório referente ao mesmo.

Com o desenvolvimento destes scripts em bash, pretende-se determinar “o espaço ocupado em disco por ficheiros com determinadas propriedades e que podem ser candidatos a ser apagados”

totalspace.sh

Tratamento de dados


O script totalspace.sh permite “a visualização do espaço ocupado pelos ficheiros selecionados em todas as subdiretorias da(s) diretoria(s) que lhe é(são) passada(s) como argumento”.

Para este efeito, existem várias opções a serem implementadas:

“-n”; “-l”; “-d”; “-L”; “-r”; “-a”.

Inicialmente, o grupo começou por explorar as funcionalidades dos comandos “ls”, aliado a “awk” e “grep” com o intuito de ser impresso o pretendido para cada opção individualmente. Com isto, deparou-se com um problema: Seria complicado imprimir dados que necessitassem de ser comparados com ficheiros de subdiretorias.

Sendo assim, a base para armazenar os dados que necessitássemos de guardar e de imprimir seria um Array Associativo:



```
declare -A ArrayGeral
```

Figura 1: ArrayGeral

Um Array Associativo é basicamente uma estrutura de dados em que a cada Key corresponde um Value, à semelhança de dicionários em Python.

Deste modo, conseguiríamos guardar os tamanhos de ficheiros/diretorias e associar o seu tamanho em bytes mais facilmente e, muito mais diretamente, possibilitar comparação entre os Values de cada Key, ou seja, comparar espaços ocupados em disco.

Neste caso específico, a Key corresponderia ao Diretório/Ficheiro e o Value ao Tamanho, uma vez que se invertêssemos esta correspondência, iria haver um grande número de colisões uma vez que seria muito provável encontrar dois ficheiros com o mesmo tamanho.

Tendo em consideração esta estrutura de dados, foi necessário implementar uma função que percorresse as diretorias e ficheiros recursivamente, de modo a listar e percorrer toda a informação necessária. Para isto foi criada uma função base:

```
function base() {  
    for dir in "$1"/* ; do  
        if [ -d "$dir" ]; then  
            size=$(dir_size $dir)  
            ArrayGeral["$dir"]=$size  
            base "$dir"  
        fi  
    done  
}
```

Figura 2: Função Base

Nesta função “\$1” corresponde à diretoria apresentada no terminal que será sempre o último argumento. A expressão [-d “\$dir”] verifica se a variável na expressão é, de facto uma diretoria ou não, à semelhança da expressão [-f “\$dir”] que verifica se é ou não ficheiro.

É de notar que nesta função aparece uma outra função que retorna/imprime o tamanho ocupado pela diretoria/ficheiro: dir_size()

```
function dir_size() {  
    y=$(du -sb $1 | awk '{printf "%10s \n", $1}')
```

Figura 3: Função para obter Tamanhos

Repare-se que o comando usado foi “du” e as opções “s” e “b” permitem uma apresentação do tamanho total e em bytes.

No conteúdo do guião referente a este trabalho, é mencionado que as diferentes opções podem ser combinadas, com exceção de “-l” e “-L”. Dado este facto, foi necessário implementar algo que nos permitisse realizar o que seria pretendido, independentemente da ordem em que as opções fossem colocadas como argumentos.

Para tal, foi usado o comando “getopts” com recurso a umas flags

```
opcaoA=0  
opcaoR=0  
opcaoL=0  
opcaoI=0  
opcaoN=0  
opcaoD=0
```

Figura 4: Flags

```
while getopts "n:l:d:L:ra" option; do
  case $option in
    n) opcaoN=${OPTARG}
      ;;
    d) opcaoD=${OPTARG}
      ;;
    l) opcaoL=1
      if [[ $opcaoL -ne 0 ]]; then
        echo "-l e -L não podem estar em simultâneo"
        exit 1
      fi
      ;;
    a) opcaoA=1
      ;;
    r) opcaoR=1;
      ;;
    L) opcaoL=${OPTARG}
      if [[ $opcaoL -ne 0 ]]; then
        echo "-l e -L não podem estar em simultâneo"
        exit 1
      fi
      ;;
    \?)
      echo "Opção inválida!"
      exit 1
      ;;
  esac
done
```

Figura 5: getopts

Usando este esquema, as informações relativas as opções ficam guardadas para serem usadas mais tarde. As flags ou são alteradas para 1, o que permite saber que essa opção foi acionada, ou altera para o seu argumento através do seguinte excerto:

```
n) opcaoN=${OPTARG}
;;
```

Figura 6: Utilização de OPTARG

A partir deste código, se as flags fossem diferentes do seu valor inicial (0), significavam que as opções das respetivas flags foram chamadas. No entanto, existia um problema:

Só com esse excerto de código, não saberíamos qual o argumento, numericamente, que correspondia à diretoria.

Contudo, como foi referido anteriormente, o argumento correspondente à diretoria seria sempre o último. Deste modo, criou-se um Array com os diferentes argumentos, chamado de inputs, a partir do qual foi possível criar uma variável com o nome do

diretório. A posição dele no Array seria igual ao numero total de argumentos menos 1, já que o Array começa em zero.

```
i=0
inputs=()
for arg in "$@"; do
    inputs[i]=$arg
    i=$((i + 1))
done
localizacao_diretoria=${#inputs[@]}
diretoria_nome=${inputs[$localizacao_diretoria-1]}
```

Figura 7: Obter Diretório a ser considerado

Para este projeto, a ideia foi criar um determinado número de funções que cobrisse algumas combinações de escolhas: “-n” e “-d” com “-l” e “-L”. Nestas combinações não entraram as opções “-r” e “-a”, uma vez que estas só alteram a ordem com que a informação seria impressa no terminal.

Para este efeito, foram criadas 12 funções referentes às combinações possíveis, no entanto 4 delas incidem na opção “-l”, outras 4 na opção “-L” e as restantes na ausência das duas opções referidas.

Opção -n

A opção “-n” tem com objetivo restringir os argumentos para apenas aqueles que tenham uma determinada extensão, como por exemplo: “*.txt”

Isto significa que, neste exemplo, para o tamanho dos diretórios e apresentação de ficheiros, só interessava os ficheiros de texto.

```
fullfilename="$dir"  
filename=$(basename "$fullfilename")  
ext="${filename##*.}"
```

Figura 8: Processo para retirar extensão a um ficheiro

Relativamente a este excerto de código, as primeiras três linhas retornam a extensão de um ficheiro, ou seja, se recebesse como variável “SOPaula09.pdf”, a variável “ext” seria “pdf”.

No então, não poderíamos comparar a extensão do argumento recebido neste excerto com o argumento da opção N, uma vez que este começa sempre por “*.”. Como tal, acrescentou-se ao “\$ext” este excerto.

```
if [ "$*.$ext" == "$opcaoN" ]; then
```

Figura 9: Comparação Extensão com argumento de N

Opção -d

A opção “-d” tem com objetivo limitar o número de ficheiros, à semelhança da opção “-n”, só que em vez de limitar pela extensão de um ficheiro, limita pela data máxima de acesso a esse ficheiro.

```
ola=$(stat -c %y "$dir" | awk '{print $1,$2}')
```

```
input1=$(date -d "$ola" +%s)
```

```
input2=$(date -d "$opcaoD" +%s)
```

Figura 10: Obter datas em segundos

Relativamente a este excerto de código, são necessárias duas datas para comparar. A data do ficheiro (\$ola) e a data máximo de acesso (\$opcaoD).

A primeira linha de código serve para retirar as Time Zones, ou seja, as diferenças de horário de região para região (+-hh:mm).

As duas seguintes linhas de código servem para tornar as datas em apenas segundos. Esta contagem é realizada tendo como referência 1 de janeiro de 1970.

```
if [[ $input1 -lt $input2 ]]; then
```

Figura 11: Comparação de datas

Finalmente apenas é necessário comparar as datas (já em segundos), onde a data do ficheiro/diretoria que se considera não deverá ser superior à data máxima estabelecida como argumento.

Opção -l

Na opção “-l”, cujo objetivo seria apresentar os maiores ficheiros em cada diretoria (em que o número de elementos apresentados seria indicado como argumento), além de uma variável local à qual correspondesse o tamanho do diretório, pretendeu-se criar um Array local (para cada diretoria) com um tamanho igual ao indicado anteriormente como argumento e preenchê-lo com zeros.

```
local tamanho_do_diretorio=0

for i in `seq 0 $opcao1`;
do
    tamanhos_dos_ficheiros[ $i ]=0
done
```

Figura 12: Array para guardar os maiores elementos

Com a ordenação do Array a cada iteração, o objetivo seria comparar o tamanho (em bytes) de cada ficheiro e, se fosse superior ao elemento mais baixo do Array, substituí-lo pelo tamanho do ficheiro em questão.

```
tamanhos_dos_ficheiros=( $(printf "%s\n" ${tamanhos_dos_ficheiros[@]} | sort -n) )
```

Figura 13: Ordenação do Array

Assim, no final de percorrer todos os ficheiros de cada diretoria, devolve-se o nome dos ficheiros e o respetivo tamanho. Nesse momento, soma-se os tamanhos dos ficheiros à variável local anteriormente referida e, posteriormente, adiciona-se ao Array Associativo a Key com o seu Value, Nome do Diretório e o seu tamanho, respetivamente.

```
for i in `seq 0 $opcao1`;
do
    elemento=${tamanhos_dos_ficheiros[ $i ]}
    tamanho_do_diretorio=$((tamanho_do_diretorio+elemento))
done
ArrayGeral+=([ $1 ]=$tamanho_do_diretorio)
```

Figura 14: Obter o tamanho do Diretório e guardar no ArrayGeral

Opção -L

Na opção “-L”, a ideia é semelhante, mas como o objetivo seria imprimir os maiores ficheiros entre todas as diretorias. Neste caso não necessitaríamos de um Array local.

O objetivo seria guardar todos os ficheiros no Array Associativo (“ArrayGeral”). Posteriormente para imprimir, necessitaríamos na impressão de fazer alguma espécie de ordenação como será abordado mais à frente neste documento.

```
function save_files() {  
    for dir in $1/*  
    do  
        if [ -d "$dir" ]  
        then  
            save_files $dir  
        fi  
        if [ -f "$dir" ]; then  
            result=$(dir_size $dir)  
            ArrayGeral+=([$dir]=$result)  
        fi  
    done  
}
```

Figura 15: Guardar ficheiros

Combinações e Impressão

Uma vez explicado e delineado as ideias gerais por cada uma das opções, mas combiná-las basta criar uma sequência de if's e elif's e depois chamar as funções respetivas.

Quando definido qual a função a ser usada e quais os argumentos respetivos, apenas falta imprimir o ArrayGeral (todas as funções armazenam lá os dados).

Foi criada uma função com a finalidade de imprimir o Array Associativo com os dados conforme os argumentos inicialmente inseridos pelo utilizador.

```
function imprimir_array() {
  if [[ $opcaoA -eq 1 ]] && [[ $opcaoR -eq 1 ]] && [[ $opcaoL -ne 0 ]]; then ...
  elif [[ $opcaoA -eq 1 ]] && [[ $opcaoR -eq 1 ]]; then ...
  elif [[ $opcaoA -eq 1 ]] && [[ $opcaoL -ne 0 ]]; then ...
  elif [[ $opcaoR -eq 1 ]] && [[ $opcaoL -ne 0 ]]; then ...
  elif [[ $opcaoA -eq 1 ]]; then ...
  elif [[ $opcaoR -eq 1 ]]; then ...
  elif [[ $opcaoL -ne 0 ]]; then ...
  else ...
  fi
}
```

Figura 16: Função para imprimir dados

```
{
  for k in "${!ArrayGeral[@]}"
  do
    echo "${ArrayGeral[$k]} ${k}"
  done } | sort -nr -k2 | head -$opcaoL
```

Figura 17: Exemplo de como ordenar impressão

Neste último exemplo, pode-se observar como a ordenação foi realizada.

O “r” utilizado tem como função inverter a ordenação, o “-k2” tem como função selecionar o \${k} a partir do qual fará ordenação alfabética em conjunto com o “-n”. “Head -\$opcaoL” é usada quando a opção “-L” é pedida e tem como função apenas imprimir os x primeiros elementos (neste caso \$opcaoL).

nespace.sh

O programa nespace.sh é em tudo igual ao programa anterior, mas acrescenta uma funcionalidade. A partir de um ficheiro texto que contem uma lista de ficheiros, estes devem ser “ignorados”, ou seja, caso a opção “-e” seja usada, os ficheiros contidos nesta lista, não deverão ser contabilizados.

Para tal foi acrescentado uma flag (opcaoE), uma opção no getopt (página 5) e foi criado uma função que permite verificar se o ficheiro pode ser contabilizado ou não.

```
opcaoE="0"
```

Figura 20: Nova Flag

```
e)
    opcaoE=${OPTARG}
;;
```

Figura 18: Captura do argumento de -e

```
function ser_proibido_v2() {
    i=0
    e_proibido=0
    while IFS=' ' read -r line || [[ -n "$line" ]]; do
        y=$(dir_size "$line")

        proibidos[ $i ]=$line
        i=$((i+1))
    done < "$2"

    for i in ${proibidos[@]};
    do
        if [ "$1" == "$i" ]; then
            e_proibido=1
        fi
    done
    echo $e_proibido
}
```

Figura 19: Função para verificar se ficheiro é elegível

Uma vez com a função e a flag criada, só é necessário acrescentar uma expressão a verificar de cada ficheiro/diretoria pode ser elegível ou não.

```
if [[ "$opcaoE" != "0" ]]; then
    proibicao=$(ser_proibido_v2 $dir $opcaoE)
    if [[ $proibicao -ne 1 ]]; then
        ...
    fi
else
    ...
fi
```

Figura 21. Condições que foram implementadas

Conclusão

O projeto era ambicioso, mas realista, o que permitiu desenvolver a capacidade de procurar soluções, o trabalho em equipa e contornar as dificuldades de enfrentar uma linguagem que ainda não tínhamos muito contacto.

Não só pelas razões acima mencionadas, temos a certeza que valeu a pena o esforço e dedicação. A principal recompensa não é só a aquisição de conhecimentos, mas também o desenvolvimento de capacidades de relação social e saber interagir em grupo.

Bibliografia

<https://www.linuxjournal.com/content/bash-associative-arrays>

<https://stackoverflow.com/questions/16661982/check-folder-size-in-bash>

<http://ldmlinux.blogspot.com/2012/10/du-estimativa-de-espaco.html>

<https://stackoverflow.com/questions/27429653/date-comparison-in-bash>

<https://tecadmin.net/how-to-extract-filename-extension-in-shell-script/?fbclid=IwAR2k0g7W4LHjulThqA7Grfg41pk9YgXNrkpV1EKyle-zmerlcKUTFWK0Oeg>