Taxas de Leitura/Escrita de processos em *bash*

Sistemas Operativos

Trabalho realizado:

Alexandre Cotorobai – 107849

Vitalie Bologa – 107854

universidade de aveiro

2022/2023

Índice

Introdução

No âmbito da unidade curricular de Sistemas Operativos, apresentamos o primeiro trabalho desta disciplina cujo objetivo consiste no desenvolvimento de um script em bash para obter estatísticas sobre as leituras e escritas que os processos estão a efetuar.

Ao longo deste relatório iremos mostrar todo o raciocínio que nos levou a desenvolver o script em questão, explicaremos as funcionalidades do código (com exemplos), assim como o fluxo de execução do mesmo e o modo como os possíveis erros ao longo do processo serão tratados.

Metodologia

Neste *script* será gerada uma tabela, com todas as informações pedidas referentes a cada um dos processos. De modo a facilitar a visualização e aumentar a praticidade na sua utilização, o *script* irá receber argumentos constituídos por *flags* e os seus valores que permitirão uma filtragem personalizada dos dados visualizados.

O *script* desenvolvido funciona do seguinte modo: começa pela função *get_input()*, cujo papel é analisar todos os elementos passados na linha de comando, onde vai validar as *flags* e seus argumentos, e caso sejam válidos as *flags* são declaradas como usadas e os valores dos seus argumentos passados para uma variável.

Já com os *inputs* recebidos, é executada a função *get_pid_status()*, onde são guardadas em variáveis todas as informações relativas a cada processo, seguindo para a função *filter()*, onde de acordo com os critérios passados pelo utilizador, vai guardar num *array* os processos já filtrados.

Para terminar, é impresso o cabeçalho da tabela e chamada a função *print()* que irá exibir o conteúdo guardado em *allSavedPids* pela ordem indicada pelas variáveis *reverse* e *sortw*.

- Declaração de Variáveis

```
#!/bin/bash

declare c_used=0    # Variável de controlo de uso da flag -c
declare e_used=0    # Variável de controlo de uso da flag -e
declare u_used=0    # Variável de controlo de uso da flag -u
declare m_used=0    # Variável de controlo de uso da flag -m
declare M_used=0    # Variável de controlo de uso da flag -M
declare p_used=0    # Variável de controlo de uso da flag -p
declare reverse=0   # Variável de controlo de uso da flag -r
declare sortw=0     # Variável de controlo de uso da flag -s
declare i=0         # Variável usada para iteração na função filter()
declare -a allSavedPids    # Array usado para guardar linhas de informação que serão impressas na tabela
declare -A saveReadBytes   # Array usado para salvar o readbytes de cada pid
declare -A saveWriteBytes  # Array usado para salvar o writebytes de cada pid
declare p=-1               # Valor default para a flag -p, quando é -1 irá dar display de todo os processos
declare -a validadeMonths=("Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec")
                           # Array que contém o formato válido de cada mês
declare argCount=0         # Variável que conta o número de flags e o numero de argumentos
```

Figura 1 - Declaração da estrutura de dados

As estruturas de dados são a essência de qualquer programa, como boa prática declaramos todas as variáveis globais no início do ficheiro. Usamos várias variáveis de controlo para detetar quais as *flags* ativas pelo utilizador ao executar o programa, assim como definimos valores *default* para outras variáveis.

As estruturas de dados aqui usadas com um grau mais elevado de complexidade são os *arrays* associativos "*saveReadBytes*" e "*saveWriteBytes*" que servirão para atribuir o valor "*readbytes*" e "*writebytes*" ao seu respetivo *pid*, numa estrutura de *key-value* (estilo dicionário) e o "*allSavedPids*" que será um array onde ficará guardada toda

a informação pós-filtragem para no final ser impressa no terminal. Todas estas estruturas de dados serão utilizadas nas funções que iremos descrever a seguir e sofrerão modificações ao longo do processo.

- Função *get_input()*

```
function get_input(){

    if [[ $# == 0 ]]; then                  # verifica se não foram passados argumentos
        echo "ERROR: Has to have as least one argument (sleep time, in seconds)"
        exit 1
    fi

    while getopts "c:s:e:u:m:M:p:rw" opt; do # lê os argumentos passados
        case $opt in
```

Figura 2.1 - Função get_input()

A função *"get_input()"* é a primeira função a ser executada pela *"main()"*. Ela é invocada através da expressão *"get_input($@)"*, onde vai analisar todos os elementos *($@)* introduzidos.

Este processo começa, através de uma condição, por verificar a existência de argumentos, através da expressão *"$#"* que nos dará o número de argumentos passados e caso esse valor seja 0, a condição imprime uma mensagem de erro e encerra o programa.

```
while getopts "c:s:e:u:m:M:p:rw" opt; do # lê os argumentos passados
    case $opt in
        c)
            c="$OPTARG"

            # check if flag is used more than once
            if [[ $c_used == 1 ]]; then
                echo "ERROR: -c flag already used"
                menu
                exit 1
            fi

            argCount=$(($argCount+2))
            c_used=1
            ;;
```

Figura 2.2 - Função get_input()

Seguidamente, entramos num *loop "while getops"*, onde serão analisadas todas as possíveis *flags* e seus argumentos (se a *flag* requisitar). As opções válidas estão descritas na seguinte expressão *"c:s:e:u:m:M:p:rw"*, onde os ":" à frente de cada letra, simboliza que a *flag* requer um argumento, logo, somente "r" e "w" não necessitam de argumentos. Assim sendo, analisamos as seguintes opções:

-c) A *flag* "c" representa o argumento de filtragem por nome. A validação começa pela condicional, onde verifica se a *flag* foi utilizada, caso sim, é impresso uma

mensagem de erro e encerrado o programa. Se não, é advertido que a mesma foi usada, *c_used=1*, e é adicionado o valor 2 à variável *argCount*.

A argCount serve para contar o número de *flags* e seus argumentos para que mais a frente seja possível prever um possível erro, que consistia no caso do último elemento adicionado ser um argumento (de uma *flag*) numérico (inteiro), podendo ser confundido com o intervalo de tempo. Situação que é resolvida mais abaixo numa condicional que iremos explicar.

Tendo em conta que o *argCount* e a condição *c_used == 1*, são ambos repetidos para a maioria das *flags*, somente iremos mencioná-la nesta opção como exemplo para as restantes.

```
s)
    s=$OPTARG
    # check if flag is used more than once
    if [[ $s_used == 1 ]]; then
        echo "ERROR: -s flag already used"
        menu
        exit 1
    fi

    # valida o formato do argumento passado como data
    local mes=${s:0:3}
    if [[ $s =~ ^[A-Za-z]{3}\ [0-9]{1,2}\ [0-9]{1,2}:[0-9]{2}$ && "${validadeMonths[*]}" =~ "${mes^}" ]]; then
        s_used=1
    else
        echo "ERROR: Invalid date format"
        menu
        exit 1
    fi
    argCount=$(($argCount+2))
    s_used=1
    ;;
e)
    e="$OPTARG"
    if [[ $e_used == 1 ]]; then
        echo "ERROR: -e flag already used"
        menu
        exit 1
    fi

    # valida o formato do argumento passado como data
    local mes=${s:0:3}
    if [[ $s =~ ^[A-Za-z]{3}\ [0-9]{1,2}\ [0-9]{1,2}:[0-9]{2}$ && "${validadeMonths[*]}" =~ "${mes^}" ]]; then
        s_used=1
    else
        echo "ERROR: Invalid date format"
        menu
        exit 1
    fi
    argCount=$(($argCount+2))
    e_used=1
    ;;
u)
    u="$OPTARG"
    # check if flag is used more than once
    if [[ $u_used == 1 ]]; then
        echo "ERROR: -u flag already used"
        menu
        exit 1
    fi
    # verifica se o utilizador passado como argumento existe
    if ! id "$u" &>/dev/null; then
        echo 'ERROR: User not found'
        menu
        exit 1
    fi
    argCount=$(($argCount+2))
    u_used=1
    ;;
```

Figura 2.3 - Função get_input()

-s) e -e) As *flags* representam o argumento de filtragem por data, limite mínimo e máximo, respetivamente. É verificado o formato da data, três letras que representam o mês, um ou dois dígitos (representam o dia), seguidos de um ou dois dígitos separados por ":" de outros dois dígitos, que representam as horas (por exemplo 17:30). O mês passado é validado de acordo com o conteúdo do *array validadeMonths*, caso seja escrito um mês que não exista nesse *array* o programa dará erro.

-u) A *flag* "u" representa o argumento de filtragem por utilizador. Através da segunda condição, é verificado a existência do utilizador. Caso a condição falhe, é impresso uma mensagem de erro e encerrado o programa.

```
m)
    m=$OPTARG
    # check if flag is used more than once
    if [[ $m_used == 1 ]]; then
        echo "ERROR: -m flag already used"
        menu
        exit 1
    fi
    # verifica se o argumento passado é um número
    if ! [[ $m =~ ^[0-9]+$ ]]; then
        echo "ERROR: -m flag must be an integer"
        menu
        exit 1
    fi
    argCount=$(($argCount+2))

    m_used=1
    ;;

M)
    M=$OPTARG
    # check if flag is used more than once
    if [[ $M_used == 1 ]]; then
        echo "ERROR: -M flag already used"
        menu
        exit 1
    fi
    # verifica se o argumento passado é um número
    if ! [[ $M =~ ^[0-9]+$ ]]; then
        echo "ERROR: -M flag must be an integer"
        menu
        exit 1
    fi
    argCount=$(($argCount+2))

    M_used=1
    ;;

p)
    p=$OPTARG
    # verifica se o argumento passado é um número
    if [[ ! "${p}" =~ ^[0-9]+$ ]]; then
        echo "ERROR: -p flag must be followed by an integer"
        menu
        exit 1
    fi
    # check if flag is used more than once
    if [[ $p_used == 1 ]]; then
        echo "ERROR: -p flag already used"
        menu
        exit 1
    fi
    argCount=$(($argCount+2))
    p_used=1
    ;;
```

Figura 2.4 - Função get_input()

-m) e -M) As *flags* representam a gama de *pid* reproduzida na tabela, limite mínimo “m” e limite máximo “M”. É verificado se o argumento é composto por dígitos (número inteiro), caso não, é impresso uma mensagem de erro e encerrado o programa.

-p) A *flag* “p” representa a quantidade de processos que serão reproduzidos na tabela. O argumento terá de ser composto por dígitos (número inteiro), se não é impresso uma mensagem de erro e terminado o programa.

```
            r)
                # check if flag is used more than once
                if [[ $reverse == 1 ]]; then
                    echo "ERROR: -r flag already used"
                    menu
                    exit 1
                else
                    reverse=1
                fi
                argCount=$(($argCount+1))

                ;;
            w)
                # check if flag is used more than once
                if [[ $sortw == 1 ]]; then
                    echo "ERROR: -w flag already used"
                    menu
                    exit 1
                else
                    sortw=1
                fi
                argCount=$(($argCount+1))

                ;;
            \?)
                echo "Invalid option"
                exit 1
                ;;
            :)
                echo "Option -$OPTARG requires an argument." >&2
                exit 1
                ;;
        esac
    done
```

Figura 2.5 - Função get_input()

-r) A *flag* "r" representa como será ordenada a informação na tabela. A validação começa na condicional, que verifica se a *flag* foi usada, caso sim, é impressa uma mensagem de erro e encerra o programa, caso não, a *flag* é declarada como usada. Aqui apenas é adicionado o valor 1 à variável *argCount*, pois a *flag* não requer argumentos.

-w) A *flag* "w" vai reproduzir a tabela de processos por ordem decrescente dos *write values*. A validação é feita consoante o uso da *flag*, caso sim, é impressa uma mensagem de erro e encerra o programa, caso não, a *flag* é declarada como usada e também é apenas adicionado o valor 1 à variável *argCount*, pois a *flag* não requer argumentos.

\?) Caso seja introduzido uma *flag* não contida na expressão *"c:s:e:u:m:M:p:rw"* (expressão usada para a declaração das *flags*), o programa imprime uma mensagem a reportar o uso de uma *flag* inválida e é terminado o programa.

:) Numa situação em que uma *flag* requer um argumento e não é declarado, é reportado no terminal a necessidade de argumento para a determinada *flag* e encerrado o programa.

Tendo em conta a facilidade da ocorrência de erros no manuseamento das *flags*, é reportado e logo em seguida impresso o menu de navegação das *flags*, de forma a esclarecer o uso de cada *flag*.

```
if ! [[ "${@: -1}" =~ ^[0-9]+$ ]]; then                    # verifica se o últ
    echo "ERROR: The last argument must be a integer (sleep time, in seconds)"
    exit 1
fi
local nrInputs=$(($#-1)) # número de argumentos passados, excluindo o último, q

if [[ $nrInputs -ne $argCount ]]; then  # verifica se o número de flags e argum
    echo "ERROR: Sleep time must exist and must be the last argument"
    exit 1
fi
```

Figura 2.6 - Função get_input()

Segue por análise o último elemento da função (*${@: -1}*) correspondente ao *sleeptime*, caso este não seja composto por dígitos, é impresso no terminal uma mensagem de erro e o programa encerra.

É declarado um valor novo à variável *nrInputs*, que representa o número total de argumentos passados, menos o último, simbolizado pelo intervalo de tempo.

Por último, a condição verifica se o número de *flags* e argumentos, introduzidos na linha de comando, coincidem com o número de *flags* e argumentos esperados. Se a condição não se verificar, é impresso uma mensagem de erro como forma a defender uma possível situação, de o último argumento de uma *flag* introduzido ser um número inteiro e ser confundido como intervalo de tempo (por exemplo: *./rwstat -p 2*)

Concluindo, esta função, tem como função declarar as flags que vão ser usadas na função *filter()*.

- Função *get_pid_status()*

```
function get_pid_stats() {
    local sleeptime=$1

    for pid in $(ps -eo pid= | tail -n +2); do                                          # percorre todos os processos
        if [ -r /proc/$pid/io ] && [ -r /proc/$pid/status ] && [ -r /proc/$pid/comm ]; then # verifica as permisoes de read de cada processo
            local readbytes=$(grep -E 'rchar' /proc/$pid/io | awk '{print $2}')
            saveReadBytes[$pid]=$readbytes                                             # guarda o readbytes inicial num dicionario associado ao pid do processo

            local writebytes=$(grep -E 'wchar' -w /proc/$pid/io | awk '{print $2}')
            saveWriteBytes[$pid]=$writebytes                                           # guarda o writebytes inicial num dicionario associado ao pid do processo
        fi
    done

    sleep $sleeptime    # tempo de espera entre uma leitura e outra para que seja possivel calcular a diferença


    for pid in $(ps -eo pid= | tail -n +2); do                                             # percorre novamente todos os processos
        if [ -r /proc/$pid/io ] && [ -r /proc/$pid/status ] && [ -r /proc/$pid/comm ]; then     # verifica as permisoes de read de cada processo
            if [[ ! ${!saveReadBytes[@]} =~ "${pid}" ]]; then                              # verifica se o pid do processo está no dicionario, ..
                continue                                                                   # .. evita crashes caso um novo processo seja criado durante o sleeptime
            fi
            local readbytes1=$(grep -E 'rchar' /proc/$pid/io | awk '{print $2}')

            declare readbytes2=$(($readbytes1 - ${saveReadBytes[$pid]}))                # calcula a diferença entre o readbytes atual e o readbytes inicial

            declare readbps=$(($readbytes2) / $sleeptime))                              # calcula o readbytes por segundo

            local writebytes1=$(grep -E 'wchar' -w /proc/$pid/io | awk '{print $2}')

            declare writebytes2=$(($writebytes1 - ${saveWriteBytes[$pid]}))             # calcula a diferença entre o writebytes atual e o writebytes inicial

            declare writebps=$(($writebytes2) / $sleeptime))                            # calcula o writebytes por segundo

            declare comm=$(cat /proc/$pid/comm)                                        # guarda o nome do processo
            comm=${comm// /}                                                           # remove os espaços do nome do processo (caso existam)

            declare creationdate=$(ps -p $pid -o lstart= | awk '{print $2 " " $3 " " substr($4,1,length($4)-3)}') # guarda a data de criação do processo

            declare user=$(ps -p $pid -o user | tail -1)                               # guarda o utilizador que criou o processo

            filter  # invocação da função filter

        fi
    done
}
```

Figura 3 - Função get_pid_status()

Assim que *inputs* acabarem de ser lidos, a próxima função a ser executada será a *get_pid_stats()*. Nela, através de um ciclo for, todos os processos a decorrer serão lidos e uma verificação inicial é efetuada para detetar se o processo em questão consegue ser lido pelo utilizador que a executa, caso não haja essa permissão *"Permission Denied"* esse processo será ignorado e passamos ao processo seguinte.

Durante a leitura de cada processo serão guardados em dois dicionários (*arrays* associativos) distintos o número de total de *bytes* de I/O associados ao seu respectivo *pid*.

Após essa primeira leitura será dado um tempo de intervalo (*sleeptime*) para dar tempo aos processos de continuar a escrever/ler dados. Terminando esse tempo, será retomada uma nova leitura (também com recurso a um ciclo *for*) e uma verificação igual à anterior para descartar os processos a que não se tem permissão de leitura.

De modo a evitar um problema encontrado durante os testes também tivemos que fazer mais uma verificação, na segunda leitura teremos que saber se o *pid* do processo lido encontra-se nos dicionários *saveReadBytes* e *saveWriteBytes*, pois, caso contrário, o programa iria fazer uma subtração de um valor inexistente, o que levaria a um erro e quebra na execução. Isto acontece no caso de, durante o *sleeptime*, iniciar-se um novo processo, o que provoca que durante a segunda leitura o *script* irá procurar nos dicionários criados anteriormente pelo *pid* desse processo, mas como este apenas surgiu após já se ter feito a primeira leitura, não vai constar em nenhum dos dicionários e portanto dará erro. Para resolver este problema, basta que quando um dos *pids* da segunda leitura não existir nos *arrays* associativos passe para a próxima iteração do ciclo ignorando esse *pid*.

Visto isto, serão novamente lidos o número de total de *bytes* de I/O de cada *pid* e será feito o cálculo da variação desse mesmo número em relação à primeira leitura (resultado: *readbytes2* e *writebytes2*), outro dado importante será o valor dessa variação por unidade de tempo (neste caso segundos), bastando dividir cada um dos valores calculados anteriormente pelo tempo de intervalo (*sleeptime*) utilizado, resultando nas variáveis *readbps* e *writebps*.

Para completar a informação exigida para este trabalho também serão guardadas em variáveis o nome do processo, *comm*, a data em que foi criado, *creationdate*, e o utilizador a que esse processo corresponde, *user*.

A cada iteração deste ciclo é chamada a função *filter()* de modo a reter todos os processos indesejados.

Durante a ordenação da tabela por colunas, foi detetado um obstáculo que não ordenava os processos de maneira correta. Após pesquisas, descobrimos que, como alguns nomes tinham espaços, cada palavra contava como uma coluna. Tendo em conta essa questão, decidimos tirar os espaços dos nomes para corrigir esse problema.

- Função *filter()*

```
function filter() {
    if [[ $c_used -eq 1 ]]; then                        # caso a flag -c tenha sido usada, verifica se o nome do processo é igual ao nome passado como argumento
        if ! [[ "$(ps -p $pid -o comm=)" =~ ^$c+$ ]]; then   # caso não seja, a função retorna e o processo é ignorado
            return
        fi
    fi
    if [[ $s_used -eq 1 ]]; then                        # caso a flag -s tenha sido usada, verifica se a creationdate é maior que a data passada como argumento
        if [[ $creationdate < $s ]]; then               # caso não seja, a função retorna e o processo é ignorado
            return
        fi
    fi
    if [[ $e_used -eq 1 ]]; then                        # caso a flag -e tenha sido usada, verifica se a creationdate é menor que a data passada como argumento
        if [[ $creationdate > $e ]]; then               # caso não seja, a função retorna e o processo é ignorado
            return
        fi
    fi
    if [[ $u_used -eq 1 ]]; then                        # caso a flag -u tenha sido usada, verifica se o utilizador que criou o processo é igual ao nome passado como argumento
        if [[ $user != $u ]]; then                      # caso não seja, a função retorna e o processo é ignorado
            return
        fi

    fi
    if [[ $m_used -eq 1 ]]; then                        # caso a flag -m tenha sido usada, verifica se o pid do processo é maior que o valor passado como argumento
        if [[ $pid -lt $m ]]; then                        # caso não seja, a função retorna e o processo é ignorado
            return
        fi
    fi
    if [[ $M_used -eq 1 ]]; then                        # caso a flag -M tenha sido usada, verifica se o pid do processo é menor que o valor passado como argumento
        if [[ $pid -gt $M ]]; then                        # caso não seja, a função retorna e o processo é ignorado
            return
        fi
    fi
    # caso o processo tenha passado por todas as verificações, é adicionado ao array allSavedPids já formatado
    allSavedPids[$i]=$(printf "%-20s %-10s %+6s %+10s %+10s %+10s %+10s %+15s" "$comm" "$user" "$pid" "$readbytes2" "$writebytes2" "$readbps" "$writebps" "$creationdate")
    i=$((i+1))
}
```

Figura 4 - Função filter()

Nesta função, será averiguada a utilização ou não utilização de cada uma das *flags* disponíveis e em caso afirmativo será aplicado o filtro respetivo.

No caso de “c” e “u” o processo é bastante simples, apenas iremos verificar se os dados introduzidos pelo utilizador correspondem aos dados contidos no processo, no caso de “s”, “e”, “m” e “M” a verificação será feita através da comparação com o valor do processo.

Visto tratar-se de uma função, o processo mais simples e eficaz neste caso passa por observarmos se não se verifica alguma das condições. Caso a *flag* seja usada e a condição dentro desse *“if”* se cumpra (ou seja, não passando no teste) a função retorna

para *get_pid_stats()* e um novo processo passará a ser analisado, voltando no final da iteração a entrar novamente na função *filter()*. No caso do nenhum dos *"ifs"* se cumprir (ou seja, de nunca se encontrar um *"return"* na função), significa que o processo passou na filtragem progredindo assim para o passo onde será guardada uma linha de texto com toda a informação num *array AllSavedPids*, que posteriormente irá facilitar o trabalho de *sort* e *reverse* da tabela.

- Função *print()*

```
function print() {
    if [ ${#allSavedPids[@]} -eq 0 ]; then
        printf "No process found matching your search\n"
    fi
    if [[ $sortw -eq 1 ]]; then
        if [[ $reverse -eq 1 ]]; then
            printf '%s \n' "${allSavedPids[@]}" | sort -k7 -n | head -n $p
        else
            printf '%s \n' "${allSavedPids[@]}" | sort -r -k7 -n | head -n $p
        fi
    else
        if [[ $reverse -eq 1 ]]; then
            printf '%s \n' "${allSavedPids[@]}" | sort -k6 -n | head -n $p
        else
            printf '%s \n' "${allSavedPids[@]}" | sort -r -k6 -n | head -n $p
        fi
    fi

}
```

Figura 5 - Função print()

A função *print()* é chamada depois da *main()* imprimir o cabeçalho. Ela tem o objetivo de mostrar todo o conteúdo do *array allSavedPids*. Nelas são impostas condicionais para selecionar a ordem que será impressa. A primeira condição e segunda condição analisam se as *flags sortw* e *reverse* estão ativas (*"$sortw -eq 1"* e *"$reverse -eq 1"*, respetivamente). Caso a variável *sortw* esteja ativa, vai ordenar por ordem decrescente de *RATEW* (comando "-k7"), caso contrário, ordena por ordem decrescente da coluna *RATER* (comando "-k6"). Se a variável *reverse* estiver ativa, a ordem da tabela passa a ser crescente, caso contrário, mantém-se como ordem decrescente. Por fim consoante o valor da *flag* -p, o número de processos é limitado de acordo com o valor de *$p*, representada pelos comandos *"head"*.

Testes de Execução

De modo a certificarmo-nos que o programa está a funcionar dentro dos conformes fizemos uma série de testes para nos certificarmos que este não possui nenhuma falha.

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -c "d.*" 5

COMM                USER          PID      READB     WRITEB      RATER      RATEW            DATE
dconf-service       alexandre    2294          0          0          0          0     Dec 1 17:38
dbus-daemon         alexandre    2214          0          0          0          0     Dec 1 17:38
```

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -c "c.t" 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW            DATE
cat                 alexandre    2593          0          0          0          0     Dec 1 17:38
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -c ".*t" 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW            DATE
nm-applet           alexandre    2442         32         56         16         28     Dec 1 17:38
csd-automount       alexandre    2267          0          0          0          0     Dec 1 17:38
cat                 alexandre    2593          0          0          0          0     Dec 1 17:38
cat                 alexandre    2592          0          0          0          0     Dec 1 17:38
blueman-applet      alexandre    2388          0          0          0          0     Dec 1 17:38
```

Figura 6 - Testar filtragem pelo nome

Nesta execução fomos observar os resultados quando diferentes expressões regulares são introduzidas e como mostram as imagens, todas as execuções apresentam os resultados expectáveis.

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -s "Dec 1 22:20" 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW            DATE
rwstat.sh           alexandre  164306  269122548      26869  134561274      13434     Dec 1 22:46
chrome              alexandre  164270       6116      26237       3058      13118     Dec 1 22:46
chrome              alexandre  151130       6116     291855       3058     145927     Dec 1 22:41
```

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -s "Dec 1 19:05" -e "Dec 1 19:20" 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW            DATE
WebKitWebProces     alexandre   65357    6299010       9164    3149505       4582     Dec 1 19:11
whatsapp-for-li     alexandre   65261        848          0        424          0     Dec 1 19:11
WebKitNetworkPr     alexandre   65356         64        128         32         64     Dec 1 19:11
sh                  alexandre   65345          0          0          0          0     Dec 1 19:11
nautilus            alexandre   64797          0          0          0          0     Dec 1 19:09
logger              alexandre   65346          0          0          0          0     Dec 1 19:11
gvfsd-burn          alexandre   64821          0          0          0          0     Dec 1 19:09
chrome              alexandre   64737          0          0          0          0     Dec 1 19:09
```

Figura 7 - Testar filtragem por data

Como se pode ver, definindo limites na data os resultados também são os esperados, independentemente se apenas se define data mínima, máxima ou ambas.

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -m 3000 -M 3100 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW           DATE
Discord             alexandre    3076     204729       4082     102364       2041     Dec 1 17:38
chrome              alexandre    3040       1521          0        760          0     Dec 1 17:38
Discord             alexandre    3083          0          0          0          0     Dec 1 17:38
Discord             alexandre    3081          0          0          0          0     Dec 1 17:38
Discord             alexandre    3080          0          0          0          0     Dec 1 17:38
chrome              alexandre    3100          0          0          0          0     Dec 1 17:38
chrome              alexandre    3025          0          0          0          0     Dec 1 17:38
```

Figura 8 - Testar filtragem por pids

Num teste mais simples, aqui averiguamos se a gama de *pids* da tabela condizem com aquilo passado como *input*.

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -p 5 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW           DATE
rwstat.sh           alexandre  262175  266191525      42492  133095762      21246     Dec 1 23:43
Discord             alexandre    3283     346075      29249     173037      14624     Dec 1 17:38
xapp-sn-watcher     alexandre    2376     299408        608     149704        304     Dec 1 17:38
Discord             alexandre    3076     207609       2947     103804       1473     Dec 1 17:38
pulseaudio          alexandre    1950      27796      29651      13898      14825     Dec 1 17:38
```

Figura 9 - Limitar número de processos a imprimir

Neste caso fizemos apenas um pequeno teste para ver se não existia nenhum problema com o número de processos impressos no terminal.

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -u alexandre -p 3 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW           DATE
rwstat.sh           alexandre   61371  210484490      35995  105242245      17997     Dec 2 17:17
chrome              alexandre    3328     726378         24     363189         12     Dec 2 15:26
xapp-sn-watcher     alexandre    2358     299408        608     149704        304     Dec 2 15:26
```

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ sudo ./rwstat.sh -u gdm -p 4 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW           DATE
goa-identity-se     gdm          1229        816        376        408        188     Dec 1 17:38
gsd-power           gdm          1608        192        320         96        160     Dec 1 17:38
gsd-media-keys      gdm          1594        192        320         96        160     Dec 1 17:38
tracker-miner-f     gdm          1111        168        280         84        140     Dec 1 17:38
```

Figura 10 - Testar filtragem por utilizador

Aqui tivemos de recorrer ao uso do *sudo*, visto que de outro modo estaríamos limitados apenas a um *user*, testamos e os resultados saíram filtrados apenas ao *user* especificado.

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -p 5 -w 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW           DATE
spotify             alexandre  117481        220     139968        110      69984     Dec 1 22:10
rwstat.sh           alexandre  265743  265736982      42464  132868491      21232     Dec 1 23:45
pulseaudio          alexandre    1950      28457      30210      14228      15105     Dec 1 17:38
Discord             alexandre    3283     256467      27483     128233      13741     Dec 1 17:38
chrome              alexandre    2587        256      12428        128       6214     Dec 1 17:38
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -p 10 -r -w 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW           DATE
agent               alexandre    2437          0          0          0          0     Dec 1 17:38
at-spi2-registr     alexandre    2217          0          0          0          0     Dec 1 17:38
at-spi-bus-laun     alexandre    2209          0          0          0          0     Dec 1 17:38
bash                alexandre  110522          0          0          0          0     Dec 1 21:27
bash                alexandre   22792          0          0          0          0     Dec 1 18:02
blueman-applet      alexandre    2388          0          0          0          0     Dec 1 17:38
blueman-tray        alexandre    2489          0          0          0          0     Dec 1 17:38
cat                 alexandre    2592          0          0          0          0     Dec 1 17:38
cat                 alexandre    2593          0          0          0          0     Dec 1 17:38
chrome              alexandre  111905          0          0          0          0     Dec 1 21:34
```

Figura 11 - Ordenar e reverse

Como era expectável, com a utilização da *flag* -w a tabela passa a estar ordenada pelo *RATEW* (*default* é ordenação por *RATER*), por ordem decrescente caso não se use a *flag* -r, ou por ordem crescente caso se use

```
alexandre@alexandre-Lenovo:~/Documents/Universidade/SO/projectso$ ./rwstat.sh -M 200 2

COMM                USER          PID      READB     WRITEB      RATER      RATEW           DATE
No process found matching your search
```

Figura 12 - Nenhum resultado encontrado

Para quando não existe nenhum processo que encaixe nos critérios definidos uma mensagem de No *process found* é exibida no terminal.

Erros

Como o programa permite a interação externa (Com o utilizador), e os resultados na tabela dependem do *input*, é normal que o *script* esteja sujeito a erros, como por exemplo uma má chamada ao programa. Consoante isso, tentamos prever o maior número de erros, a fim de alertar e comunicar o erro cometido.

```
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh
ERROR: Has to have as least one argument (sleep time, in seconds)
```

Figura 13 - Chamar o programa sem argumentos

Para testar o caso de não ser passado nenhum argumento, executámos o *script* sem nenhum argumento e o resultado obtido foi um erro de não inserção de argumentos, sendo que tem de ter pelo menos o intervalo de tempo.

```
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh 2.4
ERROR: The last argument must be a number (sleep time, in seconds)
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh a
ERROR: The last argument must be a number (sleep time, in seconds)
```

Figura 14 - Testar como último argumento, um número não inteiro

Com o objetivo de testar a utilização de um número não inteiro no último argumento, é obrigatório a utilização de um número inteiro no *sleeptime*.

```
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh -p 1500 -p 1200 2
ERROR: -p flag already used
```

Figura 15 - Repetir o uso de flags

No intuito de testar o uso da mesma *flag*, mais do que 1 vez, o programa informa que a *flag* já foi usada. Esta verificação está presente em todas as *flags*, sendo esta um exemplo.

```
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh -s "01/12/2022 09:34" 2
ERROR: Invalid date format
```

Figura 16 - Testar flag com formato inválido de data

Quanto às *flags* que usam datas, s) e e), é preciso introduzir a data num determinado formato, apresentado no menu inicial, caso falhe será emitido um erro de formato inválido.

```
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh -u spos10 2
ERROR: User not found
```

Figura 17 - Testar flag com um usuário inexistente

No teste com os *users*, a introdução de um *user* inexistente, vai imprimir uma mensagem de erro, a avisar que o mesmo não existe no dispositivo.

```
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh -M 1698.6 3
ERROR: -M flag must be an integer
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh -m 550.7 5
ERROR: -m flag must be an integer
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh -M 550.7 5 -m 244 7
ERROR: -M flag must be an integer
```

Figura 18 - Testar flag com argumentos não inteiro

No teste de limite de gama de *pid*, a inserção de número não inteiros, vai imprimir uma mensagem de erro e encerrar o programa.

```
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh -a
./rwstat.sh: illegal option -- a
Invalid option
```

Figura 19 - Inserir uma flag não declarada

No menu apresenta-se as *flags* que se podem usar, caso não se use uma das declaradas, é emitido no terminal duas mensagens de erro.

```
vitalie@vitalie-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/SO/project1$ ./rwstat.sh -p
./rwstat.sh: option requires an argument -- p
Invalid option
```

Figura 20 - Inserir uma flag sem argumentos

Na situação de inserir uma flag sem argumento, é emitido uma mensagem a notificar a falta de argumento.

Conclusão

Durante o processo de desenvolvimento tentámos ao máximo melhorar a otimização deste script, tornando-o o mais simples e eficiente possível. Todas as funcionalidades pedidas foram implementadas com êxito, sendo possível apresentar os resultados esperados num tempo útil bastante curto.

Explicamos todo o funcionamento do programa e aplicamos várias variantes na execução e obtivemos diferentes resultados conforme pedido no enunciado, apresentando diversos exemplos nos testes realizados anteriormente e o motivo de o serem feitos. No final obtivemos um trabalho que supre as nossas expectativas e o intuito para que foi feito.

Este trabalho foi maioritariamente baseado no conteúdo lecionado nas aulas teórico-práticas e práticas e complementado com pesquisas na *internet*.

Bibliografia

https://www.cyberciti.biz/faq/category/bash-shell/

https://serverfault.com/

https://www.networkworld.com/article/2693361/unix-tip-using-bash-s-regular-expressions.html

https://www.xmodulo.com/key-value-dictionary-bash.html

https://www.javatpoint.com/bash-arrays

https://www.stackoverflow.com/

Material disponibilizado no e-learning