

# Taxas de Leitura/Escrita de processos em bash

# Índice

INTRODUÇÃO.....	2
METODOLOGIA USADA PARA SOLUCIONAR O PROBLEMA .....	3
<i>Validação de Argumentos</i> .....	3
<i>Função get_read_write_stats()</i> .....	8
<i>Função Sort_Results()</i> .....	9
<i>Print Final</i> .....	10
<i>Testes do Programa</i> .....	11
<i>Testes de erros</i> .....	13
CONCLUSÃO .....	15
BIBLIOGRAFIA .....	15

## Introdução

No âmbito da disciplina Sistemas Operativos, o presente trabalho tem como principal objetivo o “desenvolvimento de um script em bash para obter estatísticas sobre as leituras e escritas que os processos estão a efetuar. Esta ferramenta permite visualizar o número total de bytes de I/O que um processo leu/escreveu e também a taxa de leitura/escrita correspondente aos últimos segundos para uma seleção de processos.”

(excerto retirado do enunciado do Projeto)

Este relatório serve para apresentar os métodos e estruturas de dados utilizados para resolver o problema do `rwstat.sh`. Ao longo do relatório será descrito as etapas e ideias que foram utilizadas para construir as funções apresentadas e para além disso todos os tratamentos de erros e todos os testes realizados para comprovar que tudo foi devidamente testado e analisado.

## Metodologia usada para solucionar o problema

A script "rwstat.sh" está organizada num "Setup", onde se declaram as variáveis que serão utilizadas, três funções que tratam de partes diferentes da resolução do problema que serão explicadas mais á frente e o código principal.

### Setup

```
#!/bin/bash

declare -A pidStatsArray # Associative array to store the read/write operations of each process
declare -A temporaryArray # Temporary array to store the pid and the desired sort method information
declare -a sortedArray # Array to store the sorted information
declare -a sortedPIDArray # Array to store the sorted PID information
cT=0; # -c: process name
uT=0; # -u: user name
sT=0; # -s: start time limit
eT=0; # -e: end time limit
mT=0; # -m: lower pid number limit
MT=0; # -M upper pid number limit
rT=0; # -r: reverse
wT=0; # -w: sort by write value
pT=0; # -p: number of results to display
```

Fig. 1 - Declaração de Arrays e variáveis.

No setup é declarado maior parte das variáveis que serão utilizadas ao longo da execução, temos a variáveis de controlo de argumentos (cT, uT, etc ...) as quais são usadas para verificar se o argumento respetivo foi passado na command line. Os arrays declarados são de dois tipos, pidStatsArray e temporaryArray são arrays associativos usados para guardar a informação relacionada com cada pid e para auxiliar a ordenação de valores, o sortedArray e o sortedPidArray são arrays normais, o primeiro é utilizado na ordenação e vai conter nele os pids e os seus valores ordenados numa string, "pid1 valor1 pid2 valor2" dependendo dos argumentos, por fim o sortedPIDArray contem os pids na ordem que são encontrados no sortedArray e é utilizado no final para imprimir a tabela.

### Validação de Argumentos

```
format() {
    if [[ $1 =~ ^-.*$ ]]; then #if an argument has - at the start of the name exit
        echo "Error: Argument format invalid -> $1"
        exit 1
    fi
}
```

Fig. 2 – Função format( )

A função **format()** é uma função auxiliar que garante o funcionamento correto dos filtros, verificando que os OPTARGs não comece por "-" assim evitando erros que poderão surgir no **ciclos getops**, sendo a mesma criada para evitar a repetição de código.

```

while getopts "c:u:s:e:m:M:p:rw" opt; do
    case $opt in
        c)
            format $OPTARG
            cArg=$OPTARG
            cT=1
            ;;
        u)
            format $OPTARG
            uArg=${OPTARG}
            uT=1
            ;;
        s)
            format $OPTARG
            if [ $eT -eq 1 ]; then
                if [[ $OPTARG > $eArg ]]; then
                    echo "Error: Start time cannot be greater than end time"
                    exit 1
                fi
            fi

            if date -d "$OPTARG" > /dev/null 2> /dev/null; then
                sArg=$OPTARG
                sT=1
            else
                echo "Error: Invalid date. Please use the format 'Month Day HH:MM'"
                exit 1
            fi

            sArg=${OPTARG}
            sT=1
            ;;

```

```

        e)
            format $OPTARG
            if [ $sT -eq 1 ]; then
                if [[ $OPTARG < $sArg ]]; then
                    echo "Error: End time cannot be less than start time"
                    exit 1
                fi
            fi

            if date -d "$OPTARG" > /dev/null 2> /dev/null; then
                eArg=$OPTARG
                eT=1
            else
                echo "Error: Invalid date. Please use the format 'Month Day HH:MM'"
                exit 1
            fi

            eArg=${OPTARG}
            eT=1
            ;;

```

```

m)
    format $OPTARG
    if ! [[ $OPTARG =~ ^[0-9]+$ ]]; then
        echo "Error: -m argument must be a number"
        exit 1
    fi

    if [ $MT -eq 1 ]; then
        if [ $OPTARG -gt $MArg ]; then
            echo "Error: Minimum PID cannot be greater than maximum PID"
            exit 1
        fi
    fi

    mArg=${OPTARG}
    mT=1
    ;;

M)
    format $OPTARG
    if ! [[ $OPTARG =~ ^[0-9]+$ ]]; then
        echo "Error: -M argument must be a number"
        exit 1
    fi

    if [ $mT -eq 1 ]; then
        if [ $OPTARG -lt $mArg ]; then
            echo "Error: Maximum PID cannot be less than minimum PID"
            exit 1
        fi
    fi

    MArg=${OPTARG}
    MT=1
    ;;

p)
    format $OPTARG
    if ! [[ $OPTARG =~ ^[0-9]+$ ]]; then
        echo "Error: -p argument must be a number"
        exit 1
    fi

    pArg=${OPTARG}
    pT=1
    ;;

r)
    rT=1
    ;;

w)
    wT=1
    ;;

\?)
    echo "ERROR: Invalid option: -$OPTARG. Valid Arguments are -c, -u, -s, -e, -m, -M, -p, -r, -w"
    exit 1
    ;;

esac
done

```

Fig. 3 – Ciclo getopt

O Ciclo getopt é usado para a validação de todos os argumentos/opções inseridos/as pelo utilizador. O utilizador consegue inserir todas as opções válidas, presentes em "c:u:s:e:m:M:p:rw", de modo que, a sua leitura é feita de forma sequencial e a ordem como são colocadas não é estática, ou seja, a ordem como as mesmas são inseridas pode variar, exceto o argumento obrigatório de Sleep Time que tem de ser inserida sempre no fim, confirmando a utilização de algum dos filtros a respetiva variável muda de 0 para 1.

Caso seja inserida uma opção não suportada, é apresentada a mensagem "Error: Invalid option: -OPTARG", também é verificado se os argumentos verificam certas condições, as quais variam de argumento para argumento.

As datas para os argumentos -e e -s tem de ser datas válidas, isto é verificado utilizando o comando "date -d", caso ambas sejam usadas também é verificado que a data mínima não é superior á data máxima e vice-versa. Os argumentos de -m, -M e -p têm que ser primeiramente números inteiros, isto é verificado a partir da expressão regular "^[0-9]+\$" (começa com numero de 0-9 "[0-9]\$" e consequentemente verifica os próximos devido ao greedy modifier "+"), caso -m e -M sejam utilizados simultaneamente é verificado que o valor de -M não é inferior ao de -m e vice versa. Caso alguma destas verificações falhe é apresentada uma mensagem de erro apropriada que serão referidas posteriormente.

```
if [ $# -eq 0 ]; then
    echo "ERROR: No arguments. Sleep_Time mandatory"
    exit 1
fi

if ! [[ $OPTARG -eq $# ]]; then
    echo "ERROR: Sleep needs to be the last argument"
    exit 1
fi

shift $((OPTARG - 1))

if ! [[ $1 =~ ^[0-9]+$ ]]; then
    echo "ERROR: Invalid Sleep_Time: $1. Sleep_Time must be a number"
    exit 1
fi
```

Fig. 4 – Verificação do argumento obrigatório

Simultaneamente caso não seja dado o argumento obrigatório, Sleep Time, é apresentada a seguinte mensagem "Error: No arguments", caso o argumento de Sleep não esteja no final apresenta a mensagem "Error: Sleep needs to be the last argument" e finalmente caso o argumento utilizado para o sleep time não seja valido, ou seja, não é um número inteiro, é apresentada a mensagem "Error: Invalid Sleep\_Time \$1".

## Filtração de PIDs

```
pidArr=()

for pid in $(ps -eo pid | tail -n +2)
do
    add=1
    if [ $cT -eq 1 ]; then
        if ! [[ $(ps -p $pid -o comm=) =~ ^$cArg$ ]]; then
            add=0
        fi
    fi
    if [ $uT -eq 1 ]; then
        if ! [[ $(ps -p $pid -o user=) =~ ^$uArg$ ]]; then
            add=0
        fi
    fi
    if [ $sT -eq 1 ]; then
        if [[ $(ps -p $pid -o lstart= | awk '{print $2 " " $3 " " substr($4,1,length($4)-3)}') < $sArg ]]; then
            add=0
        fi
    fi
    if [ $eT -eq 1 ]; then
        if [[ $(ps -p $pid -o lstart= | awk '{print $2 " " $3 " " substr($4,1,length($4)-3)}') > $eArg ]]; then
            add=0
        fi
    fi
    if [ $mT -eq 1 ]; then
        if [[ $pid -lt $mArg ]]; then
            add=0
        fi
    fi
    if [ $MT -eq 1 ]; then
        if [[ $pid -gt $MArg ]]; then
            add=0
        fi
    fi
    if [ $add -eq 1 ]; then
        pidArr+=($pid)
    fi
done

get_read_write_stats $1
```

Fig. 5 – Ciclo for Pid

Sabendo quais filtros estão a ser utilizados numa execução do programa, podemos iterar por todos os processos e dependendo das variáveis de controlo atualizadas em cima, cada processo é adicionado ao array **pidArr** por defeito apenas se as suas características não forem compatíveis com o/os filtros o mesmo é então “recusado”.

Por exemplo se a script for corrida com o argumento opcional ‘-c “d.\*”’ todos os PIDs cujo o nome de comando não comece com d não serão adicionados ao array, a lógica de filtração para o resto dos argumentos opcionais de comparação de strings é igual, entretanto para as datas é utilizado awk para a comparação permitindo fazer uma comparação simples de maior ou menor entre as duas strings, e por ultimo os argumentos de limite mínimo e máximo de PID (e.g. m e M) são comparados de forma a ver se o PID é menor ou maior, respetivamente.



## Função get\_read\_write\_stats()

```
function get_read_write_stats() {

    local timetoSleep=$1
    for pid in "${pidArr[@]}"
    do
        pidstats=()
        if (cat /proc/$pid/io > /dev/null 2> /dev/null); then

            local cmd=$(ps -p $pid -o comm=)

            local user=$(ps -p $pid -o user=)

            local startTime=$(ps -p $pid -o lstart= | awk '{print $2 " " $3 " " substr($4,1,length($4)-3)}')

            local readStatsBeforeSleep=$(cat /proc/$pid/io | grep rchar | awk '{print $2}')

            local writeStatsBeforeSleep=$(cat /proc/$pid/io | grep wchar | awk '{print $2}')
            pidstats+=("$cmd|$pid|$user|$readStatsBeforeSleep|$writeStatsBeforeSleep|")

            pidStatsArray[$pid]=${pidstats[@]}
        fi
    done

    sleep $timetoSleep

    for pid in "${pidArr[@]}"
    do
        if (cat /proc/$pid/io > /dev/null 2> /dev/null); then
            pidstats=${pidStatsArray[$pid]}
            IFS='|' read -r -a pidstattempArr <<< "${pidStatsArray[$pid]}"
            cmd=${pidstattempArr[0]}
            pid=${pidstattempArr[1]}
            user=${pidstattempArr[2]}
            readStatsBeforeSleep=${pidstattempArr[3]}
            writeStatsBeforeSleep=${pidstattempArr[4]}

            local readStatsAfterSleep=$(cat /proc/$pid/io | grep rchar | awk '{print $2}')

            local writeStatsAfterSleep=$(cat /proc/$pid/io | grep wchar | awk '{print $2}')

            local readStats=$((readStatsAfterSleep - readStatsBeforeSleep))

            local writeStats=$((writeStatsAfterSleep - writeStatsBeforeSleep))

            local readRate=$(echo "scale=0; ($readStatsAfterSleep - $readStatsBeforeSleep) / $timetoSleep" | bc)

            local writeRate=$(echo "scale=0; ($writeStatsAfterSleep - $writeStatsBeforeSleep) / $timetoSleep" | bc)

            pidStatsArray[$pid]="$cmd|$pid|$user|$readStats|$writeStats|$readRate|$writeRate|$startTime"
        fi
    done
}
```

Fig. 6 – Função get\_read\_write\_stats()

A função **get\_read\_write\_stats()** é utilizada para obter as informações dos processos depois da filtração, se a mesma acontecer, e adicionar as informações num array associativo com os pids como keys e os values que são obtidos através do comando ps e o ficheiro IO do respetivo PID, obtemos assim as estatísticas desejadas que serão adicionadas temporariamente ao array associativo **pidStatsArray** na forma de uma string separada por barras, "|", isto é feito para depois, caso seja necessário facilitar a separação dos valores.

Após o comando sleep acabar, iteramos novamente por todos os PIDs, retirando toda a informação guardada anteriormente, que será separada utilizando o comando IFS, depois é lido o ficheiro IO de novo para se obter as estatísticas do processo após o sleep. Utilizando os dois conjuntos, é calculado o **read stats**, o **write stats**, o **write rate** e o **read rate**. Por fim é colocada a

string final que inclui a informação toda do processo com as informações guardadas na ordem que vão ser apresentadas na tabela, também separadas por barras pela razão anterior.

## Função Sort\_Results()

```
function Sort_Results() {
    local rT=$1
    local wT=$2

    # if rt = 0, wt = 0, sort by read rate
    if [ $rT -eq 0 ] && [ $wT -eq 0 ]; then
        for pid in "${pidArr[@]}"
        do
            IFS='|' read -r -a pidstattemp <<< "${pidStatsArray[$pid]}"
            local readRate=$(echo ${pidstattemp[5]})
            temporaryArray[$pid]=$readRate
        done
        sortedArray=$(for pid in "${!temporaryArray[@]}"; do echo "$pid ${temporaryArray[$pid]}"; done | sort -k2 -n -r))
    fi

    # wt = 0, rt = 1, sort by reversed read rate
    if [ $rT -eq 1 ] && [ $wT -eq 0 ]; then
        for pid in "${pidArr[@]}"
        do
            IFS='|' read -r -a pidstattemp <<< "${pidStatsArray[$pid]}"
            local readRate=$(echo ${pidstattemp[5]})
            temporaryArray[$pid]=$readRate
        done
        sortedArray=$(for pid in "${!temporaryArray[@]}"; do echo "$pid ${temporaryArray[$pid]}"; done | sort -k2 -n)
    fi

    # wt = 1, rt = 0, sort by write Stats
    if [ $rT -eq 0 ] && [ $wT -eq 1 ]; then
        for pid in "${pidArr[@]}"
        do
            IFS='|' read -r -a pidstattemp <<< "${pidStatsArray[$pid]}"
            local writeRate=$(echo ${pidstattemp[6]})
            temporaryArray[$pid]=$writeRate
        done
        sortedArray=$(for pid in "${!temporaryArray[@]}"; do echo "$pid ${temporaryArray[$pid]}"; done | sort -k2 -n -r))
    fi

    # wt = 1, rt = 1, sort by reversed write Stats
    if [ $rT -eq 1 ] && [ $wT -eq 1 ]; then
        for pid in "${pidArr[@]}"
        do
            IFS='|' read -r -a pidstattemp <<< "${pidStatsArray[$pid]}"
            local writeRate=$(echo ${pidstattemp[6]})
            temporaryArray[$pid]=$writeRate
        done
        sortedArray=$(for pid in "${!temporaryArray[@]}"; do echo "$pid ${temporaryArray[$pid]}"; done | sort -k2 -n )
    fi

    for (( i=0; i < ${#sortedArray[@]}; i=i+2 ))
    do
        sortedPIDArray+=("${sortedArray[$i]})"
    done
}
```

Fig. 7 – Sort\_Results()

Com a informação já armazenada é utilizada a função **Sort\_Results()** para ordenar a mesma, para conseguir tal efeito são utilizadas duas variáveis de controlo (**wt** e **rt**), **wt** muda para 1 se o filtro se encontrar ativo tendo por defeito valor 0, o mesmo se aplica com a variável **rt**.

Existem assim 4 tipos de ordenamento (por **read\_rate** (ordem **decrecente**), **read\_rate** invertido (ordem **crecente**), **writeB** e **writeB** invertido).

Todos utilizam o mesmo método apenas tendo como diferença a **estatística** que é usada para o ordenar e se a ordem é **crecente** ou **decrecente**.

O método consiste em percorrer os **pids** do array **pidArr** (array com todos os **pids** para usar), de seguida obter o valor da **estatística** que vai ser usada para ordenar, isso é realizado primeiramente através do comando **IFS** (porque as informações são previamente separadas por uma " | "), é lida a informação do respetivo processo e guardada a mesma separadamente no array temporário **pidstattemp**, obtendo-se assim a **estatística** desejada através do índice da mesma no array, o próximo passo é armazenar num array associativo temporário **temporaryArray** a key **pid** e o value **estatística**, estando este processo finalizado é então adicionado ao array **sortedArray** a string "pid valor" do **temporaryArray** associado a esse **pid** para todos os **pids** sendo também usado o comando **sort** com os respetivos parâmetros (**-k2** (coluna 2, que será o valor), **-n** (numeric) e **-r** (se for desejado a ordem decrecente)) dependendo da ordem desejada para ordenar o **sortedArray**.

Para terminar é executado um for loop que itera uma string com tudo o que o array **sortedArray** contem e são adicionados apenas os elementos pares (os **pids**) ao array **SortedPIDArray** onde os mesmos vão estar ordenados.

## Print Final

```
Sort_Results $rT $wT

printf "%-30s %-20s %20s %20s %20s %20s %20s %20s\n" "COMM" "USER" "PID" "READB" "WRITEB" "RATER" "RATEW" "DATE"

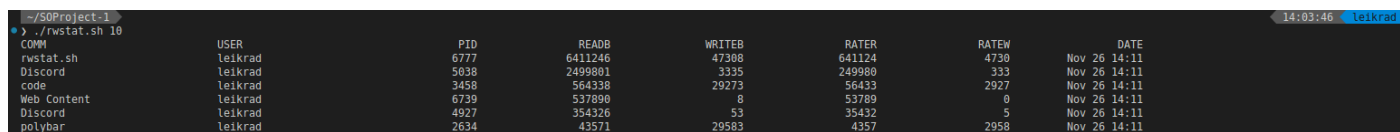
counter=0;
for i in "${sortedPIDArray[@]}"
do
    if [ $pT -eq 1 ]; then
        if [ $counter -eq $pArg ]; then
            break
        fi
    fi
    if [ -z "${pidStatsArray[$i]}" ]; then #some processes may have blank stats
        continue
    fi
    IFS='|' read -r -a pidstatprintarr <<< "${pidStatsArray[$i]}"
    printf "%-30s %-20s %20s %20s %20s %20s %20s %20s\n" "${pidstatprintarr[0]}" "${pidstatprintarr[2]}" "${pidstatprintarr[1]}" "${pidstatprintarr[3]}"
    | "${pidstatprintarr[4]}" "${pidstatprintarr[5]}" "${pidstatprintarr[6]}" "${pidstatprintarr[7]}"
    ((counter=counter+1))
done
```

Fig. 8 – Print final

Finalmente é utilizado um for loop para iterar o array **sortedPIDArray** que já tem os **pids** devidamente ordenados, para de seguida dar print dos processos com a ordem correta, para tal é realizado o comando **IFS** para a separação da string obtida do respetivo processo no array associativo **pidStatsArray**, seguidamente é lida a informação do respetivo processo e guardada a mesma separadamente no array temporário **pidstatprintarr**, por fim é dado o print das estatísticas do processo no formato desejado.

É também aqui no final que é implementado o filtro **-p** para tal acontecer simplesmente é criado um **if** a ver se o mesmo se encontra ativo e dentro deste existe outro **if** com uma variável de controlo **counter** que aumenta com cada print até alcançar o valor desejado.

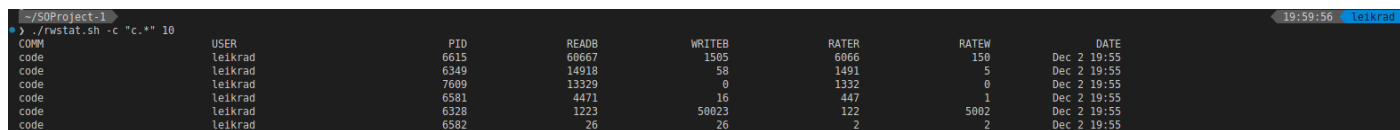
## Testes do Programa



COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
rwstat.sh	leikrad	6777	6411246	47308	641124	4730	Nov 26 14:11
Discord	leikrad	5038	2499801	3335	249980	333	Nov 26 14:11
code	leikrad	3458	564338	29273	56433	2927	Nov 26 14:11
Web Content	leikrad	6739	537890	8	53789	0	Nov 26 14:11
Discord	leikrad	4927	354326	53	35432	5	Nov 26 14:11
polybar	leikrad	2634	43571	29583	4357	2958	Nov 26 14:11

Fig. 9 - `./rwstat.sh 10`

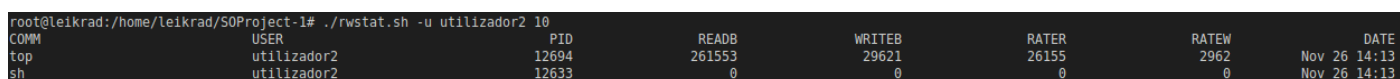
Este teste é utilizado para verificar a correta execução do programa sem qualquer filtro. O teste revela-se bem-sucedido.



COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
code	leikrad	6615	68667	1505	6866	150	Dec 2 19:55
code	leikrad	6349	14918	58	1491	5	Dec 2 19:55
code	leikrad	7609	13329	0	1332	0	Dec 2 19:55
code	leikrad	6581	4471	16	447	1	Dec 2 19:55
code	leikrad	6328	1223	50023	122	5002	Dec 2 19:55
code	leikrad	6582	26	26	2	2	Dec 2 19:55

Fig. 10 - `./rwstat.sh -c "c.*" 10`

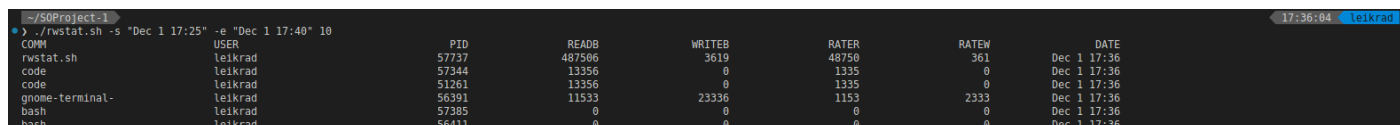
Este teste é utilizado para verificar a correta execução do programa apenas com o filtro `-c` cujo objetivo é mostrar apenas os processos que respeitam o argumento inserido. O teste revela-se bem-sucedido.



COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
top	utilizador2	12694	261553	29621	26155	2962	Nov 26 14:13
sh	utilizador2	12633	0	0	0	0	Nov 26 14:13

Fig. 11 - `./rwstat.sh -u utilizador2 10`

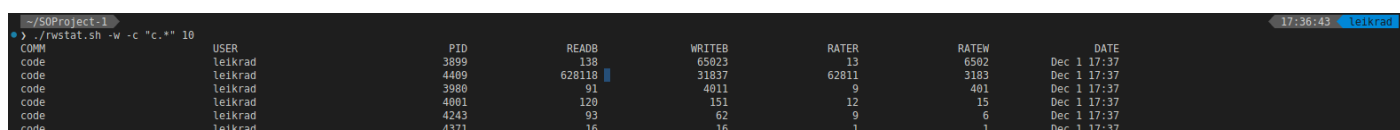
Este teste é utilizado para verificar a correta execução do programa apenas com o filtro `-u` cujo objetivo é mostrar apenas os processos com o usuário desejado, é de notar que foi utilizado o comando `sudo` pelo simples facto de o usuário "utilizador2" não ser o usuário a ser utilizado para executar o programa logo sem o `sudo` (superuser do) não seria possível aceder aos processos do mesmo. O teste revela-se bem-sucedido.



COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
rwstat.sh	leikrad	57737	487506	3619	48750	361	Dec 1 17:36
code	leikrad	57344	13356	0	1335	0	Dec 1 17:36
code	leikrad	51261	13356	0	1335	0	Dec 1 17:36
gnome-terminal	leikrad	56391	11533	23336	1153	2333	Dec 1 17:36
bash	leikrad	57385	0	0	0	0	Dec 1 17:36
bash	leikrad	56411	0	0	0	0	Dec 1 17:36

Fig. 12 - `./rwstat.sh -s "Dec 1 17:25" -e "Dec 1 17:40" 10`

Este teste é utilizado para verificar a correta execução do programa com o filtro `-s` e o filtro `-e` cujo objetivos são respetivamente mostrar os processos criados depois de "\*" (sendo \* uma data) e mostrar os processos criados antes de "\*" (sendo \* uma data) os mesmos funcionam corretamente separadamente, sendo que neste teste estão a ser utilizados simultaneamente logo nesta situação obtemos os processos que satisfazem ambas as condições ao mesmo tempo, ou seja os processos que começam depois de `-s "*"` e antes de `-e "*"`. O teste revela-se bem-sucedido.



COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
code	leikrad	3899	138	65023	13	6502	Dec 1 17:37
code	leikrad	4409	628118	31837	62811	3183	Dec 1 17:37
code	leikrad	3988	91	4911	9	491	Dec 1 17:37
code	leikrad	4901	120	151	12	15	Dec 1 17:37
code	leikrad	4243	93	62	9	6	Dec 1 17:37
code	leikrad	4371	16	16	1	1	Dec 1 17:37

Fig. 13 - `./rwstat.sh -w -c "c.*" 10`

Este teste é utilizado para verificar a correta execução do programa com o filtro `-c` e o filtro `-w`, o uso do filtro `-c` encontra-se explicado previamente (Fig. 10) logo não vai ser repetida a sua explicação, sobre o filtro `-w` é de notar que o mesmo tem como objetivo reordenar os processos para os mesmos serem apresentados por ordem decrescente de **RateW**. O teste revela-se bem-sucedido.

COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
code	leikrad	4409	612926	31466	61292	3146	Dec 1 17:47
code	leikrad	51261	13365	0	1336	0	Dec 1 17:47
code	leikrad	61524	13358	2	1335	0	Dec 1 17:47
code	leikrad	4243	93	62	9	6	Dec 1 17:47
code	leikrad	4371	16	16	1	1	Dec 1 17:47
code	leikrad	4431	0	0	0	0	Dec 1 17:47

Fig. 14 - `./rwstat.sh -m 4200 -c "c.*" 10`

Este teste é utilizado para verificar a correta execução do programa com o filtro -c e o filtro -m, o uso do filtro -c encontra-se explicado previamente (Fig. 10) logo não vai ser repetida a sua explicação, sendo o objetivo do filtro -m mostrar apenas os processos com PID superior ao desejado. O teste revela-se bem-sucedido.

COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
code	leikrad	4431	0	0	0	0	Dec 1 16:44
code	leikrad	4371	16	16	1	1	Dec 1 16:44
code	leikrad	4243	39	26	3	2	Dec 1 16:44
code	leikrad	4409	640181	37798	64018	3779	Dec 1 16:44

Fig. 15 - `./rwstat.sh -w -r -m 4200 -M 5000 -c "c.*" 10`

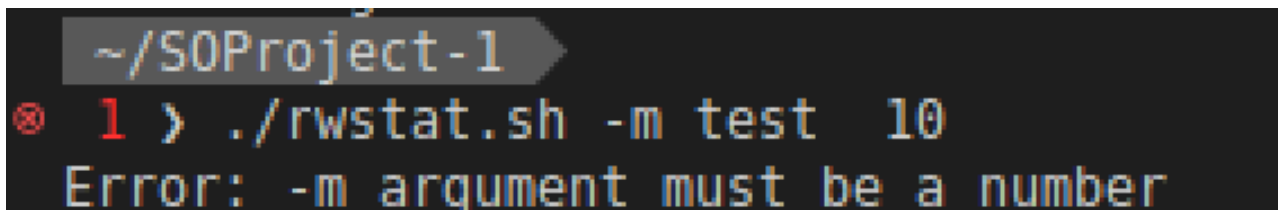
Este teste é utilizado para verificar a correta execução do programa com o filtro -c, -m, -M, -w e -r o uso do filtro -c, -m, e -w encontra-se explicado previamente (Fig. 10) (Fig.14)(Fig. 13) respetivamente, logo não vão ser repetidas as suas explicações, o filtro -r tem como objetivo inverter a ordenação dos processos, ou seja, invés de serem ordenados por ordem decrescente passam a ser por ordem crescente e o filtro -M tem como objetivo "remover" os processos com PID superior ao desejado (mostrar os com PID inferior ao desejado), é de notar que os filtros -m e -M interagem da mesma forma que os filtros -s e -e (Fig. 11) . O teste revela-se bem-sucedido.

COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
code	leikrad	4431	0	0	0	0	Dec 1 16:44

Fig. 16 - `./rwstat.sh -w -r -m 4000 -M 5000 -c "c.*" -p 1 10`

Este teste é utilizado para verificar a correta execução do programa com o filtro -c, -m, -M, -w, -r e -p o uso do filtro -c, -m, -w, -r, -M encontra-se explicado previamente (Fig. 10) (Fig.14) (Fig. 13) (Fig. 15) respetivamente, logo não vão ser repetidas as suas explicações, o filtro -p tem como objetivo limitar a quantidade de processos a tabela vai mostrar. O teste revela-se bem-sucedido.

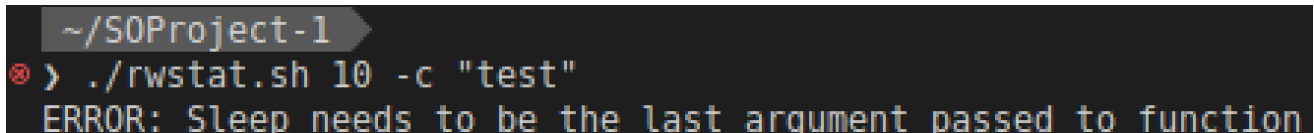
## Testes de erros



```
~/S0Project-1
❶ 1 > ./rwstat.sh -m test 10
Error: -m argument must be a number
```

Fig. 17 – Erro de validação de argumento de tipo número

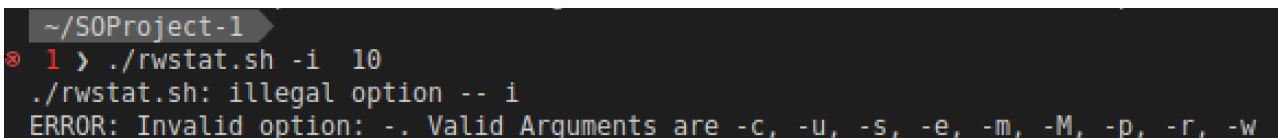
Este error\_code foi criado com o objetivo de validar os argumentos opcionais que são obrigatoriamente números inteiros não se verificando que o argumento é um número inteiro o error\_code é mostrado e a execução do programa é abortada. Os filtros que utilizam esta validação são o -m, -M e o -p.



```
~/S0Project-1
❶ > ./rwstat.sh 10 -c "test"
ERROR: Sleep needs to be the last argument passed to function
```

Fig. 18 – Erro de posição do argumento obrigatório

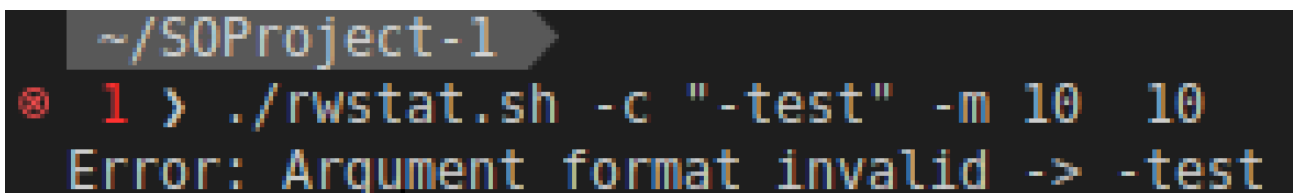
Este error\_code foi criado com o objetivo de validar a posição do sleep\_time, a qual tem de ser sempre a última nos argumentos do programa, se isso não se verificar é então apresentado o error\_code e a execução do programa é abortada.



```
~/S0Project-1
❶ 1 > ./rwstat.sh -i 10
./rwstat.sh: illegal option -- i
ERROR: Invalid option: -. Valid Arguments are -c, -u, -s, -e, -m, -M, -p, -r, -w
```

Fig. 19 – Erro de argumento opcional inválido

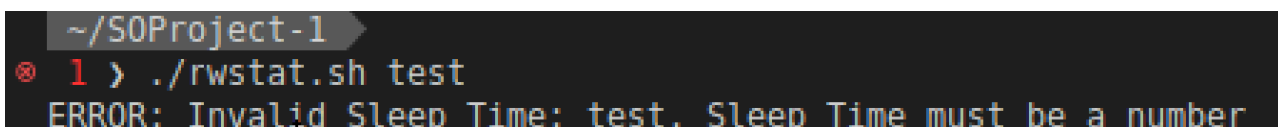
Este error\_code foi criado com o objetivo de validar os argumentos opcionais. Os argumentos opcionais possíveis são declarados previamente e sendo usado um diferente dos possíveis (um argumento opcional inválido) o error\_code é apresentado e a execução do programa é abortada.



```
~/S0Project-1
❶ 1 > ./rwstat.sh -c "-test" -m 10 10
Error: Argument format invalid -> -test
```

Fig. 20 – Erro de caracter inválido no argumento

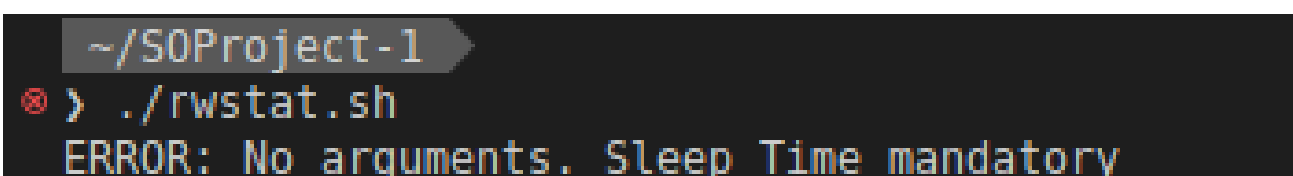
Este error\_code foi criado com o objetivo de validar os argumentos dos argumentos opcionais para os mesmos ao não terem o seu argumento avisarem invés de usarem o filtro seguinte, verificando-se essas condições o error\_code é apresentado e a execução do programa é abortada.



```
~/S0Project-1
❶ 1 > ./rwstat.sh test
ERROR: Invalid Sleep Time: test. Sleep Time must be a number
```

Fig. 21 – Erro de validação do tipo do argumento obrigatório

Este error\_code foi criado com o objetivo de validar o valor do sleep\_time, ou seja, verificar se o mesmo é um número inteiro se isso não se verificar o error\_code é apresentado e a execução do programa é abortada.



```
~/S0Project-1
❶ > ./rwstat.sh
ERROR: No arguments. Sleep_Time mandatory
```

Fig. 22 – Erro de falta de argumento obrigatório

Este error\_code foi criado com o objetivo de validar se o programa recebeu o argumento obrigatório, ou seja, se não se verificar o argumento sleep\_time não é executado o resto do programa e é apresentado o error\_code e a execução é abortada.

```
~/S0Project-1  
⊗ 1 > ./rwstat.sh -e "Dec 1 18:00" -s "Dec 1 18:20"  
Error: Start time cannot be greater than end time
```

Fig. 23 – Erro de argumentos contraditórios -e e -s 1

```
~/S0Project-1  
⊗ 1 > ./rwstat.sh -s "Dec 1 19:00" -e "Dec 1 18:00" 2  
Error: End time cannot be less than start time
```

Fig. 24 – Erro de argumentos contraditórios -e e -s2

Estes error\_codes foram criados com o objetivo de prevenir datas inválidas ao ser usado simultaneamente os argumentos -e e -s, isto é a data de começo não pode ser superior á do fim e vice versa, caso isto se verifica é apresentado o error\_code apropriado e a execução é abortada.

```
~/S0Project-1  
⊗ > ./rwstat.sh -s "test" 2  
Error: Invalid date for argument -s. Please use the format 'Month Day HH:MM'
```

Fig. 25 – Erro de validação de data

Este error\_code foi criado para a validação das datas, caso estas não seja data válidas, por exemplo "Nov 54 12:21", é apresentado o error\_code e a execução é abortada. Esta verificação está presente em ambos, argumentos -e e -s.

```
~/S0Project-1  
⊗ > ./rwstat.sh -M 10 -m 100 2  
Error: Minimum PID cannot be greater than maximum PID
```

Fig. 26 – Erro de argumentos contraditórios -m e -M 1

```
~/S0Project-1  
⊗ 1 > ./rwstat.sh -m 100 -M 10 2  
Error: Maximum PID cannot be less than minimum PID
```

Fig. 27 – Erro de argumentos contraditórios -m e -M 2

Estes error\_codes foram criados com o objetivo de prevenir argumentos contraditórios de -m e -M, caso este sejam utilizados simultaneamente, isto é, m não pode ser superior a M e vice versa. Caso isto se verifica é apresentado o error\_code apropriado e a execução é abortada.

Através destes testes conseguimos verificar a correta funcionalidade de todos os erros.

## Conclusão

O programa rwstats.sh foi realizado com êxito, ou seja, todos os resultados estão em conformidade com o pedido e apresentado nos exemplos do enunciado.

Este trabalho foi feito com base em muita pesquisa, sendo que grande parte do conhecimento foi adquirido nas aulas teóricas e práticas previamente. De modo geral, o maior desafio foi perceber todos os comandos necessários para a implementação do projeto a organização das ideias e estruturas de dados, visto que todo o código teria de ter uma estrutura e um funcionamento apropriado conforme o enunciado do projeto.

Finalmente, todos os testes realizados foram bem-sucedidos.

## Bibliografia

<https://www.artificialworlds.net/blog/2012/10/17/bash-associative-array-examples/>

<https://www.geeksforgeeks.org/ps-command-in-linux-with-examples/>

<https://pt.stackoverflow.com/>

<https://devhints.io/bash>