

# TAXAS DE LEITURA E ESCRITA DE PROCESSOS EM BASH

## SISTEMAS OPERATIVOS

Trabalho realizado por:

- Pedro Ramos
  - > N°Mec: 107348
- Daniel Madureira
  - > N°Mec: 107603



universidade  
de aveiro



deti  
universidade de aveiro



# ÍNDICE

INTRODUÇÃO .....	3
PROPOSIÇÃO DO PROBLEMA .....	3
METODOLOGIA DO CÓDIGO .....	3
ESTRUTURAS DE DADOS UTILIZADAS.....	4
FLUXO DE EXECUÇÃO .....	5
ARGUMENTOS .....	6
IDENTIFICAR OS PROCESSOS RELEVANTES .....	9
FUNÇÕES.....	11
usage.....	11
generateArray .....	12
O PROBLEMA DO ARRAY BIDIMENSIONAL.....	12
COMPARAÇÃO DE DATAS.....	13
readWriteData.....	14
O PROBLEMA .....	16
A SOLUÇÃO .....	17
clearArray.....	18
sortArray .....	19
printArray.....	20
RESULTADOS E TESTES .....	21
CONCLUSÃO.....	25
WEBGRAFIA .....	26



# INTRODUÇÃO

## PROPOSIÇÃO DO PROBLEMA

O propósito deste trabalho prático é criar um script em *bash* que consiga identificar **processos relevantes** e **mostrar as estatísticas da utilização do espaço de armazenamento** de cada uma deles, ou seja, o número de *bytes* lidos e escritos por cada processo em *s* segundos.

Neste documento iremos apresentar a nossa abordagem e descrever a metodologia do código utilizado.

## METODOLOGIA DO CÓDIGO

Para chegar aos resultados esperados, o nosso programa deve realizar as seguintes operações:

- **Analisar e validar** os argumentos obrigatórios e opcionais introduzidos;
- **Obter os Process IDs** relevantes dos processos a correr no sistema com base nos **argumentos escolhidos** e na **informação encontrada** sobre o processo;
- Gerar um array 2D com a informação necessária sobre cada processo baseado no seu ID;
- **Ordenar o array** segundo os critérios escolhidos (*taxa de leitura ou taxa de escrita*);
- **Apresentar o array** ao utilizador.

# ESTRUTURAS DE DADOS UTILIZADAS

O nosso programa utiliza essencialmente 4 estruturas de dados para realizar as operações necessárias:

→ Várias variáveis (*c, u, s, e, etc.*) que guardam os valores dos argumentos passados ao programa e processados pelo *'getops'*;

```
# Default Values, podem ser sobreescritos pelo utilizador
c=".*"      # Regex para filtrar processos pelo seu COMM
u=".*"      # Regex para filtrar processos pelo seu USER
s="0"       # Data mínima inicial dos processos
e="0"       # Data máxima inicial dos processos
m=0         # PID mínimo dos processos
M=0         # PID máximo dos processos
p=0         # Num máximo de processos a mostrar
w=0         # Ordenar a tabela pela taxa de escrita
r=0         # Ordenar a tabela inversamente
```

→ Um vetor *PIDVector* que guarda os Process IDs encontrados inicialmente;

0	363	4	1076
1	397	5	1235
2	545	6	2890
3	986	...	...

→ Um array associativo *procData* que guarda os dados processados de cada PID relevante. Este array funciona como um array de 2 dimensões na nossa implementação;

```
declare -A procData
```

Index	Comm	User	PID	...
1	Python	Root	363	...
2	Firefox	User1	545	...
3	Spotify	User2	986	...
...	...	...	...	...

→ Uma variável *maxProc* que contém o número total de PIDs no array e que é atualizada quando este muda.

# FLUXO DE EXECUÇÃO

```
# Função de geração da maior parte do array de dados
function generateDataArr() {}

# Função que obtém os dados de leitura e escrita
function readWriteData() {}

# Função que remove a maior parte dos processos baseado nos critérios escolhidos
function clearArray() {}

# Função que ordena o array
function sortArray() {}

# Função que imprime o array no formato de tabela
function printArray() {}
```

Gerar a maior parte do *array*

```
> generateDataArr procData
```

Ler e fazer a média das taxas  
*read/write*

```
> readWriteData maxProc
```

Remover as entradas do array que não interessam

```
> clearArray procData maxProc
```

Ordenar o *array*

```
> sortArray procData maxProc
```

Apresentar o *array* na forma de tabela

```
> printArray procData
```

# ARGUMENTOS

Para aumentar o potencial do código e a sua utilidade para o utilizador numa situação real, o código deve adaptar-se aos requisitos do utilizador.

Para tal é necessário utilizar uma ferramenta que possa transmitir as variáveis introduzidas pelo utilizador para o código em si. Assim, usamos a função do sistema *getopts* para obter os argumentos opcionais introduzidos pelo utilizador.

## SYNOPSIS

```
getopts optstring name [arg....]
```

## DESCRIPTION

The *getopts* utility shall retrieve options and option-arguments from a list of parameters. [...]

Each time it is invoked, the *getopts* utility shall place the value of the next option in the shell variable specified by the *name* operand and the index of the next argument to be processed in the shell variable *OPTIND*.

from: *getopts(1p)* - Linux manual Page.

A função *getopts* vai processar os seguintes argumentos opcionais:

**-h** Mostra ao utilizador as operações possíveis.

```
# Mostrar a usagem caso seja requisitado
h)
    usage
    ;;
```

**-c<regex>** Filtra os processos de acordo com o seu nome.

```
# Regex a aplicar sobre o nome do processo
c)
    c=${OPTARG}
    ;;
```

**-u<regex>** Filtra os processos de acordo com o nome de utilizador;

```
# Data mínima inicial do processo
s)
    s=${OPTARG}
    (( ${#s} != 0 )) || usage
    ;;
```

**-s<data>** Elimina os processos cuja data de início seja menor que a especificada;

```
# Regex a aplicar sobre o nome do
# utilizador do processo
u)
    u=${OPTARG}
    ;;
```

<b>-e&lt;data&gt;</b>	Elimina os processos cuja data de início seja maior que a especificada;	<pre># Data máxima inicial do processo e)     e=\${OPTARG}     (( \${#e} != 0 ))    usage     ;;</pre>
<b>-m&lt;int&gt;</b>	Elimina os processos cujo Process ID seja menor que o integer especificado;	<pre># Número mínimo do ID do processo m)     m=\${OPTARG}     ((m &gt; 0))    usage     ;;</pre>
<b>-M&lt;int&gt;</b>	Elimina os processos cujo Process ID seja maior que o integer especificado;	<pre># Número máximo do ID do processo M)     M=\${OPTARG}     ((M &gt; 0))    usage     ;;</pre>
<b>-p&lt;int&gt;</b>	Determina o número de processos a mostrar na tabela final;	<pre># Número de processos a mostrar na tabela p)     p=\${OPTARG}     ((p &gt; 0    p &lt; 1000))    usage     ;;</pre>
<b>-w</b>	Ordena a tabela final em ordem à taxa de escrita ao invés da ordem à taxa de leitura ( <i>default</i> );	<pre># Ordenar por leitura (0) ou escrita (1) w)     w=1     ;;</pre>
<b>-r</b>	Inverte a ordem da tabela final;	<pre># Ordenar pelo número maior (0) # ou pelo número menor (1) r)     r=1     ;;</pre>
<b>Outro argumento</b>	Apenas mostra as operações possíveis.	<pre># Mostrar a usagem caso seja m # intruduzidas opções inválidas *)     usage     ;;</pre>

Na nossa implementação, o último argumento é sempre o tempo entre as duas leituras para as taxas de escrita e leitura.



Após o *getopts* processar os argumentos iniciais (começados por "-") a variável "\$\*" ganha o valor das opções que não foram processadas, que neste caso é o número de segundos pretendido.

```
# Tempo entre as duas leituras das taxas read/write
updateTime="$*"
```

Após obter todos os argumentos, os valores que definem conjuntos de PIDs/Datas tem de ser testados para garantir que são válidos.

O último argumento (tempo entre leituras) também é verificado, tendo este de pertencer a {1, 2, ..., 1000}.

```
if [[ "$M" -ne "0" ]] && [[ "$m" -ne "0" ]]; then
    # Asserta que "M" é maior ou igual que "m"
    ((M >= m)) || usage
fi
if [[ "$s" -ne "0" ]] && [[ "$e" -ne "0" ]]; then
    # Asserta que "s" é menor ou igual a "e"
    (( $(date -d "${s}" +%s) <= $(date -d "${e}" +%s) )) || usage
fi

# Testar se o ultimo argumento (segundos entre leituras)
# não foi utilizado, ou é inválido
if [[ "$*" -eq "" ]] || [[ "${updateTime}" =~ '^[0-9]+$' ]] ||
    [[ "${updateTime}" -lt 1 ]] || [[ "${updateTime}" -gt 1000 ]]; then
    usage
fi
```



# IDENTIFICAR OS PROCESSOS RELEVANTES

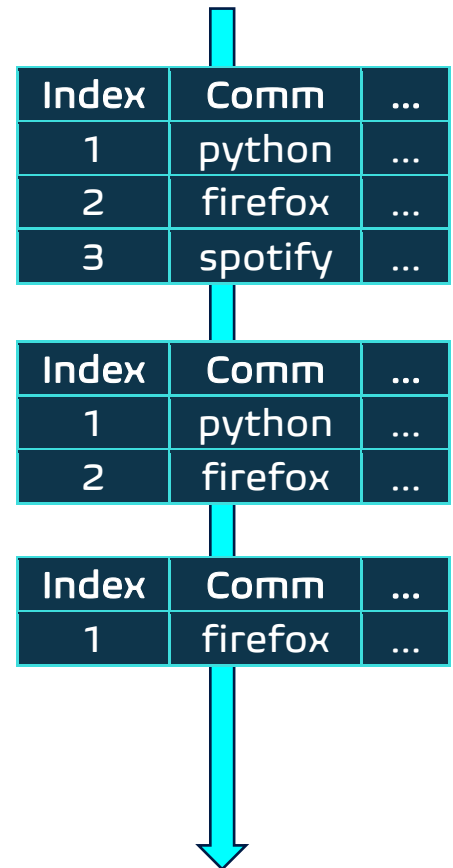
Para a execução do código essencial do projeto, devemos primeiro **identificar os processos relevantes** e obter os seus respetivos Process IDs, preenchendo assim o PIDVector.

No nosso código a eliminação de processos ocorre **gradualmente ao longo da execução**, removendo processos com base na **informação que temos sobre eles a cada momento** (ex: não precisamos de esperar até obtermos o Start Date de um processo se já sabemos que este não cumpre o regex do Comm).

Logo, a remoção de processos da nossa tabela final é feita **separadamente ao longo do percurso do código**, sacrificando simplicidade em benefício de uma maior eficiência.

Para preencher o vetor PIDVector inicial **utilizamos o comando "ps"**, mas o comando **"pgrep"** também pode ser utilizado.

Com o comando **"ps"** temos de **eliminar o cabeçalho da tabela** que este retorna com o comando **"tail"**. Com a tabela resultante podemos aplicar o comando **"awk"**, que usaremos para **verificar o nome do processo e o seu utilizador** contra o **regex especificado** através dos argumentos opcionais, sendo este um exemplo de remoção de processos ao longo do código.



```
# ps com PID, user, comm | ignorar cabeçalho |
PIDVector=$(sudo ps -Ao pid,user,comm | tail -n +2 |
  awk -v userName="$u" -v comName="$c" '$3~comName && $2~userName { print $1 }' ))
# user e comm filtrados com regex output só os PIDs selecionados
```

Deste modo conseguimos remover imediatamente PIDs que iriam **aumentar exponencialmente o tempo de execução** das funções seguintes, visto que estas iteram sobre todos os PIDs recebidos múltiplas vezes, logo quanto mais rápido "limparmos" os PIDs a tratar, melhor.

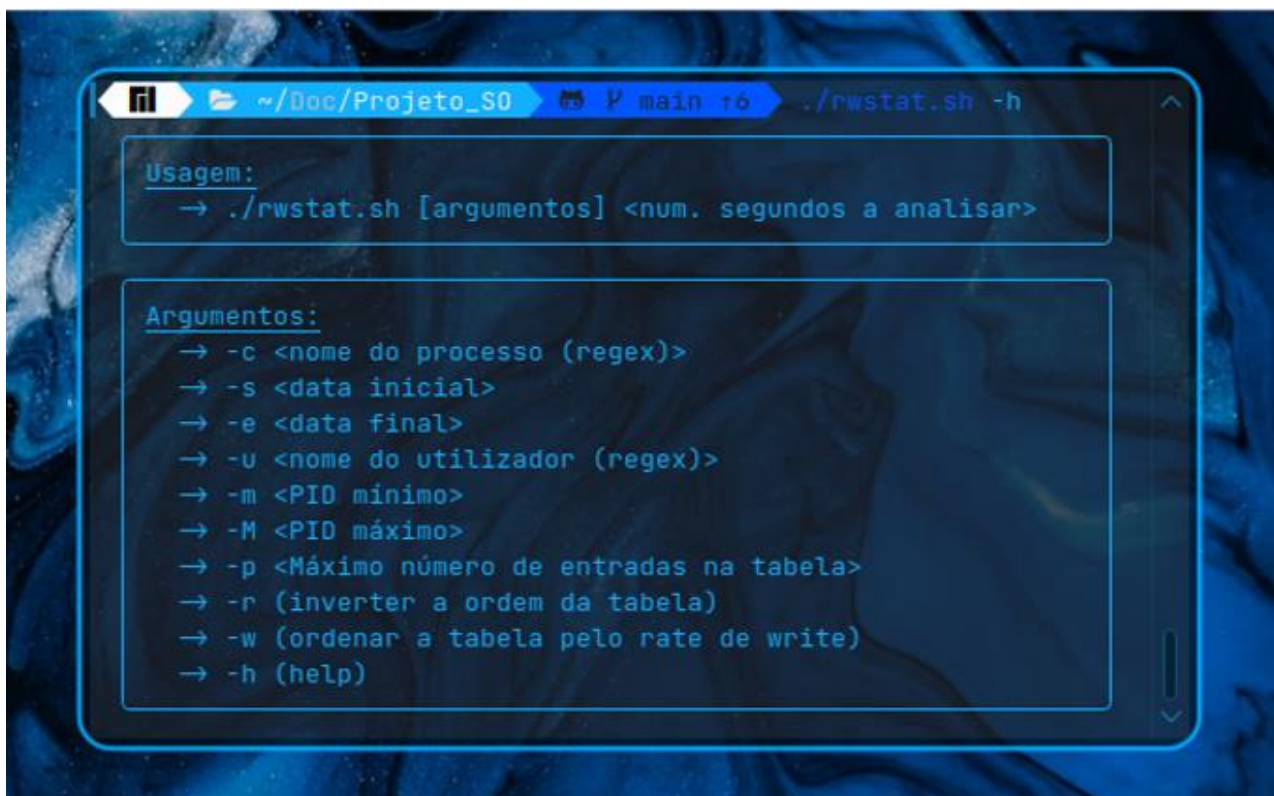
# FUNÇÕES

O nosso código está dividido em várias funções para **melhor encapsular e descrever o fluxo do programa**.

## USAGE

A função `usage` simplesmente **imprime no terminal um menu com a explicação da utilização do programa**.

Aqui o utilizador pode descobrir quais as operações possíveis que este pode introduzir, se necessitam de argumento e qual o objetivo de cada uma delas. **Esta função também acaba a execução do código**, visto que ela é chamada quando existe algum erro nos argumentos introduzidos (**exit code 1**).



```
~/Doc/Projeto_S0 main ✎ ./rwstat.sh -h

Usagem:
→ ./rwstat.sh [argumentos] <num. segundos a analisar>

Argumentos:
→ -c <nome do processo (regex)>
→ -s <data inicial>
→ -e <data final>
→ -u <nome do utilizador (regex)>
→ -m <PID mínimo>
→ -M <PID máximo>
→ -p <Máximo número de entradas na tabela>
→ -r (inverter a ordem da tabela)
→ -w (ordenar a tabela pelo rate de write)
→ -h (help)
```

# GENERATEARRAY

Esta função utiliza o array de Process IDs "PIDVector".

A função itera sobre cada PID capturado e verifica:

- Se o programa tem acesso aos ficheiros necessários para analisar o processo em questão (e se eles existem);
- Se o processo escreveu ou leu algo desde que foi chamado, pois processos sem leitura ou escrita não são relevantes;
- Se o PID pertence ao conjunto  $[m; M]$  definido pelo utilizador (isto só é verificado caso "m" ou "M" tenham sido definidos);
- Se a data de criação do processo pertence ao conjunto  $[s; e]$  definido pelo utilizador (isto só é verificado caso "s" ou "e" tenham sido definidos).
- Se um processo passa por estes critérios, a informação adicional deste é lida e colocada no array bidimensional procData. O número total de processos apanhados, "maxProc", é também incrementado.

## O PROBLEMA DO ARRAY BIDIMENSIONAL

Para guardar as informações de cada processo, a nossa implementação usa um array de bidimensional.

O problema é que, em *bash*, não existem arrays com mais do que uma dimensão, ou seja, o endereçamento de um array do tipo `arr[3]=16` é possível, mas um endereçamento do tipo `arr[3][2]=16` já não é.

Felizmente, encontrámos uma forma de passar por esta limitação com o uso de arrays associativos, que funcionam como um dicionário (de python) ou uma hash table, ao associar um valor não a um index, mas sim a uma string.

Com isto pudemos converter `arr[3][2]=16` para `arr["3,2"]=16`, que é muito fácil de realizar em *bash* visto que a esta trata todas as variáveis como se fossem apenas de um *type*, fazendo a conversão de 2 números para "int1, int2" sem problemas.



Por exemplo:

```
for i in $(seq 0 $(( numProcMax - 1 )); do
    procRead=$(sudo cat /proc/${procPid}/io | g
    procWrite=$(sudo cat /proc/${procPid}/io |

    procData[$i, 1]=${procRead#-}
    procData[$i, 2]=${procWrite#-}
done
```

Neste caso, se  $i$  fosse 3, estaríamos a endereçar o valor de `procData` ("3, 1") e ("3, 2").

É de notar que o array `procData` está dividido da seguinte forma:

Index	0	1	2	3	4	5	6	7
nProc	Comm	User	PID	ReadB	WriteB	RateR	RateW	Date

## COMPARAÇÃO DE DATAS

Para comparar datas é utilizado o comando "`date -d $varData +%s`", que transforma as datas num número int de segundos desde 1/1/1970 (*epoch time*) até à data definida na variável.

Sendo assim, é possível facilmente comparar qualquer data como se fosse um valor int (o "date" aceita vários formatos de datas diferentes).

A data do processo é dada pela data de criação da sua pasta `/proc/PID`, que podemos obter com o "`ls -ld`" e "`awk`".

```
procDate=$(ls -ld /proc/${procPid} | awk '{ print $6, $7, $8 }')
```

*# Só compara os tempos se estes forem restringidos pelo utilizador*

```
if [ "${s}" != "0" ] || [ "${e}" != "0" ];then
    dateProcess=$(date -d "${procDate}" +%s)
    dateStart=$(date -d "${s}" +%s)
    dateEnd=$(date -d "${e}" +%s)

    # Ignorar as linhas em que a data não está no conjunto especificado
    if [ $dateProcess -lt $dateStart ] || [ $dateProcess -gt $dateEnd ]; then
        continue
    fi
fi
```

Para preencher a informação necessária de cada processo precisamos ainda de obter o **nome do processo**, contido no ficheiro `/proc/PID/comm` e o **nome do utilizador** que o chamou, contido na informação de quem criou a pasta `/proc/PID`.

```
procComm=$(sudo cat /proc/${procPid}/comm)
procUser=$(stat --format '%U' /proc/${procPid})

procData[$maxProc, 0]=$(procComm)
procData[$maxProc, 1]=$(procUser)
procData[$maxProc, 2]=$(procPid)
procData[$maxProc, 7]=$(procDate[@])

# Num total de processos analisados
((maxProc=maxProc+1))
#fi
```

## READWRITEDATA

A função `readWriteData` lê os dados de leitura e escrita de cada processo e coloca os resultados no array de dados `procData`.

Após completar isto, a função **calcula quanto tempo demorou** a realizar este ciclo inicial, e dorme durante o tempo especificado pelo utilizador.

Quando a função **acorda**, volta a ler os dados de leitura/escrita de cada processo e **calcula a sua média** (taxa de leitura e escrita). Os valores dos dados totais escritos e lidos também são atualizados.

A função começa por guardar em `$t1` o tempo inicial (*epoch* com milissegundos).

Após isto a função **itera pelos processos** no array bidimensional `procData`, copiando o PID de cada processo (coluna 2 no array `procData`) e copiando os valores do número de caracteres lidos e escritos pelo processo para as colunas vazias 5 e 6 do `procData`.

Estes valores são obtidos do ficheiro `/proc/PID/io`, a que pudemos aceder e retirar a informação necessária com os comandos `"cat"`, `"grep"` e `"awk"`.

```

t1=$(date +%s%3N)

for i in $(seq 0 $(( numProcMax - 1 ))); do

    procPid=${procData[$i, 2]}

    # Ignorar os processos a que não se tem acesso (ou não existe) o ficheiro
    if ! [[ -f "/proc/${procPid}/io" ]]; then
        continue
    fi

    procRead=$(sudo cat /proc/${procPid}/io | grep "rchar" | awk '{ print $2 }')
    procWrite=$(sudo cat /proc/${procPid}/io | grep "wchar" | awk '{ print $2 }')

    procData[$i, 5]=${procRead#-}
    procData[$i, 6]=${procWrite#-}
done

```

Depois de terminar a leitura, é calculado o tempo que o ciclo "for" demorou a correr. Isto vai ser útil para resolver o grande problema desta função, mencionado mais à frente.

Após percorrer os processos pela primeira vez, a função espera o tempo especificado e volta a iterar pelos processos, lendo outra vez o número de caracteres que cada processo leu e escreveu e colocando os dados nas colunas vazias 3 e 4 do procData.

```

# Reler o número de caracteres lido/escrito por processo
for i in $(seq 0 $(( numProcMax - 1 ))); do

    procPid=${procData[$i, 2]}

    # Ignorar os processos a que não se tem acesso (ou não existe) o ficheiro
    if ! [[ -f "/proc/${procPid}/io" ]]; then
        continue
    fi

    procRead=$(sudo cat /proc/${procPid}/io | grep "rchar" | awk '{ print $2 }')
    procWrite=$(sudo cat /proc/${procPid}/io | grep "wchar" | awk '{ print $2 }')

    procData[$i, 3]=${procRead#-}
    procData[$i, 4]=${procWrite#-}
done

```

Finalmente, os processos são corridos uma terceira vez, na qual a função lê os valores guardados nas colunas 3 a 6 do procData (preenchidas com os dados de leitura e escrita) e, armazenando-as de novo no procData (escrevendo por cima dos valores das colunas 3 a 6, visto que estes já não serão precisos).

```
# Fazer as médias de lida/escrita
for i in $(seq 0 $(( numProcMax - 1 )); do

    # Diferença entre as duas leituras
    numRead="$(( procData[$i, 3] - procData[$i, 5] ))"
    numWrite="$(( procData[$i, 4] - procData[$i, 6] ))"

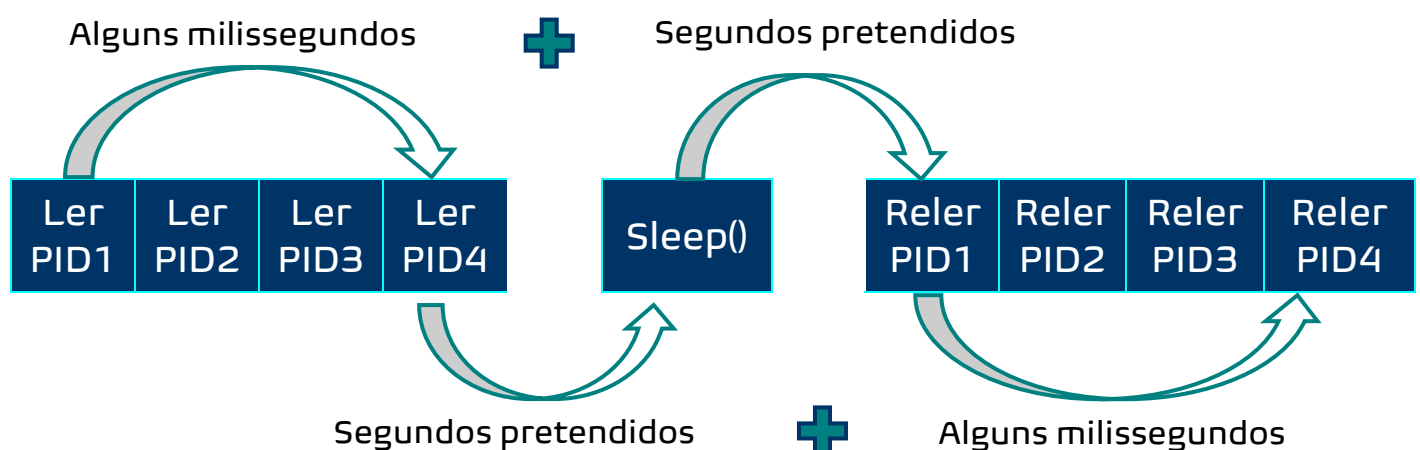
    # Taxas lida/escrita
    numMedioRead="$(( numRead / updateTime ))"
    numMedioWrite="$(( numWrite / updateTime ))"

    # Atualiza-mos a escrita e lida total só por conveniência
    procData[$i, 3]=${numRead#-}
    procData[$i, 4]=${numWrite#-}
    procData[$i, 5]=${numMedioRead#-}
    procData[$i, 6]=${numMedioWrite#-}
done

return
```

## O PROBLEMA

O tempo que a função espera entre as leituras não pode ser exatamente o tempo especificado pelo utilizador, visto que, por exemplo, o primeiro PID a ser iterado no primeiro ciclo "for" vai ter de esperar os segundos especificados mais o tempo que demora a percorrer os outros PIDs (antes de chegar ao sleep), o que levaria a que os cálculos das médias fossem incorretos. O mesmo acontece para todos os outros PIDs, sendo que o tempo a mais é exatamente o tempo de correr um dos ciclos "for".





# A SOLUÇÃO

Para resolver isto, pudemos calcular o tempo que o primeiro ciclo "for" demora a ser executado e removê-lo do tempo total a esperar antes de percorrer o próximo "for". Esta solução não é absolutamente correta visto que os dois ciclos "for" não tem tempos de execução exatamente iguais.

O terceiro ciclo "for", que calcula os valores das diferenças e das médias, só existe para fazer os dois primeiros ciclos "for" demorarem aproximadamente o mesmo tempo a serem executados, para que o resultado seja muito mais próximo do correto. Caso contrário, as ações executadas no terceiro ciclo "for" poderiam ser absorvidas para o segundo ciclo "for".

```
t2=$(date +%s%3N)
# Calcular o tempo a esperar (tempo escolhido menos tempo gasto no primeiro for)
sleepTime=$(echo | awk -v secs="$updateTime" -v t2="$t2" -v t1="$t1"
| | | | 'BEGIN {print (secs*1000 - (t2-t1)) / 1000}')
# Esperar o tempo especificado
sleep ${sleepTime#-}
```

É de notar que os cálculos do tempo de espera são feitos através do "awk" visto que a *bash* não consegue calcular nativamente valores numéricos decimais ("float" ou "double") e o "sleep" apenas aceita valores em segundos e não milissegundos.

# CLEARARRAY

A função `clearArray` é a última etapa de remoção de processos do array `procData`.

Esta função remove os processos do array em que não foi detetada nenhuma atividade de leitura ou escrita ao longo do código, ou seja, nos quais as taxas de leitura e escrita foram nulas.

```
# for i = [0, 1, ..., numProcs-1]
for i in $(seq 0 $(( numProcs - 1 ))); do

    # Ignorar as linhas que tenham rates Write/Read de 0
    if [[ "${dataArray[$i, 6]}" -eq 0 ]]
    || [[ "${dataArray[$i, 5]}" -eq 0 ]]; then
        continue
    fi

    # Colocar a linha do dataArray no tempArray,
    # agora que sabemos que esta é importante
    for ((n = 0; n < 8; n++)); do
        tempArray[$indexTemp, $n]="${dataArray[$i, $n]}"
    done
    ((indexTemp=indexTemp+1))
done
```

Como tecnicamente o nosso array `procData` é associativo, uma vez que arrays bidimensionais não existem em *bash*, não é possível remover uma linha do array sem ter de mudar a "chave" das linhas consequentes, logo a melhor forma de atualizar o array removendo as linhas pretendidas é ao criar um array novo temporário e no fim escrever esse array temporário por cima do array `procData`.

```
# Clonar o array temporário para procData
unset procData
declare -gA procData
for key in "${!tempArray[@]}"; do
    procData[$key]="${tempArray[$key]}"
done
maxProc=${indexTemp}

return
```

# SORTARRAY

A função `sortArray` utiliza o algoritmo do tipo “Bubble sort” para ordenar o array de dados de acordo com as taxas de leitura (ou escrita, se o utilizador assim definir).

Outros algoritmos seriam mais rápidos, mas visto que são muito mais complexos de traduzir para *bash*, e que o número de processos relevantes nunca é muito grande (sempre inferior a 200), a diferença de tempo não seria notável o suficiente para justificar a criação dum algoritmo muito mais complexo.

```
# Fazer bubble Sort no array
for i in $(seq 0 $(( numProcMax - 1 ))); do
  for j in $(seq $i $(( numProcMax - 1 ))); do
    # Caso o valor da linha i < j
    if [[ "${dataArray[$i, $sortN]}" -lt "${dataArray[$j, $sortN]}" ]]; then
      # Copiar a linha i para temp
      for ((n = 0; n < 8; n++)); do
        temp[$n]="${dataArray[$i, $n]}"
      done
      # Copiar a linha j para i e a linha temp para j
      for ((m = 0; m < 8; m++)); do
        dataArray[$i, $m]="${dataArray[$j, $m]}"
        dataArray[$j, $m]="${temp[$m]}"
      done
    fi
  done
done
```

# PRINTARRAY

O printArray imprime o array de dados final através de um ciclo for.

Este ciclo começa na linha 0 e **incrementa** um contador até chegar à linha final ou até mostrar o número de linhas especificado.

Caso o argumento "-p <int>" seja especificado, o ciclo acaba após a linha número p seja alcançada.

Caso o argumento "-r" seja especificado, o ciclo **começa por mostrar a linha final e decrementa** até chegar ao número pretendido de linhas ou até ao início do array (reverte o array).

```
# Função que imprime o array no formato de tabela
function printArray() {
    declare -n local dataArray="$1" # Cópia local de procData
    local start=0                   # Número da linha inicial
    local increment=1               # Incremento (positivo ou negativo)
    local end=$(( ${#dataArray[@]} / 8 )) # Número da linha final

    # Determinar o fim da tabela caso "-p <int>" seja definido
    if [[ "$p" -ne "0" ]]; then
        end="$p"
    fi

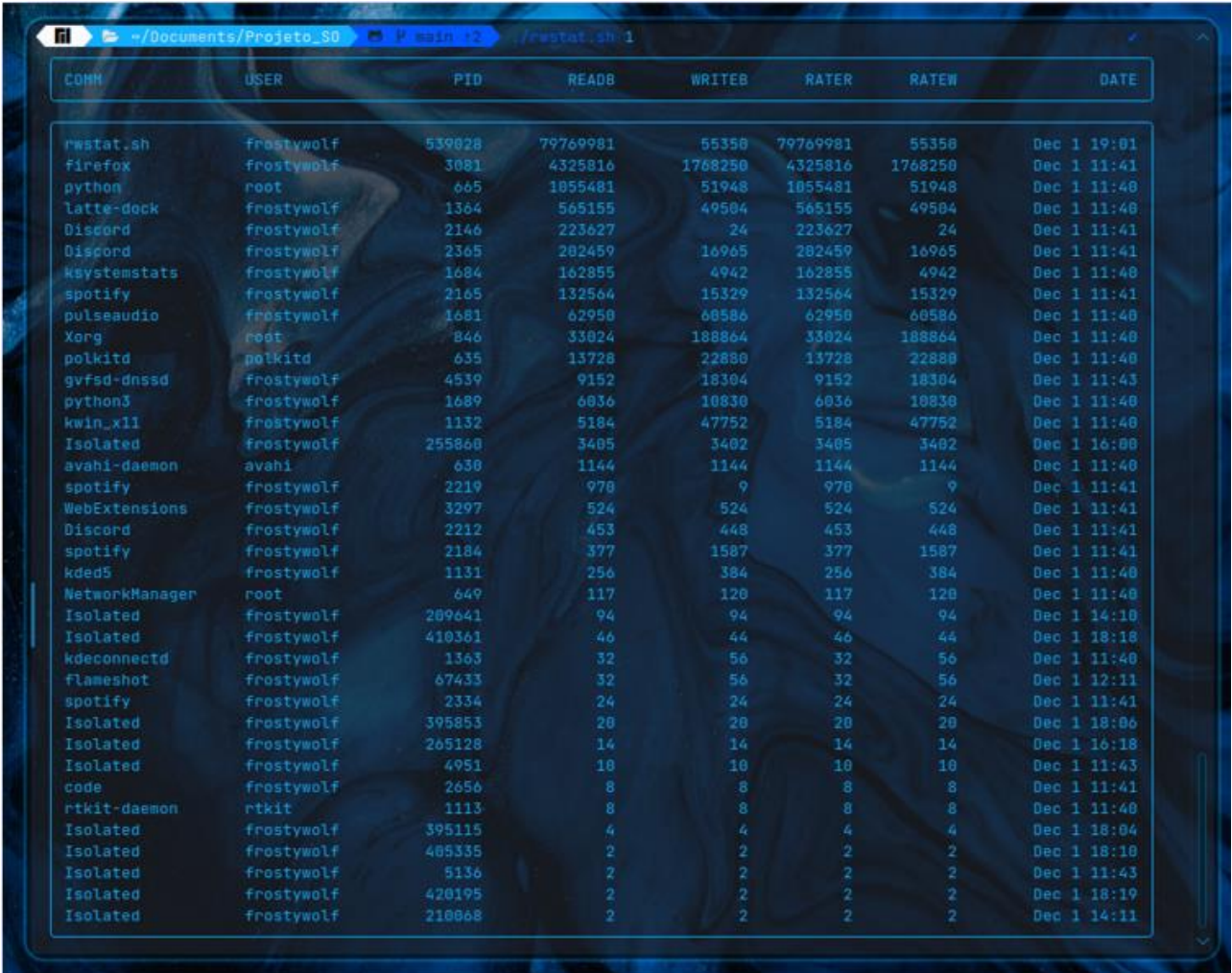
    # Onde começar/acabar a tabela, caso "-r" seja definido
    if [ "$r" -eq 1 ]; then
        start=$(( end - 1 )) # Começar no fim do array
        end="-1"             # Acabar no início do array
        increment="-1"       # Incrementar negativamente
    fi
```

```
#for j in $(seq $start $end); do
for (( j = ${start} ; j != ${end} ; j = ${j} + ${increment} )); do
    printf " | %-18s %-18s %6s %13s %13s %10s %10s %18s | \n"
        "${dataArray[$j, 0]}" "${dataArray[$j, 1]}" "${dataArray[$j, 2]}
done
```



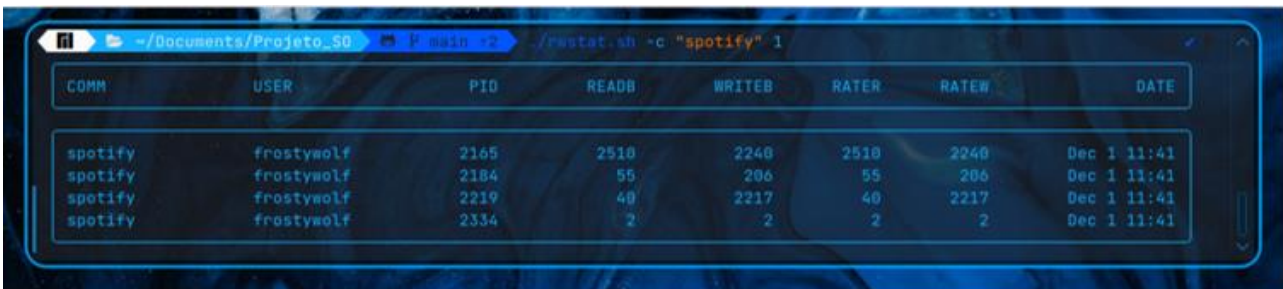
# RESULTADOS E TESTES

→ Teste com 1 segundo:



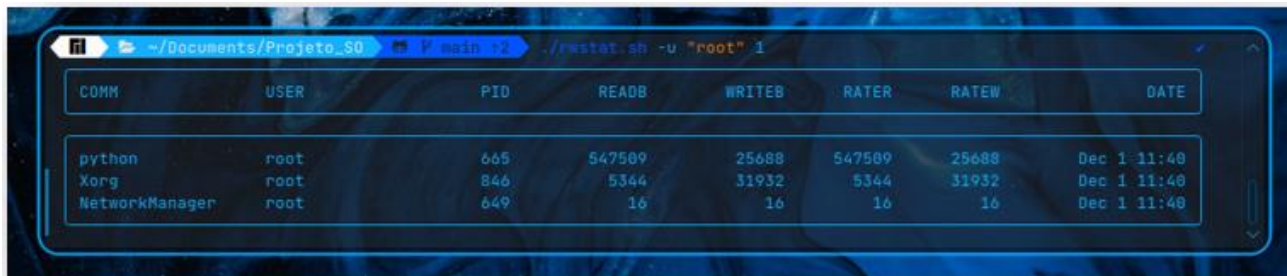
COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
/rwstat.sh	frostywolf	539028	79769981	55358	79769981	55358	Dec 1 19:01
firefox	frostywolf	3081	4325816	1768250	4325816	1768250	Dec 1 11:41
python	root	665	1855481	51948	1855481	51948	Dec 1 11:40
latte-dock	frostywolf	1364	565155	49504	565155	49504	Dec 1 11:40
Discord	frostywolf	2146	223627	24	223627	24	Dec 1 11:41
Discord	frostywolf	2365	282459	16965	282459	16965	Dec 1 11:41
systemstats	frostywolf	1684	162855	4942	162855	4942	Dec 1 11:40
spotify	frostywolf	2165	132564	15329	132564	15329	Dec 1 11:41
pulseaudio	frostywolf	1681	62950	60586	62950	60586	Dec 1 11:40
Xorg	root	846	33024	188864	33024	188864	Dec 1 11:40
polkitd	polkitd	635	13728	22880	13728	22880	Dec 1 11:40
gvfsd-dnssd	frostywolf	4539	9152	18304	9152	18304	Dec 1 11:43
python3	frostywolf	1689	6036	10830	6036	10830	Dec 1 11:40
kwin_x11	frostywolf	1132	5184	47752	5184	47752	Dec 1 11:40
Isolated	frostywolf	255860	3405	3405	3405	3405	Dec 1 16:00
avahi-daemon	avahi	630	1144	1144	1144	1144	Dec 1 11:40
spotify	frostywolf	2219	970	9	970	9	Dec 1 11:41
WebExtensions	frostywolf	3297	524	524	524	524	Dec 1 11:41
Discord	frostywolf	2212	453	448	453	448	Dec 1 11:41
spotify	frostywolf	2184	377	1587	377	1587	Dec 1 11:41
kd5	frostywolf	1131	256	384	256	384	Dec 1 11:40
NetworkManager	root	649	117	120	117	120	Dec 1 11:40
Isolated	frostywolf	289641	94	94	94	94	Dec 1 14:10
Isolated	frostywolf	410361	46	44	46	44	Dec 1 18:18
kdeconnectd	frostywolf	1363	32	56	32	56	Dec 1 11:40
Flameshot	frostywolf	67433	32	56	32	56	Dec 1 12:11
spotify	frostywolf	2334	24	24	24	24	Dec 1 11:41
Isolated	frostywolf	395853	20	20	20	20	Dec 1 18:06
Isolated	frostywolf	265128	14	14	14	14	Dec 1 16:18
Isolated	frostywolf	4951	10	10	10	10	Dec 1 11:43
code	frostywolf	2656	8	8	8	8	Dec 1 11:41
rtkit-daemon	rtkit	1113	8	8	8	8	Dec 1 11:40
Isolated	frostywolf	395115	4	4	4	4	Dec 1 18:04
Isolated	frostywolf	485335	2	2	2	2	Dec 1 18:10
Isolated	frostywolf	5136	2	2	2	2	Dec 1 11:43
Isolated	frostywolf	420195	2	2	2	2	Dec 1 18:19
Isolated	frostywolf	210068	2	2	2	2	Dec 1 14:11

→ Teste com restrição do nome do processo (-c spotify)



COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
spotify	frostywolf	2165	2510	2240	2510	2240	Dec 1 11:41
spotify	frostywolf	2184	55	206	55	206	Dec 1 11:41
spotify	frostywolf	2219	40	2217	40	2217	Dec 1 11:41
spotify	frostywolf	2334	2	2	2	2	Dec 1 11:41

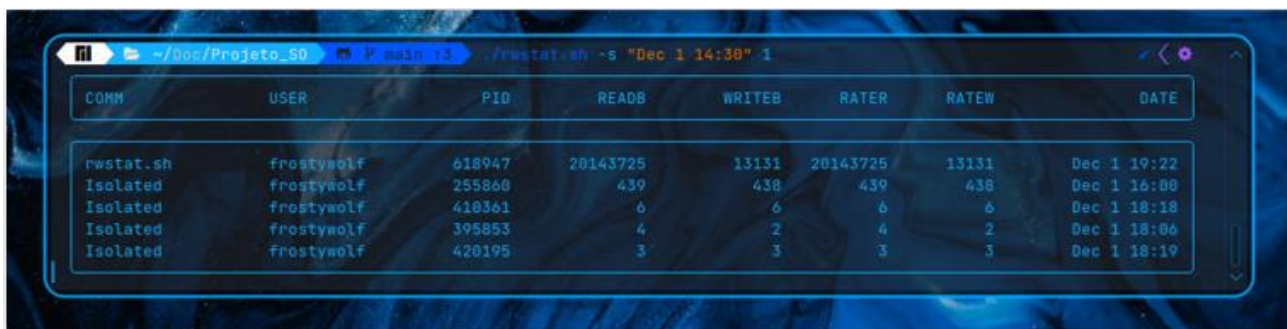
→ Teste com restrição do nome do utilizador (-u root)



Terminal screenshot showing the command `./rwstat.sh -u "root" 1` and its output table. The table lists processes for the user 'root'.

COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
python	root	665	547509	25688	547509	25688	Dec 1 11:40
Xorg	root	846	5344	31932	5344	31932	Dec 1 11:40
NetworkManager	root	649	16	16	16	16	Dec 1 11:40

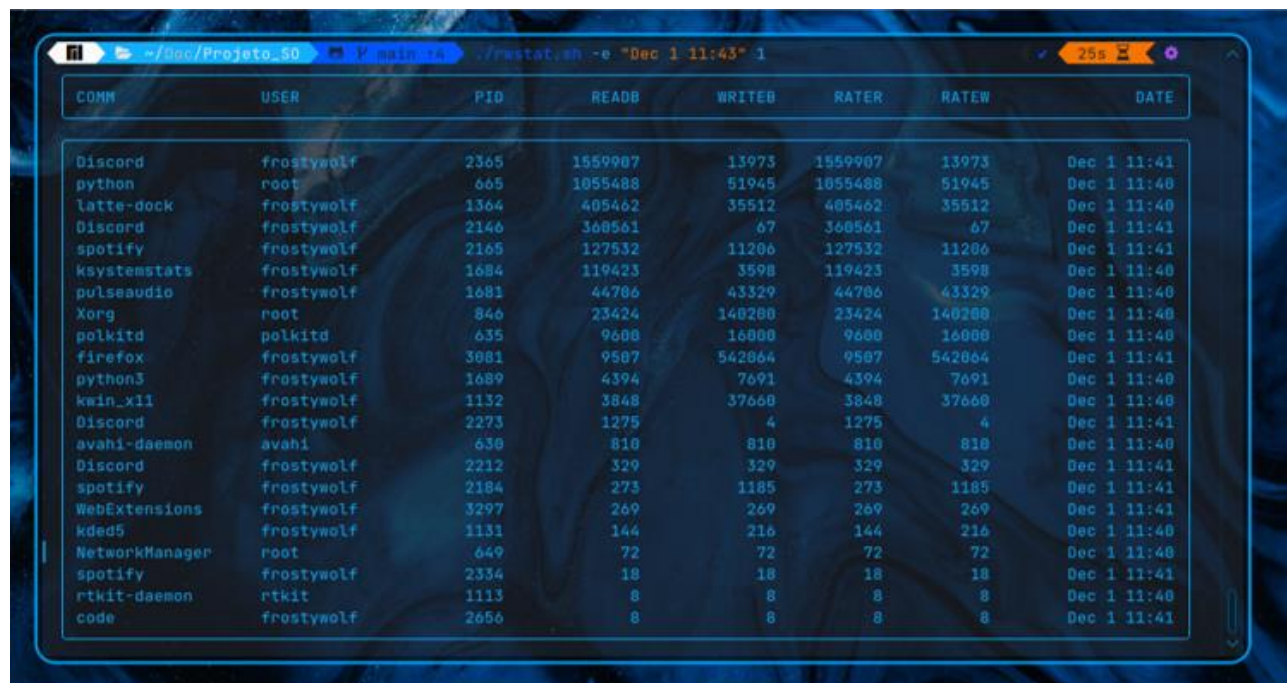
→ Teste com restrição de data de início mínimo do processo (-s "Dec 1 14:30")



Terminal screenshot showing the command `./rwstat.sh -s "Dec 1 14:30" 1` and its output table. The table lists processes starting from Dec 1 14:30.

COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
rwstat.sh	frostywolf	618947	20143725	13131	20143725	13131	Dec 1 19:22
Isolated	frostywolf	255860	439	438	439	438	Dec 1 18:08
Isolated	frostywolf	410361	6	6	6	6	Dec 1 18:18
Isolated	frostywolf	395853	4	2	4	2	Dec 1 18:06
Isolated	frostywolf	420195	3	3	3	3	Dec 1 18:19

→ Teste com restrição de data de início máximo do processo (-s "Dec 1 11:43")

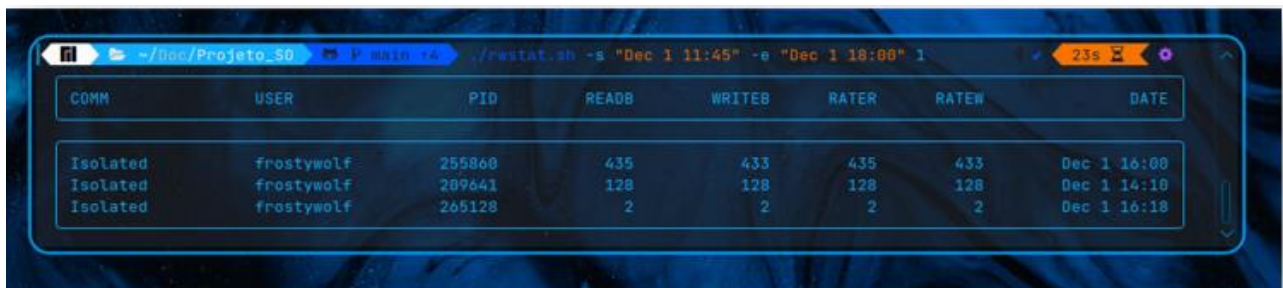


Terminal screenshot showing the command `./rwstat.sh -e "Dec 1 11:43" 1` and its output table. The table lists processes starting before Dec 1 11:43.

COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
Discord	frostywolf	2365	1559907	13973	1559907	13973	Dec 1 11:41
python	root	665	1055488	51945	1055488	51945	Dec 1 11:40
latte-dock	frostywolf	1364	405462	35512	405462	35512	Dec 1 11:40
Discord	frostywolf	2146	360561	67	360561	67	Dec 1 11:41
spotify	frostywolf	2165	127532	11206	127532	11206	Dec 1 11:41
systemstats	frostywolf	1684	119423	3598	119423	3598	Dec 1 11:40
pulseaudio	frostywolf	1681	44706	43329	44706	43329	Dec 1 11:40
Xorg	root	846	23424	140200	23424	140200	Dec 1 11:40
polkitd	polkitd	635	9600	16000	9600	16000	Dec 1 11:40
firefox	frostywolf	3081	9507	542064	9507	542064	Dec 1 11:41
python3	frostywolf	1689	4394	7691	4394	7691	Dec 1 11:40
kwin_x11	frostywolf	1132	3848	37660	3848	37660	Dec 1 11:40
Discord	frostywolf	2273	1275	4	1275	4	Dec 1 11:41
avahi-daemon	avahi	630	810	810	810	810	Dec 1 11:40
Discord	frostywolf	2212	329	329	329	329	Dec 1 11:41
spotify	frostywolf	2184	273	1185	273	1185	Dec 1 11:41
WebExtensions	frostywolf	3297	269	269	269	269	Dec 1 11:41
kded5	frostywolf	1131	144	216	144	216	Dec 1 11:40
NetworkManager	root	649	72	72	72	72	Dec 1 11:40
spotify	frostywolf	2334	18	18	18	18	Dec 1 11:41
rtkit-daemon	rtkit	1113	8	8	8	8	Dec 1 11:40
code	frostywolf	2656	8	8	8	8	Dec 1 11:41

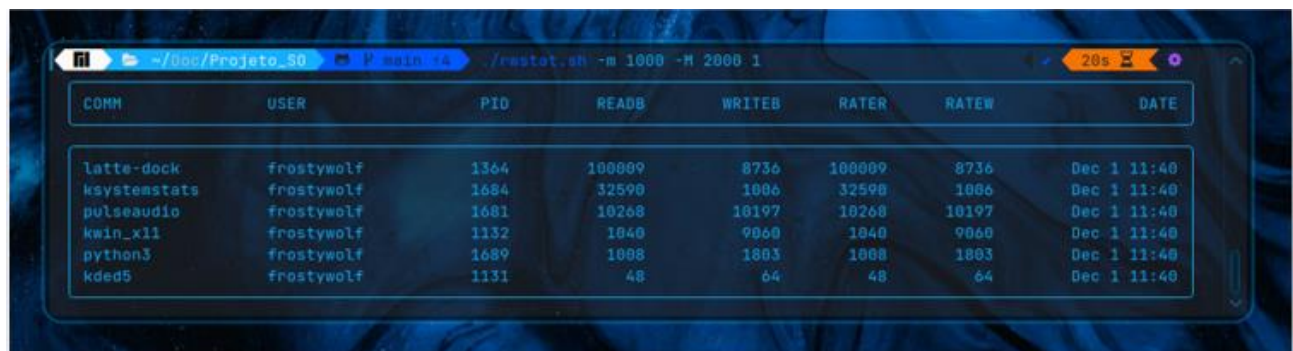


→ Teste com restrição de data de início mínimo e máximo do processo (-s "Dec 1 11:45" -e "Dec 1 18:00" 1)



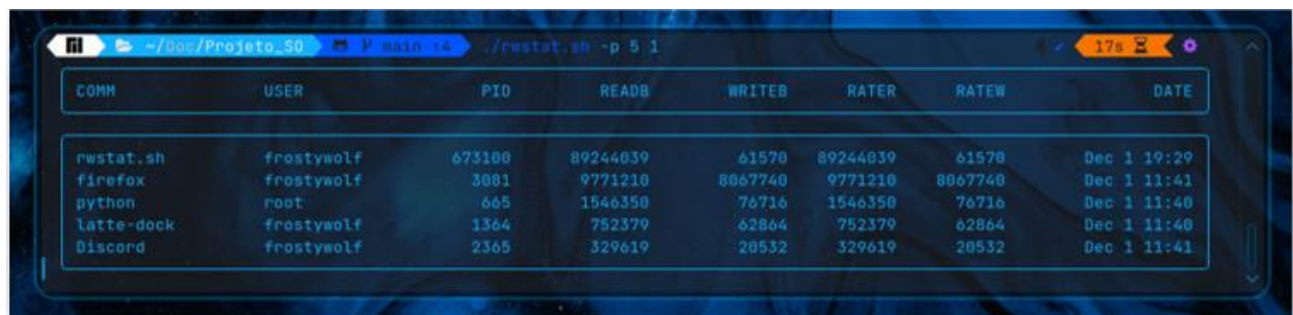
COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
Isolated	frostywolf	255860	435	433	435	433	Dec 1 16:00
Isolated	frostywolf	209641	128	128	128	128	Dec 1 14:10
Isolated	frostywolf	265128	2	2	2	2	Dec 1 16:18

→ Teste com restrição de mínimo e máximo de PID (-m 1000 -M 2000 1)



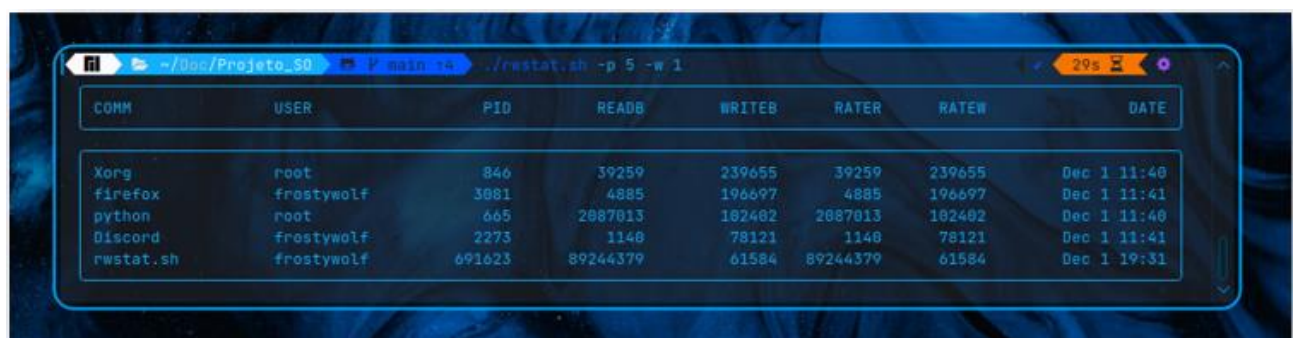
COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
latte-dock	frostywolf	1364	100009	8736	100009	8736	Dec 1 11:40
kssystemstats	frostywolf	1684	32590	1006	32590	1006	Dec 1 11:40
pulseaudio	frostywolf	1681	10268	10197	10268	10197	Dec 1 11:40
kwin_x11	frostywolf	1132	1040	9060	1040	9060	Dec 1 11:40
python3	frostywolf	1689	1008	1803	1008	1803	Dec 1 11:40
kdcd5	frostywolf	1131	48	64	48	64	Dec 1 11:40

→ Teste com máximo número de linhas (-p 5)



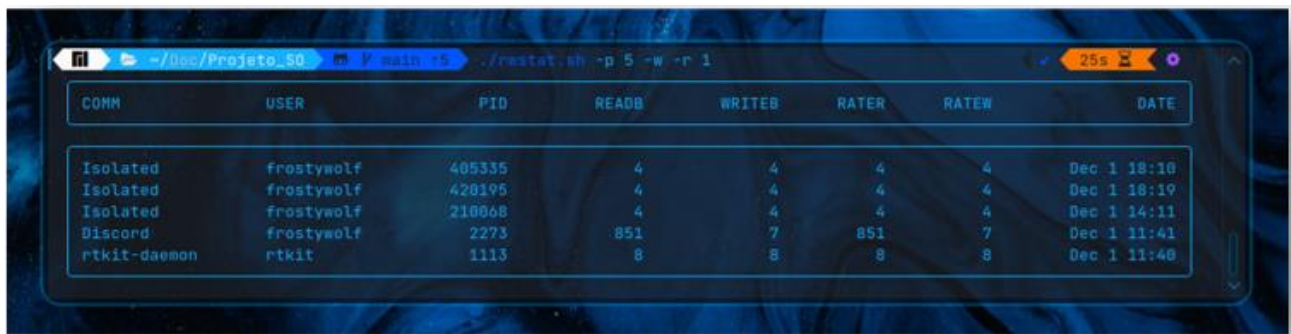
COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
rwstat.sh	frostywolf	673100	89244039	61570	89244039	61570	Dec 1 19:29
firefox	frostywolf	3081	9771210	8067740	9771210	8067740	Dec 1 11:41
python	root	665	1546350	76716	1546350	76716	Dec 1 11:40
latte-dock	frostywolf	1364	752379	62864	752379	62864	Dec 1 11:40
Discord	frostywolf	2365	329619	20532	329619	20532	Dec 1 11:41

→ Teste com máximo número de linhas e tabela invertida (-p 5 -r)



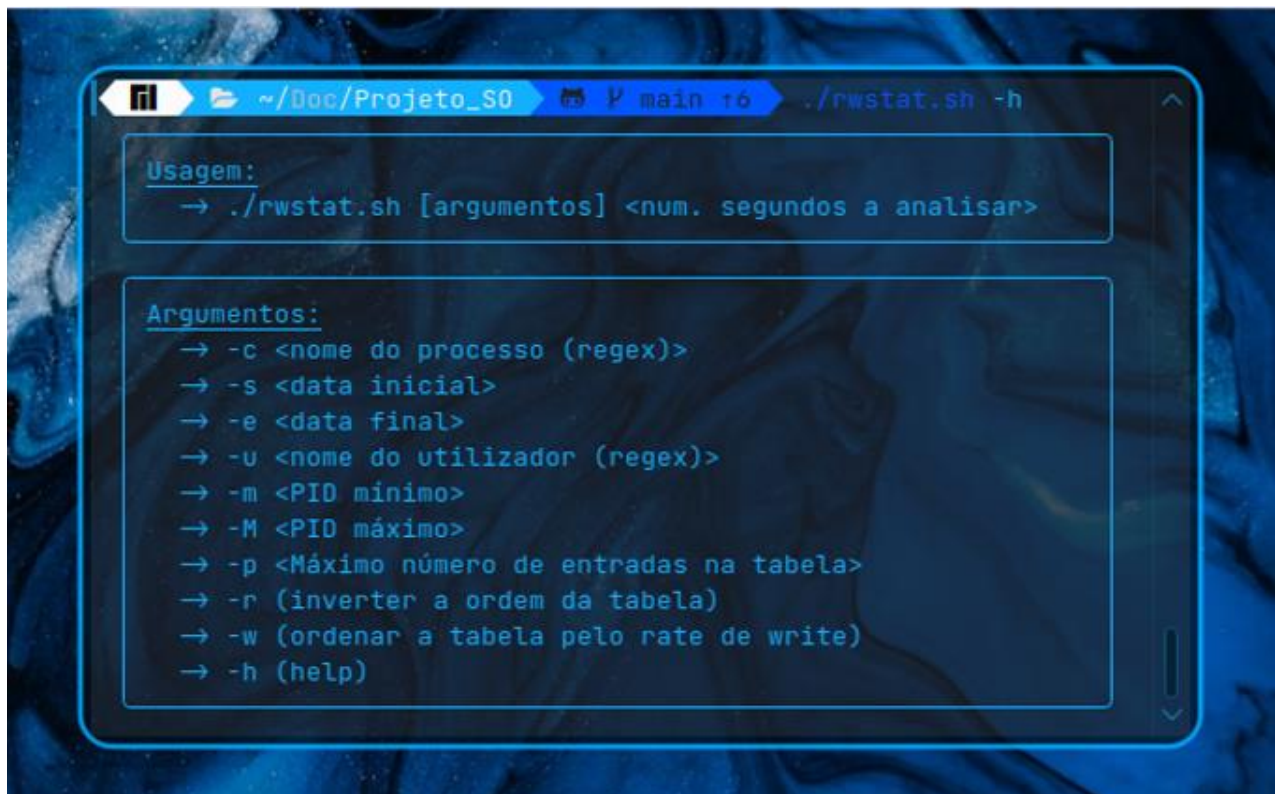
COMM	USER	PID	READB	WRITEB	RATER	RATEW	DATE
Xorg	root	846	39259	239655	39259	239655	Dec 1 11:40
firefox	frostywolf	3081	4885	196697	4885	196697	Dec 1 11:41
python	root	665	2087013	102402	2087013	102402	Dec 1 11:40
Discord	frostywolf	2273	1140	78121	1140	78121	Dec 1 11:41
rwstat.sh	frostywolf	691623	89244379	61584	89244379	61584	Dec 1 19:31

→ Teste com máximo número de linhas, tabela invertida e ordenado por taxa de leitura (-p 5 -r -w)



COMM	USER	PID	READB	WRITES	RATEB	RATEW	DATE
Isolated	frostywolf	405335	4	4	4	4	Dec 1 18:18
Isolated	frostywolf	428195	4	4	4	4	Dec 1 18:19
Isolated	frostywolf	210868	4	4	4	4	Dec 1 14:11
Discord	frostywolf	2273	851	7	851	7	Dec 1 11:41
rtkit-daemon	rtkit	1113	8	8	8	8	Dec 1 11:48

→ Teste com pedido de ajuda na utilização (-h)



```
Usagem:
→ ./rwstat.sh [argumentos] <num. segundos a analisar>

Argumentos:
→ -c <nome do processo (regex)>
→ -s <data inicial>
→ -e <data final>
→ -u <nome do utilizador (regex)>
→ -m <PID mínimo>
→ -M <PID máximo>
→ -p <Máximo número de entradas na tabela>
→ -r (inverter a ordem da tabela)
→ -w (ordenar a tabela pelo rate de write)
→ -h (help)
```





# CONCLUSÃO

Após correremos estes testes e muitos mais, em máquinas diferentes e com distribuições Linux diferentes, podemos concluir que **todas as operações foram implementadas corretamente** e que o **programa funciona como o esperado**, tendo em conta os objetivos a atingir com o projeto.

Com este projeto pudemos perceber a real potência que a linguagem *bash* tem, pois embora seja bastante simples é possível **utilizar diversas ferramentas e “work arounds”** que nos permitem contruir um projeto de grandes dimensões e facilmente portátil para inúmeros sistemas operativos diferentes, tomando partido da personalização e modularidade da *bash*.

Assim sendo, podemos concluir que este trabalho se tornou possível graças ao conhecimento e pratica desenvolvidos nas aulas práticas e teóricas, tanto de *bash* com de tratamento de processos. Valorizamos também as muitas pesquisas que fizemos na web de modo a concluirmos este projeto com sucesso.



# WEBGRAFIA

- > <https://stackoverflow.com/>
- > <https://unix.stackexchange.com/>
- > <https://askubuntu.com/>
- > <https://superuser.com/>
- > <https://linuxhint.com/>
- > <https://www.tutorialkart.com/>