Sistemas Operativos

2º Ano Licenciatura em Ciências da Computação

Grupo 20 - Trabalho Prático

SDStore: Relatório de Desenvolvimento

Alef Keuffer	Alexandre Baldé	Ivo Lima
(A91683)	(A70373)	(A90214)

21 de maio de 2022

Resumo

Neste relatório explicar-se-á a abordagem utilizada para construir o serviço de armazenamento seguro de ficheiros SDStore, utilizando a linguagem C e os conhecimentos práticos desenvolvidos nas UC de *Sistemas Operativos*.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Este relatório contém a descrição do projeto realizado pelo Grupo 20 para o Trabalho Prático de Sistemas Operativos, para o ano letivo de 2021/2022.

Estrutura do Relatório

A estrutura do relatório é a seguinte:

- No capítulo 2 faz-se uma análise do comportamento dos programas cliente e servidor desenvolvidos para a aplicação SDStore.
- No capítulo 3 explicam-se alguns aspetos mais técnicos e concretos da implementação.
- No capítulo 4, explica-se como compilar, executar e testar o projeto.
- No capítulo 5 termina-se o relatório com as conclusões e o trabalho futuro.

Capítulo 2

Arquitetura da aplicação

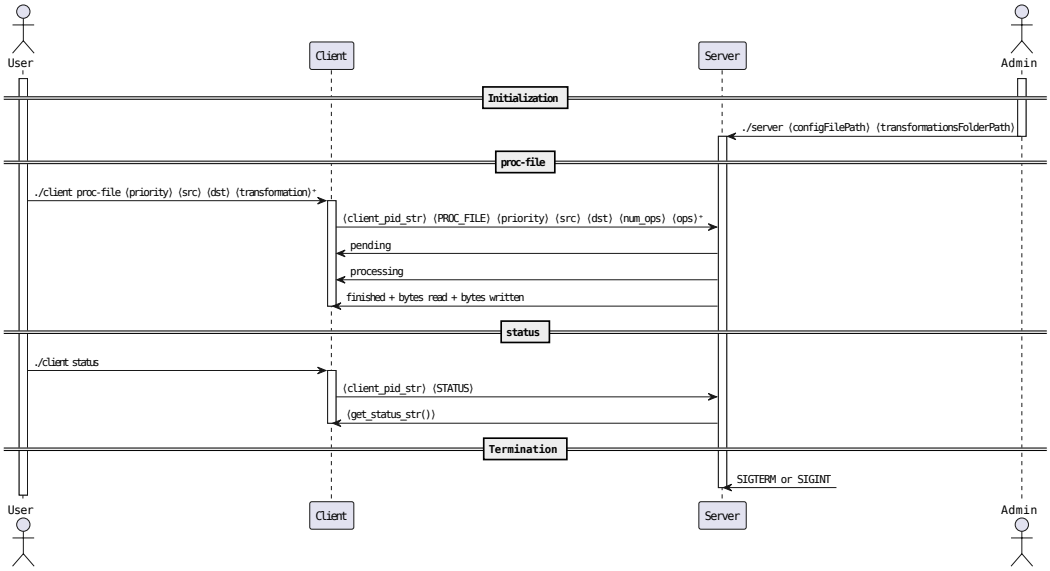

Figura 2.1: Interação entre Cliente e Servidor via terminal

2.1 Notas sobre Cliente, Servidor

- A aplicação é composta por dois executáveis, `server` e `client` (que correspondem a `sdstore` e `sdstored`, resp.).
- Servidor:
  - O servidor lê um ficheiro de configuração igual àquele descrito no enunciado, cuja informação ditará os limites de concorrência para cada transformação que pode correr.
  - O servidor também precisa receber como argumento a pasta onde se encontram os programas que efetuam as transformações. Ou seja, como pedia o enunciado: `./server etc/sdstored.conf bin/sdstore-transformations`.
  - A única forma de parar o servidor é enviar `SIGINT` ou `SIGTERM`, o que não terminará quaisquer processos em curso, ou que já tenham sido submetidos por clientes.
- Cliente:
  - Tal como pedido no enunciado, o cliente pode submeter pedidos de transformação de ficheiros, com uma prioridade associada. A prioridade pode ser qualquer não negativo.
  - O cliente também pode, através do comando `./client status`, ver quais são as transformações atualmente em curso, e quais os limites de concorrência do servidor.

2.2 Comunicação entre Servidor e Cliente

- Servidor e clientes comunicam entre si através de `mkfifo()`s.
- Existe um só FIFO para comunicação de todos os clientes para o servidor, chamado `SERVER`
- Cada cliente envia o seu PID em cada pedido que faz, porque:
- Cada cliente tem um FIFO próprio para receber informação relativa ao seu pedido, identificado pelo seu PID.
- **Cada cliente**, ao comunicar com o servidor, cria uma estrutura de dados A com a informação relativa ao pedido (ou só `status` se for esse o caso), que depois envia ao servidor.
  O protocolo de comunicação entre cliente e servidor pode ser aproximado pela seguinte gramática:

$$
\begin{aligned}
\langle task\_message \rangle &::= \langle proc\_file \rangle \mid \langle status \rangle \mid \langle finished\_task \rangle \\
\langle proc\_file \rangle &::= \langle client\_pid\_str \rangle \text{ PROC\_FILE } \langle priority \rangle \langle src \rangle \langle dst \rangle \langle num\_ops \rangle \langle ops \rangle^{+} \\
\langle status \rangle &::= \langle client\_pid\_str \rangle \text{STATUS} \\
\langle finished\_task \rangle &::= \text{FINISHED\_TASK} \langle monitor\_pid\_str \rangle \\
\langle num\_ops \rangle &::= \langle int \rangle
\end{aligned}
$$

- O servidor não executa os pedidos que recebe dos clientes.
  Cria um **Monitor** por cada uma delas, que depois fica responsável por notificar o cliente e o servidor da sua terminação. Ver imagem 2.2.

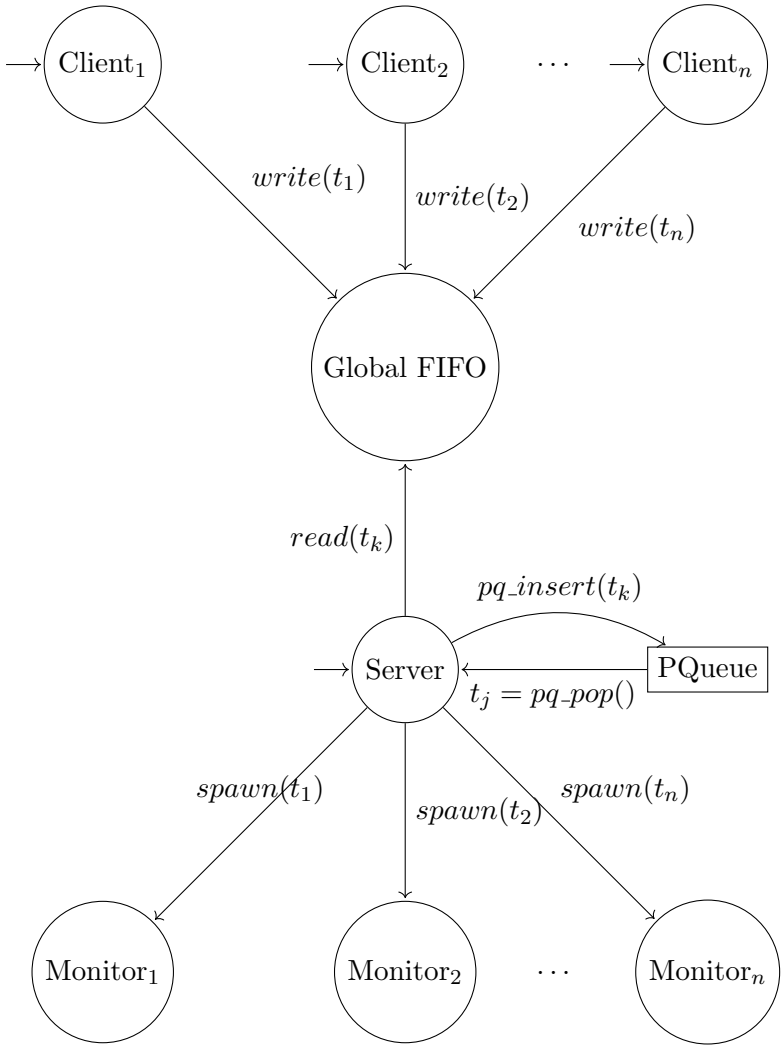

Figura 2.2: Arquitetura global da aplicação

Na figura 2.2, vê-se uma especificação informal do funcionamento global da aplicação.

- Vários clientes podem comunicar em simultâneo com o servidor
- O servidor lê os pedidos do FIFO, insere-os na sua "*priority queue*", e depois, de acordo com a sua lógica interna para controlar os limites de concorrência, cria monitores para as próximas tarefas, que ficam responsáveis pela "*pipeline*"de transformações.
- São os monitores que comunicam ao cliente a conclusão da tarefa, como referido anteriormente 2.2, assim como ao servidor, através de uma escrita no FIFO global de uma mensagem especial `FINISHED_TASK` 2.2. A figura seguinte ilustra melhor o funcionamento dos monitores.

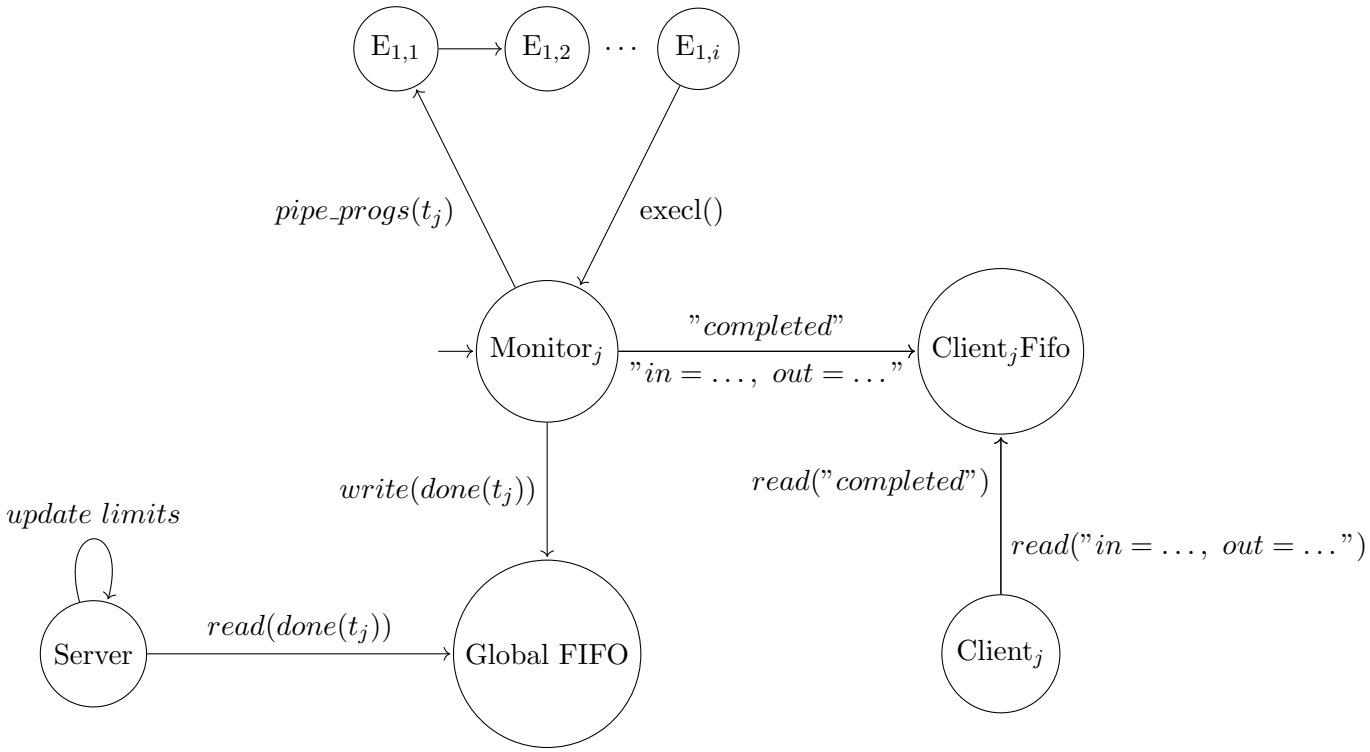

Figura 2.3: Relação entre Monitores, Clientes e Servidor

Capítulo 3

Funcionamento do servidor

- O servidor faz, essencialmente, um ciclo while(1) read(fifo);.
- O servidor guarda os pedidos em execução numa lista, e utiliza "arrays"para armazenar os limites de cada transformação, assim como o número de transformações em execução: ver anexo A.
- Durante o desenvolvimento do projeto, consideraram-se alternativas que fariam os monitores enviar sinais ao servidor aquando do término da sua execução, para ser poder fazer waitpid (...) , colher o processo monitor, e libertar os filtros em uso.

  No entanto, optou-se contra esta solução devido aos problemas de não-determinismo envolvidos no

tratamento de sinais e.g. dois processos terminam exatamente ao mesmo tempo, só haverá um desses sinais na fila do servidor, só se fará wait uma vez, etc.

- O servidor usa uma "*priority queue*"[1] para guardar os pedidos que lê do FIFO global.
- Após cada monitor terminar e escrever no FIFO global que já o fez, o servidor lê essa mensagem do FIFO e atualiza depois os limites de concorrência de cada tipo de transformação.
- O servidor obtém a próxima tarefa a executar através de pqueue_peek(). Se a próxima tarefa não puder ser executada, nunca se fazem esperas ativas: o servidor regressa à leitura do FIFO (que, recorde-se, é bloqueante), para esperar ou por novos pedidos de clientes, ou por monitores que terminam, levando à atualização dos limites de concorrência.

A imagem seguinte tem mais detalhes sobre a lógica interna do servidor.

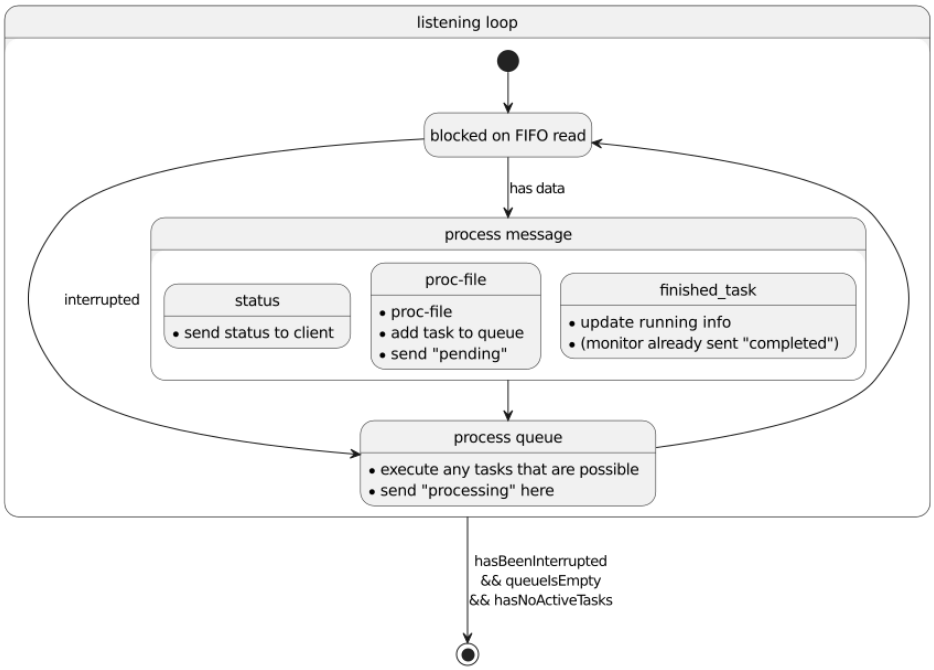

Figura 3.1: Funcionamento interno do Servidor

Note-se que na figura acima 3.1, as mensagens que o servidor processa correspondem ao protocolo definido em 2.2:

- status para clientes que querem saber o estado do servidor
- proc−file para submeter pedidos de processamento
- finished_task para monitores que terminaram o pedido de um cliente.

3.1 Funcionalidades Avançadas do Servidor

Foram pedidas 3 funcionalidades avançadas para o serviço, que se enumeram de seguida.

[1] implementação disponível em `https://github.com/vy/libpqueue`

3.1.1 Tamanho de ficheiros de entrada/saída

Para o servidor (através dos monitores que atribui a cada tarefa) reportar o número de bytes lidos do ficheiro de *input* e escritos no ficheiro de *output*, faz-se somente lseek(fd, 0, SEEK_END) para cada ficheiro.

3.1.2 Terminação graciosa de servidor

Para terminar de forma graciosa com SIGTERM/SIGQUIT, executa-se, em cada processo monitor, o código

```
signal (SIGINT, SIG_IGN);
signal (SIGTERM, SIG_IGN);
```

Caso contrário, CTRL^C também os terminará —monitores são obtidos através de fork() do servidor, herdando o seu tratamento de sinais.

No servidor, faz-se

Listing 3.1: Tratamento de sinais no servidor

```
void sig_handler (__attribute__((unused)) int signum) {
    unlink (SERVER);
    g.has_been_interrupted = 1;
}

struct sigaction sa;
sigemptyset (&sa.sa_mask);
sa.sa_handler = sig_handler;
sa.sa_flags = 0;

sigaction (SIGINT, &sa, NULL);
sigaction (SIGTERM, &sa, NULL);
```

O comando unlink(SERVER) previnirá mais clientes de fazer novos pedidos para o FIFO global.

3.1.3 Prioridades de pedidos de processamento

Como referido anteriormente, usa-se uma implementação de "*priority queues*" 2.2 em C para decidir a ordem de execução dos pedidos.
Note-se que pedidos com a mesma prioridade não têm distinção, e o processo de escolha é não-determinístico.

Capítulo 4

Testes realizados e Resultados

Para desenvolver e testar a aplicação, utilizaram-se as seguintes ferramentas:

- Cmake: `https://cmake.org/`
- Tmux: `https://en.wikipedia.org/wiki/Tmux`
- Tmuxinator: `https://github.com/tmuxinator/tmuxinator`
- Git: `https://git-scm.com/`

O Tmuxinator utiliza configurações YAML (ver exemplos em anexo) que especificam cenários de teste concretos, que depois podem ser executados através de scripts Bash que compilam o projeto A, criam ficheiros de teste e depois executam os programas A.

Para exemplificar, veja-se como obter, construir, e testar e.g. a concorrência do servidor (há outros cenários). Assume-se que todas as ferramentas mencionadas estão instaladas e disponíveis no `PATH`.

```
git clone https://github.com/Alef-Keuffer/SDStore/
cd SDStore
chmod +x ./compile.sh
./compile.sh
cd tests
chmod +x ./test.sh
./test.sh conc
```

Para ver os testes disponíveis, ver A.

Capítulo 5

Conclusão

Conclui-se desta forma a apresentação do Projeto Prático de *Sistemas Operativos* do Grupo 20 para o ano letivo 2021/2022.

5.1 Comentários

Terminar o projeto foi gratificante e transmitiu aos autores informação sobre como arquitetar, estruturar e implementar pequenos programas em C que fazem uso de "*system calls*".

No entanto, embora o Professor Paulo Almeida tenha dito, várias vezes e em tom humoroso, que o trabalho se completaria numa tarde, os autores discordam.

Embora o projeto não chegue a ter 1000 LOC em C, e algumas centenas em `YAML/Bash`, foi completamente não-trivial projetar as aplicações servidor/cliente tendo em conta as restrições impostas (comunicação através de FIFOs, pequeno conjunto de syscalls permissíveis, etc), e a novidade do paradigma concorrente, que acrescenta os seus problemas —memória partilhada, sincronização, limites de concorrência do servidor —e, cujas estratégias de abordagem não são abordados na parte teórica da disciplina, e só parcialmente na parte prática.

Acresça-se a isto as dificuldades inerentes a "*debugging*"em C, e todas as idiossincrasias da linguagem, e este projeto foi facilmente o mais difícil que os autores completaram durante o curso inteiro.

5.2 Trabalho Futuro

Este projeto foi escrito num contexto académico, e não terá qualquer uso futuro adicional.
Se tivesse, a primeira necessidade seria melhor "*error handling*". Por exemplo, o servidor termina prematuramente se receber um pedido de uma transformação que não exista.

Apêndice A

Excertos de Código Utilizado no Projeto

Listing A.1: Estrutura para pedidos de processamento de ficheiros

```
  typedef struct task_t {
    size_t pos; //private
    pqueue_pri_t pri;
    char *client_pid_str;
    char *src;
    char *dst;
    char *ops;
    int num_ops;
    pid_t monitor;
    char ops_totals[NUMBER_OF_TRANSFORMATIONS];
} task_t;
```

Listing A.2: Amostra de Estrutura global do servidor

```
struct {
  volatile sig_atomic_t has_been_interrupted;
  const int max_parallel_task;
  task_t *active_tasks[GLOBAL_MAX_PARALLEL_TASKS];
  const char *TRANSFORMATIONS_FOLDER;
  int server_fifo_rd;
  int server_fifo_wr;
  pqueue_t *queue;
  int num_active_tasks;
  int get_transformation_active_limit[NUMBER_OF_TRANSFORMATIONS];
  int get_transformation_active_count[NUMBER_OF_TRANSFORMATIONS];
} g = {
    .has_been_interrupted = 0,
    .num_active_tasks = 0,
    .get_transformation_active_count = {0},
    .active_tasks = {NULL}
};
```

Listing A.3: Script Bash de compilação

```
#!/bin/bash

# Build SDStore executables, and return to project root.
# Using subshell to avoid having to cd ..
```

```
# https://www.shellcheck.net/wiki/SC2103
(
    cd ./bin/sdstore-transformations  return;
    make clean;
    make;
)

# Build the project executables, and return to project's root.
# Subshell, same as above.

# If IDE cmake is working, this step may be skipped.
mkdir -p build
#cmake --build ./build --config Release --target all -j 18 --

rm -f tests/server
rm -f tests/client
cp build/server tests
cp build/client tests

cp ./etc/sdstored.conf tests/sdstored.conf

mkdir -p tests/bin
cp -r ./bin/sdstore-transformations/* ./tests/bin/
```

Listing A.4: Script Bash de teste

```
#!/bin/bash

# How many test files to generate, both input and output.
num_files=5

# Size of each input file. Should be 10M+ to create scenarios with interesting delay.
file_size="100M"

for ((i=1;i<=num_files;i++)); do
    rm -f filein"$i";
    rm -f fileout"$i";
done

for ((i=1;i<=num_files;i++)); do
    head -c $file_size </dev/urandom >filein"$i";
    touch fileout"$i";
done

# Argument passed to bash script dictates what test to run
if [ "$1" == "conc" ]; then
    tmuxinator start -p concurrency.yml
elif [ "$1" == "prio" ]; then
    tmuxinator start -p priority.yml;
elif [ "$1" == "status" ]; then
    tmuxinator start -p status.yml;
elif [ "$1" == "impossible" ]; then
    tmuxinator start -p impossible.yml;
elif [ "$1" == "ctrl_c" ]; then
    tmuxinator start -p ctrl_c.yml;
```

```
else
  echo "Unknown test parameter! Please rerun with an appropriate argument."
fi
```

Listing A.5: Exemplo de configurações Tmuxinator

```
name: concurrency


windows:
  - concurrency_and_limit:
      panes:
        - ./server sdstored.conf bin/
        - sleep 1; ./client proc-file 1 filein1 fileout1 bcompress bdecompress
        - sleep 1; ./client proc-file 1 filein2 fileout2 bcompress bdecompress
        - sleep 1; ./client proc-file 1 filein3 fileout3 bcompress bdecompress

name: ctrl_c

windows:
  - priority:
      panes:
        - ./server sdstored.conf bin/
        - sleep 1; ./client proc-file 1 filein1 fileout1 bcompress bcompress nop nop nop no
        - sleep 1; ./client proc-file 3 filein2 fileout2 bcompress gcompress encrypt decryp
        - sleep 2; ./client status;
echo -e CTRL^C IN SERVER;
echo -e "RUN ./client proc-file 1 filein1 fileout1 bcompress bcompress nop nop nop nop nop nop"

name: priority

# Resultado esperado: que o primeiro pedido bloqueie os outros, e que
# o pedido com prioridade mais alta, feito no fim, corra em segundo lugar.
windows:
  - priority:
      panes:
        - ./server sdstored.conf bin/
        - sleep 1; ./client proc-file 1 filein1 fileout1 nop nop nop nop nop nop bcompress
        - sleep 2; ./client proc-file 3 filein2 fileout2 nop bcompress gcompress gcompress
        - sleep 3; ./client proc-file 5 filein3 fileout3 nop bcompress gcompress gcompress

name: impossible

# Resultado esperado: que o servidor recuse o pedido por exceder os
# limites de concorrencia para o comando gcompress.
windows:
  - priority:
      panes:
        - ./server sdstored.conf bin/
        - sleep 2; ./client proc-file 1 filein1 fileout1 gcompress gcompress gcompress gcor

name: status

# Resultado esperado: que o servidor mostre a segunte informacao.
```

```
# [...] Message sent to 'SERVER'
# Number of active tasks: 2
# task [pid=...]: proc-file 1 filein1 fileout1 nop nop bcompress encrypt decrypt bdecompre
# task [pid=...]: proc-file 1 filein2 fileout2 bcompress gcompress encrypt decrypt gdecomp
# transf nop: 2/6 (running/max)
# transf bcompress: 2/4 (running/max)
# transf bdecompress: 2/4 (running/max)
# transf encrypt: 2/3 (running/max)
# transf decrypt: 2/3 (running/max)
# transf gcompress: 1/2 (running/max)
# transf gdecompress: 1/2 (running/max)
# [...] Unlinked ...
windows:
  - priority:
      panes:
        - ./server sdstored.conf bin/
        - sleep 1; ./client proc-file 1 filein1 fileout1 nop nop bcompress encrypt decrypt
        - sleep 1; ./client proc-file 1 filein2 fileout2 bcompress gcompress encrypt decryp
        - sleep 2; ./client status
```