



Universidade do Minho

Sistemas Operativos

MIEI - 2º ANO - 2º SEMESTRE

UNIVERSIDADE DO MINHO

CONTROLO E MONITORIZAÇÃO DE PROCESSOS E COMUNICAÇÃO

Tiago Gomes a69853

Braga, 15 de Junho de 2020

Índice

1	Introdução	2
1.1	Background	2
2	Conceção e Estrutura	3
2.1	Contextualização	3
2.2	Abordagem e Estrutura	3
2.3	Funcionalidade Mínima	4
2.3.1	Tempo de Inactividade	4
2.3.2	Tempo de Execução	4
2.3.3	Executar Tarefa	5
2.3.4	Listar Tarefas em Execução	5
2.3.5	Histórico de Tarefas Terminadas	6
2.3.6	Terminar Tarefa em Execução	7
2.3.7	Menu Ajuda	7
2.4	Funcionalidade Adicional	8
2.4.1	Consultar Output	8
2.4.2	Preservação do estado do servidor	8
3	Conclusão	9
3.1	Trabalho Futuro	9

1. Introdução

O presente trabalho prático insere-se no âmbito da Unidade Curricular de Sistemas Operativos, e tem como objectivo implementar um serviço de monitorização de execução e de comunicação entre processos, de forma a aplicar os conhecimentos adquiridos durante o semestre. Estes conhecimentos foram essenciais para a execução deste trabalho prático.

1.1 Background

Um *fork* consiste numa chamada de sistema utilizada para criar um novo processo (processo filho), que por sua vez corre concorrentemente com o processo que realiza a chamada inicial (processo pai). Após a criação de um novo processo filho, ambos os processos irão executar a próxima instrução a seguir à execução do *fork*, excepto se houver a verificação do valor de retorno do *fork* em que para o processo filho é 0 e para o processo pai é o id do processo criado. Um processo filho usa os mesmos recursos do processo pai (registos de cpu, descritores de ficheiros, etc..).

Para executar uma instrução, utilizamos a família *exec* (presente na biblioteca C <unistd.h>) que substituem o processo a decorrer por um novo processo, podendo ser utilizado, por exemplo, para correr um comando através de um programa em C. Existem vários membros na família *exec* (*execvp*, *execlp*, *execv*, *execl*, *execle*, *execve*) que variam fundamentalmente a nível de argumentos e funcionalidade.

A nível de redireccionamento de descritores, os descritores de ficheiros nada mais são do que *ints* associados a streams de input e output (ex: *stdin*, *stdout*, *stderr*). Os conteúdos de uma stream podem ser redireccionados para outro. Existem várias funcionalidades possíveis com esta técnica, implementada neste projecto, como por exemplo o redireccionamento da stream de output de um *exec* para a stream de input de outro.

Um *pipe* consiste num canal de comunicação entre dois processos. Um processo com o acesso a um ponta pode comunicar com outro processo que tem acesso na outra. É importante distinguir a diferença entre pipes anónimos e pipes com nome. Um pipe anónimo gere a comunicação unidireccional, sendo tipicamente utilizada para comunicar entre um processo pai e um processo filho. Relativamente aos pipes com nome, estes podem tanto gerir comunicação unidireccional como bidireccional entre dois processos não relacionados, sendo possível ter duas aplicações comunicando entre um pipe, como é o caso do cliente e servidor implementado neste projecto.

2. Conceção e Estrutura

2.1 Contextualização

O objectivo proposto é desenvolver um serviço de monitorização de execução e de comunicação entre processos. Permitindo diversas funcionalidades descritas na próxima secção. A interface com o utilizador contempla duas vertentes: interface textual interpretada(shell), e através de linha de comando.

2.2 Abordagem e Estrutura

Para responder às necessidades de implementação impostas pelo enunciado, abordei as mesmas por fases. Inicialmente implementei um cliente e servidor, que comunicam entre eles através de pipes com nome, onde a comunicação de cliente para servidor é por um único pipe. Aquando da necessidade de comunicação de servidor para cliente, é criado um pipe pelo cliente, cujo nome contém o *pid* deste, de forma a garantir que as mensagens são enviadas ao cliente que fez o pedido. De forma a garantir uma interoperabilidade semântica separamos todas as secções de uma mensagem pelo carácter "#". Seguidamente foi necessário preparar o cliente e o servidor para tratar os dados perante as funcionalidades mínimas propostas, começando pela única implementada do lado do cliente, o menu ajuda e posteriormente a escrita no pipe pelo comando executar. No lado do servidor, para uma correta execução de comandos com pipe, implementei em primeiro lugar a execução das tarefas, no qual foi necessário criar um array de pipes anónimos de tamanho variável, sendo o número de pipes proporcional ao número de comandos que o executar indicar (ex: executar "cut -f7 -d: /etc/passwd | uniq | wc -l"). Posteriormente, com a necessidade da criação de histórico das tarefas terminadas e listagem de tarefas em execução, surgiu a necessidade de criação de uma estrutura de dados auxiliar capaz de guardar o número da tarefa, tipologia do comando, estado e o *pid* (este *pid* guardado refere-se ao processo responsável pela execução dos comandos recebidos).

```
typedef struct instructions
{
    int n;
    int estado; // [0- concluido, 1-max Inativo, 2-max Exec, 3-terminada pelo utilizador, 4-execução]
    char *task;
    pid_t pid;
    struct instructions *next;
} * Instructions;
```

Figura 2.1: Estrutura de dados para gestão de tarefas.

De seguida, implementei tanto o tempo de execução como o tempo de inactividade onde foi necessário ter em conta os sinais e alarmes responsáveis por alterar o estado da tarefa e contar o tempo decorrido. Por fim, aquando da implementação do comando terminar tarefa, foi necessário ter em atenção se a função já se encontrava concluída, caso contrário, utiliza o *pid* guardado anteriormente como forma correcta de *matar* o processo.

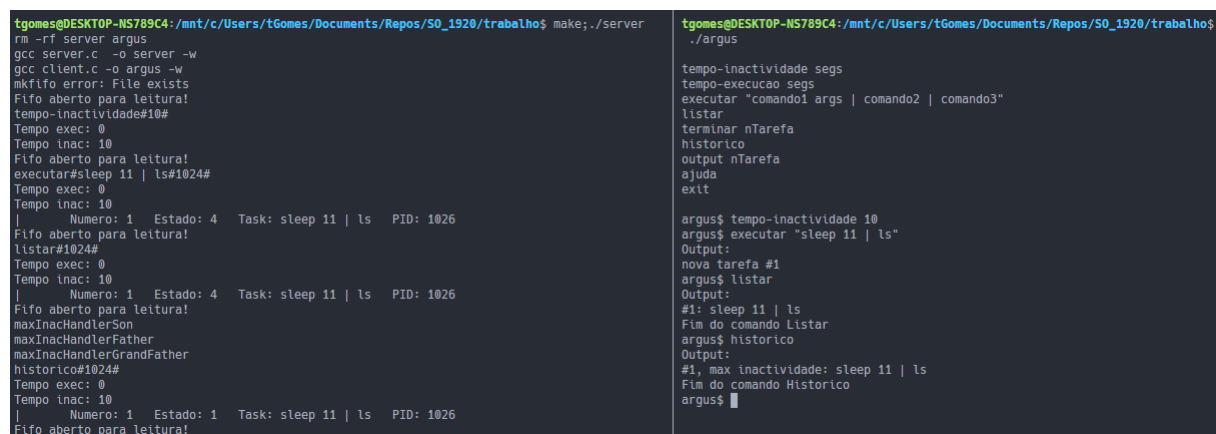
2.3 Funcionalidade Mínima

2.3.1 Tempo de Inatividade

Interpretando o tempo de inatividade como o intervalo máximo de tempo entre a execução dos comandos separados por pipes, implementei um sinal que fica atento a um alarme que é accionado se cada um dos pipes exceder um tempo especificado pelo utilizador. De seguida, este sinal executa uma função que por sua vez envia um sinal para o pai a referir que atingiu o limite de execução para este alterar o seu estado na estrutura de dados e por fim para a sua execução.

Exemplo Prático

Na figura seguinte encontra-se um exemplo do funcionamento do tempo de inatividade. Após inicialização do servidor e do cliente, aquando do input (do lado do cliente) do comando *tempo-inatividade 10*, o servidor recebe-o e define o tempo máximo de inatividade a 10 (como é evidenciado pelo `fprintf` efetuado no servidor para fins de debug tempo inac : 10). Após o comando *executar "sleep 11 / ls"* ser enviado pelo cliente, faz-se um *listar* para ver que realmente a mesma está em execução. Após um intervalo de 10 segundos passar, o servidor "mata" a tarefa por exceder o tempo máximo de inatividade (`fprintfs` referentes aos handlers para efeito de debug no lado de servidor). Para verificação, executamos o comando *histórico* e aí vemos que a tarefa #1 foi parada por exceder o máximo do tempo de inatividade.



```
tgomes@DESKTOP-NS789C4:/mnt/c/Users/tgomes/Documents/Repos/S0_1920/trabalho$ make; ./server
rm -rf server argus
gcc server.c -o server -w
gcc client.c -o argus -w
mkfifo error: File exists
Fifo aberto para leitura!
tempo-inatividade#10#
Tempo exec: 0
Tempo inac: 10
Fifo aberto para leitura!
executar#sleep 11 | ls#1024#
Tempo exec: 0
Tempo inac: 10
|      Numero: 1  Estado: 4   Task: sleep 11 | ls   PID: 1026
Fifo aberto para leitura!
listar#1024#
Tempo exec: 0
Tempo inac: 10
|      Numero: 1  Estado: 4   Task: sleep 11 | ls   PID: 1026
Fifo aberto para leitura!
maxInacHandlerSon
maxInacHandlerFather
maxInacHandlerGrandFather
historico#1024#
Tempo exec: 0
Tempo inac: 10
|      Numero: 1  Estado: 1   Task: sleep 11 | ls   PID: 1026
Fifo aberto para leitura!
```

```
tgomes@DESKTOP-NS789C4:/mnt/c/Users/tgomes/Documents/Repos/S0_1920/trabalho$ ./argus
tempo-inatividade segs
tempo-execucao segs
executar "comando1 args | comando2 | comando3"
listar
terminar nTarefa
historico
output nTarefa
ajuda
exit

argus$ tempo-inatividade 10
argus$ executar "sleep 11 | ls"
Output:
nova tarefa #1
argus$ listar
Output:
#1: sleep 11 | ls
Fim do comando Listar
argus$ historico
Output:
#1, max inatividade: sleep 11 | ls
Fim do comando Historico
argus$
```

Figura 2.2: Tempo de Inatividade (servidor / cliente).

2.3.2 Tempo de Execução

À semelhança da funcionalidade anterior, esta funcionalidade tem como objectivo sinalizar que o comando a ser executado atingiu o limite de tempo de execução estipulado pelo utilizador. Para este efeito, tal como anteriormente, é utilizado um *alarm* que é accionado quando o limite é atingido e executa uma função que envia um sinal para o pai com o objectivo de mudar o estado na estrutura de dados e para a execução do processo.

Exemplo Prático

Na figura seguinte exemplificado o funcionamento do tempo de execução. Após o servidor e o cliente serem inicializados, o cliente efetua o comando *tempo-execucao 10*, no lado do servidor é definido o tempo de execução máximo de uma tarefa para 10 (evidenciado pelo `fprintf` tempo exec: 10, para efeitos de debug). Posteriormente o cliente envia o comando *executar "ls / sleep 1"*

seguido de um comando *historico* e verificamos que a mesma foi efetuada, além de um "ls | sleep 9"(guardada em logs derivada de uma execução anterior). Quando o cliente envia o comando *executar "ls | sleep 11"*, logo de seguida o mesmo efetua um *listar* e vemos que a tarefa está em execução. Passados 10 segundos, o servidor para a tarefa (evidenciado pelos fprints referentes aos handlers para efeito de debug no lado do servidor). Finalmente quando o cliente executa o comando *histórico* onde podemos verificar que a última tarefa (referente ao ls | sleep 11) foi parada por atingir o tempo máximo de execução.

```

tfgomes@DESKTOP-NS789C4: /mnt/c/Users/tfgomes/Documents/Repos/SO_1920/trabalhos ./server
mkfifo error: File exists
Fifo aberto para leitura!
tempo-execucao#10#
Tempo exec: 10
Tempo lnac: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
executar#ls | sleep 1#1048#
Tempo exec: 10
Tempo lnac: 0
| Numero: 2 Estado: 4 Task: ls | sleep 1 PID: 1049
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
concludeInstructionsHandler
historico#1048#
Tempo exec: 10
Tempo lnac: 0
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 1049
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
executar#ls | sleep 11#1048#
Tempo exec: 10
Tempo lnac: 0
| Numero: 3 Estado: 4 Task: ls | sleep 11 PID: 1054
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 1049
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
Fifo aberto para leitura!
Fifo aberto para leitura!
Tempo exec: 10
Tempo lnac: 0
| Numero: 3 Estado: 4 Task: ls | sleep 11 PID: 1054
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 1049
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
maxExeHandlerSon
maxExeHandlerFather
historico#1048#
Tempo exec: 10
Tempo lnac: 0
| Numero: 3 Estado: 2 Task: ls | sleep 11 PID: 1054
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 1049
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!

```

```

tfgomes@DESKTOP-NS789C4: /mnt/c/Users/tfgomes/Documents/Repos/SO_1920/trabalhos ./argus
./argus
tempo-inactividade segs
tempo-execucao segs
executar "comando1 args | comando2 | comando3"
listar
terminar nTarefa
historico
output nTarefa
ajuda
exit

argus$ tempo-execucao 10
argus$ executar "ls | sleep 1"
Output:
nova tarefa #2
argus$ historico
Output:
#2, concluido: ls | sleep 1
#1, concluido: ls | sleep 9
Fin do comando Historico
argus$ executar "ls | sleep 11"
Output:
nova tarefa #3
argus$ listar
Output:
#3, ls | sleep 11
Fin do comando Listar
argus$ historico
Output:
#3, max execucao: ls | sleep 11
#2, concluido: ls | sleep 1
#1, concluido: ls | sleep 9
Fin do comando Historico
argus$

```

Figura 2.3: Tempo de Execução(servidor / cliente).

2.3.3 Executar Tarefa

Após interpretação do comando dado pelo utilizador, em que este é separado nos seus diversos pipes, são criados processos individuais para cada um destes pipes (processos filho), onde é redireccionado o output de cada instrução para o input da instrução seguinte, e no fim o output da última instrução é redireccionado para um ficheiro log (funcionalidade adicional).

2.3.4 Listar Tarefas em Execução

Através da estrutura de dados implementada é possível acedermos directamente às tarefas no qual o campo estado têm o valor de 4 (em execução), e de seguida é encaminhada a informação necessária para o cliente que fez o pedido.

Exemplo Prático

Na figura seguinte é demonstrada a funcionalidade do comando *listar*. Após o servidor e o cliente serem inicializados, o cliente executa uma série de comandos com tempo de execução alto (sleep 100, sleep 90, ls | sleep 55), de seguida, o mesmo executa o comando *listar*. Com este, verificamos que existem três tarefas em execução referentes aos últimos 3 comandos inseridos pelo cliente.

```
tgomes@DESKTOP-NS789C4:/mnt/c/Users/tgomes/Documents/Repos/SO_1920/trabalho$ ./server
mkfifo error: File exists
Fifo aberto para leitura!
executar#sleep 100#1063#
Tempo exec: 0
Tempo inac: 0
|
| Numero: 3 Estado: 4 Task: sleep 100 PID: 1064
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
executar#sleep 90#1063#
Tempo exec: 0
Tempo inac: 0
|
| Numero: 4 Estado: 4 Task: sleep 90 PID: 1067
| Numero: 3 Estado: 4 Task: sleep 100 PID: 1064
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
executar#ls | sleep 55#1063#
Tempo exec: 0
Tempo inac: 0
|
| Numero: 5 Estado: 4 Task: ls | sleep 55 PID: 1070
| Numero: 4 Estado: 4 Task: sleep 90 PID: 1067
| Numero: 3 Estado: 4 Task: sleep 100 PID: 1064
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
listar#1063#
Tempo exec: 0
Tempo inac: 0
|
| Numero: 5 Estado: 4 Task: ls | sleep 55 PID: 1070
| Numero: 4 Estado: 4 Task: sleep 90 PID: 1067
| Numero: 3 Estado: 4 Task: sleep 100 PID: 1064
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
[]

tgomes@DESKTOP-NS789C4:/mnt/c/Users/tgomes/Documents/Repos/SO_1920/trabalho$ ./argus
./argus
tempo-inactividade segs
tempo-execucao segs
executar "comando1 args | comando2 | comando3"
listar
terminar nTarefa
historico
output nTarefa
ajuda
exit

argus$ executar "sleep 100"
Output:
nova tarefa #3
argus$ executar "sleep 90"
Output:
nova tarefa #4
argus$ executar "ls | sleep 55"
Output:
nova tarefa #5
argus$ listar
Output:
#5: ls | sleep 55
#4: sleep 90
#3: sleep 100
Fim do comando Listar
argus$
```

Figura 2.4: Listar Tarefas (servidor / cliente).

2.3.5 Histórico de Tarefas Terminadas

À semelhança da instrução anterior, com a possibilidade de aceder directamente ao estado das tarefas (estado 0,1,2,3), é possível encaminhar a informação sobre as tarefas que já não se encontram em execução para o cliente que fez o pedido.

Exemplo Prático

Na figura seguinte encontra-se exemplificado o funcionamento do comando *histórico*. Após a execução de 3 comandos (sleep 100, sleep 90, ls | sleep 55), é executado o comando *histórico* e verificamos que 2 das 3 tarefas foram executadas, o que é correcto pois tinham decorrido apenas 95 segundos, logo o "sleep 100" ainda se encontra em execução.

```
tgomes@DESKTOP-NS789C4:/mnt/c/Users/tgomes/Documents/Repos/SO_1920/trabalho$ ./server
mkfifo error: File exists
Fifo aberto para leitura!
executar#sleep 100#1063#
Tempo exec: 0
Tempo inac: 0
|
| Numero: 3 Estado: 4 Task: sleep 100 PID: 1064
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
executar#sleep 90#1063#
Tempo exec: 0
Tempo inac: 0
|
| Numero: 4 Estado: 4 Task: sleep 90 PID: 1067
| Numero: 3 Estado: 4 Task: sleep 100 PID: 1064
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
executar#ls | sleep 55#1063#
Tempo exec: 0
Tempo inac: 0
|
| Numero: 5 Estado: 4 Task: ls | sleep 55 PID: 1070
| Numero: 4 Estado: 4 Task: sleep 90 PID: 1067
| Numero: 3 Estado: 4 Task: sleep 100 PID: 1064
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
listar#1063#
Tempo exec: 0
Tempo inac: 0
|
| Numero: 5 Estado: 4 Task: ls | sleep 55 PID: 1070
| Numero: 4 Estado: 4 Task: sleep 90 PID: 1067
| Numero: 3 Estado: 4 Task: sleep 100 PID: 1064
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
concludeInstructionsHandler
concludeInstructionsHandler
historico#1063#
Tempo exec: 0
Tempo inac: 0
|
| Numero: 5 Estado: 0 Task: ls | sleep 55 PID: 1070
| Numero: 4 Estado: 0 Task: sleep 90 PID: 1067
| Numero: 3 Estado: 4 Task: sleep 100 PID: 1064
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
|
Fifo aberto para leitura!
concludeInstructionsHandler
[]

tgomes@DESKTOP-NS789C4:/mnt/c/Users/tgomes/Documents/Repos/SO_1920/trabalho$ ./argus
./argus
tempo-inactividade segs
tempo-execucao segs
executar "comando1 args | comando2 | comando3"
listar
terminar nTarefa
historico
output nTarefa
ajuda
exit

argus$ executar "sleep 100"
Output:
nova tarefa #3
argus$ executar "sleep 90"
Output:
nova tarefa #4
argus$ executar "ls | sleep 55"
Output:
nova tarefa #5
argus$ listar
Output:
#5: ls | sleep 55
#4: sleep 90
#3: sleep 100
Fim do comando Listar
argus$ historico
Output:
#5, concluida: ls | sleep 55
#4, concluida: sleep 90
#2, concluida: ls | sleep 1
#1, concluida: ls | sleep 9
Fim do comando Historico
argus$
```

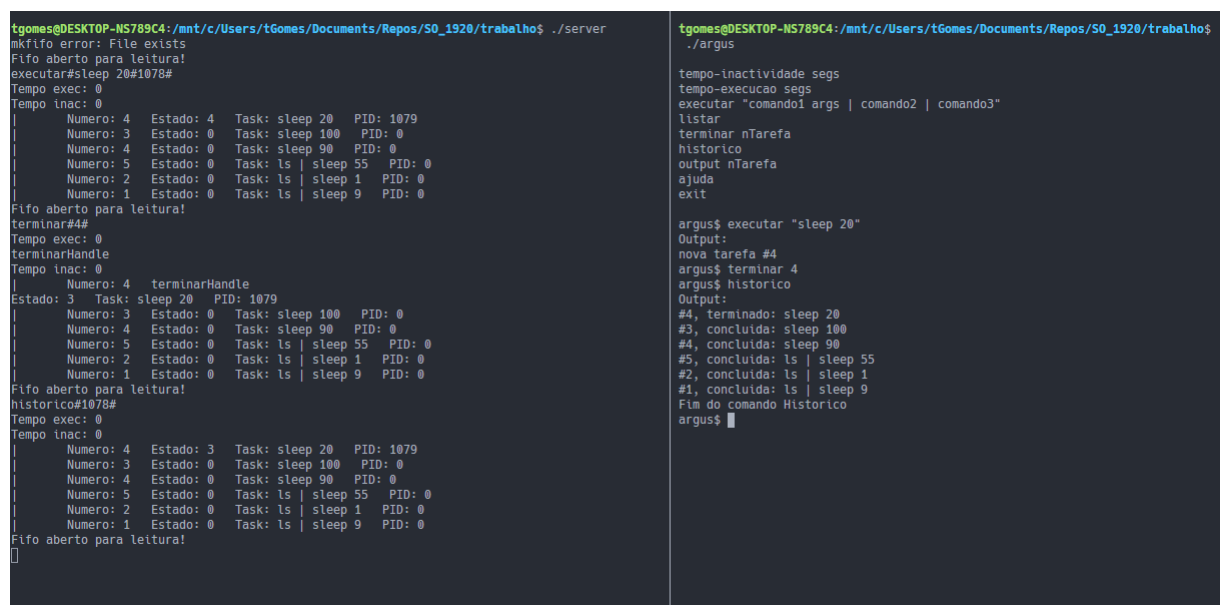
Figura 2.5: Histórico (servidor / cliente).

2.3.6 Terminar Tarefa em Execução

Ao receber o pedido para termino de uma tarefa, através da estrutura de dados implementada, é possível aceder directamente ao id do processo (pid) que está a executar o comando e enviar um sinal para este parar a sua execução e alterar o estado (para 3) na estrutura de dados. Na recepção deste sinal, o processo filho para a execução de todas as instruções, tal como a sua própria execução.

Exemplo Prático

Na figura seguinte encontra-se demonstrado o funcionamento do comando *terminar*. O cliente envia o comando *executar "sleep 20"*, de seguida executa *terminar 4* (id da tarefa). Para verificarmos o correto funcionamento, é executado um comando *historico* e verificamos que a tarefa 4 foi realmente terminada pelo utilizador.



```
tgomes@DESKTOP-NS789C4: /mnt/c/Users/tGomes/Documents/Repos/S0_1920/trabalho$ ./server
mkfifo error: File exists
Fifo aberto para leitura!
executar#sleep 20#1078#
Tempo exec: 0
Tempo inac: 0
| Numero: 4 Estado: 4 Task: sleep 20 PID: 1079
| Numero: 3 Estado: 0 Task: sleep 100 PID: 0
| Numero: 4 Estado: 0 Task: sleep 90 PID: 0
| Numero: 5 Estado: 0 Task: ls | sleep 55 PID: 0
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
Fifo aberto para leitura!
terminar#4#
Tempo exec: 0
terminarHandle
Tempo inac: 0
| Numero: 4 terminarHandle
Estado: 3 Task: sleep 20 PID: 1079
| Numero: 3 Estado: 0 Task: sleep 100 PID: 0
| Numero: 4 Estado: 0 Task: sleep 90 PID: 0
| Numero: 5 Estado: 0 Task: ls | sleep 55 PID: 0
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
Fifo aberto para leitura!
historico#1078#
Tempo exec: 0
Tempo inac: 0
| Numero: 4 Estado: 3 Task: sleep 20 PID: 1079
| Numero: 3 Estado: 0 Task: sleep 100 PID: 0
| Numero: 4 Estado: 0 Task: sleep 90 PID: 0
| Numero: 5 Estado: 0 Task: ls | sleep 55 PID: 0
| Numero: 2 Estado: 0 Task: ls | sleep 1 PID: 0
| Numero: 1 Estado: 0 Task: ls | sleep 9 PID: 0
Fifo aberto para leitura!
[]

tgomes@DESKTOP-NS789C4: /mnt/c/Users/tGomes/Documents/Repos/S0_1920/trabalho$ ./argus
tempo-inactividade segs
tempo-execucao segs
executar "comando1 args | comando2 | comando3"
listar
terminar nTarefa
historico
output nTarefa
ajuda
exit

argus$ executar "sleep 20"
Output:
nova tarefa #4
argus$ terminar 4
argus$ historico
Output:
#4, terminado: sleep 20
#3, concluida: sleep 100
#4, concluida: sleep 90
#5, concluida: ls | sleep 55
#2, concluida: ls | sleep 1
#1, concluida: ls | sleep 9
Fim do comando Historico
argus$
```

Figura 2.6: Terminar tarefa (servidor / cliente).

2.3.7 Menu Ajuda

Dependendo da interface optada pelo utilizador, linha de comandos ou shell, são apresentados flags ou comandos respectivamente (por exemplo: *-e* para linha de comandos e *"executar"* para shell). Esta funcionalidade está implementada do lado do cliente.



```
tgomes@DESKTOP-NS789C4: /mnt/c/Users/tGomes/Documents/Repos/S0_1920/trabalho$ ./argus -h
tempo-inactividade: -i segs
tempo-execucao: -m segs
executar: -e "comando1 args | comando2 | comando3"
listar: -l
terminar: -t nTarefa
historico: -r
output: -o nTarefa
ajuda: -h

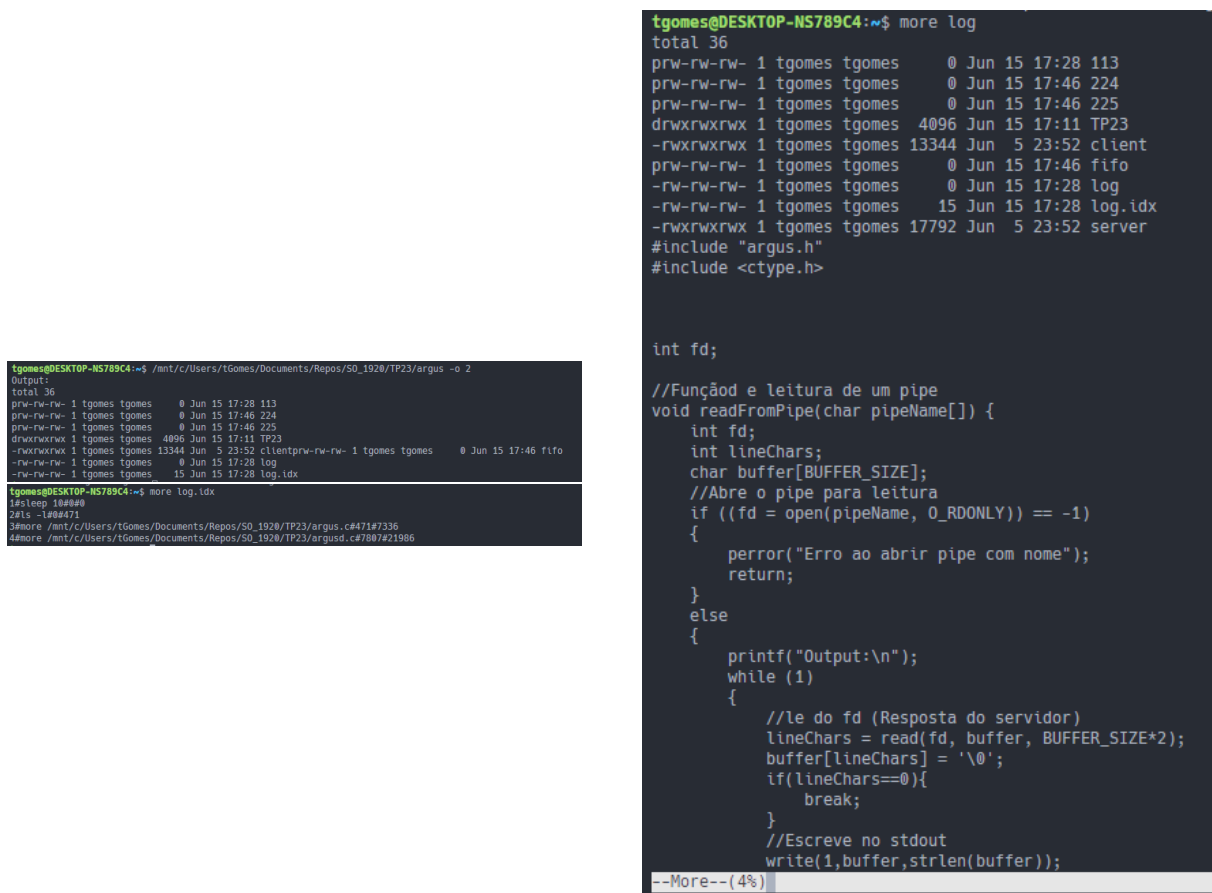
tgomes@DESKTOP-NS789C4: /mnt/c/Users/tGomes/Documents/Repos/S0_1920/trabalho$ ./argus
tempo-inactividade segs
tempo-execucao segs
executar "comando1 args | comando2 | comando3"
listar
terminar nTar
historico
output tarefa
ajuda
exit
argus$
```

Figura 2.7: Menu Ajuda Linha de Comandos e Shell

2.4 Funcionalidade Adicional

2.4.1 Consultar Output

Tal como mencionado na executar tarefa, o output de cada tarefa é escrito num ficheiro de log e guardado num ficheiro log.idx o número da tarefa, o comando, o índice do caracter inicial e a quantidade de caracteres que tem o output da instrução. Tendo estes ficheiros criados, consulto no ficheiro log.idx as informações relativas à tarefa pedida pelo utilizador e, de seguida, com a informação do caracter inicial e da quantidade de caracteres que tem a instrução, consulto o ficheiro de log onde analiso o output da tarefa, localizando-o através das informações recolhidas no log.idx. Na figura seguinte encontra-se um exemplo de um ficheiro de log gerado após a execução de alguns comandos.



The figure consists of two terminal screenshots. The left screenshot shows the output of the 'more log' command, displaying a list of tasks with their permissions, user, group, and timestamps. The right screenshot shows the C code for the 'readFromPipe' function, which reads from a pipe and writes to stdout.

```
tgomes@DESKTOP-NS789C4:~$ more log
total 36
prw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:28 113
prw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:46 224
prw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:46 225
drwxrwxrwx 1 tgomes tgomes 4096 Jun 15 17:11 TP23
-rwxrwxrwx 1 tgomes tgomes 13344 Jun  5 23:52 client
prw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:46 fifo
-rw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:28 log
-rw-rw-rw- 1 tgomes tgomes   15 Jun 15 17:28 log.idx
#include "argus.h"
#include <ctype.h>

int fd;

//Função de leitura de um pipe
void readFromPipe(char pipeName[]) {
    int fd;
    int lineChars;
    char buffer[BUFFER_SIZE];
    //Abre o pipe para leitura
    if ((fd = open(pipeName, O_RDONLY)) == -1)
    {
        perror("Erro ao abrir pipe com nome");
        return;
    }
    else
    {
        printf("Output:\n");
        while (1)
        {
            //le do fd (Resposta do servidor)
            lineChars = read(fd, buffer, BUFFER_SIZE*2);
            buffer[lineChars] = '\0';
            if(lineChars==0){
                break;
            }
            //Escreve no stdout
            write(1,buffer,strlen(buffer));
        }
    }
}
```

```
tgomes@DESKTOP-NS789C4:~$ /mnt/c/Users/tGomes/Documents/Repos/SO_1928/TP23/argus -o 2
Output:
total 36
prw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:28 113
prw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:46 224
prw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:46 225
drwxrwxrwx 1 tgomes tgomes 4096 Jun 15 17:11 TP23
-rwxrwxrwx 1 tgomes tgomes 13344 Jun  5 23:52 clientprw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:46 fifo
-rw-rw-rw- 1 tgomes tgomes    0 Jun 15 17:28 log
-rw-rw-rw- 1 tgomes tgomes   15 Jun 15 17:28 log.idx
tgomes@DESKTOP-NS789C4:~$ more log.idx
1$sleep 10###
2$ls -l###471
3$more /mnt/c/Users/tGomes/Documents/Repos/SO_1928/TP23/argus.c#471#7336
4$more /mnt/c/Users/tGomes/Documents/Repos/SO_1928/TP23/argus.c#471#7336
```

Figura 2.8: Menu Ajuda Linha de Comandos e Shell

2.4.2 Preservação do estado do servidor

Com a implementação da funcionalidade acima descrita, e tendo acesso a um ficheiro de indexação das tarefas previamente executadas, é possível manter o estado do servidor em relação a estas. Aquando da inicialização do servidor, há um carregamento e interpretação do ficheiro log.idx de forma a analisar as informações relativas às tarefas existentes no fim da última execução do programa.

3. Conclusão

A meu ver, este projecto respondeu a todas as necessidades requeridas pelo enunciado, bem como algumas funcionalidades adicionais. É de salientar a importância que este projecto teve na minha compreensão de várias particularidades referentes à comunicação entre processos, quer seja através de pipes com nome ou anónimos, quer seja a comunicação efectuada de filhos para pais através de sinais e mesmo o controlo de tempo possível com a implementação destes sinais com alarmes. Mesmo com os conteúdos aplicados nos guiões práticos, aquando da realização do trabalho é que fiquei com um nível de compreensão correcto sobre a criação de processos filho e as suas particularidades principalmente a nível de herança de memória, execução de comandos e redireccionamento de descritores. Considero esta incidência sobre o funcionamento intrínseco de processos e os detalhes inerentes aos mesmos, fulcrais para uma verdadeira compreensão de paralelização, da comunicação entre processos e da interoperabilidade semântica de forma a programar mais eficientemente, sendo isto, a meu ver, fundamental para um futuro engenheiro informático.

3.1 Trabalho Futuro

Penso que este projecto poderia ser expandido a nível de funcionalidade, contemplando mais necessidades que um utilizador possa ter, como por exemplo saber o tempo que uma dada tarefa demorou a executar, um histórico associado a um dado cliente e as tarefas executadas por esse cliente, entre outras, pois a base de cliente e servidor implementada permite futura escalabilidade.