

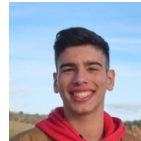
Sistemas Operativos

Relatório do Trabalho Prático - Rastreamento e Monitorização da Execução de Programas

LEI - 2022/2023

Grupo 43

Rui Chaves João Rodrigues Mateus Martins
A83693 A100711 A100645



Universidade do Minho

Índice

1	Introdução	3
2	Estrutura do Projeto	3
2.1	Instruções de Interação - <i>Usage</i>	4
2.2	Makefile	4
2.3	Ficheiro <i>Header</i> - common.h	5
2.4	Servidor - monitor.c	6
2.5	Cliente - tracer.c	7
3	Funcionalidades Básicas	8
3.1	Execução de programas do utilizador	8
3.1.1	execute -u	8
3.2	Consulta de programas em execução	9
3.2.1	status	9
4	Funcionalidades Avançadas	10
4.1	Execução encadeada de programas	10
4.1.1	execute -p	10
4.2	Armazenamento de informação sobre programas terminados	11
4.3	Consulta de programas terminados	11
4.3.1	stats-time	11
4.3.2	stats-command	12
4.3.3	stats-uniq	13
5	Comentário Final	14

1 Introdução

No âmbito da Unidade Curricular de **Sistemas Operativos** foi proposto o desenvolvimento, conceção e implementação de um serviço de monitorização de programas executados numa máquina.

O seguinte relatório irá expressar todas as funcionalidades que permitem que um utilizador consiga executar programas, através do cliente, e obter o seu tempo de execução, entre outras informações, e que permitem que um administrador de sistemas consiga consultar, através do servidor, todos os programas que se encontram atualmente em execução, incluindo o tempo despendido pelos mesmos, entre outras informações.

2 Estrutura do Projeto

O Projeto encontra-se partido em dois ficheiros `monitor.c` e `tracer.c`, um ficheiro `header` comum a ambos e um ficheiro `Makefile`. Seguidamente, iremos abordar o conteúdo de cada um destes ficheiros no contexto das várias funcionalidades que implementam, básicas e avançadas.

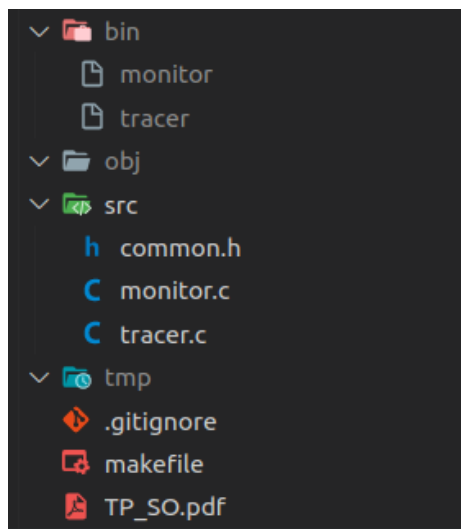


Figura 1. Estrutura do Projeto

2.1 Instruções de Interação - *Usage*

Tal como está exposto no enunciado deste trabalho prático, a forma de interagir com os programas criados é, primeiramente, recorrer à *Makefile* do projeto para compilar o código - fazendo *make* - e, em seguida, para executar o código e tirar proveito de todas as suas funcionalidades, correr um dos comandos expostos na secção seguinte.

2.2 Makefile

De forma a apresentar estas orientações de interação de uma forma mais intuitiva e prática para o utilizador comum, estas foram adicionadas ao ficheiro *Makefile* em forma de *target*, resultando na mensagem *Usage* seguinte:

```
##### BEM VINDO AO RASTREAMENTO E MONITORIZAÇÃO DA EXECUÇÃO DE PROGRAMAS! #####
TARGET
-Executar um programa individual:
./tracer execute -u "prog arg1 (...) argN"

-Executar uma pipeline de programas:
./tracer execute -p "progA arg1 (...) argN | progB arg1 (...) argN | progC arg1 (...) argN"

-Listar programas em execução:
./tracer status

-Total de tempo de execução de um conjunto de programas terminados:
./tracer stats-time PID1 PID2 (...) PIDN

-Número de vezes que um certo programa foi executado:
./tracer stats-command progA PID1 PID2 (...) PIDN

-Listar programas únicos executados:
./tracer stats-uniq PID1 PID2 (...) PIDN

SERVIDOR
-Executar o Servidor:
./monitor PIDS-folder
#####
```

Figura 2. Mensagem *Usage* apresentada ao fazer *make*

Adicionalmente, e no decorrer da criação do *header* file, sendo este incluído em ambos, *tracer* e *monitor*, houve a necessidade de atualizar os *targets* de compilação dos ficheiros *.c* para dependerem também deste ficheiro *common.h*, para considerar alterações feitas neste, caso hajam, aquando da compilação.

Estas duas alterações descrevem todas as alterações feitas à *Makefile* fornecida pelos docentes da UC no enunciado do trabalho e servem de base para a correta implementação e execução das funcionalidades sugeridas. O código relativo à *Makefile* está incluído nos anexos deste documento.

2.3 Ficheiro *Header* - `common.h`

Este ficheiro corresponde a um ficheiro *header* que é incluído em ambos `tracer` e `monitor`. Este ficheiro é essencial para o funcionamento correto destes, incluindo várias bibliotecas *standard* do C como `stdio.h`, que é utilizado para o `sprintf` e o `sys/types.h` e `sys/stat.h`, que permitem a utilização de FIFOs e a abertura de ficheiro, entre outras.

Para além disto, neste ficheiro é também definida uma estrutura de dados `exec` que irá servir de principal forma de comunicação Cliente→Servidor. Esta guarda a informação correspondente a novo pedido feito por um cliente através do `tracer` e é posteriormente enviado ao servidor para tratamento. A estrutura contém os seguintes valores:

- **int query_int:** Corresponde a um valor inteiro que identifica o tipo de pedido:
 - 1 - Início de Execução simples de programa - `execute -u`
 - 2 - Início de Execução encadeada de programas - `execute -p`
 - 3 - Fim de uma Execução
 - 4 - Consulta de programas em execução - `status`
 - 5 - Consulta de programas terminados - `stats-time`
 - 6 - Consulta de programas terminados - `stats-command`
 - 7 - Consulta de programas terminados - `stats-uniq`
- **int pid_pai:** Guarda o valor do Process ID correspondente ao processo da `bash` a executar o `tracer`;
- **int pid:** Guarda o valor do Process ID correspondente ao processo que faz uma execução (valor do primeiro processo no caso de uma execução encadeada);
- **prog_name[100]:** Guarda o nome do `commando/pipeline` a executar no caso de uma execução ou o nome do programa a pesquisar no caso de um pedido `stats-command` (vazio nos pedidos de tipo 4, 5 e 7);
- **struct timeval start:** Guarda o valor do timestamp correspondente ao início de uma execução (apenas não vazio em pedidos do tipo 1,2 e 3);
- **struct timeval end:** Guarda o valor do timestamp correspondente ao fim de uma execução (apenas não vazio em pedidos do tipo 3);
- **int pids_search[100]:** Guarda os valores dos Process ID a pesquisar no contexto de pedidos `stats-x` (vazio nos restantes pedidos);
- **int pids_search_size:** Guarda a quantidade de valores Process ID do array em cima (vazio nos restantes pedidos).

2.4 Servidor - monitor.c

Neste programa é onde se encontra o processamento principal de todo o projeto. De certa forma, tentou-se fazer com que maior parte do trabalho computacional realizado acontecesse do lado do Monitor (à parte das próprias execuções).

Neste programa definimos uma lista ligada de execuções atuais (que se encontram a correr) que irá ser essencial para a implementação da funcionalidade de consulta de programas em execução. Esta lista ligada guarda num nodo a informação de uma execução e um apontador para o seguinte nodo.

A execução do Servidor inicia-se através da passagem como argumento do nome de uma pasta. Nesta irá ser guardada, aquando do fim de uma execução, os vários ficheiros relativos à informação de execuções terminadas. A estrutura destes ficheiros irá ser abordada posteriormente. De notar que o *Monitor* é capaz de verificar a já existência deste pasta na *source* do projeto e/ou ocorrência de um erro e mostrar uma das seguintes mensagens informativas sobre o processo:

- "Folder ../X/ already exists"
- "Folder ../X/ created successfully"
- "Finished executions folder name not provided. Please try again..."
- "Error creating folder ../X/ for finished executions"

Em seguida, a lista ligada mencionada em cima é inicializada e o monitor cria um FIFO que irá servir de comunicação e ponto de entrada de pedidos (structs `exec`) enviados por todas as eventuais instâncias de Clientes/*Tracers* ao Servidor/*Monitor*. Assim, neste FIFO irão existir vários escritores e apenas um leitor. Ambos, monitor e tracer, conseguem abrir este FIFO e, por conseguinte, sabem o seu nome pois este foi definido estaticamente no ficheiro partilhado header como "fifo_client_server". Este nome faz, de certa forma, alusão à direção em que a informação circula no FIFO unidirecional, de Clientes para Servidor. Pode-se verificar a correta criação deste FIFO confirmando a sua existência na pasta `/tmp/` na *source* do projeto ao correr uma instância Monitor. Caso este pipe com nome já exista nenhuma mensagem de erro é mostrada e o programa continua.

Com o FIFO criado, e de forma a que os pedidos sejam lidos continuamente recorreu-se à criação de um ciclo infinito `while(1)`. Dento do *loop* o programa lê os pedidos dos clientes de tamanho fixo `sizeof(struct(exec))` assegurando-se que este valor é sempre menor que o tamanho de `PIPE_BUF`.

Após a leitura de um pedido, este é apresentado no terminal devidamente formatado consoante o tipo de pedido que é (através de `query_int`).

Para pedidos `status/stats-x` vai ser necessário haver comunicação de informação por parte do Monitor para a instância Tracer específica que fez o pedido. Como tal e considerando que a comunicação é unidirecional, houve necessidade de criar um novo FIFO (pipe com nome) de comunicação Servidor→ClienteX. Quando se recebe um pedido destes, ao contrário dos pedidos relativos a execuções, é criado um novo FIFO com o nome "fifo_server_client_X". O valor de X corresponde ao Process ID do processo da bash que criou a instância Tracer de fez o pedido e foi exatamente por esta razão que este valor foi incluído na estrutura de dados. De notar que este FIFO é eliminado do lado do Tracer aquando do fim da execução, pelo que o mesmo terminal pode fazer vários pedidos sequencialmente sem que haja a possibilidade de erros na leitura derivados de informações de execuções anteriores.

Nestes pedidos após a abertura do FIFO, tratam-se estes pedidos no contexto de um processo filho. Esta decisão foi motivada pelo facto de aquando da criação deste a informação da lista ligada que guarda as execuções a decorrer estar acessível e, também pelo facto, destas operações `status/stats-x` serem bastante mais demoradas obrigando a uma maior tempo de processamento do que as relativas a execuções (do tipo 1, 2 e 3) e assim não se bloquear a interação de outros clientes com o servidor. O objetivo foi, assim, permitir o processamento concorrente de pedidos.

2.5 Cliente - `tracer.c`

No programa *Tracer* é primeiramente feita a verificação dos argumentos passados por terminal e, caso não estejam corretos, mostrada uma mensagem de erro. É posteriormente, aberto o FIFO já criado pela instância do *Monitor* a correr de comunicação Cliente→Servidor e também criado o FIFO Servidor→Cliente X. Este último será apenas aberto, tanto do lado do Servidor como do Cliente, caso seja necessário. É, depois, feito o *parse* dos argumentos e tratado o pedido correspondente.

Em seguida, vai ser detalhado como cada funcionalidade em si foi implementada.

3 Funcionalidades Básicas

3.1 Execução de programas do utilizador

3.1.1 `execute -u` O cliente é responsável por executar programas dos utilizadores (p.ex., `cat`, `grep`, `wc`) através da opção `execute` e comunicar ao servidor o estado dessa execução. O servidor é responsável por armazenar essa informação de execução e disponibilizá-la para consulta.

Para implementar esta funcionalidade começa-se por fazer o *parse* do comando, separando o nome do programa do(s) seu(s) argumento(s), tendo o cuidado de definir o último elemento a *NULL* para indicar o fim dos argumentos.

De seguida, guarda-se o *timestamp* atual para ser adicionado na estrutura como o tempo inicial de execução, através da função `gettimeofday` (como sugerido pelos docentes). Isto é feito imediatamente antes da criação do processo filho que irá correr o comando fornecido de forma a obter posteriormente o intervalo de tempo mais correto possível. Preferimos guardar o tempo antes do *fork* e não no contexto do processo filho, porque isto implicaria adicionar um pipe anónimo extra desnecessário (na nossa opinião) apenas para passar esse valor (ainda que mais correto) imediatamente antes de correr o comando, para o processo pai.

De seguida, o processo filho corre o programa e os seu argumentos através da *call* `execvp`, retornando o valor de retorno desta *call*. Concorrentemente, o processo pai trata de criar a estrutura que irá ser enviada pelo FIFO para o servidor, que inclui o inteiro que identifica uma execução simples - 1 -, o PID do processo Pai, o PID do processo resultante do *fork* e o nome do comando. De seguida, informa no terminal a mensagem "Running PID: X", X correspondendo ao valor PID do processo filho. Após enviar a estrutura para o FIFO, o processo pai espera que o processo X termine. Quando termina, é guardado o *timestamp* final e iniciado o processo de sinalizar o Servidor de que o programa terminou.

Este processo de terminação, altera a estrutura passada no processo anterior, atualizando o valor que identifica uma execução simples para uma que identifica o término da execução - de 1 para 3 - e definindo o valor *end* com o valor do *timestamp* final. Esta nova estrutura é enviada pelo FIFO para o Servidor e é mostrado no terminal a mensagem: "Ended in y ms", y correspondendo ao valor em milissegundos da diferença entre o *timestamp* de início e final de execução.

Do lado do servidor, informa-se que foi recebido um novo pedido de início de uma execução através de uma mensagem de *debug* mostrada no terminal. Após a receção, é criado um novo nodo e adicionada à lista ligada a execução corrente. Aquando da receção do pedido de fim de uma execução, é igualmente mostrada uma mensagem e itera-se sobre a lista à procura do nodo correspondente através do PID e, caso se encontre, o nodo é removido e a informação de execução terminada guardada num ficheiro.

O código implementado inclui controlo de erros.

3.2 Consulta de programas em execução

3.2.1 status Através da opção `status` devem ser listados, um por linha, os programas em execução no momento. A informação de cada programa deve conter o seu PID, nome, e tempo de execução até ao momento (em milissegundos).

Para implementar esta funcionalidade cria-se uma estrutura de dados *exec* do lado do cliente com o identificador do pedido 4 e o PID do processo Pai e é enviado para o FIFO para o Servidor ler. Depois, é aberto o FIFO previamente criado para leitura das mensagens enviadas pelo Servidor e caso o valor seja 0 é apresentada a mensagem "There aren't any programs currently running", tanto do lado do Cliente como do lado do Servidor. Caso não o seja, o PID, o nome do comando e o tempo de execução até ao momento de cada execução são apresentados um em casa linha.

Do lado do Servidor, para obter a lista de programas em execução, obtém-se o tamanho da lista e caso seja 0, envia-se essa informação para o cliente. Caso não o seja, itera-se pela lista ligada e em cada nodo calcula-se o tempo de execução até ao momento e constrói-se a string que contém toda a informação necessária. Para passar cada string para o tracer envia-se primeiro o tamanho da string e depois a string em si. De relembrar, que esta operação é feita no contexto de um processo filho pelo que, caso chegue ao fim da lista, o processo termina com o código 0.

A escolha da lista ligada como estrutura deveu-se ao facto de permitir um tempo constante de inserção e remoção de elementos, e $O(n)$ para encontrar o nodo em si. A situação crítica neste projeto é a inserção e remoção em si. Um *hashmap* baseado no PID foi considerado para uma procura mais rápida mas a lista ligada pareceu-nos ideal para um projeto desta dimensão.

4 Funcionalidades Avançadas

4.1 Execução encadeada de programas

4.1.1 `execute -p` Através da opção `execute -p`, o cliente deve suportar a execução encadeada de programas do utilizador (pipelines) tal como acontece na linha de comandos quando usado o operador

A implementação desta funcionalidade, apesar de mais complexa e de necessitar alguma coordenação, é algo análoga à de uma execução simples. Todo o processo de sinalização da terminação de execução é até comum à execução simples pelo que a explicação não será repetida (query_int de 2 para 3, ao invés de 1 para 3). A parte de receção, leitura, adicionar na struct e remoção e guardar num ficheiro do lado do Servidor é também comum.

Do lado do Cliente, é necessário fazer parse dos vários comandos da pipeline através do *parse* pelo carácter `ẽ` em seguida dos programas e argumentos em casa comando em si. Após isto, a principal diferença é que é necessário a criação de tantos processos filho quantos os comandos presentes na pipeline e de tantos pipes anónimos como o comandos-1 para a comunicação entre os processos. O ponto chave e o que permite a comunicação da informação de execução entre os processos é a implementação cuidadosa dos `closes` e das duplicações de *file descriptors*. Temos que considerar 3 casos: o caso de ser o primeiro comando a ser executado, em que é necessário fechar o ponto de entrada do primeiro pipe anónimo e trocar o `stdout` para o ponto de entrada do pipe anónimo seguinte; o caso de ser o último comando a ser executado, em que é necessário mudar o `stdin` para ser o ponto de entrada do último pipe anónimo; e, por fim, o caso de ser uma execução intermédia, em que é necessário trocar tanto o `stdin` como o `stdout` para o entrada do pipe anterior e saída do pipe com o mesmo índice respetivamente.

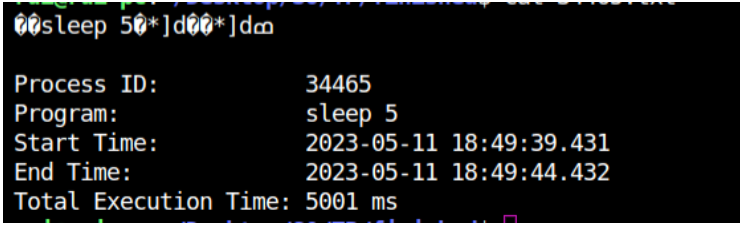
Nestes três casos é também necessário fechar os *file descriptors* corretos para que um certo processo não fique continuamente aberto à custa de um que não era suposto estar aberto, estar. Por segurança, antes da execução do comando destinado a um certo processo filho, todos os pipes são fechados devidamente.

O processo pai trata apenas de guardar num array os PIDs dos processos filho criados, permitindo assim que este corram concorrentemente (limitados apenas pelas correta execução dos processos filho/comandos anteriores na pipeline).

O processo pai espera depois pelos processos filhos, construindo e enviando a estrutura *exec* com o PID do 1º processo filho do array e esperando pelo último para iniciar o processo final de sinalização de término.

4.2 Armazenamento de informação sobre programas terminados

Aquando do término de uma execução, o nodo correspondente é eliminado da lista ligada e a sua informação é guardada em ficheiro. Este ficheiro é guardado na pasta passada como argumento ao *Monitor* destinada a guardar estas mesmas informações e tem a seguinte estrutura:



```
00sleep 50*d00*d00
Process ID:      34465
Program:        sleep 5
Start Time:     2023-05-11 18:49:39.431
End Time:       2023-05-11 18:49:44.432
Total Execution Time: 5001 ms
```

Figura 3. Estrutura do ficheiro de uma execução terminada "sleep 5"

Como é possível verificar, no início do ficheiro que é identificado pelo PID do processo filho que correu o programa - [PID].txt - guarda-se a estrutura em si e, depois, de uma forma legível, as únicas informações necessárias da estrutura. De notar que os *timestamps* são guardados de acordo com o formato local. A razão pela qual a estrutura foi guardada neste ficheiro foi para facilitar a implementação das funcionalidades avançadas *stats-x*.

4.3 Consulta de programas terminados

4.3.1 stats-time Através do comando *stats-time*, deve ser impresso no standard output o tempo total (em milisegundos) utilizado por um dado conjunto de programas identificados por uma lista de PIDs passada como argumento

Do lado do Cliente, é primeiramente criada uma nova estrutura de dados *exec* com o identificador 5, o PID do processo Pai, o array com os PIDs através dos quais vai ser feita a pesquisa e o tamanho deste mesmo array, e depois, esta é enviada para o Servidor. Em seguida é aberto o FIFO já criado para ler as mensagens enviadas pelo Servidor e por este

FIFO é lida a mensagem final "Total execution time is y ms", onde y é o valor calculado pelo Servidor para os PIDs da estrutura.

Do lado do Servidor, o array de PIDs a pesquisar é lido e depois são criados tantos processos filhos quanto os passados. Um pipe para onde os processos filho vão escrever e um pai ler também foi necessário. A cada processo filho é atribuído um PID e faz as seguintes operações:

Tenta abrir o ficheiro corresponde ao PID; Se falhar, escreve no pipe o valor 0. Se suceder, tenta ler do ficheiro o tamanho corresponde a uma struct *exec*, depois lê os valores dos *timestamps* inicial e final, calcula a diferença em milissegundos e escreve para o pipe o valor. Todos estes processos filhos correm concorrentemente.

O processo pai, por sua vez, espera que todos terminem e depois lê os vários valores *int* do pipe e soma-os. Em seguida constrói a string final e comunica-a ao Cliente.

De forma a evitar potenciais erros e a ter a certeza de que o processo Pai apenas lê do pipe quando todos os processos filho terminarem a utilização de *wait* foi necessária. De lembrar, que esta operação é feita no contexto de um processo filho pelo que, caso chegue ao fim da lista, o processo termina com o código 0.

4.3.2 stats-command Através do comando *stats-command*, deve ser impresso no standard output o número de vezes que foi executado um certo programa, cujo nome é passado como argumento, para um dado conjunto de PIDs, também passado como argumento.

Do lado do Cliente, é primeiramente criada uma nova estrutura de dados *exec* com o identificador 6, o PID do processo Pai, o nome do programa a pesquisar, o array com os PIDs através dos quais vai ser feita a pesquisa e o tamanho deste mesmo array, e depois, esta é enviada para o Servidor. Em seguida é aberto o FIFO já criado para ler as mensagens enviadas pelo Servidor e por este FIFO é lida a mensagem final "X was executed Y times", onde x é o nome do programa e y é o número de vezes calculado pelo Servidor para os PIDs da estrutura.

Do lado do Servidor a implementação é análoga à da funcionalidade *stats-time* no sentido em que também são criados os vários forks, apenas se diferenciando na forma como o pai recebe a informação calculada pelo filho. Considerando que o número que pretendemos obter dos filhos é binário, 0 caso o programa não seja o do PID ou 1 caso seja, não foi necessário a criação de um pipe anónimo e achamos por bem recorrer ao *WEXITSTATUS*. Para recorrer ao código de saída do *exit*, consideramos que os erros que poderiam acontecer no contexto do processo filho não

seriam de todo importante e limita-mos este código a 1 ou 0. O valor 0 é passado inclusive caso o ficheiro com o valor de PID não exista, por exemplo.

Desta forma e tal como no **stats-time**, do lado do pai apenas foi necessário realizar a soma dos valores, construir a string final e comunicá-la ao Cliente. Recorrer ao **WEXITSTATUS** verificou-se uma implementação simples e direta, capaz de produzir os resultados esperados com uma eficiência suficiente, não havendo a necessidade de aumentar a complexidade através de pipes anónimos.

De notar que a atual implementação permite que o comando a pesquisar seja completo ou apenas parcial - `sleep` ou `"sleep 1"`. Foi relativamente fácil de implementar mas aumenta a versatilidade da funcionalidade.

4.3.3 stats-uniq Através do comando **stats-uniq**, deve ser impressa no standard output a lista de nomes de programas diferentes (únicos), um por linha, contidos na lista de PIDs passada como argumento.

Do lado do Cliente, é primeiramente criada uma nova estrutura de dados *exec* com o identificador 7, o PID do processo Pai, o array com os PIDs através dos quais vai ser feita a pesquisa e o tamanho deste mesmo array, e depois, esta é enviada para o Servidor. Em seguida é aberto o FIFO já criado para ler as mensagens enviadas pelo Servidor. Após o processamento do pedido do lado do Servidor e através de um *loop* é lido do FIFO os vários nomes de programas únicos, sendo cada um apresentado numa linha separadamente.

Do lado do Servidor a implementação é análoga à da funcionalidade **stats-time** no sentido em que também são criados os vários forks, mas em vez de ser criado apenas um pipe anónimo são criados tantos quanto os processos filho, em que cada um vai escrever num pipe. Em caso de o ficheiro com um determinado PID não existir, é escrito o valor 0 e, caso exista, é escrito o tamanho do nome do programa (`int` e o nome em si. Estes filhos vão correr concorrentemente.

O processo pai vai depois ler de cada pipe anónimo sequencialmente e caso o valor não corresponda a 0 e o comando não seja ainda conhecido vai ser guardado para que seja comparado com os seguintes comandos. Em seguida, cada programa único é enviado sequencialmente pelo FIFO para o Cliente.

5 Comentário Final

Finalizando, sentimos que este projeto explorou a nossa criatividade e capacidade de superar dificuldades, o que nos levou a descobrir aspectos da Linguagem *C* e de Sistemas Operativos com os quais até então não estávamos familiarizados, nomeadamente a utilização *System Calls*, a capacidade de emular um Sistema Operativo recorrendo à manipulação e criação de processos e estabelecer comunicações entre processos através de pipes anónimos e FIFOs.

Apesar das adversidades encontradas ao longo da realização do projeto, conseguimos resultados bastante satisfatórios. A nível geral, e tendo em conta o que foi explicado nos capítulos anteriores, podemos afirmar que temos um projeto bem conseguido e que, este, nos ajudou a aprofundar e consolidar conhecimentos adquiridos ao longo da Unidade Curricular

Referências Bibliográficas

1. Draw.io: Ferramenta para construção gráfica da Topologia
2. StackOverflow: Ferramenta para resolver problemas relacionados com código e erros associados
3. Youtube - "pipe fork dup exec explained"
4. OverLeaf: Plataforma de colaboração em tempo real para a criação e edição de documentos *LaTeX*