

---

# Projeto Prático de Sistemas Operativos

---

## TRABALHO REALIZADO POR:

BIANCA ARAÚJO DO VALE  
MATILDE MARIA FERREIRA DE SOUSA FERNANDES  
PEDRO DANTAS DA CUNHA PEREIRA

GRUPO 108



A95835  
Bianca Vale



A95319  
Matilde Fernandes



A97396  
Pedro Dantas

---

# Índice

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Comunicação Cliente-Servidor</b>	<b>2</b>
<b>3</b>	<b>Arquitetura do Programa</b>	<b>2</b>
<b>4</b>	<b>Funcionalidades do Programa</b>	<b>2</b>
4.1	Funcionalidades Básicas . . . . .	2
4.2	Aspetos Gerais . . . . .	2
4.2.1	Execução de Programas do Utilizador . . . . .	3
4.2.2	Consulta de Programas em Execução . . . . .	3
4.3	Funcionalidades Avançadas . . . . .	4
4.3.1	Armazenamento de Informação Sobre Programas Terminados . . .	4
4.3.2	Consulta de Programas Terminados . . . . .	4
4.3.3	Execução Encadeada de Programas . . . . .	4
<b>5</b>	<b>Conclusão</b>	<b>5</b>

---

## 1 Introdução

No âmbito da Unidade Curricular de Sistemas Operativos do segundo ano da licenciatura em Engenharia Informática, foi-nos proposta a implementação de um serviço de monitorização dos programas executados numa máquina, com auxílio da linguagem de programação imperativa C, pondo, deste modo, em prática, toda a aprendizagem adquirida durante as aulas da respetiva UC.

De forma sucinta e muito simplificada, deverá ser desenvolvido um cliente (programa *tracer*) que ofereça uma interface com o utilizador via linha de comandos. Deverá ser também desenvolvido um servidor (programa *monitor*), com o qual o cliente irá interagir e comunicar. O servidor deve manter em memória e em ficheiros a informação relevante para suportar as funcionalidades descritas neste enunciado.

## 2 Comunicação Cliente-Servidor

Tal como foi referido anteriormente, o projeto passa pelo desenvolvimento de um programa servidor e um programa cliente que comunicam entre si. Para isto, tanto o cliente como o servidor deverão ser programas escritos em C e deverão comunicar via *pipes* com nome. Será utilizado o *standard output* pelo cliente para apresentar as respostas necessárias ao utilizador, e pelo servidor para apresentar informação de debug que julgue necessária.

## 3 Arquitetura do Programa

O nosso programa está dependente de duas peças fundamentais: o *tracer*(cliente) e o *monitor*(servidor), sendo que ambos comunicam através de *pipes*.

O programa que desenvolvemos possui um *pipe* principal que recebe todos os pedidos do cliente, que no nosso caso são os *PIDs* dos processos a ser executados. Cada vez que o cliente quer executar um programa é criado um ou mais *fifos* para comunicar com o servidor durante a sua execução. Os *PIDs* que o programa principal recebe servem para abrir os *fifos* que foram criados no programa cliente que está a ser executado, estes programas criam *fifos* com o nome equivalente ao *PID* do processo que está a ser executado e servem para o servidor comunicar com o cliente durante a execução de um programa.

## 4 Funcionalidades do Programa

Tal como foi indicado no enunciado do projeto, existe um número de funcionalidades que devem estar presentes no programa para que este possa ser valorizado, sendo estas divididas em duas categorias: básicas e avançadas.

### 4.1 Funcionalidades Básicas

Em relação às funcionalidades básicas propostas, podemos dizer que, no geral, conseguimos implementá-las na sua totalidade e com um funcionamento correto. Em termos de dificuldades, destacamos a implementação do *status* (aquando da sua chamada devem ser listados, um por linha, os programas em execução no momento).

### 4.2 Aspetos Gerais

De modo a facilitar a forma como os dados relativos aos programas executados são guardados e consultados, criamos a seguinte *struct* com as informações relevantes para o

---

funcionamento do projeto:

```
typedef struct Informacao{
    pid_t pid;
    char nome[50];
    long tempo;
} Informacao;
```

A partir daqui, começamos por testar o uso de *hashtables* (com o auxílio da *glib*) como forma de guardar as *structs* (e os dados contidos nas mesmas) anteriormente referidas. Apesar disto, devido a um erro que julgamos poder estar relacionado ao uso das *hashtables*, decidimos substituí-las por listas ligadas, o que acabou por revelar que o erro, de facto, não era o que nós esperávamos, sendo que acabamos por não conseguir encontrá-lo e resolvê-lo a tempo. Tendo isto em conta, decidimos manter o uso das listas ligadas ao invés de *hashtables*.

#### 4.2.1 Execução de Programas do Utilizador

Nesta função, inicialmente, é alocado espaço para uma cópia da *string* passada como argumento. Essa *string* é usada para iterar sobre cada argumento do comando e armazená-lo num *array* de *strings*.

De seguida, é verificado se o comando começa com a opção **-u**. Caso se verifique, é criado um processo filho que executa o comando, enquanto o processo pai regista informações sobre o processo filho e comunica com o programa **monitor** através de um FIFO cujo nome será o *PID* do processo.

Antes de executar o comando, é registado o tempo atual com o uso da função *gettimeofday*, armazenando-o numa variável que será usada posteriormente para calcular o tempo de execução do comando.

Após, é criado um novo processo através de um *fork*, sendo que este será responsável por executar o comando (com o auxílio da função *execvp*). A execução do processo filho é bloqueada até que o processo pai escreva num *pipe*.

Antes de escrever no *pipe*, são registadas informações sobre o processo filho, como o PID, o nome do comando e o momento em que começou a execução do programa, numa estrutura de dados do tipo *Informacao*, sendo esta enviada para o monitor.

De seguida, a função escreve estas informações no FIFO e, após isto, o processo pai escreve no *pipe* para desbloquear o processo filho e espera pela finalização da execução do comando.

Finalmente, é registado o momento em que a execução do comando termina (novamente com o auxílio da função *gettimeofday*) e são escritas as informações finais no FIFO, incluindo o tempo de término e o tempo de execução do comando. No final, é libertada a memória anteriormente alocada.

#### 4.2.2 Consulta de Programas em Execução

Tal como referido anteriormente, tivemos dificuldades na implementação desta funcionalidade, uma vez que nos deparamos com um erro, também já referido, que não conseguimos resolver e não sabemos o que o poderá estar a causar. O erro em questão passa por, aquando da execução do comando *status*, existem casos em que certas informações que não deviam estar presentes na lista ligada de programas em execução (sendo que foram previamente removidas), ainda conseguem ser acedidas, o que parece ocorrer

---

de forma aleatória, tendo em conta que, por exemplo, em três execuções seguidas do *status*, na primeira o *output* é o correto, na próxima o *output* é incorreto (estando presentes as informações que deveriam ter sido removidas) e, numa terceira vez, voltamos a obter o *output* correto.

Este erro ocorre tanto com listas ligadas como com *hashtables*, como referido anteriormente. Aquando da testagem do comando no *gdb*, obtivemos o *output* correto em todas as execuções, pelo que não sabemos o que poderá estar a causar o erro.

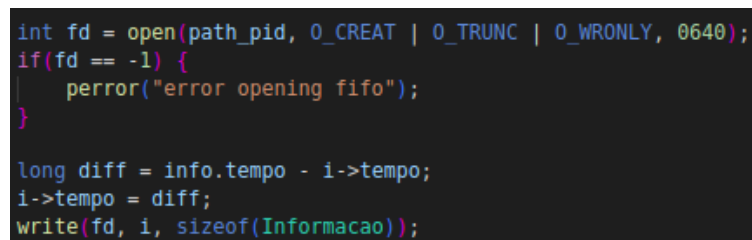
No que diz respeito à implementação desta funcionalidade, tudo o que temos de fazer é percorrer a lista ligada anteriormente criada e que armazena as informações dos programas em execução. Estas informações são posteriormente enviadas para o cliente via *write*, lidas pelo cliente via *read* e impressas no *standard output*.

### 4.3 Funcionalidades Avançadas

Em termos de funcionalidades avançadas, conseguimos implementar tudo o que foi pedido no enunciado do projeto, ou seja, consulta de programas terminados, armazenamento de informação sobre programas terminados e execução encadeada de programas.

#### 4.3.1 Armazenamento de Informação Sobre Programas Terminados

Para armazenar a informação sobre os programas terminados, quando executamos o comando *execute -u*, o cliente manda uma mensagem ao servidor para o sinalizar que o programa terminou e o servidor, ao receber esta mensagem, vai executar a função *executeU\_final*, na qual o armazenamento da informação vai ser feito.



```
int fd = open(path_pid, O_CREAT | O_TRUNC | O_WRONLY, 0640);
if(fd == -1) {
    perror("error opening fifo");
}

long diff = info.tempo - i->tempo;
i->tempo = diff;
write(fd, i, sizeof(Informacao));
```

Excerto do código da *executeU\_final*

Na imagem acima, temos a parte do código onde é criado/aberto um ficheiro binário em modo de escrita, onde posteriormente são guardadas as structs com as informações do programa terminado.

#### 4.3.2 Consulta de Programas Terminados

Em relação à consulta de programas terminados, ou seja, os comandos *stats-time*, *stats-command* e *stats-uniq*, e dadas as semelhanças entre estes 3 comandos, a sua resolução foi, também ela, idêntica (tendo em consideração as suas diferenças fundamentais).

Em todos os casos, começamos por guardar a lista *PIDs* passados como argumento num *array*, de modo a podermos procurar as informações necessárias e associadas a cada um deles no *PIDS\_folder*. A partir deste ponto, a solução passa por procurar cada um dos *PIDs* nesta pasta e, assim que o ficheiro associado a estes for encontrado, as informações que precisamos são retiradas do mesmo, por exemplo, no comando *stats\_time* é feito um somatório com os todos os tempos de execução relativos aos *PIDs* fornecidos.

#### 4.3.3 Execução Encadeada de Programas

Esta função executa uma sequência de comandos encadeados por *pipes*. O objetivo é criar um processo filho para cada comando, ligando o *standard output* do processo pai

---

ao *standard input* do processo filho saída padrão do processo pai à entrada padrão do processo filho através da utilização de *pipes*.

Depois disto, cada processo filho executa o seu respectivo comando e passa o *output* para o processo pai.

Finalmente, é usado um terceiro *loop for* para criar processos filhos para cada comando, ligando o *standard input* e *standard output* dos processos pai e filho usando os *pipes* criados anteriormente. Em seguida, cada processo filho executa seu respectivo comando usando a função *execvp* e o processo pai aguarda até que todos os processos filhos sejam concluídos. O tempo total necessário para executar a sequência de comandos é calculado através da função *gettimeofday* e é impresso no final da execução.

## 5 Conclusão

De forma geral, consideramos que o trabalho realizado foi bom, tendo em conta que tanto as funcionalidades básicas como as avançadas sugeridas foram implementadas na sua totalidade. Apesar disto, e tal como foi referido e explicado anteriormente, o nosso programa contém um erro que não fomos capazes de resolver a tempo da entrega final, pelo que não está totalmente correto.

Posto isto, é seguro dizer que os conhecimentos adquiridos nesta UC e, mais especificamente, com a realização do projeto são de grande valor, pelo que esperamos poder vir a poli-los e utilizá-los de forma mais eficiente e sábia no futuro.