



UNIVERSIDADE DO MINHO

LICENCIATURA EM ENGENHARIA INFORMÁTICA

## **Sistemas Operativos**

### **Trabalho Prático - Rastreamento e Monitorização da Execução de Programas**

#### **Grupo 84**

Ana Rita Poças (A97284)      Inês Ferreira (A97372)

13 de maio de 2023

# Introdução

Este trabalho prático surge no âmbito da Unidade Curricular de Sistemas Operativos, e consiste na implementação de um serviço de monitorização dos programas executados numa máquina. O objetivo é que os utilizadores do sistema consigam executar programas, através do cliente (programa *tracer*), e obter o seu tempo de execução e ainda que um administrador de sistemas consiga consultar, através do servidor (programa *monitor*), todos os programas em execução e o tempo despendido pelos mesmos.

## Arquitetura do sistema

### Comunicação entre cliente (tracer) - servidor (monitor)

No nosso trabalho, de forma a que o servidor consiga atender os pedidos de vários clientes de forma concorrente, procedemos à criação de um *pipe* com nome (FIFO) no servidor com o nome **CLIENT\_TO\_SERVER**, que é partilhado com os diferentes clientes e fica guardado na pasta *tmp*. Por sua vez, de forma a que o servidor seja capaz de comunicar com o cliente, decidimos que o cliente ao efetuar um pedido deve criar um pipe único com o nome do PID do processo cliente, desta forma, o servidor tendo acesso ao PID do processo recebido é capaz de enviar a resposta para o devido cliente. O pipe com nome criado pelo cliente é também guardado na pasta *tmp*.

### Estruturas de dados do programa

De forma a suportar os pedidos recebidos pelo cliente, procedemos à criação das seguintes estruturas de dados:

#### PEDIDO

```
//struct que define um novo pedido do cliente
typedef struct pedido
{
    int pid;
    struct timeval initial_timestamp;
}PEDIDO;
```

Figura 1: Estrutura PEDIDO do programa

Esta estrutura, usada unicamente no lado do cliente, possui os campos **PID** (identificador do processo) e **initial\_timestamp** (tempo imediatamente antes do início da execução do programa) e serve essencialmente para que o cliente seja capaz de calcular o tempo de execução de um dado programa, isto é, assim que o pedido é recebido guardamos o tempo inicial na estrutura de dados, associado ao pid do processo em questão e posteriormente à execução do programa, de forma a que o cliente seja capaz de notificar o utilizador, via standard output, do tempo de execução do processo, subtraímos o valor guardado na struct do tempo atual.

## PEDIDOSEXECUCAO

```
// struct que guarda os pedidos em execução num dado momento
typedef struct pedidos_execucao{
    char nome_programa[512];
    int pid;
    long initial_timestamp;
} PEDIDOSEXECUCAO;
```

Figura 2: Estrutura PEDIDOSEXECUCAO do programa

Esta estrutura, usada unicamente no lado do servidor, possui os campos **nome\_programa** (nome do programa a executar), **pid** (identificador do processo) e **initial\_timestamp** (tempo imediatamente antes do início da execução do programa). Basicamente, quando o cliente envia informação ao servidor sobre a execução de um novo programa contendo o nome do programa, pid do processo e tempo inicial em milissegundos, o servidor recebe esta informação, efetua o seu *parsing* e cria uma nova instância da estrutura **PEDIDOSEXECUCAO** e atribui a informação devida a cada um dos campos da estrutura. De forma a armazenar a informação sobre os pedidos em execução o servidor cria também o seguinte array:

```
PEDIDOSEXECUCAO *array_processos_running[MAX];
int num_processos_running = 0;
```

Figura 3: Array que alberga os PEDIDOSEXECUCAO do programa

Após atribuir todas as informações necessárias relativas ao pedido recebido pelo cliente na instância da estrutura **PEDIDOSEXECUCAO**, o servidor adiciona ao array **array\_processos\_running** essa instância e incrementa o valor de **num\_processos\_running**. Por sua vez quando, o cliente informa o servidor da terminação de um dado programa, o servidor percorre o array **array\_processos\_running** e remove a instância de **PEDIDOSEXECUCAO** associada com o pedido em causa e decrementa o valor de **num\_processos\_running**.

# Funcionalidades Implementadas

## Funcionalidades Básicas

De acordo com o que foi requerido pela equipa docente, o serviço deverá ser capaz de suportar as seguintes funcionalidades básicas:

### Execução de programas no utilizador

Nesta funcionalidade, o nosso programa cliente (tracer) recebe do utilizador os argumentos da linha de comandos:

- execute -u
- nome do programa a executar
- argumentos do programa

Após calcular o tempo atual em milissegundos, de forma a determinar o tempo inicial do processo, o cliente efetua o parsing destes argumentos, o cliente abre o fifo **CLIENT\_TO\_SERVER** para escrita e envia para o servidor um buffer com a string iniciada por **"Add"** que contém o pid do processo, o nome do programa e o tempo imediatamente antes do início da execução do programa. O servidor, por sua vez abre o fifo **CLIENT\_TO\_SERVER** para leitura e quando verifica que o conteúdo é iniciado por **"Add"**, cria uma instância da estrutura **PEDIDOSEXECUCAO** e adiciona cada um dos valores aos campos respetivos, adicionando a instância criada ao array **array\_processos\_running** e incrementa o valor do array. No lado do cliente, após o envio ao servidor das informações sobre o pedido prestes a ser executado, o cliente notifica ainda o utilizador, via standard output, de que está a correr o programa com o PID que lhe foi atribuído, e efetua a execução do programa, utilizando o **execvp**. Posteriormente à execução do programa, o cliente calcula o tempo atual, de forma a determinar o tempo no final da execução. Com este valor o cliente:

- envia ao servidor, através do fifo **CLIENT\_TO\_SERVER** um buffer que contém a string **"Ended in"** o PID do processo que terminou e o timestamp no final da execução. O servidor quando verifica que o conteúdo presente no fifo **CLIENT\_TO\_SERVER** é iniciado por **"Ended in"**, faz parsing do comando recebido e guarda o pid recebido como pid do processo a remover do array **array\_processos\_running**. Desta forma, o servidor percorre o array, procurando o elemento **PEDIDOSEMEXECUCAO** do array que contém o pid em questão e remove-o invocando a função *remove\_from\_processos\_running*, decrementando, também, o número de elementos do array.
- notifica o utilizador, via standard output, do tempo de execução que o programa demorou.

Por fim fecha-se o fifo **CLIENT\_TO\_SERVER**.

## Consulta de programas em execução

Nesta funcionalidade, o nosso programa cliente (tracer) recebe do utilizador o seguinte argumento da linha de comandos: `./tracer status`. Aquando da receção deste comando, o cliente abre o fifo `CLIENT_TO_SERVER` e envia um buffer que contém uma string iniciada por "status" e seguida do nome dado ao **fifo de resposta do servidor para o cliente** (que é basicamente `tmp/valor do pid do processo`). Isto, para que o servidor perante múltiplos pedidos saiba exatamente para onde enviar a resposta. Do lado do servidor, após a abertura do fifo do `CLIENT_TO_SERVER` para leitura, ao verificar que o cliente enviou um status, o servidor percorre o array **array\_processos\_running** e para cada um dos `PROCESSOSEMEXEUCAO` lá presentes o servidor guarda o seu pid, nome do programa dentro de um buffer, juntamente com tempo de execução do processo calculado após efetuar a diferença entre o tempo inicialmente guardado quando o cliente informou o servidor de que iria executar o processo em questão e o tempo atual. Caso o número de elementos do array seja 0, o servidor envia ao cliente, através do nome do fifo fornecido pelo cliente, uma mensagem que informa que não existem processos a correr. Caso contrário, o cliente receberá a lista de processos em execução.

## Funcionalidades Avançadas

De entre as funcionalidades avançadas propostas pela equipa docente, o grupo foi capaz de executar a seguinte funcionalidade:

### Armazenamento de informação sobre programas terminados

Se no arranque do servidor (monitor), este receber um argumento com o caminho para a pasta onde devem ser guardados os ficheiros que guardam a informação sobre os programas já terminados, o servidor cria um ficheiro *log* para cada processo terminado cuja designação corresponde ao pid do processo em causa. Na figura em baixo, é possível verificar que após a execução dos processos em causa, os ficheiros *log* são criados:

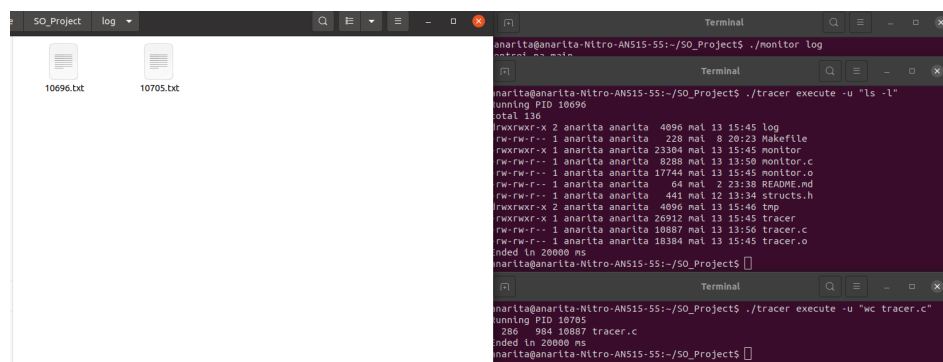


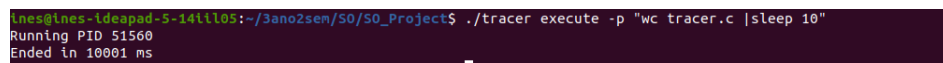
Figura 4: Criação dos ficheiros *log*

# Funcionalidades que não estão corretamente implementadas

## Execução encadeada de programas

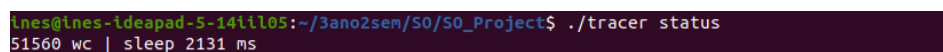
O grupo tentou implementar a funcionalidade de execução encadeada de programas, no entanto, infelizmente, esta funcionalidade não está a funcionar na sua totalidade, uma vez que nos apresenta problemas na consulta de programas em execução. Por algum motivo que o grupo não foi capaz de decifrar, após a execução desta funcionalidade, a consulta de programas do sistema apresenta um comportamento anormal e o programa deve ser reiniciado para apresentar um comportamento correto novamente.

A ideia que o grupo concebeu para a implementação desta funcionalidade foi que no *tracer*, o cliente, permite executar vários programas simultaneamente através de uma pipeline, usando a chamada de sistema *dup2()* para fazer redirecionamento de entrada e saída padrão. O programa recebe uma string de entrada(*argv[3]*) com os nomes e argumentos dos programas a serem executados em cadeia, separados por *pipes*, e então cria uma *pipeline*. Em seguida, o programa faz um *fork()* para criar processos filho que executarão cada um dos programas especificados na *pipeline*, fazendo os devidos redirecionamentos com *dup2()*. O código é finalizado com a espera pelo término dos processos filhos usando a chamada de sistema *wait()* e o *close* do descritor de arquivo associado ao *fifo* usado para se comunicar com o servidor.



```
lmes@lmes-ldeapad-5-1411l05:~/3ano2sen/S0/S0_Project$ ./tracer execute -p "wc tracer.c |sleep 10"
Running PID 51560
Ended in 10001 ms
```

Figura 5: Tentativa de execução encadeada de programas: execute -p



```
lmes@lmes-ldeapad-5-1411l05:~/3ano2sen/S0/S0_Project$ ./tracer status
51560 wc | sleep 2131 ms
```

Figura 6: Tentativa de execução encadeada de programas: status

A consulta de programas em execução não funciona após a primeira execução encadeada de programas, pelo que o programa deixa de funcionar e deve ser reiniciado.

## Exemplos de Execução

De seguida apresentamos alguns exemplos de execução de modo a testar e comprovar o funcionamento do nosso programa tendo em conta as funcionalidades pedidas no enunciado.

De forma a sermos capazes de visualizar programas como o "ls" ou "wc", que são rápidos, no status acrescentamos no código um sleep(20) antes da remoção do pedido de execução.

The image shows three terminal windows. The top-left window shows the output of a program that prints 'entrou na main', 'recebi pedido', 'add;10912;ls;1083989470555;', 'entrou added', 'added', 'recebi pedido', and 'add;10912;ls;1083989470555;'. The top-right window shows the output of a program that prints 'Running PID 10912', 'tracer tmp monitor.o tracer.c structs.h Makefile', and 'log monitor tracer.o monitor.c README.md'. The bottom window shows the output of a program that prints 'Running PID 10914', '211 702 8288 monitor.c', and 'Ended in 20000 ms'.

Figura 7: Execução de dois pedidos (com sleep(20)) enquanto verificamos a sua presença no status, usando a opção de armazenamento de informação sobre programas terminados

The image shows a terminal window with the following output: 'anarita@anarita-Nitro-AN515-55:~/SO\_Project\$ ./tracer execute -u "ls -u"', 'Running PID 10912', 'tracer tmp monitor.o tracer.c structs.h Makefile', 'log monitor tracer.o monitor.c README.md', 'Ended in 20000 ms', and 'anarita@anarita-Nitro-AN515-55:~/SO\_Project\$'.

Figura 8: Execução do pedido "ls -u"terminada (com sleep(20))

The image shows a terminal window with the following output: 'anarita@anarita-Nitro-AN515-55:~/SO\_Project\$ ./tracer execute -u "wc monitor.c"', 'Running PID 10914', '211 702 8288 monitor.c', 'Ended in 20000 ms', and 'anarita@anarita-Nitro-AN515-55:~/SO\_Project\$'.

Figura 9: Execução do pedido "wc monitor.c"terminada (com sleep(20))

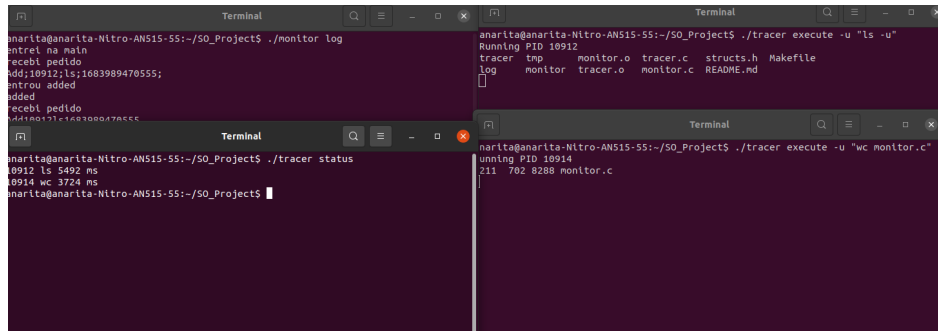


Figura 10: Execução de dois pedidos enquanto verificamos a sua presença no status, usando a opção de armazenamento de informação sobre programas terminados

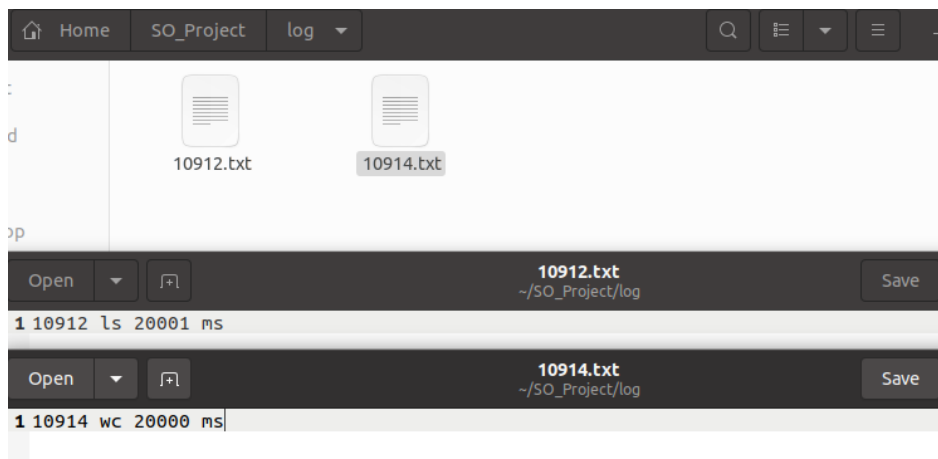


Figura 11: Verificação a informação sobre programas terminados foi armazenada

## Makefile do programa

Na Makefile do programa procedemos à criação das diretorias *tmp* (onde são armazenados os *fifos* do programa) e *log* (onde armazenamos a informação dos programas já executados) e procedemos à criação dos executáveis do projeto. Por fim quando efetuamos *make clean* as diretorias criadas são eliminadas, assim como os executáveis.



# Conclusão

Por fim, consideramos que, apesar de o grupo ter ficado apenas com 2 elementos dada a desistência do terceiro elemento do grupo, fomos capazes de desenvolver, com sucesso, as funcionalidades básicas propostas e a funcionalidade avançada de armazenamento de informação sobre programas terminados, cumprindo todos os requisitos para elas propostos. O grupo tentou ainda implementar a funcionalidade de execução encadeada de programas, que apesar de estar presente no código não se encontra totalmente funcional. Neste sentido, através deste trabalho prático fomos capazes de consolidar os conceitos adquiridos desde o início do semestre, em especial os conceitos de concorrência, escalonamento de processos, ficheiros, entre outros.