Sistemas Operativos

Trabalho Prático

Rastreamento e Monitorização da Execução de Programas

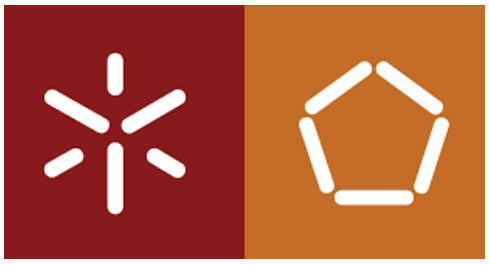
LEI - 2022/2023

Maurício Pereira
A95338

Carlos Costa
A88551

Guilherme Gonçalves
A88280

Grupo 11



Universidade do Minho

Índice

1 Introdução

O trabalho prático de Sistemas Operativos, ***Tracer***, consiste na implementação de um serviço que permita monitorizar os programas em execução numa máquina. O nosso programa deve permitir que os utilizadores executem ***programas externos*** através do cliente (tracer) e possam saber o tempo que demoraram a executar, desde o momento em que solicitaram ao cliente a execução do programa até ao fim.

Além disso, um cliente também deve poder consultar todos os programas que estão ***atualmente em execução*** e saber quanto tempo cada um está a demorar. Isto ajuda a gerir melhor os recursos da máquina e a identificar possíveis problemas de desempenho.

O cliente (tracer) e o servidor (monitor) comunicam através de dois pipes com nome, ***FIFO´s***. No caso dos pedidos que envolvém informação sobre outros comandos, o cliente envia o pedido e o servidor responde com o output pretendido. No caso de ser um pedido para executar, o cliente informa o servidor que o comando se iniciou e, posteriormente, termiou.

2 Funcionalidades

Os objetivos e funcionalidades deste trabalho podem ser divididos em duas categorias: as funcionalidades básicas e as funcionalidades avançadas. As funcionalidades básicas envolvem tanto o servidor quanto o cliente e incluem os objetivos já descritos anteriormente.

Em primeiro lugar, o servidor deve receber um argumento através da linha de comando que corresponde à diretoria onde as informações sobre os comandos terminados serão armazenadas.

```
1   Por exemplo:
2   $ ./monitor cache
```

2.1 Help

Opção ***help*** fornece ao utilizador todas as informações de utilização.

```
❯ ./tracer help
usage: ./tracer [option] commands(or pids) arguments(or not)
Submit jobs to be executed.
Options:
execute -u    : submit a job to execute command, requires command name and arguments(./tracer execute -u ...)
status        : display a status message containing the status of the server (./tracer status)
help          : display this message (./tracer help)
stats-time    : give the total execution time of the jobs you want(./tracer stats-time [pids])
stats-command : give the total execution times of command in the jobs you want(./tracer stats-command cmd [pids])
stats-uniq    : give the commands that the jobs execute(./tracer stats-uniq [pids])
❯
```

Figura 1: Help

2.2 Execução de programas do utilizador

O cliente se pretende executar apenas um comando deve fornecer a opção ***execute***, a flag ***-u*** e de seguida, o comando e os seus argumentos, se os tiver.

```
1   Por exemplo:
2   $ ./tracer execute -u ls -l
```

2.3 Consulta de programas em execução

Para sabermos os programas que estão atualmente a ser executados e quanto tempo estão a demorar, recorremos à opção ***status***.

```
1   Por exemplo:
2   $ ./tracer status
```

2.4 Consulta de programas terminados

Através da opção ***stats-time***, recolhemos o tempo total (em milissegundos) utilizado por um dado conjunto de programas identificados por uma lista de **PIDs** passada como argumento.

```
1   Por exemplo:
2   $ ./tracer stats-time 88218 88251 88349
```

Através do comando ***stats-uniq***, recolhemos a lista de nomes de programas diferentes (únicos), um por linha, contidos na lista de **PIDs** passada como argumento.

```
1   Por exemplo:
2   $ ./tracer stats-uniq 88218 88251 88349
```

Através da opção ***stats-command***, recolhemos o número total de vezes que foi executado um certo programa, cujo nome é passado como argumento, para um dado conjunto de **PIDs**, também passado como argumento.

```
1   Por exemplo:
2   $ ./tracer stats-command ls 88218 88251 88349
```

3 Arquitetura da Aplicação

Como já referido, o programa é composto pelo cliente e pelo servidor. O cliente comunica com o servidor e envia-lhe pedidos, enquanto o servidor fica à espera de novos pedidos, lê-os e processa-os.

3.1 O cliente

Um aspeto importante que tem de ser analisado é o facto de, para comunicar entre cliente e servidor, pode ser usado apenas um Pipe com nome. Porém, para comunicar entre servidor e cada cliente, é necessário usar um *FIFO* específico para cada cliente. Mas como garantir a sua **unicidade**?

Para garantir que cada ligação através de pipe com nome seja única, devemos usar o identificador do processo do cliente atual. Assim, para cada cliente, será criado um *FIFO* próprio. A criação do *FIFO* entre o servidor e o cliente é feita no cliente e, quando é enviado um pedido ao servidor, o seu nome também é enviado. Deste modo, garantimos que cada cliente tem um canal de comuniação próprio e que o servidor sabe sempre por onde enviar mensagens para comunicar com cada cliente.

Após a abertura do *FIFO* do cliente-servidor (cts) e a criação do *FIFO* do servidor-cliente (stc), podemos começar a formatar a mensagem. A mensagem a enviar ao servidor será uma *Response struct*. Esta mensagem contém o identificador do processo, o nome do comando executado, o tempo de início ou o tempo de fim. Em seguida, a mensagem é enviada ao servidor e entramos em estado de espera.

O estado de espera, nada mais é,que uma chamada à função read(), ficando a aguardar a resposta do servidor.

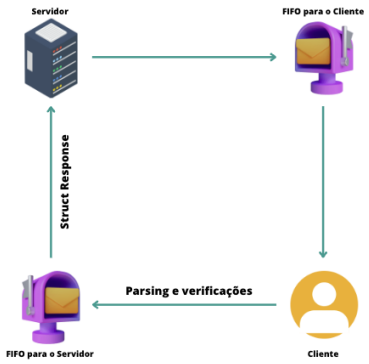


Figura 2: Arquitetura do Cliente

3.2 O servidor

Ao contrário do cliente, esta parte do programa é responsável por receber pedidos, determinar o tipo do pedido e executá-lo. Existem três tipos de pedidos: **Execute**, **Status**, **Stats-(...)** e **Help**.

Quando o servidor recebe um pedido, realiza uma análise inicial da mensagem para determinar o seu tipo.

Se for um pedido de **Help**, é enviado uma mensagem de ajuda com todas as funcionalidades do programa, através de um *FIFO* único para cada cliente, fornecido na mensagem.

O processo **"Execute"** é responsável pela manipulação da Queue. Ele verifica o tipo de comando execute e, se for o início de um comando, adiciona-o à Queue. Se for o fim de um comando, remove-o da Queue e determina o tempo total de execução. Neste momento, o servidor armazena um arquivo com as informações sobre o comando que acaba de ser concluído, na diretoria especificada durante a execução do servidor.

Se for um pedido de **Status**, é realizada uma conversão para string do estado atual da Queue.

Os pedidos **Stats-uniq**, **Stats-command** e **Stats-time** funcionam de forma semelhante. Eles acessam os arquivos com base nos identificadores fornecidos no pedido e executam as operações necessárias em cada um deles.

Essas informações são mais claras na Figura 3, anexada abaixo.

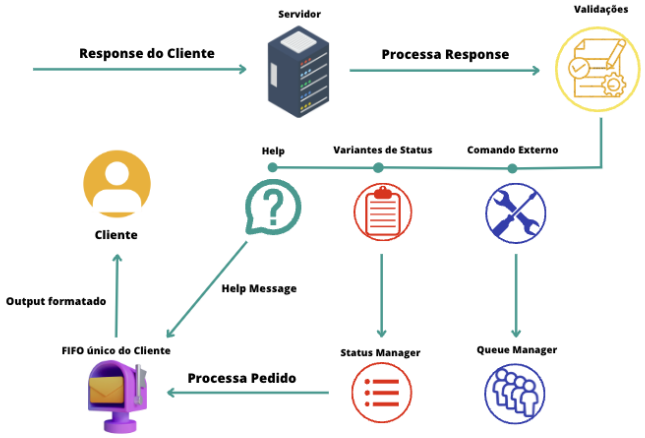


Figura 3: Arquitetura do Servidor

4 Testes

Apresentamos agora alguns dos testes manuais que fizemos para testar o funcionamento do nosso programa. Nas imagens abaixo, se não houver indicação em contrário, podem se observar a execução de comandos e os respectivos resultados(estes encontram-se em imagens sequênciais ou na mesma).

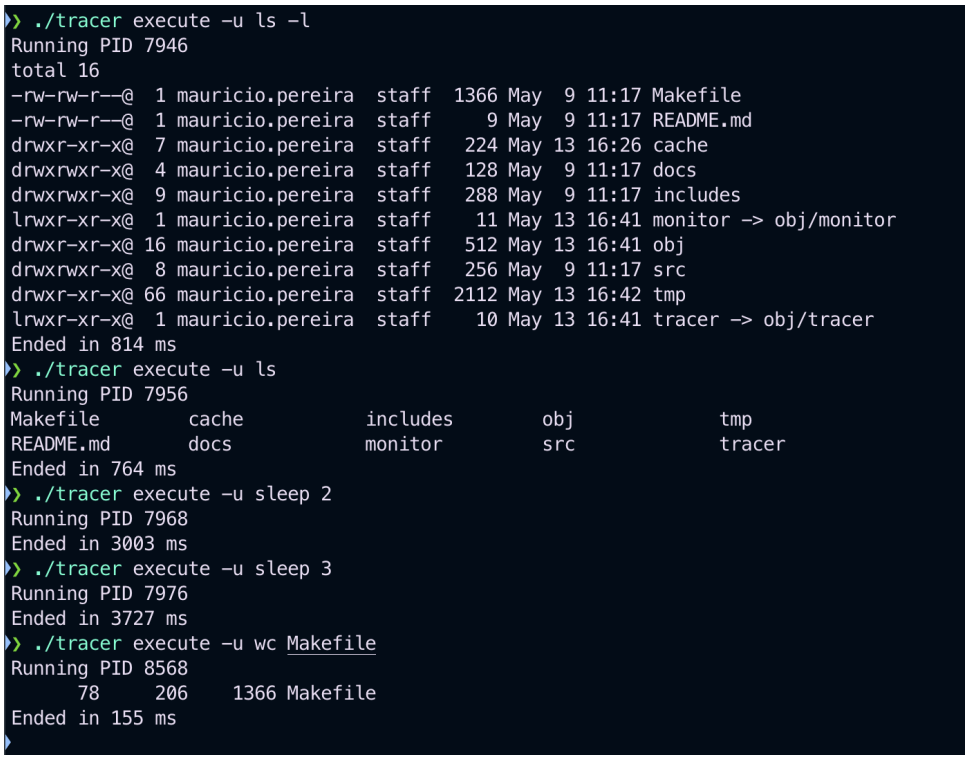


Figura 4: Execute de programas com e sem argumentos

Figura 5: Status

Nas imagens abaixo expostas, podemos observar a excução das diferentes funções stat e os respetivos resultados da chamada das mesmas, com a complementação dos ficheiros que foram gerados em resposta às mesmas.

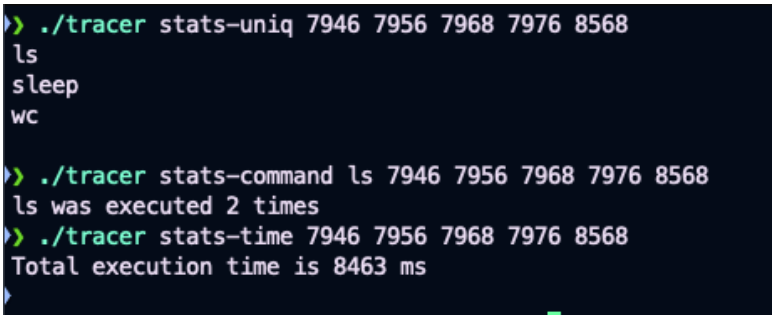


Figura 6: Comandos Stats

Figura 7: Ficheiros Utilizados

5 Problemas

5.1 Fluxo de Respostas

O primeiro problema encontrado esteve relacionado com o que enviariamos ao servidor. Inicialmente pensamos em enviar uma string formatada através do **FIFO**, mas deparamos-nos com *excessivos parsings* e formatações de *strings*. Optamos então pelo fluxo atualemente implementado, uma *struct Response*. Esta serve para todo o tipo de mensagens que enviamos para o servidor, ou seja, está generalizada.

```
typedef struct response
{
    int pid;
    char cmd[64];
    char pids[64];
    struct timeval start;
    struct timeval end;
    int flag;
    unsigned long final_time;
    char fifo[64];
} Response;
```

5.2 Tempo de execução

O segundo problema encontrado foi na forma como calcularíamos o tempo final de execução do programa no cliente. Optamos por utilizar um pipe anônimo para permitir a comunicação entre o processo filho e o processo pai, possibilitando obter o tempo de início do comando.

5.3 Comunicação com "aquele"Cliente

O terceiro problema encontrado, conforme mencionado anteriormente, foi a necessidade de estabelecer um canal de comunicação separado entre o servidor e cada cliente individualmente. Cada cliente deve possuir o seu próprio FIFO para receber os dados enviados pelo servidor.

5.4 Concurrência

O último problema está relacionado com o facto de que o servidor deveria suportar, sempre que possível, o processamento concorrente de pedidos, evitando que os clientes que realizam pedidos que exigem maior tempo de processamento possam bloquear a interação de outros clientes com o servidor. Percebemos que este era um problema grave pouco tempo antes da data limite de entrega (13/09/2023 19:37). Sabemos como resolvê-lo: basta converter os campos necessários da estrutura "Response"para uma string usando o sprintf para formatá-la, enviar a string para o servidor e, posteriormente, recriar a estrutura com os campos presentes na string. Poderiamos usar a flag como o primeiro elemento da string para tornar a reconstrução mais eficiente, pois com a flag saberiamos exatamente que campos teriamos dentro daquela *string*.

6 Conclusão

Como é possível constatar, o nosso trabalho possui todas as funcionalidades básicas pedidas a funcionar perfeitamente, para além da maioria das funcionalidades adicionais/avancadas, com a exceção do último problema que devido ao facto de ter sido apenas encontrado encima da data de entrega, este não foi corrigido a tempo. Apesar disso, acreditamos que o nosso trabalho encontra-se sólido, mesmo com todos os problemas que encontramos ao longo do seu desenvolvimento(referidos no capitulo anterior). Este trabalho permitiu-nos consolidar a informação que fomos adquirindo na cadeira de Sistemas Operativos.