



Universidade do Minho

UNIVERSIDADE DO MINHO

LICENCIATURA EM CIÊNCIAS DA COMPUTAÇÃO

Rastreamento e Monitorização da Execução de
Programas

Grupo 25

João Luís da Cruz Pereira (a95375)

Maria Helena Alves Machado Marques Salazar (a75635)

Miguel António Vaz Pinto Pereira (a72161)

13 de maio de 2023

Conteúdo

1	Introdução	3
2	Funcionalidades Básicas	4
2.1	Execução de programas por parte do utilizador	4
2.1.1	Novo Programa do Utilizador e Respectiva Terminação	5
2.1.2	Consulta dos Programas em Execução	6
3	Funcionalidades Avançadas	7
3.1	Execução Encadeada de Programas	7
3.1.1	Primeiro Caso	7
3.1.2	Segundo Caso	7
3.1.3	Terceiro Caso	8
3.2	Armazenamento de Informação sobre Programa Terminados	8
3.3	Consulta de Programas Terminados	9
3.3.1	Stats-Time	9
3.3.2	Stats-Command	9
3.3.3	Status-Uniq	10
4	Conclusão	11

Capítulo 1

Introdução

Através deste projeto, pretendemos implementar um serviço de monitorização eficaz de um ou vários programas.

A monitorização e rastreio da execução de programas são aspetos críticos do desenvolvimento e manutenção de software, que consistem na observação do comportamento do programa para identificar erros, problemas de desempenho, entre outros. Uma das formas eficazes de rastrear a execução de programas é através de chamadas de sistema, que possibilitam ao programa solicitar serviços ao **kernel** do sistema operacional e interagir com o sistema operativo subjacente de forma segura e padronizada.

Neste projeto serão utilizados serão usados pipes e **FIFOS** (pipes com nome) para implementar uma variedade de funcionalidades que permitem o fluxo de dados entre diferentes processos num sistema operativo, permitindo uma comunicação eficiente e em tempo real entre programas. No caso deste projeto, esta comunicação será vital para o bom funcionamento entre o cliente e o servidor, bem como entre os vários processos filhos a executar (*por vezes, em simultâneo*).

O objetivo deste projeto é concretizar uma comunicação sólida e segura entre um possível cliente e servidor, usando pipes, chamadas de sistema e uma variedade de outras técnicas para monitorizar, rastrear e guardar a informação relativa ao programas em execução.

Espera-se que no término do desenvolvimento feito pelo grupo, todas as funcionalidades (*básicas e avançadas*) estejam corretamente implementadas e toda a conexão esteja operacional e segura.

Capítulo 2

Funcionalidades Básicas

2.1 Execução de programas por parte do utilizador

Em geral,

Para prosseguir com a execução do programa em questão, definiu-se uma função `execute` onde se processa a inicialização com um **fork** (*para permitir a execução sem um término prévio do programa*) e o comando procede com os respectivos argumentos.

Foram também escritas mensagens de erro caso aconteçam durante o correr do código, desde a impossibilidade de criação de um processo filho à execução através do **execvp**.

Os argumentos em questão foram separados por uma função `tokenizer` e durante todo o processamento de dados do código será maioritariamente suportado por uma struct `prog`. Seguem-se então breves explicações sobre a construção e funcionamento destes.

Tokenizer

Derivado da criação da funcionalidade **execute -u**, onde será necessário analisar o conteúdo da string fornecida pelo utilizador ao cliente, surgiu a necessidade da criação de um sistema auxiliar que nos permita dividir esta em seus vários componentes (*cujo parâmetro de divisão seja o caracter espaço*).

Com isto em mente, recorreu-se a uma função `tokenizer`, aonde se dá a separação do input introduzido pelo utilizador em tokens de formato string (*que posteriormente serão armazenados array*), o que então possibilita o uso do conteúdo do programa em si – o seu comando e respetivos argumentos – em outras funcionalidades do projeto.

Struct PROG

Para a transição de informação entre o cliente e o servidor de um programa (*entre os pipes e, por continuidade, as pipelines*), procedeu-se a criação de uma struct que armazena todos os dados aos quais pretendemos trabalhar com; desde o comando em si, argumentos associados, o tempo de iniciação do processo e respetivo tempo de execução. Deste modo, haverá menos perda de informação e esta estará organizada de uma maneira mas sucinta.

2.1.1 Novo Programa do Utilizador e Respectiva Terminação

Cliente: A funcionalidade básica do projeto, o **execute**, procede com a cálculo de um processo filho que estará conectado a um pipe que servirá para futuro intercâmbio de informação com os processos a correr. Prossegue-se então para a execução os comandos que serão introduzidos pelo utilizador seguido da função. Com a função **gettimeofday**, procedemos a atribuição do tempo inicial ao qual um programa começa a executar e passa-se essa informação através de uma struct previamente criada a fim de guardar o **PID** (*do filho em causa*) e esse mesmo tempo para o processo pai através de um pipe anónimo e ao servidor através de um **FIFO**.

Neste processo, também será enviado o nome do comando que foi executado com sucesso. Assim que o processo filho termina a sua execução, o processo pai (que estava em modo **WAIT**, ou seja, em modo espera) chama também a função **gettimeofday** para obter o tempo de término e com a informação recebida do filho, este calcula a diferença entre ambos – isto é, o tempo que o programa demorou a executar. Essa duração em milissegundos juntamente com o **PID** do processo filho que acabou de concluir serão então enviados para o servidor por **FIFO**.

Durante o correr das introduções deste pedaço de código, o utilizador é informado do **PID** do programa que está a correr e no final, o tempo de execução do mesmo. O utilizador também é informado de vários erros que posso acontecer durante a execução.

Store

No seguimento de obtenção de uma melhor dinâmica de consulta de programas ainda em execução (*e suas respectivas informações*, decidiu-se proceder a criação de um array chamado store que armazena em cada posição uma **struct prog** com esses mesmo dados. Sendo assim, torna-se uma das peças chave no código do projeto.

Server Block

Para manter o servidor a correr, foi aberto um descritor de escrita que se mantém bloqueado enquanto não houver descritores de leitura abertos, mantendo assim o servidor a correr para "sempre".

Kill

Para evitar possíveis consequências (*como o não fazer unlink e close*) com o fechar do servidor usando o **CTRL + C**, decidiu-se criar o **comando kill** para que este ponha um fim ao estado acima definido como **Server Block**.

Este comando interrompe o ciclo de leitura com um break, obrigando o processo a entrar na fase de encerramento dos descritores e unlink dos pipes.

Servidor: Para simplificar o processamento de informação dos vários programas que estão a execução, criou-se um array store que vai armazenar em cada índice struct prog. Tendo este array, inicializamos a leitura do **PID** enviado e vai consultar se este já existe dentro de algum índice do store. Se encontrado, quer dizer que o programa em questão já terminou; se não encontrar, armazena-o.

```
joao@bin (main) $ ./tracer execute -u "ls -l"
Running PID 9276
total 60
-rwxrwxr-x 1 joao joao 26456 mai 12 18:06 monitor
-rw-rw-r-- 1 joao joao    0 mai 12 18:24 pipe_to_client
-rw-rw-r-- 1 joao joao    0 mai 12 18:43 pipe_to_server
-rwxrwxr-x 1 joao joao 32064 mai 12 17:55 tracer
Ended in 11 ms
```

Figura 2.1: Demonstração da execução do execute-u.

2.1.2 Consulta dos Programas em Execução

A função `status` quais são os programas que estão a executar e o seu tempo até ao momento (milisegundos). Os resultados serão apresentados ao utilizador pelo cliente, listados um por linha.

Cliente: Quando é lido o comando `status`, abre-se um pipe onde a informação passada por este será feita em **struct prog**¹ onde seguirá o comando 'status' para o servidor então processar o código associado a esta função. Dado como terminado o reconhecimento dos programas a executar pelo lado do servidor, o cliente lê toda a data enviada e imprime no standard output o **PID**, o nome do programa e o seu tempo de execução até ao momento.

Servidor: Foi pedido ao servidor o tempo de execução do programa até ao momento e para tal, usa-se a função `gettimeofday` (como visto previamente). Procedemos com a abertura de um pipe para o cliente onde será enviada a informação pedida pelo mesmo, desta vez só usando **strings**². Com o array store passado na função, é feito o cálculo dos vários programas armazenados em struct, sendo depois formatado numa string e enviado pelo pipe acima criado. Depois do ciclo ser dado como terminado, o processo é encerrado com o encerramento desse mesmo pipe e o cliente terá acesso a informação pedida, se nenhum erro ocorrer.

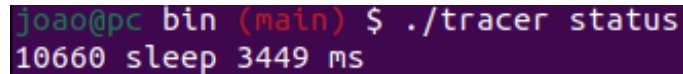
A terminal window with a dark background and light-colored text. The prompt is 'joao@pc bin (main) \$'. The command entered is './tracer status'. The output is '10660 sleep 3449 ms'.

Figura 2.2: Demonstração da execução do `status`.

¹Neste caso, decidiu-se comunicar com o servidor pois este está sempre com um processo de leitura direcionado à informação no formato da struct prog.

²Quando estabelecidas conexões do servidor para o cliente, o pipe não está dependente de structs. Então, com isto em mente, procurou-se usar strings para o envio desta mesma informação pois facilita o processo em causa.

Capítulo 3

Funcionalidades Avançadas

3.1 Execução Encadeada de Programas

Cliente: Inicialmente abre-se um **FIFO** onde irá ser escrito, tal como no resto das funcionalidades deste programa, uma **struct prog** que guarda o **PID** do pipeline (*no caso, **PID** do processo pai*), os comandos (*provenientes do utilizador*), e o ‘start’ (*o valor resultante da função `gettimeofday`*). Consecutivamente, envia-se a informação guardada para o servidor pelo pipe. Verificado o envio, o utilizador recebe notificação sobre o ID do processo da pipeline em execução - **PID** do processo pai.

Para executar as funções solicitadas pelo utilizador, o grupo implementou um loop que itera sobre um array de arrays de tamanho dois, onde será criado um pipe em cada posição do array usando esse mesmo par. Para uma melhor organização do código, o loop é dividido em três casos distintos.

3.1.1 Primeiro Caso

No primeiro caso, ou seja, no caso de ser o primeiro programa, é criado um pipe que estabelece comunicação entre este e o segundo programa. Para tal, é necessário alterar o file descriptor 1 para o file descriptor de escrita do pipe, de forma a permitir que o output do primeiro comando seja redirecionado para o próximo pipe em vez de ser escrito diretamente no **STDOUT**.

Após a criação do processo filho, este é responsável pela execução do comando fornecido enquanto o processo pai continua a executar o loop. No entanto, caso ocorra algum erro durante a execução do comando pelo processo filho, a substituição completa do código deste pela chamada ao sistema **execvp** não ocorrerá, e o programa terminará.

3.1.2 Segundo Caso

No segundo caso, isto é, no último programa da pipeline, o processo pai criará um novo filho, seguindo um procedimento semelhante ao anteriormente descrito. A diferença reside no facto de que, ao contrário do primeiro caso, o descritor de leitura deve ser alterado em vez do descritor de escrita, uma vez que o objetivo é ler o output do pipe em vez do **STDIN**. Para esse fim, a função **dup2()** é novamente utilizada.

Neste caso, é mantido o file descriptor de escrita para o **STDOUT** pois estamos perante o último programa a ser executado.

3.1.3 Terceiro Caso

No terceiro caso, ou seja, para os programas intermédios, é necessário alterar ambos os descritores de ficheiros, uma vez que não se pretende que este pipe leia do **STDIN** ou escreva no **STDOUT**, mas sim no descritor de leitura do pipe anterior e no descritor de escrita do pipe atual, respetivamente. Para realizar esta operação, é necessário utilizar a system call **dup2** duas vezes, para alterar os descritores de ficheiros adequadamente.

Deve-se salientar que é necessário fechar os file descriptors que não serão mais utilizados no processo pai. Quando se faz **fork**, é obtida uma cópia dos mesmos, e se não forem mais utilizados, devem ser fechados para evitar vazamento de memória. O mesmo deve ser feito nos processos filhos, fechando os descritores após o uso, para o bom funcionamento do programa. Finalmente, o processo pai espera que todos os processos filhos terminem através de um loop de **wait()**.

O código responsável pela execução da pipeline calcula também o tempo de execução total da mesma, utilizando o tempo atual e o tempo inicial armazenado na variável **start**. A diferença entre ambos é calculada em microssegundos e guardada na **struct prog**, juntamente com o **PID** da pipeline. Esta struct é então enviada para o servidor.

```
joao@pc bin (main) $ ./tracer execute -p "grep -v ^# /etc/passwd | cut -f7 -d: | uniq | wc -l"
Running PID 11536
13
Ended in 3 ms
```

Figura 3.1: Demonstração da execução da pipeline.

3.2 Armazenamento de Informação sobre Programa Terminados

Cliente: Quando o utilizador fornece ao cliente uma das duas opções de execução, este prepara a informação necessária numa **struct prog** para o servidor avaliar o rumo a tomar. Finalizando o processo, é enviada outra struct com o **PID** e o tempo de execução.

Servidor: Se o servidor não corresponder a nenhuma opção (*status*, *stats-x* e *kill*), então vai usar o **PID** recebido previamente do cliente para verificar se este é um programa novo ou é um programa que terminou (*através de comparações feitas com as posições do array store*).

Tendo o programa encontrado o seu término, é criado um ficheiro na diretoria fornecida pelo utilizador ao executar o monitor. Este ficheiro contém a informação fornecida através da **struct**, isto é, os comandos e o tempo de execução e cujo nome é o **PID** do programa em questão.

```
joao@pc Projeto-S0 (main) $ ls PIDS-folder/
10019.txt 10287.txt 11076.txt 11233.txt 22712.txt 6759.txt 7912.txt
10029.txt 10321.txt 11088.txt 11243.txt 25285.txt 6766.txt 7919.txt
10036.txt 10361.txt 11105.txt 11255.txt 25445.txt 6773.txt 7966.txt
10043.txt 10384.txt 11121.txt 11485.txt 25467.txt 7046.txt 8023.txt
10050.txt 10523.txt 11133.txt 11520.txt 25745.txt 7471.txt 8358.txt
10057.txt 10625.txt 11143.txt 11533.txt 30929.txt 7478.txt 8371.txt
10064.txt 10660.txt 11155.txt 11536.txt 31043.txt 7498.txt 8925.txt
10071.txt 10704.txt 11166.txt 11540.txt 32210.txt 7599.txt 9049.txt
10078.txt 10760.txt 11177.txt 12763.txt 6507.txt 7668.txt 9225.txt
10085.txt 10783.txt 11187.txt 20237.txt 6529.txt 7719.txt 9241.txt
10092.txt 10815.txt 11198.txt 20352.txt 6738.txt 7767.txt 9276.txt
10125.txt 10849.txt 11209.txt 20397.txt 6745.txt 7842.txt 9907.txt
10200.txt 11045.txt 11219.txt 22104.txt 6752.txt 7882.txt
```

Figura 3.2: Demonstração da funcionalidade e respetivos ficheiros.

3.3 Consulta de Programas Terminados

3.3.1 Stats-Time

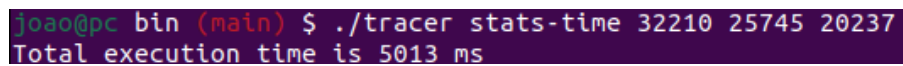
Cliente: Na funcionalidade stats-time, o utilizador passa ao cliente uma série de **PIDs** onde quer que seja calculado o tempo total de execução nos programas dos mesmos. No lado do cliente é verificado a opção e é passado ao servidor uma struct com o comando “stats-time” e no campo args seguem os **PIDs** passados pelo utilizador.

Todo o processo é efetuado no servidor e no final, o cliente recebe a string previamente formatada através de um **FIFO**, optando assim por uma solução mais simples em vez de voltar-se a utilizar a struct.

Servidor: É enviado ao servidor uma **struct prog** e essa é lida, entrando na opção stats-time onde a função **status_time** será chamada para execução.

Com isto, percorre-se os **PIDs** fornecidos, onde segue a abertura do ficheiro correspondente a cada um deles. A leitura é efetuada byte a byte para se conseguir encontrar o final da linha, sabendo assim quando termina o valor em milisegundos do seu tempo de execução (*que se encontra na primeira linha*) e adicionar ao total. Quando a primeira linha termina, o ciclo é interrompido pois já foi obtida a informação pedida.

No final de percorrer todos os ficheiros dos **PIDs**, passa-se o valor total em formato string ao cliente que por sua vez mostra a mesma recebida ao utilizador.



```
joao@pc bin (main) $ ./tracer stats-time 32210 25745 20237
Total execution time is 5013 ms
```

Figura 3.3: Demonstração da execução da stats-time.

3.3.2 Stats-Command

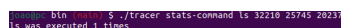
A funcionalidade **stats-command** partilha uma certa similaridade com a funcionalidade **stats-time**. Porém, a nova recebe mais um argumento para a sua execução proceder conforme o esperado.

Cliente: É passado ao servidor o comando **stats-command** na **struct prog** no seu respetivo campo, enquanto que no campo **args**, na primeira posição do array vai o comando a ser procurado enquanto que no resto do array segue os **PIDs** onde queremos procurar o comando em questão.

Terminada a execução por parte do servidor, o cliente recebe uma string formatada para mostrar ao utilizador o resultado pretendido.

Servidor: Recebida a **struct prog** necessária, este na opção correta e chama a respetiva função. Em seguida, repete-se o processo descrito em cima em relação **PIDs**, abrindo os respetivos ficheiros. A leitura é na mesma efetuada byte a byte, porém o processo não termina na primeira linha pois é sempre comparado o valor lido nessa linha com o comando pretendido (*na primeira linha lê o valor em milisegundos e compara com o comando na mesma, e certamente que não é obtida nenhuma igualdade*).

Sendo o valor lido equivalente ao comando, é incrementado total o número de vezes quantas aquelas o comando em si é repetido nos **PIDs** previamente enviados. Lido o último **PID**, a informação obtida é então formatada numa string e de novo, através de um **FIFO** é enviada ao cliente.



```
bin $ ./tracer stats-command ls 32210 25745 20237
ls was executed 1 times
```

Figura 3.4: Demonstração da execução da stats-command.

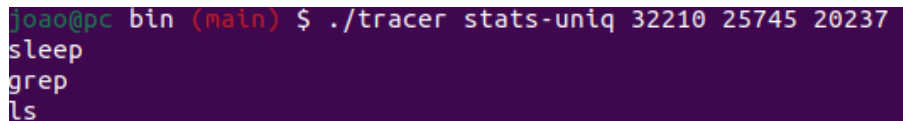
3.3.3 Status-Uniq

No caso do stats-uniq, a similaridade é maior com a primeira funcionalidade descrita na Consulta de Programas Terminados, visto que o campo *args* da **struct prog** só leva os **PIDs** e no campo *cmd* leva o **stats-uniq**.

Cliente: Como descrito em cima, é enviado ao servidor uma **struct prog** com as informações pedidas para o servidor atuar sobre a função definida como **stats-uniq**. Visado o término da função no servidor, o cliente recebe as strings dos comandos executados separados por um `\n` e apresenta-os ao utilizador.

Servidor: Semelhante as outras duas funções acima descritas, serão percorridos os **PIDs** fornecidos pelo cliente, abrir os ficheiros correspondentes e verificar os comandos guardados em cada um deles. Com a ajuda de um array **store**, onde serão armazenados os comandos dos **PIDs** lidos, serão introduzidos os comandos uma única vez (mesmo havendo repetição). A leitura para obtenção do comando é de novo efetuada byte a byte, ignorando a primeira linha pois esta está reservada para o tempo de execução.

No caso de o **store** estar vazio, adiciona-se imediatamente o comando em questão; senão tem-se que percorrer o array para garantir que esse comando já não está armazenado numa posição. Com o término da leitura de **PIDs**, é passado ao cliente os comandos guardados (*depois de convertidos e formatados em string*), um a um, através de um **FIFO**.

A terminal window with a dark purple background. The prompt is 'joao@pc bin (main) \$'. The command './tracer stats-uniq 32210 25745 20237' has been entered. Below the prompt, the output shows three lines of commands: 'sleep', 'grep', and 'ls', each on a new line.

```
joao@pc bin (main) $ ./tracer stats-uniq 32210 25745 20237
sleep
grep
ls
```

Figura 3.5: Demonstração da execução da stats-uniq.

Capítulo 4

Conclusão

Com o desenvolvimento do projeto, conseguimos cumprir todas as funcionalidades básicas e avançadas propostas. Segundo os testes que nós efetuamos, não encontramos falhas nas funcionalidades mas as mesmas podem existir de qualquer das formas.

Em termos de melhorias possíveis, podíamos tentar com que quando o utilizador por exemplo executa `./tracer execute -u ls` o programa retornasse erro pois não colocou o programa a executar dentro de aspas. Existe tratamento de erros para quando é colocado `./tracer execute -u ls -l` devido ao número errado de argumentos mas com o número correto não.

Outro problema é nas funções stats, no caso de o utilizador não inserir PIDs válidos aparece um erro no monitor (que continua a correr) mas no lado do tracer o utilizador ter que terminar com o programa à bruta, uma possível solução para isto seria do lado do monitor escrever a mensagem de erro no pipe fazendo assim com que o programa desbloqueasse e termina-se normalmente. Com mais algum tempo implementaríamos isto.

A grande maioria dos problemas que tivemos foi mais com C do que propriamente com a comunicação entre programas/ficheiros, com a exceção de algumas leituras dos pipes que acabaram por ser resolvidas.

Foi um trabalho interessante que envolveu toda a matéria usada nas aulas práticas e que apesar de desafiador foi também muito produtivo, pois para além de aprendermos mais sobre a cadeira de *Sistemas Operativos* também aprendemos mais sobre *Git/Github*, *LaTeX/Markdown* e mesmo a linguagem *C* em si.