Sistemas Operativos - Trabalho Prático

Miguel Ângelo Sousa Martins[1 [A89584]] and Pedro Marcelo Bogas Oliveira[1 [A95076]]

Universidade do Minho, R. da Universidade 4710-057 Braga, Portugal

1 Introdução

O presente documento serve de relatório sobre os resultados obtidos pelo grupo 22 durante a realização do trabalho prático da unidade curricular de sistemas operativos do curso de Licenciatura em Engenharia Informática da Universidade do Minho no ano letivo de 2022/2023.

Nas secções seguintes é exposto o objetivo do trabalho, as soluções adotadas para a implementação de cada funcionalidade com a confirmação através de testes e por fim uma breve conclusão sobre os resultados obtidos e a proposta de trabalho futuro.

2 Objetivos

O objetivo deste projeto foi o desenvolvimento de duas aplicações, o *tracer* e o *monitor*. Nas subsecções seguintes vamos expor brevemente as funcionalidades básicas e avançadas requisitadas para este projeto.

2.1 Funcionalidades Básicas

Execução de programas do utilizador O *tracer* deve ser capaz de executar um programa do utilizador e comunicar ao *monitor* o estado dessa execução.

O *monitor* é apenas responsável por armazenar a informação relativa ás execuções para posterior consulta.

Na sequência da execução de um programa um utilizador deve passar como argumentos da linha de comando a opção *execute -u*, o nome do programa a executar e os diversos argumentos do mesmo no formato:

```
$ ./tracer execute -u "prog arg1 arg2 ... argN"
```

O *output* esperado via linha de comandos deve ser idêntico à execução do programa via mesma e segue o formato:

```
$ ./tracer execute -u "prog arg1 arg2 ... argN"
Running PID pid
(output do programa)
Ended in x ms
```

onde *pid* represente o PID do processo que executa o programa e x o tempo que este demora a executar.

O *tracer* deve informar o *monitor* do nome do programa a ser executado, o PID do processo a executá-lo e o *timestamp* imediatamente antes da execução do mesmo. No final da execução deve também informar o PID do programa que terminou e o *timestamp* imediatamente após a execução.

Consulta de programas em execução O *tracer* deve permitir aos utilizadores a visualização dos programas em execução.

Para tal o utilizador deve passar via linha de comando a opção *status* no formato:

```
$ ./tracer status
```

O *output* esperado conta com uma linha para cada programa ainda em execução indicando o PID do processo a executá-lo, o seu nome e o tempo decorrido até ao momento da execução da opção *status*. O formato seria então

```
$ ./tracer status
PID1 prog1 m1 ms
PID2 prog2 m2 ms
...
PIDN progN mN ms
```

Servidor O servidor, *monitor*, deve sempre que possível suportar o processamento concorrente de pedidos para evitar que pedidos com um maior tempo de execução bloqueie outros clientes do uso dos serviços providenciados.

2.2 Funcionalidades Avançadas

Execução encadeada de programas Muito semelhante à execução de programas descrita na subsecção anterior com a principal diferença nesta sendo o programa a executar ser um *pipeline* de programas.

Os argumentos a passar via linha de comando são semelhantes com a alteração da opção de execução para *execute -p* no formato:

```
$ ./tracer execute -p "prog1 arg11 arg12 ... arg1N | prog2
arg21 arg22 ... arg2N | ... | progN argN1 argN2 ... argNN"
```

O *output* esperado é o mesmo que na execução descrita anteriormente.

Armazenamento de informação sobre programas terminados O *monitor* deve ser capaz de armazenar em ficheiro informações sobre programas terminados. Cada programa deve ter um ficheiro independente. Devem ser conservados, no mínimo, o nome do programa executado e o tempo de execução total do mesmo.

Os ficheiros devem ser gerados numa pasta que deve ser indicada como argumento via linha de comandos ao *monitor* no formato:

```
$ ./monitor pids
```

onde *pids* indica o *path* para a pasta.

Consulta de programas terminados O *tracer* deve poder suportar a execução das seguintes interrogações estatísticas sobre programas já terminados:

- Com recurso à opção *stats-time* verificar o tempo de execução total de um conjunto de programas identificados por uma lista de PIDs passada via linha de comando no formato:

  ```
  $ ./tracer stats-time PID1 PID2 ... PIDN
  ```

  O *output* esperado seria o seguinte:

  ```
  $ ./tracer stats-time PID1 PID2 ... PIDN
  Total execution time is x ms
  ```

  onde x representa o tempo de execução total em milissegundos.
- Com recurso à opção *stats-command* verificar o número de vezes que um dado programa identificado pelo seu nome foi executado num dado conjunto de programas identificados por uma lista de PIDs passada via linha de comando no formato:

  ```
  $ ./tracer stats-command prog PID1 PID2 ... PIDN
  ```

  O *output* esperado seria o seguinte:

  ```
  $ ./tracer stats-command prog PID1 PID2 ... PIDN
  prog was executed x times
  ```

  onde x representa o número de vezes que *prog* foi executado.
- Com recurso à opção *stats-uniq* verificar a lista de programas únicos executados de um dado conjunto de programas identificados por uma lista de PIDs passada via linha de comando no formato:

  ```
  $ ./tracer stats-uniq PID1 PID2 ... PIDN
  ```

  O *output* esperado seria o seguinte:

  ```
  $ ./tracer stats-uniq PID1 PID2 ... PIDN
  prog1
  prog2
  ...
  progM
  ```

3 Arquitetura e Funcionalidades Implementadas

Nesta secção expõe-se as funcionalidades implementadas sendo a identificação delas através dos nomes apresentados na secção anterior e que aspeto da arquitetura permite a implementação dessa funcionalidade.

No entanto antes é preciso primeiro explicar como é que a aplicação *tracer* comunica com o *monitor* e o formato das mensagens que são trocadas entre eles então a primeira subsecção não diz respeito a nenhuma funcionalidade implementada mas sim ao suporte dado a todas elas.

3.1 Comunicação *tracer-monitor*

Como o *tracer* e o *monitor* são duas aplicações distintas *ergo* dois processos diferentes a única maneira de assegurar um meio de comunicação entre eles é através de *pipes* com nome, FIFOs.

Assim, o *monitor* antes de aceitar qualquer pedido gera um FIFO na pasta */tmp/* com o nome de *monitor* do qual ele vai receber os pedidos do *tracer*.

Por sua vez o *tracer* a cada execução abre esse mesmo FIFO em modo de leitura e escreve para lá uma mensagem definida no seguinte excerto de código:

```
#define CREATE 0
#define END 1
#define STATUS 2
#define STATS_TIME 3
#define STATS_COMMAND 4
#define STATS_UNIQUE 5

typedef struct message
{
    int type;
    int pid;
    char name[420];
    struct timeval timestamp;
} Message;
```

Como se pode verificar cada mensagem contém um tipo que identifica o tipo de pedido que mensagem carrega, um PID, uma *string* que embora se chame nome no final trata-se de um meio para carregar outra informação também e um *timestamp*.

Esta informação é entregue ao *monitor* que por sua vez interpreta o tipo da mensagem e lida com ela adequadamente.

3.2 Execução de programas do utilizador e Armazenamento de informação sobre programas em execução

O *tracer* inicia esta funcionalidade com a recolha do *timestamp* atual para o posterior cálculo do tempo de execução do programa.

Depois segue-se a criação de um processo de auxílio que executará o programa passado como argumento. A razão para tal deve-se à necessidade de apresentar o tempo total da execução do programa no *output* pois a execução do programa só é possível com recurso ás *system calls* da família *exec* que não permitem o posterior controlo sobre a sequência de eventos na execução de um programa depois da sua chamada.

Assim, o processo filho envia ao *monitor* uma mensagem do tipo *CREATE* indicando que um novo programa entrou em execução. Nos restantes campos vai o PID do processo filho no *pid*, o nome do programa no *name* e o *timestamp* previamente recolhido.

O *monitor* ao receber uma mensagem do tipo *CREATE* só tem uma função que é guardar a informação obtida em memória numa *hashtable* que utiliza o PID presente na mensagem como identificador único para o *hash*. O tratamento aqui não é concorrente pois a abertura de um processo para a escrita numa estrutura de dados não resultaria na escrita na estrutura original e embora esse problema pudesse ser resolvido com recurso a *pipes* anónimos ter-se-ia que esperar pela escrita no *pipe* de qualquer das formas resultando numa espera semelhante ou superior aquela realizada pela simples inserção sem o uso de um processo auxiliar.

Aquando do fim da execução do programa, o processo pai do *tracer* toma a responsabilidade de recolher o *timestamp* final e enviar ao *monitor* uma mensagem com o tipo *END* indicando que um programa terminou a execução, o PID para a identificação do programa e o *timestamp final*. Seguidamente imprime o *output* esperado através do cálculo do tempo de execução e termina a execução fechando o extremo de escrita do FIFO.

O *monitor* ao receber uma mensagem do tipo *END* procede à remoção da entrada correspondente ao programa terminado da *hashtable* também sem recurso a estratégias de concorrência pelas mesmas razão referidas anteriormente, e à criação do ficheiro que guarda a informação relevante ao programa terminado.

A criação do ficheiro já conta com concorrência e portanto inicia-se com a criação de um processo filho para o restante tratamento do pedido. O pai apenas realiza uma *waitpid* com a *flag WNOHANG* de maneira a não criar processos *zombies*. O filho por sua vez abre, truncando o que já lá está ou criando, o ficheiro com nome do PID do programa em tratamento e escreve para lá a seguinte estrutura em binário:

```
typedef struct info
{
    char name[420];
    long exec_time;
} Info;
```

que conta com o nome do programa e o tempo total de execução. O filho termina com o fecho do dito ficheiro.

3.3 Consulta de programas em execução

O *tracer* começa por abrir, em modo de leitura, um *pipe* com nome na pasta */tmp/* cujo nome é o seu PID. Este FIFO servirá como meio de resposta para o *monitor* comunicar com o *tracer*. As restantes funcionalidades fazem uso deste mecanismo também pelo que ele só explicitamente explicado aqui. De seguida, envia uma mensagem do tipo *STATUS* na qual vai apenas o PID do programa a executar o pedido.

O *monitor* ao receber uma mensagem do tipo *STATUS* começa por criar um novo processo para o tratar. O pai faz uso da função *waitpid* para evitar a criação de processos *zombie* - esta funcionalidade também é uma constante para o tratamento de pedidos pelo *monitor* pelo que esta é a última vez que será referida no documento. O filho por sua vez percorre a *hashtable* que coleciona a informação dos programas a executar e para cada um deles gera a linha esperada no *output* no *tracer*. De seguida escreve essa linha no FIFO de comunicação com o *tracer* que vai imprimindo-as à medida que as recebe. Ambos terminam com fecho de qualquer descritor de ficheiros aberto.

3.4 Execução encadeada de programas

A estratégia adotada nesta funcionalidade bem como as mensagens trocadas entre as duas aplicações são muito semelhantes aquela descrita para a "Execução de programas do utilizador" pelo que nesta secção foca-se mais na estratégia adotada para o real encadeamento dos programas.

O *tracer* começa então por criar tantos filhos como comandos na cadeia do *pipeline* e tantos *pipes* sem nome como *pipes* na cadeia. Depois cada processo filho deve fechar os extremos dos *pipes* que não vai utilizar segundo o seguinte algoritmo:

- O primeiro processo filho, o que irá executar o primeiro programa na cadeia deve fechar todos os extremos exceto o de escrita do primeiro *pipe*. Depois deve redirecionar através da função *dup2* o seu *standard output* para o extremo de escrita do primeiro *pipe*.
- O último filho deve fechar todos os extremos exceto o de leitura do último *pipe* e redirecionar o seu *standard input* para esse mesmo extremo.
- Cada filho intermédio deve fechar todos os extremos exceto o de leitura do *pipe* que lhe precede e o de escrita do *pipe* que lhe sucede, por exemplo o segundo filho deve manter aberto apenas o extremo de leitura do primeiro *pipe* e o extremo de escrita do segundo. Depois basta redirecionar o seu *standard input* para o extremo de leitura ainda aberto e o seu *standard output* para o extremo de escrita também aberto.

A partir daí cada filho executa o programa que lhe é designado.

A restante execução do processo pai é muito semelhante à da execução de apenas um programa com a apropriação para execução encadeada.

3.5 Consulta de programas terminados

O *tracer* começa por enviar uma mensagem do tipo *STATS_ TIME*, *STATS_ COMMAND* ou *STATS_ UNIQUE* dependo do pedido a executar.

A estratégia adotada para o *monitor* é semelhante em todas. Depois de criar um processo filho para o tratamento do pedido, o filho por sua vez cria tantos processos quantos PIDs tem de analisar.

Para cada processo existe também associado um *pipe* sem nome cujo objetivo é comunicar ao respetivo pai as suas descobertas. Assim cada processo filho começa por fechar o extremo de leitura do respetivo *pipe*, abre o ficheiro correspondente ao PID que deve analisar, lê o conteúdo do ficheiro e escreve no *pipe* um resultado de acordo com o que o pedido procura saber, por exemplo num *STATS_ TIME* devolveria o tempo de execução lido. O pai coleta essa informação e agrega-a para a posterior escrita no FIFO de comunicação com o cliente.

4 Teste e Comprovação de Resultados

Nesta secção ficam apenas algumas figuras a comprovar a execução de alguns comandos para efeito de teste das funcionalidades descritas e implementadas. Todas as figuras referenciadas nesta secção encontram-se presentes nos anexos no final do ficheiro

Na figura 1 vemos, apesar de não se verificar nenhum *output* ilustrativo o *monitor* correr.

Seguidamente na figura 2 vemos o resultado de executar o comando "ls -l" através do *tracer*.

Note-se que o PID de execução deste programa é 46013 e o tempo é de 12 ms.

De seguida nas figuras 3, 4 e 5 vemos a execução do comando *sleep* com vários argumentos diferentes.

Na figura 6 vemos a execução da opção *status* durante a execução dos programas em cima onde podemos verificar:

- Na primeira execução do *status*, vemos 3 programas identificados como *sleep* a correr. Se repararmos vemos que os PIDs destes respetivos programas coincidem com os dos da execução dos programas anteriores e que o tempo de execução do programa 46361 é inferior aos 10000 ms, ou seja, menor que 10.
- Na segunda execução do *status* o tempo de execução do programa 46363 já é superior aos 10 segundos e visto que os três começaram mais ou menos ao mesmo tempo vemos que o primeiro programa já terminou a sua execução.
- Na terceira execução do *status* o tempo de execução do programa 46365 já indica cerca de 17 segundos implicando que os restantes já terminaram.
- E por fim a última execução mostra um *output* vazio indicando que no momento não à programas em execução.

Na figura 7 vemos o resultado de correr o comando "grep -v ^# /etc/passwd | cut -f7 -d: | uniq | wc -l" pelo tracer. Note-se que o PID referente à execução é 46625 e o tempo de execução de 4 ms.

Depois na figura 8 vemos a pasta passada como argumento ao *monitor* onde se pode ver os diversos ficheiros com o nome dos PIDs dos programas já executados. Note-se que para cada programa há apenas um ficheiro com o PID associado.

De seguida, vemos na figura 9 a execução da opção *stats-time* onde são passados como argumentos os PIDs 46013 e 46625 referentes à execução do programa "ls -l" e do pipeline "grep -v ^# /etc/passwd | cut -f7 -d: | uniq | wc -l". Como já tinhamos referido antes o temp de execução do primeiro foi de 12 ms e a do segundo foi de 4 ms resultando num tempo total de execução de 16 ms como mostrado na figura.

Seguidamente na figura 10 vemos a execução da opção *stats-command* para os programas "ls", "sleep" e "grep | cut | uniq | wc" e com os PIDs vistos até agora. Recorde-se que houve apenas uma instância da execulção do "ls" e do "grep | cut | uniq | wc" e três do "sleep" como se pode confirmar também nos *outputs* obtidos na figura.

Para finalizar esta secção na figura 11 conseguimos verificar o resultado da opção *stats-uniq* no *tracer* para todos os PIDs já verificados anteriormente e como se pode ver pelo *output* todos os programas executados estão presentes uma e uma única vez independentemente do número de vezes que foram executados.

5 Conclusão e Trabalho Futuro

Este trabalho tinha como objetivo final a familiarização com os conceitos lecionados nas aulas práticas e a aplicação de todos eles em conjunto para criar duas aplicações que trabalham em conjunto para satisfazer os objetivos e funcionalidades descritas ao longo deste documento. Nesse sentido acreditamos que o trabalho desenvolvido foi bastante satisfatório tendo conseguido implementar todas as funcionalidades requisitadas levando connosco todo o conhecimento necessário a essa implementação.

No entanto, devido a algumas limitações nomeadamente à *deadline* definida não foi possível limpar o código todo e, embora não seja diretamente o objetivo do trabalho, gostaríamos de propor como trabalho futuro a limpeza do mesmo desde a escrita de mais comentários, à libertação de memória alocada e possivelmente ainda à reorganização do código para uma maior facilidade de leitura.

Anexos

Fig. 1. *Monitor* a correr.

```
(base) ➜  tracer git:(main) ✗ ./tracer execute -u "ls -l"
Running PID 46013
total 208
-rw-r--r--   1 zyveth  staff   1021 Apr 30 01:14 Makefile
drwxr-xr-x  11 zyveth  staff    352 Apr 30 01:10 include
-rwxr-xr-x   1 zyveth  staff  33425 Mar 21 16:05 main
-rw-r--r--   1 zyveth  staff    102 Mar 21 16:05 main.c
drwxr-xr-x  11 zyveth  staff    352 Apr 30 14:05 obj
drwxr-xr-x  11 zyveth  staff    352 Apr 30 00:54 src
-rwxr-xr-x   1 zyveth  staff  58771 Apr 30 14:05 tracer
Ended in 12 ms
```

Fig. 2. Resultado da execução do comando "ls -l" pelo *tracer*.

```
(base) ➜  tracer git:(main) ✗ ./tracer execute -u "sleep 10"
Running PID 46361
Ended in 10009 ms
```

Fig. 3. Resultado da execução do comando "sleep 10" pelo *tracer*.

```
(base) ➜  tracer git:(main) ✗ ./tracer execute -u "sleep 15"
Running PID 46363
Ended in 15008 ms
```

Fig. 4. Resultado da execução do comando "sleep 15" pelo *tracer*.

```
(base) ➜  tracer git:(main) ✗ ./tracer execute -u "sleep 20"
Running PID 46365
Ended in 20008 ms
```

Fig. 5. Resultado da execução do comando "sleep 20" pelo *tracer*.

```
(base) ➜  tracer git:(main) ✗ ./tracer status
46361 sleep 9240 ms
46363 sleep 7079 ms
46365 sleep 5625 ms
(base) ➜  tracer git:(main) ✗ ./tracer status
46363 sleep 10928 ms
46365 sleep 9473 ms
(base) ➜  tracer git:(main) ✗ ./tracer status
46365 sleep 16956 ms
(base) ➜  tracer git:(main) ✗ ./tracer status
(base) ➜  tracer git:(main) ✗
```

Fig. 6. Resultado da execução da opção *status* no *tracer*.

```
(base) ➜  tracer git:(main) ✗ ./tracer execute -p  "grep -v ^# /etc/passwd | cut -f7 -d: | uniq | wc -l"
Running PID 46625
       7
Ended in 4 ms
```

Fig. 7. Resultado da execução do *pipeline* "grep -v ^# /etc/passwd | cut -f7 -d: | uniq | wc -l" pelo *tracer*.

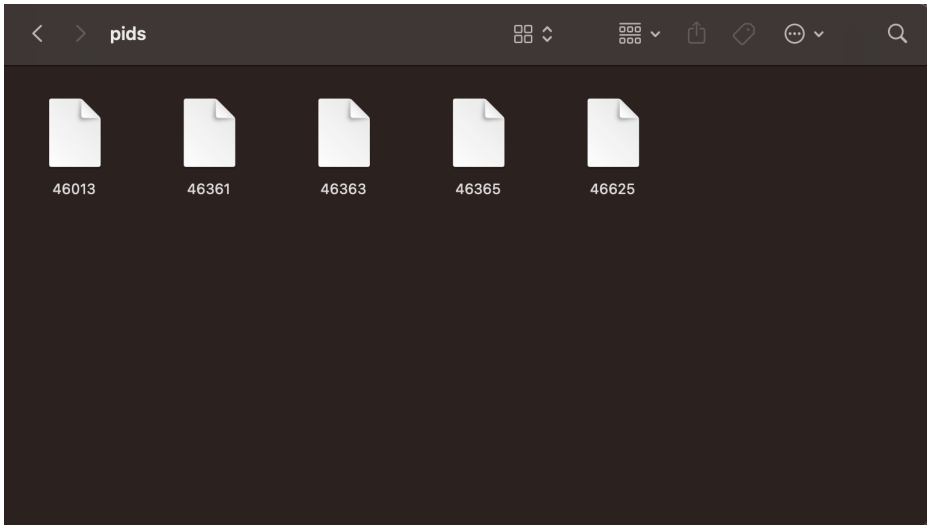

Fig. 8. Pasta que contém os ficheiros PID dos comandos executados anteriormente.

```
(base) ➜  tracer git:(main) ✗ ./tracer stats-time 46013 46625
Total execution time is 16 ms
```

Fig. 9. Resultado da execução da opção *stats-time* no *tracer*.

```
(base) ➜  tracer git:(main) ✗ ./tracer stats-command ls 46013 46625 46361 46363 46365
ls was executed 1 times
(base) ➜  tracer git:(main) ✗ ./tracer stats-command sleep 46013 46625 46361 46363 46365
sleep was executed 3 times
(base) ➜  tracer git:(main) ✗ ./tracer stats-command "grep | cut | uniq | wc" 46013 46625 46361 46363 46365
grep | cut | uniq | wc was executed 1 times
```

Fig. 10. Resultado da execução da opção *stats-command* no *tracer*.

```
(base) ➜  tracer git:(main) ✗ ./tracer stats-uniq 46013 46625 46361 46363 46365
ls
grep | cut | uniq | wc
sleep
```

Fig. 11. Resultado da execução da opção *stats-uniq* no *tracer*.