



Universidade do Minho
Escola de Engenharia

Universidade Do Minho

Engenharia Informática - Sistemas Operativos

Rastreamento e Monitorização da Execução de Programas

Grupo 101:

Luís Gomes Ferreira A91672

Bernardo Garcia de Freitas Lima A93258

João Pedro Martins da Silva Freitas Cardoso A94595

Conteúdo:

1. Introdução.....	3
2. Funcionalidades Básicas.....	4
2.1. Estrutura da Comunicação.....	4
2.2. Execução de comandos.....	5
2.3. Execução de status.....	6
2.4. Armazenamento de informação sobre programas terminados.....	6
2.5. Funções auxiliares.....	8
3. Conclusão.....	9

Capítulo 1

Introdução

Este projeto tem como objetivo desenvolver um sistema de monitorização eficiente aplicada a um, ou vários programas em execução. A monitorização e rastreamento da execução de programas são tarefas cruciais para a manutenção e desenvolvimento de software, a observação do comportamento dos programas é importantíssimo para identificar problemas de desempenho, erros, entre outros.

Uma das formas mais eficazes de rastrear a execução de programas é através de chamadas de sistema, estas permitem que os programas solicitem serviços ao kernel do sistema operacional e interajam com o sistema operativo subjacente de forma segura e padronizada.

No contexto deste projeto, foram utilizados *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.

Essa comunicação será crucial para o bom funcionamento entre o cliente e o servidor, bem como entre os vários processos filhos a executar em simultâneo.

Espera-se que, no final do desenvolvimento realizado pelo grupo, a conexão entre o cliente e o servidor esteja operacional e segura. O resultado final será um serviço de monitorização eficaz para um ou vários programas, utilizando conceitos como chamadas de sistema e pipes para rastrear e guardar informações importantes sobre a execução dos programas em questão.

Capítulo 2

Funcionalidades

Este projeto tem duas componentes, a primeira o cliente (*tracer.c*) e o servidor (*monitor.c*), estes comunicam entre si e, tal como previsto, tem certas funcionalidades, as mesmas se categorizam em básicas e avançadas.

Estas funcionalidades devem ser implementadas de acordo com a metodologia de Sistemas Operativos, ou seja, não podem recorrer a funções da biblioteca do C, como a *bash* ou o *system()*.

Capítulo 2.1. - Estrutura da Comunicação

Para que exista comunicação entre o *tracer* e o *monitor*, recorremos a dois *named pipes* para essa comunicação, o *read* e o *write* (armazenados na pasta *fifo*s), que, dedutivamente, têm capacidades inversas. Num o *tracer* escreve enquanto o *monitor* lê, no outro os papéis são invertidos.

Para além de *named pipes* recorremos também a um *unnamed pipe* para a comunicação interna do *tracer*, de forma a que filhos e pais conseguissem transmitir a informação relevante, sem esta se perder.

Por fim, para agilizar a comunicação entre os diversos clientes e o servidor, recorreu-se a uma estrutura (*struct*) , idealizada de maneira a uniformizar o tamanho das mensagens entre os programas. Assim, criámos a *struct pedido*:

```
typedef struct pedido {  
  
    pid_t pid;  
  
    char commando[60];  
  
    struct timeval inicial,final;  
  
}pedido;
```

Assim, através do *pid* poderemos comunicar com o servidor, visto que o *pid* do programa a executar, importante para identificar.

Também o char *commando*, que conterà a string do comando a executar e, duas *struct's* do tipo *timeval*, pois utilizámos, tal como sugerido pelos docentes da UC, a função *gettimeofday()*, onde na *inicial* guardamos o instante em que o programa é executado e na *final*, o instante em que o programa concluiu a execução.

Capítulo 2.2. - Execução de comandos

A principal funcionalidade deste projeto é a execução de programas via input do utilizador. Este, terá ao seu dispor uma diversidade de comandos da *bash*, através do cliente (*tracer.c*), utilizando “*./tracer execute -u <comando> <arg1> ... <argn>*”.

Para executar estes comandos o *tracer.c* recorre a uma função *execute*, que recebendo o file descriptor do pipe de leitura do servidor e os comandos a executar, vai enviar a informação que estará na *struct pedido* (ainda não completa pois falta o tempo final) ao monitor, onde este estará à espera do resto da informação para, após a receber, poder eliminar da lista de programas em execução a entrada correspondente ao seu PID.

Apesar de na função *execute()* estar presente uma chamada a *execvp()*, que substitui o código de execução de um programa pelo presente nos seus argumentos, nós recorremos a um *fork()* para gerar um processo filho do *tracer* para que apenas este seja substituído pelo comando a executar pelo *execvp()*, enquanto que o processo pai espera pelo filho através do *waitpid()*, de forma a poder enviar de novo a informação na forma da *struct pedido* após a sua conclusão, desta vez, para a conclusão do processo.

```
bodacios@pop-os:~/Documents/S0/projetolatestcommit/bin$ ./tracer execute -u sleep 3
Running PID 11819
Ended in 3002 ms
bodacios@pop-os:~/Documents/S0/projetolatestcommit/bin$ ./tracer execute -u ls /home/
bodacios/
Running PID 11833
bodacios.cabal Documents MEGA Music Public Videos
CHANGELOG.md Downloads MEGAsync MyLib.hs Setup.hs
Desktop Main.hs 'MEGAsync Downloads' Pictures Templates
Ended in 3 ms
```

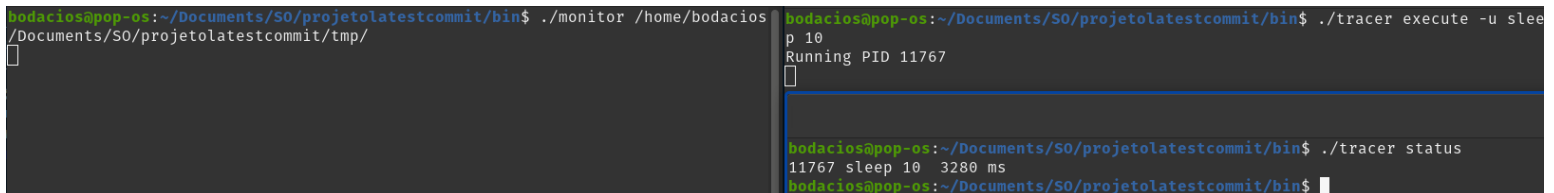
Figura 1: Execução de Comando ‘ls’ através de “*./tracer execute -u ls directory*”

Capítulo 2.3. - Execução de status

Em termos funcionais, o status tem semelhanças com a execução de comandos.

Continua a ser enviada a *struct* através dos pipes para notificar o monitor do pedido, só que para esta implementação preenchemos o ‘*pedido.comando*’ com o valor “status” em vez de um comando a executar, para que assim que chegar a informação ao servidor a função *status()* seja executada.

Como neste caso o servidor tem a necessidade de comunicar com o cliente, para obter a informação do estado dos programas em execução, tem que ser aberto o pipe de escrita do servidor “write” para ser aí escrita a informação (*resposta*), de modo a que, depois o cliente, através de um loop com a função *read()*, faça a leitura da string enviada pelo *monitor*, e de seguida a escreva para o *standard output*, de maneira que possa ser lida pelo utilizador.



```
bodacios@pop-os:~/Documents/SO/projetolatestcommit/bin$ ./monitor /home/bodacios
/
Documents/SO/projetolatestcommit/tmp/
bodacios@pop-os:~/Documents/SO/projetolatestcommit/bin$ ./tracer execute -u slee
p 10
Running PID 11767
bodacios@pop-os:~/Documents/SO/projetolatestcommit/bin$ ./tracer status
11767 sleep 10 3280 ms
bodacios@pop-os:~/Documents/SO/projetolatestcommit/bin$
```

Figura 2: Representação do Status através de “./tracer status”

Capítulo 2.4. - Armazenamento de informação sobre programas terminados

O armazenamento das informações dos programas foi implementado no servidor para que, este, ao receber um pedido de execução de comandos, pudesse guardar a informação dos comandos que terminavam em ficheiros de texto.

Primeiramente, ocorre uma verificação se os PIDs dos processos que terminaram estão presentes no array global que armazena os pedidos. Caso estejam, o tempo de execução dos processos é calculado utilizando a função *calcExecMonitor()*, que recebe como argumento o pedido que acabou de terminar e o pedido armazenado no array global

correspondente ao mesmo processo. Essa função utiliza a diferença entre os campos *final* dos dois pedidos para calcular o tempo de execução em milissegundos.

Em seguida, o código escreve as informações do processo num ficheiro de texto utilizando a função *snprintf* para formatar a string de saída. O nome do ficheiro é criado a partir da concatenação da pasta *pids_folder* (passada como argumento na linha de comando) com o pid do processo que terminou, seguido pela extensão ".txt". O tempo de execução e o comando que foi executado são escritos no ficheiro utilizando a função "write".

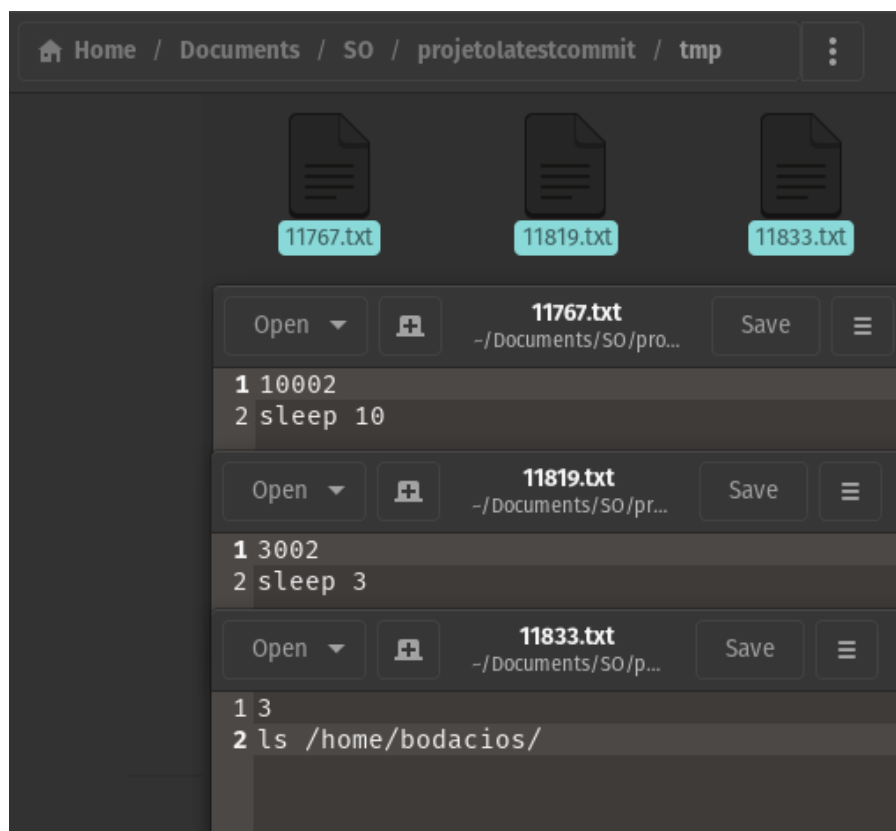


Figura 3: Ficheiros gerados após execução de comandos no *tracer*

Capítulo 2.5. - Funções auxiliares

- *void itoa(int n, char s[])*

Para informar o cliente que o seu pedido se encontra em execução, recorreremos ao *itoa* para transformar o *pid* num *char**, de maneira a escrevê-lo para o *stdout*. Esta função também recorre a outra para executar, o *void reverse(char s[])*, que simplesmente inverte a string resultante.

- *suseconds_t calcExec(pedido pedido)*

De forma a calcular o tempo de execução do comando, utilizámos esta função, que recorre à *struct timeval* *inicial* e *final* onde recorreremos tanto ao campo dos microsegundos como o campo dos segundos e microsegundos, *tv_sec* e *tv_usec*, respetivamente, através de uma conta simples que nos permitiu o cálculo na unidade esperada, o milissegundo.

- *char *extraiComandoString(int argc, char **argv)*

De forma a não transmitir informação desnecessária, criámos esta função simplesmente para remover o “*./tracer execute -u*”, de forma a que o comando na *struct pedido* contivesse apenas a parte do comando relevante.

- *void extraiComandoArray(char **str, int argc, char *argv[])*

Para a utilização do *execvp* foi-nos útil criar uma função que colocasse o comando a executar num array, para estar num formato permitido pelo *execvp* de maneira a executar o programa.

- *suseconds_t calcExecMonitor(struct pedido pedido, struct pedido global)*

Para a execução da funcionalidade do *status* decidimos recorrer a uma função semelhante à *calcExec()* do *tracer*, com a diferença que esta é chamada no *monitor* para calcular a duração dos programas a executar no instante em que a *status* é chamada.

Capítulo 3

Conclusão

Durante a evolução do trabalho foram notáveis as dificuldades em diversas áreas às quais me irei referir.

Primeiramente, a nível de alocação e gestão de memória entre os dois programas, surgiram diversos erros, como por exemplo, aquando o envio da *struct* semi-completa, o monitor sofreu diversos *segmentation fault* devido a alocação da memória para a *struct* não estar implementada da mesma maneira em ambos os programas.

Em segundo, o cálculo do tempo do pedido também suscitou algumas dúvidas inicialmente, uma vez que a ferramenta sugerida não fornecia o tempo de execução nas unidades desejadas.

Também relativamente ao debug dos eventuais erros de compilação, foi necessário um estudo mais aprofundado da ferramenta *gdb* de maneira a ter um melhor controlo entre os processos pai/filho, de maneira a corrigir estes erros eficazmente e apesar disso, não nos foi possível a implementação da funcionalidade avançada da *pipeline*, pois o debug de alguns erros revelou-se ser extremamente difícil.

Focando agora nos aspetos mais positivos, pensamos que as aulas práticas nos prepararam bem para a implementação do projeto no geral, pois desde o início que foi claro a abordagem que queríamos implementar, o que facilitou bastante o progresso deste projeto, assim como o conhecimento da grande maioria das ferramentas que tivemos de utilizar, como o *fork()*, *mkfifo()*, entre outros, que foram indispensáveis para a finalização destes dois programas.