Universidade do Minho
Departamento de Informática

Sistemas Operativos

Rastreamento e Monitorização da Execução de Programas

Realizado por:
José Rodrigues (a100692)
João Coelho (a100596)
Duarte Araújo (a100750)

2 de março de 2024
Ano Letivo 2022/23

Índice

1 Introdução

No âmbito da disciplina de Sistemas Operativos foi-nos apresentado um projeto que requere o desenvolvimento de um serviço de monitorização dos programas executados numa máquina.

Desta forma, o projeto irá se focar em duas partes importantes, o servidor e o cliente. O utilizador deverá conseguir executar programas através do cliente e obter o tempo de execução do mesmo. Por outro lado, um admistrador, através do servidor, deverá conseguir consultar informações sobre os programas que estão a ser executados no momento, incluindo o tempo de execução que passou até ao momento da consulta. Mais tarde, o servidor poderá ainda apresentar estatísticas sobre programas já terminados, tais como, o tempo de execução agregado de um certo conjunto de programas, os programas únicos que foram executados e quantas vezes um programa foi executado.

Para além disso, a comunicação entre o servidor e cliente deverá ser feita apenas por *pipes com nome*. Durante todo o desenvolvimento do projeto a realização de *prints* no terminal foram feitas com recurso à função *write()* no *standart output*.

2 Funcionalidades Básicas

2.1 Comunicação entre Servidor e Cliente

A primeira tarefa que precisámos concluir foi o desenvolvimento do método de comunicação entre o cliente e o servidor. Para que esta comunicação fosse possível, e se mantivesse fiel ao enunciado dado, o servidor cria um *pipe com nome*, *FIFO*, que é acedido pelo cliente para escrita e pelo servidor para leitura.

O primeiro obstáculo a ultrapassar foi o facto do servidor desligar quando se lê o primeiro conjundo de dados vindo do *pipe*, no entanto, tal problema foi resolvido deixando o servidor sempre ligado, em memória, permitindo que este pudesse receber sempre novos clientes sem desligar. Para que tal fosse possível o servidor possuiu um ciclo infinito que fica sempre à espera de conseguir ler algo do *FIFO* e, quando lê, devido ao ciclo referido, ele vai imediatamente retomar a espera por mais informações vindas pelo *pipe*.

De seguida precisávamos de definir uma mensagem de tamanho fixo para ser escrita no *pipe* pois, se tal mensagem não existisse, seria bastante dificultado o reconhecimento, por parte do servidor, do que necessita ler do *FIFO*. Assim foi criada uma estrutura conhecida tanto ao servidor como ao cliente.

```c
typedef struct informacao_processo {
    int pid;
    char nome[50];
    struct timeval time;
    long ms;
} informacao_processo;
```

Figura 1: Struct informacao_processo

A estrutura é constituída pelo *PID* do processo pai do cliente, o nome do programa, o tempo em milisegundos que o programa utilizou, inicialmente definido como -1, e uma *struct timeval* que guarda o instante em que o programa começou, sendo posteriormente utilizada para calcular o tempo total.

2.2 Execução de programas do utilizador

A primeira funcionalidade que impletámos foi a execução de programas.

O utilizador sinaliza que quer executar um programa com os argumentos *execute* e *-u* (ex: ./bin/tracer execute -u "ls /etc"), de seguida, após o programa *tracer* identificar tal instrução do utilizador, recorrendo à função *strcmp()*, é preenchida a struct introduzida anteriormente com os valores adequados. Esta *struct* é então enviada para o servidor através do *FIFO* e o utilizador também é informado, via terminal, do *PID* do processo que está a executar o programa.

Ao mesmo tempo, enquanto todo este processo do *tracer* é executado, o servidor já se encontra à espera de receber a *struct* do cliente, sendo que, quando finalmente este a consegue ler e após a verificação de que a variável *ms* se encontra a -1, a *struct* é adicionada a um *array* dinâmico de *informacao_processo* formado durante a criação do servidor.

Voltando ao programa *tracer*, após o servidor ter sido notificado do seu processo, este começa a execução do programa pedido. Primeiramente é realizado o *parse* dos argumentos do programa, desta forma os mesmos podem ser organizados num *array* de *char**. De seguida, é possível passar este *array* à função *execvp()* que irá executar o programa pretendido.

Após a execução, é novamente enviada a estrutura criada anteriormente para o servidor, recorrendo ao *FIFO*, no entanto, a mesma terá o valor da variável *ms* igual ao tempo de execução do programa que foi finalizado, com o auxílio de uma nova *struct timeval* obtida no final, é importante referir que até o momento o valor da variável *ms* era igual a -1. A fórmula utilizada para calcular o tempo em milissegundos é a seguinte:

$$(fim.tv\_sec - inicio.tv\_sec) * 1000 + (fim.tv\_usec - inicio.tv\_usec)/1000 \quad (1)$$

O resultado desta conta também é apresentado ao utilizador no terminal.

Por sua vez, o servidor, ao receber novamente a estrutura, elimina a mesma do array, já que o processo acabou a sua execução.

2.3 Consulta de programas em execução

A outra funcionalidade básica implementada é a *status*, sendo que esta funcionalidade tem como objetivo apresentar os processos a ser executados no momento, bem como os tempos de execução no momento da realização do pedido.

Para que tal funcionalidade fosse implementada foi necessário criar outro *pipe com nome* seguindo a seguinte regra de nomenclatura, *FIFO* mais o *PID* do processo que está a tratar do *status*. Por exemplo, se o processo a realizar o comando tiver o *PID* 770, o nome do fifo será *FIFO770*.

Desta forma, à semelhança da execução de programas, é preenchida uma estrutura *informacao_processo*, no entanto, a variável nome é preenchida com *status* em vez do nome de um possível programa. De seguida, o servidor verifica que o nome na estrutura corresponde ao comando, assim, para responder ao pedido, o array de processos é percorrido e, em cada iteração, é calculado o tempo de execução seguindo a fórmula introduzida anteriormente, 1, sendo a estrutura final a *struct timeval* recebida pelo *pipe*. No fim de cada iteração é escrito no *pipe FIFO_PID*, previamente aberto para escrita com recurso ao *PID* recebido na estrutura lida no *pipe FIFO*, a frase "PID NOME TEMPO ms\n".

3 Funcionalidades Avançadas

3.1 Execução encadeada de programas

A primeira funcionalidade avançada que implementámos foi a execução encadeada de programas. Esta funcionalidade tinha como objetivo simular um *pipeline* de programas à semelhança do que acontece quando se coloca o operador '|' no terminal, sendo que a mesma é identificada pelo uso do argumento "-p" em vez de "-u". Para que a execução separada dos programas fosse possível foi necessário alterar o *parse* realizado no caso da execução de programas sozinhos. Desta forma, antes de começar a separar os programas por espaços, é necessário separar primeiro pelo caracter '|' e o resultado de cada *token* é guardado num *array* de *char**.

Para além disso, também foi necessário mudar o nome que a estrutura *informacao_processo* carrega, sendo que, a partir deste momento, o mesmo é constituído pelo nome de cada programa separado pela seguinte expressão: " | ". Desta forma, para cada elemento do *array* previamente referido, é separado o nome do programa e, por sua vez, é adiconada a expressão referida anteriormente, ou seja, a servir de exemplo, no caso de ser executado "./bin/tracer execute -p "ls /etc | wc -l"", o nome do programa será "ls | wc".

Por fim, para a fase de execução do programa, decidimos usar *pipes anónimos*. De seguida encontra-se uma imagem que descreve melhor o desafio a codificar.

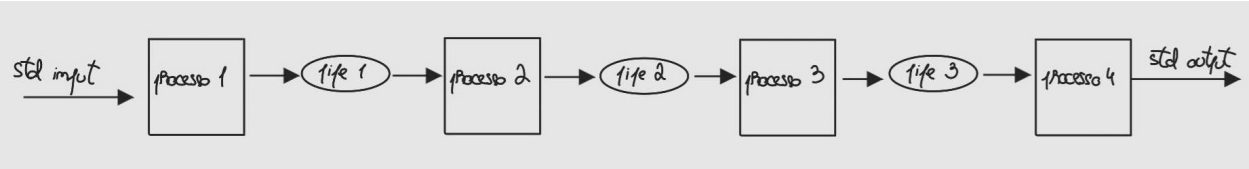

Figura 2: Esquema da execução de programas encadeados

Observando a imagem é possível notar que, para a execução de N programas, precisamos de N-1 *pipes anónimos*, logo foi criada uma matriz com N-1 linhas e 2 colunas para representar estes *pipes*. É importante referir que existem 3 fases diferentes na execução da pipeline.

A primeira é a cabeça, que irá ler do *standart input* mas irá escrever no primeiro *pipe*.

De seguida, a segunda fase é a ponte entre os programas intermédios, sendo que, para a execução funcionar corretamente, o programa a executar deve ler do *pipe* anterior e escrever no *pipe* seguinte.

Por fim, a terceira e última fase trata-se da cauda da pipeline, onde o último programa deverá ler do último *pipe*, mas irá escrever no *standart output*. Foi necessário ter bastante atenção no término dos *pipes* para que o programa funcionasse devidamente. Para além disso, este processo é todo realizado dentro de um *fork*, já que a escrita do tempo de execução por vezes aparece antes do resultado da execução da pipeline. Desta maneira, é possível que o processo pai espere pelo filho antes de acabar.

Excluindo o nome do programa, cujas alterações já foram explicadas, toda a interação com o servidor não foi alterada.

3.2 Armazenamento de informação sobre programas terminados

A segunda funcionalidade a implementar foi a criação de ficheiros para os programas já terminados.

Assim, para a realização desta funcionalidade, foi aproveitado o facto de já estar implementada uma maneira de diferenciar se um programa está a iniciar ou a acabar, o valor dos *ms* da estrutura

informacao_processo, como foi referido em 2.2. De seguida, após receber a estrutura de finalização, o servidor cria um ficheiro com o nome igual ao *PID* do processo recebido na pasta cujo caminho é fornecido como argumento no início da execução do servidor e é passado como argumento às funções necessárias e, por fim, escreve a própria estrutura no ficheiro pois já contém o nome e tempo de execução do programa.

3.3 Consulta de programas terminados

Para a última funcionalidade avançada é pedida a realização de três tipos de consulta dos programas terminados: *stats-time*, *stats-command* e *stats-uniq*. O servidor identifica o tipo de comando recebido de maneira semelhante para as três instruções. Primeiramente, o *tracer* envia na estrutura utilizada ao longo do projeto o nome do comando a ser executado, de seguida, o servidor filtra estes nomes e chama as funções respetivas. Cada função é executada dentro de um *fork()* para que a execução de novos clientes não seja atrasada devido à espera de leitura de *pipes* que ocorrem dentro das funções.

O comando *stats-time* tem como objetivo obter o tempo total de execução utilizado por um dado conjunto de programas identificados por uma lista de *PIDs*.

Para a realização do mesmo decidimos criar dois *pipes com nome*, o nome do primeiro segue a mesma regra do criado em 2.3 e o segundo tem o nome da própria funcionalidade, ou seja, "stats-time". O *tracer* escreve primeiro uma *string* contendo todos os *PIDs* requesitados no primeiro *pipe*. O servidor, por sua vez, irá ler do mesmo *fifo* esta *string* e percorre todos os *PIDs* contidos na mesma. Para cada *PID* lido é aberto o respetivo ficheiro na pasta passada como argumento ao servidor e é incrementado o tempo a uma variável de resultado. Após todos os PIDs terem sido percorridos é então aberto o segundo *fifo* onde será escrita a seguinte frase "Total execution time is X ms\n", onde X representa o tempo total calculado. Para concluir, o programa *tracer* lê esta frase do mesmo *pipe* e escreve no terminal.

O comando *stats-command* pretende que se obtenha o número de vezes que um programa foi executado numa lista de *PIDs*.

À semelhança do comando anterior, este também necessita de dois *pipes com nome* que têm a mesma função, no entanto, o nome do segundo *pipe* é "stats-command". Em adição, a *string* escrita pelo *tracer* no primeiro *pipe* possui ainda, no ínicio, o nome do programa que se pretende contar o número de execuções. O servidor, antes de começar a abrir todos os ficheiros dos *PIDs* recebidos, separa o nome do programa do resto da *string*. Por sua vez, o servidor abre todos os ficheiros dos PIDs e percorre todos os programas no nome de execução, utilizando a função *strtok_r()* com o delimitador " | ", pois, se o programa foi executado como é explicado em 3.1, é necessário analisar todos os programas executados na pipeline. Por fim, após contar todas as execuções, é escrito no segundo *pipe* a seguinte frase: "PROGRAM was executed X times\n", onde X é o número de vezes que foi executado. De seguida, o programa *tracer* lê a frase do *pipe* e imprime no terminal.

Para finalizar, o comando *stats-uniq* tem o objetivo de listar os programas únicos que foram executados numa lista de *PIDs*.

Tal como os últimos dois comandos explicados, este também utiliza dois *pipes com nome*, onde o nome do primeiro segue a mesma regra de nomenclatura e o nome do segundo será "stats-uniq". O processo no *tracer* é bastante semelhante aos outros comandos, é enviada uma *string* de *PIDs* no primeiro *pipe*, é lida a resposta do segundo e, por sua vez, é realizada a impressão no terminal. O servidor também tem um processo bastante parecido, são abertos os ficheiros dos *PIDs*, um a um, e são percorridos os nomes das execuções para encontrar os programas únicos, no entanto, neste caso, o servidor cria um *array* de *strings* para guardar os programas diferentes. Aquando a escrita da resposta, ao contrário dos comandos anteriores, o servidor percorre o array e escreve os nomes dos programas separadamente, o que implica que o *tracer* também leia os nomes isoladamente.

4 Conclusão

Um dos maiores problemas que surgiram ao longo do trabalho foi a sicronização dos *pipes* com e sem nome, sendo que nos primeiros também foi necessário prestar muita atenção aos *close()* no código responsável pela execução de programas encadeados, 3.1. Para além disso, pensamos que o trabalho poderia ser melhorado se abrangêssemos casos em que as listas de *PIDs* dadas nas funcionalidades avançadas tivessem um tamanho muito mais elevado. Em suma, com este trabalho, foi possível compreender e aplicar melhor o funcionamento dos *pipes com nome* e os *pipes anónimos*, bem como o manejamento de ficheiros binários.