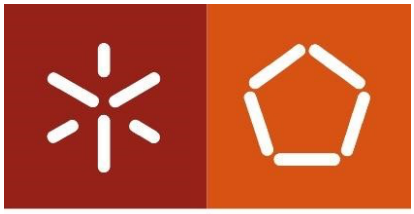

Universidade do Minho

Licenciatura em Engenharia Informática

Sistemas Operativos - Trabalho Prático

Grupo 15

David da Silva Teixeira (A100554)

João Luís Ferreira Magalhães (A100740)

Jorge Nuno Gomes Rodrigues (A101758)

Ano Letivo 2022/2023

Conteúdo

Capítulo 1

Introdução

Este projeto descreve o processo de implementação de um serviço de monitorização de programas executados em uma máquina. O objetivo é permitir que os utilizadores executem programas e obtenham o tempo de execução dos mesmos, enquanto o Servidor do sistema pode consultar todos os programas que estão atualmente em execução bem como, os que já terminaram.

O relatório detalha as etapas necessárias para desenvolver esse serviço, incluindo a análise de requisitos, o design da arquitetura do sistema, a implementação do Cliente e do Servidor, e a validação do modelo. Também são apresentados os resultados e a avaliação do desempenho do serviço.

O projeto implementa vários conceitos estudados nas aulas práticas como por exemplo a comunicação entre o Cliente e o Servidor, que comunicam através de dois "pipes com nome", em que um envia comandos do Cliente para o Servidor, e o outro envia o output dos comandos do Servidor para o Cliente.

Capítulo 2

Funcionalidades disponíveis no Servidor

2.1 Execução de programas do utilizador

Este ponto representa uma das funcionalidades básicas do Servidor e do Cliente responsáveis por executar todos os programas. O Cliente realiza o “parsing” da input, que é feito para obter o programa e os argumentos necessários a executar utilizando apenas um "execvp".

Um filho é então criado usando o comando "fork()", que cria uma mensagem. Essa mensagem é inicializada com as informações relevantes, incluindo o PID do filho, o nome do programa e um timestamp. A mensagem é então enviada para o Servidor através de um “FIFO” e para o processo pai através de um “pipe anónimo”.

Figura 1 – Estrutura de “Message” e de “Type”

Depois de enviar a mensagem, o processo filho executa o programa usando o "execvp()". Quando o processo filho termina, o processo pai espera pelo final da execução do porcesso filho.Assim que o programa terminar, o pai recorre à mensagem recebida pelo filho para atualizar os campos necessários para serem enviados ao Servidor. Esses campos incluem o estado do processo (“Start” ou “Finish”), o PID do processo filho e o tempo total de execução.

Assim, o Servidor mantém um registro em memória de todas as execuções de programas, incluindo informações relevantes para monitorizar o seu desempenho e garantir que tudo está a funcionar como pretendido.

2.2 Consulta de programas em execução

Ainda dentro das funcionalidades básicas, é necessário que o Servidor consiga apresentar todos os programas que estão em execução ao Cliente que fez o pedido. A estratégia utilizada consiste em fazer com que o Cliente crie uma "pipe com nome" com o PID atual do processo e envia uma mensagem com o estado "Info" para o Servidor através do "pipe com nome". Em seguida, após o Servidor verificar que se trata de uma mensagem do tipo "Info", este cria um processo filho para atender o pedido. O monitor de seguida invoca uma função que percorre a estrutura de dados e que escreve para o Cliente as mensagens que ainda não passara ao estado "Finish". O Cliente fica responsável por fazer o cálculo do tempo que passou e imprimir via terminal.

2.3 Execução encadeada de programas

A função é responsável por executar um comando que contêm vários "pipes". Ela recebe como parâmetro uma "string" que contém o comando a ser executado. Primeiramente, tal como a "execute -u" é criada a mensagem com as informações necessárias e enviada ao Servidor.

De seguida, é individualizado cada comando na "string" criando "pipes anónimas" para a comunicação entre os comandos. A função executa cada comando utilizando a função "execute_command". Se o comando for o primeiro da lista de argumentos, não tem um "pipe" de entrada, então é passado "-1" como descritor de "standard input". Se for o último comando, não tem um "pipe" de saída, então é passado "-1" como descritor de "standard output". Para os comandos intermédios, são passados os descritores "pipe" de entrada e o descritor do "pipe" de saída, respetivamente.

Após a execução de todos os comandos, a função calcula o tempo total de execução e atualiza a mensagem com o estado, "Finish", e o tempo total de execução. Por fim, ela envia a mensagem para o Servidor através de o mesmo "pipe com nome" usado na primeira mensagem.

2.4 Consulta de programas terminados

2.4.1 Comando "stats-time"

Esta função é responsável por enviar uma mensagem para o Servidor solicitando informações sobre o tempo total de execução de uma lista de programas. O processo atual cria um fifo com o nome do seu próprio PID e envia uma mensagem contendo o nome da lista de programas e o PID com o estado "InfoTime".

O Servidor recebe a mensagem, processa a informação e envia de volta o tempo total de execução através do mesmo fifo.

Em seguida, o processo atual lê o tempo total de execução a partir do fifo, converte-o para milissegundos e exibe a mensagem no "stdout" e o fifo é removido, através do comando "unlink ()".

2.4.2 Comando “stats-command”

Este comando é semelhante ao anterior, mas agora tem como objetivo obter o número de vezes que um determinado programa foi executado. A função recebe dois argumentos: o primeiro é uma lista de programas (separados por vírgulas) e o segundo é o nome de um programa específico cujo número de execuções será solicitado.

O código começa por criar um processo para executar a função “getpid ()”, que retorna o PID do processo atual. Em seguida, a função cria um FIFO com o nome do PID do processo atual, para que o Servidor possa enviar a resposta diretamente para este processo.

Depois disso, o nome do programa que se pretende verificar é concatenado com a lista de programas recebida como argumento. A mensagem é então preenchida com o nome do programa, o PID do processo atual e o estado “InfoCommand”, indicando que se pretende obter o número de execuções de um programa específico.

A escrita e leitura no “fifo” é igual à função anterior.

Finalmente, é criado um “buffer” para armazenar a mensagem que será escrita no terminal. Esta contém o nome do programa e o número de vezes que foi executado que será escrita no terminal usando a função “write”.

2.4.3 Comando “stats-uniq”

Este comando cria um fifo com o nome do seu próprio PID e envia uma mensagem contendo o nome da lista de programas e o PID com o estado “InfoUniq”**,** indicando que deseja receber uma lista de programas únicos em execução no momento.

O Servidor é responsável por ver quais os processos únicos entre a lista de processos enviada. Para isso, o Servidor constrói uma lista de mensagens em que os programas não apareçam repetidos. Para esse efeito, o Servidor inicializa a lista com os programas não repetidos do primeiro PID enviado e, a partir desse ponto, junta à lista programas que não se encontrem na mesma.

Em seguida, o Servidor escreve para o “pipe com nome” criado com o PID do Cliente e escreve a lista. Para realizar a leitura, o Cliente fica a ler continuamente as mensagens de resposta do Servidor até que este feche a conexão, “! Read (...) > 0”.

Para cada mensagem lida do fifo, a função imprime o nome do programa, contido na mensagem. Por fim, a função exclui o fifo criado com seu próprio PID, através do comando “unlink ()”.

2.5 Armazenamento de informação sobre programas terminados

A última funcionalidade do Servidor é o armazenamento dos programas terminados. Aquando a receção de uma mensagem de “Finish”, o Servidor para além de atualizar as estruturas de dados utilizadas também escreve para um ficheiro tal como referido no enunciado. Para esse efeito, o Servidor usa algumas funções auxiliares de construção de “path” e do nome do ficheiro pretendido. Ainda a pedido do enunciado, o “path” é passado ao monitor.

Capítulo 3

Testes

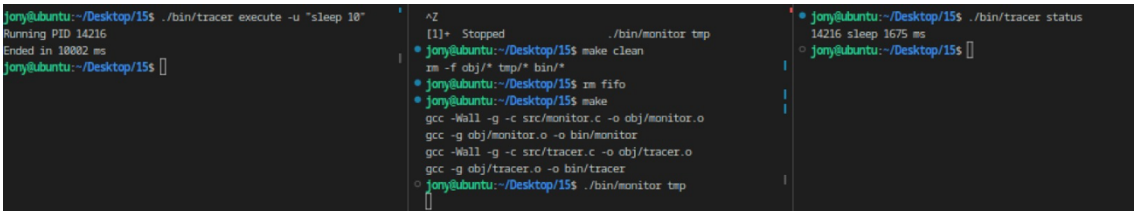

Figura 2 – Teste à funcionalidade "status" e "execute -u"

Tal como foi explicado anteriormente a função status retorna o tempo que o programa está em execução. Uma vez que, "./bin/tracer status" foi executado pouco tempo depois, aproximadamente 1 segundo depois de ./bin/tracer execute -u "sleep 10", faz sentido o resultado obtido.

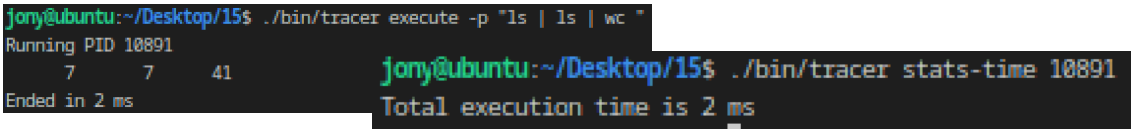

Figura 3 – Teste à funcionalidade "execute -p" e "stats-time"

Uma vez que o valor do tempo despendido para a realização do comando "execute -p" é igual em ambos, podemos concluir que o teste foi concluído com sucesso.

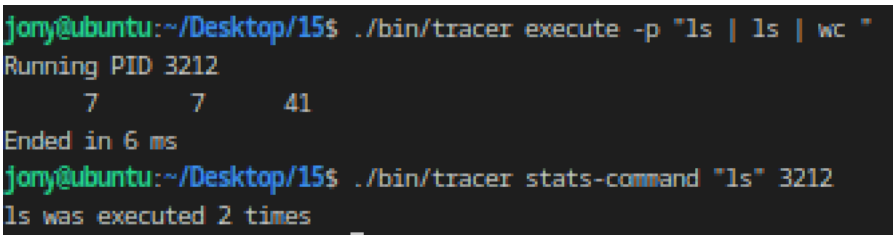

Figura 4 – Teste à funcionalidade "stats-command"

Dado que o comando, "execute -p "ls | ls | wc", foi executado apenas uma vez, tal como exemplificado na figura 3, e na figura 4 o "output" mostra que o processo "ls" foi executado duas vezes, concluímos que a função está a funcionar corretamente.

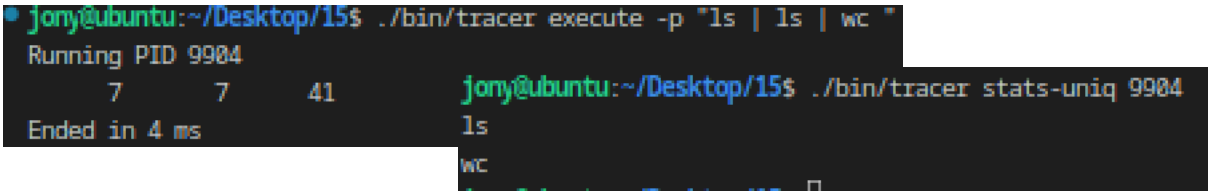

Figura 5 – Teste à funcionalidade "stats-uniq" em dois casos distintos

Testando a função “stats-uniq” para o caso onde o programa está a executar em paralelo dois programas diferentes e para um outro quando o mesmo executa para dois programas iguais, percebemos que o “output” está de encontro com o espero, confirmando o bom funcionamento da função.

Após testar as diferentes funcionalidades, também quisemos mostrar que o Servidor permite processamento em paralelo. Para isso utilizamos um comando que envia 5 processos “sleep 10” para o Servidor. Caso o Servidor esteja a correr sequencialmente, o tempo de execução real seria o somatório do tempo de execução individual de cada programa. Contudo, o nosso Servidor possibilita o processamento em paralelo, tendo apenas demorado 10 segundos, que corresponde ao tempo execução máximo entre os vários processos enviados.

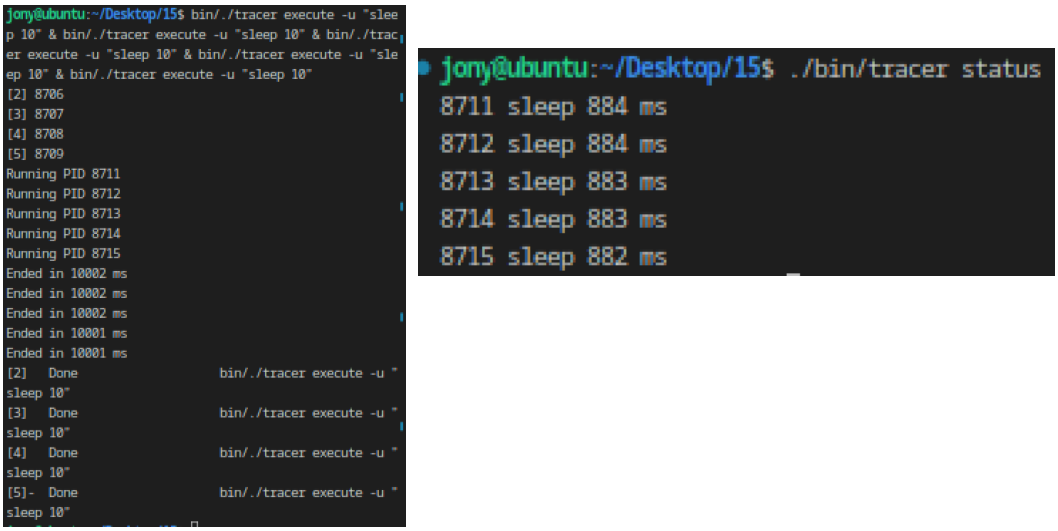

Figura 6 – Teste ao processamento em paralelo de vários processos.

Por fim, testamos se o Servidor bloqueava ou se entrava em espera ativa. Para isso enviamos um “ls” ao Servidor e com o comando “top” num terminal separado ficamos a monitorizar o uso de “cpu” do monitor. O que observamos foi que de facto o Servidor bloqueia no comando “write” do “pipe com nome” que recebe informação do Cliente. Este comportamento deve se ao facto de o próprio Servidor abrir um descritor de escrita no “pipe” em questão, o que faz com que haja sempre algum processo disponível para escrever, bloqueando assim o Servidor.

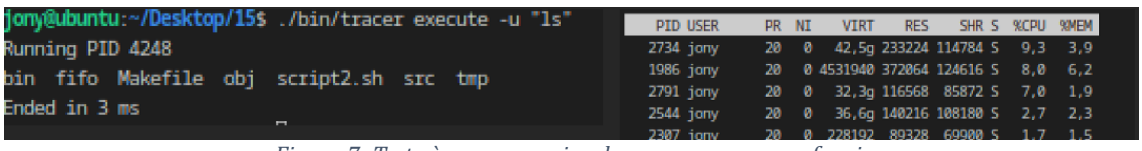

Figura 7- Teste à espera passiva do nosso programa a funcionar

Capítulo 4

Arquitetura e decisões tomadas

Face ao problema apresentado ficou claro desde o início que iriamos ter dois programas principais: Cliente e Servidor, e que o desafio seria criar uma forma de comunicação entre ambos que permitisse processamento paralelo por parte do Servidor, uma forma rápida de guardar e consultar dados, e também que não recorresse a esperas ativas. Deste modo, o grupo escolheu uma arquitetura que garantisse os pontos descritos anteriormente.

Primeiramente, o mecanismo escolhido para a comunicação entre Servidor e Cliente são as "pipes com nome". A "pipe" inicial é criada pelo Servidor e é sempre utilizada pelo Cliente para fazer um pedido ao Servidor ou a dar a conhecer informação relativa aos processos em execução. Além dessa "pipe", em funcionalidades onde o Servidor precisa de dar informação ao Cliente é utilizada uma "pipe" diferente que utiliza o PID do Cliente, pois é um dado que apenas o Cliente em questão e o Servidor conhecem e nunca é repetido. Neste tipo de funcionalidades, de forma a não bloquear o Servidor na interação Servidor-Cliente, o Servidor cria um processo filho capaz de fazer a comunicação pretendida. Este fenómeno é possível, uma vez que todos os processos filhos recebem uma cópia da memória do pai.

Como foi mencionado anteriormente, é importante escolher uma estrutura de dados que permita acessos rápidos aos dados dos processos guardados. Deste modo, optámos por utilizar uma "tabela de Hash" que permite processos ilimitados por "slot", pois é importante não perder informação. Assim, o Servidor está feito de forma a que consiga lidar rapidamente com as informações do Cliente permitindo na mesma travessias em que não haja alvos concretos, como por exemplo o comando "status". No Servidor não temos uma "tabela de Hash" mas sim duas que possuem propósitos diferentes. Uma das tabelas é destinada a controlar processos em execução enquanto a outra estrutura serve de histórico do programa. A primeira tabela dispõe de um mecanismo de limpeza que elemina, caso não houver nenhum programa em execução de momento, processos que já estejam em histórico dando espaço a novos processos.

Ainda nas estruturas de dados, é relevante mencionar o formato das mensagens e o seu papel na interação. A estrutura das mensagens foi escolhida tendo em conta fatores como, uma dimensão fixa, que informação era útil manter e a capacidade de poder serem guardadas dinamicamente. De forma a responder a esses requisitos, a nossa estrutura para as mensagens tomou a forma exposta na *Figura 1*, onde o nome do programa é guardado num "array" de tamanho fixo, o que garante o ponto de as dimensões ser iguais entre mensagens, contém um "enum" para o estado da mensagem de forma a ser ótima e, por fim, um apontador para outra lista, o que permite o desenvolvimento de estruturas dinâmicas por base na estrutura Mensagem.

Após concluirmos as funcionalidades básicas, passamos para as funcionalidades de estatísticas e para a execução em pipeline. O funcionamento de cada uma destas funcionalidades já foi mencionado anteriormente no relatório, mas há aspetos que merecem ser mencionados face ao conteúdo deste capítulo de forma a reforçar as decisões do grupo face a questões como as estruturas de dados. Analisando o processamento em pipeline, não há muita diferença no uso de estruturas face à execução de programas do utilizador. Contudo, observando as estatísticas, detetamos padrões no que é pedido que realçam a

importância de o uso de “Tabelas de Hash”. Um fator comum a todos as estatísticas é um conjunto finito de “PID” de programas que já executaram. Com isto em mente, e tendo em conta as propriedades de uma “Tabela de Hash”, vemos que o tempo de acesso à tabela é muito mais rápido e eficiente do que uma outra estrutura de dados que obrigasse uma travessia da estrutura de dados. Além disso, para a comando “stats-uniq” é bastante importante, além das propriedades já referidas, a capacidade de montar uma estrutura dinâmica de mensagens a ser enviadas ao Cliente.

Um aspeto importante a considerar do nosso projeto é a questão de guardar informação dos programas em memória. Um dos problemas que as “Tabelas de Hash” possuem é a necessidade de utilizar memória do programa, o que origina uma perda de histórico assim que o Servidor é desligado. Visto que uma das funcionalidades requisitadas corresponde à criação de ficheiros com informações de programas, o grupo achou por bem aproveitar esse histórico. Assim, quando o Servidor é executado, a primeira funcionalidade é carregar de novo os processos terminados para a estrutura de dados. Deste modo, certificamo-nos que não há programas que ficam perdidos para as diversas estatísticas.

Capítulo 5

Conclusão

Diante do exposto, podemos afirmar que o nosso trabalho cumpre os requisitos pedidos incluindo as funcionalidades avançadas. Os testes que o grupo fez, para além dos pequenos excertos mostrados, incluem variedade tanto em tamanho como no tipo de programas pedidos Cliente que, por sua vez, ambicionam provar que o trabalho foi concretizado sem nenhuma falha e, mais importante, que as escolhas a nível de conteúdos da Unidade Curricular foram devidamente aplicadas. As questões de processamento em paralelo e de espera passiva, apesar de possuírem soluções simples, foram as mais desafiantes e provaram ter na prática um desempenho interessante em contexto académico.

Por fim, achamos que o trabalho está bem concebido e que demonstramos o uso correto dos conceitos da cadeira de Sistemas Operativos.