Universidade do Minho

Braga, Portugal

TRABALHO PRÁTICO - RELATÓRIO

Orquestrador de Tarefas

Sistemas Operativos

Departamento de Informática

Engenharia Informática 2023/24

Equipa de Trabalho:

AXXXXXX - Nome

AXXXXXX - Nome

AXXXXXX - Nome

AXXXXXX - Nome

Índice

1. Introdução

O objetivo deste relatório é apresentar a implementação de um orquestrador de tarefas. O sistema, desenvolvido em C, é composto por um cliente e um servidor que comunicam através de pipes com nome (`FIFOS`). O cliente, que opera através do terminal, envia comandos para o servidor, que os interpreta e executa as tarefas correspondentes, suportado por uma política de escalonamento pré-definida e pela capacidade de poder executar múltiplos processos em simultâneo, informando assim o cliente sobre o identificador da tarefa, cujo resultado é posteriormente guardado em um ficheiro de texto unicamente identificado. O servidor é capaz de executar várias tarefas em paralelo e de as escalonar de acordo com a politica de escalonamento definida previamente no arranque do servidor. A comunicação entre o cliente e o servidor é assegurada através do uso de `FIFOS`, garantindo que os comandos e as respostas sejam transmitidos de forma segura e eficiente. Este projeto foi desenvolvido com recurso às chamadas de sistema do UNIX, tais como `fork`, `pipe`, `dup2`, `execvp`, `wait`, entre outras, que são cruciais para o desenvolvimento de um sistema de orquestração de tarefas eficiente e responsivo. Assim, este relatório apresenta a implementação do sistema, descrevendo a comunicação entre o cliente e o servidor, o modo de execução de tarefas e o escalonamento das mesmas. Adicionalmente, são apresentados os resultados de testes de desempenho realizados ao orquestrador de tarefas, que permitem avaliar a eficiência e o uso de diferentes politicas de escalonamento.

2. Comunicação entre Cliente e Servidor

2.1. Servidor

Após a sua inicialização e da verificação dos parâmetros de entrada, o servidor cria um `FIFO` geral para receber os vários pedidos dos clientes. Este `FIFO` serve como um canal de comunicação centralizado através do qual todos os pedidos são encaminhados para o servidor. Assim, após a leitura do pedido enviado pelo cliente, o servidor utiliza a função command_interpreter, definida no módulo command_interpreter.c, para interpretar o request. Esta função é responsável por identificar não só o comando enviado do cliente, como também obter o id do cliente. Este id é crucial pois vai ser usado posteriormente para identificar o `FIFO` do cliente, com o objetivo de enviar uma resposta em função do pedido feito.

2.2. Cliente

Após o arranque do servidor, o cliente pode começar com os seus pedidos. Inicalmente, o cliente cria um `FIFO` com o nome do seu `PID`, que será utilizado para receber as respostas do servidor. De seguida, o cliente envia um pedido ao servidor através do `FIFO` geral. O envio de toda a informação é assegurado pela função command_sender, que está definida no módulo command_sender.c. Esta função encapsula a lógica necessária para formatar e transmitir os comandos de forma eficiente e sem erros, garantindo que o servidor recebe a informação completa. Este pedido é lido pelo servidor, que o interpretará e executará a tarefa correspondente. Após a análise do pedido, o servidor envia uma resposta ao cliente através do `FIFO` do cliente, que é idenificado unicamente pelo `PID` do cliente.

3. Execução de Tarefas

No sistema de orquestração desenvolvido, a execução de tarefas é a funcionalidade chave que permite ao servidor processar e executar os pedidos feitos pelos clientes, manipulando tanto tarefas simples como as compostas por pipelines de comandos. Aliado a todos estes fatores, o escalonamento de tarefas é também uma funcionalidade importante, que permite ao servidor gerir de forma eficiente o processamento de tarefas, garantindo que o sistema não fica sobrecarregado e que as tarefas são executadas de forma eficiente. Assim, tal como referido anteriormente, o uso das chamadas ao sistema foi a chave para a implementação de todas estas funcionalidades e pelo correco funcionamento do sistema de orquestração de tarefas. Em seguida, serão apresentadas as principais funcionalidades do sistema, nomeadamente a execução de tarefas simples e compostas, bem como a estratégia utilizada para o escalonamento de tarefas.

3.1. Limite de Tarefas em Paralelo

Na inicialização do servidor, um dos parâmetros de entrada é o número máximo de tarefas que podem ser executadas em paralelo, o que permite uma maior flexibilidade conforme as necessidades e capacidades do servidor. Este limite é crucial para garantir que o servidor não fica sobrecarregado com um número excessivo de tarefas, o que poderia levar a uma degradação do desempenho.É mantida uma lista de tarefas em execução, que é atualizada sempre que uma tarefa termina e um contador de tarefas em execução, que é incrementado sempre que uma nova tarefa é iniciada e decrementado sempre que uma tarefa termina. Assim, o servidor é capaz de gerir de forma eficiente o número de tarefas em execução. O uso do mecanismo de `fork` é fundamental para alcançar o processamento de multiplas tarefas de forma eficaz , permitindo que o servidor crie um novo processo para cada tarefa a ser executada. Assim, o servidor pode executar várias tarefas em simultâneo, garantindo que o sistema é capaz de processar pedidos de forma eficiente e sem bloquear o servidor/cliente. Todas as tarefas que não conseguem ser executadas de forma imediata, são colocadas numa fila de espera, que é processada assim que o número de tarefas em execução seja inferior ao limite definido.

4. Processos simples

A execução de tarefas simples é feita através da função execute_simple_process, que é responsável não só pela execução do comando, como também por redirecionar a saída do mesmo para um ficheiro de texto, que é unicamente identificado. O uso do mecanismo de dup2 é crucial para redirecionar a saída do comando e dos erros para o ficheiro de texto, garantindo que a informação é guardada de forma correta e sem erros. Para facilitar a execução do comando,a instrução recebida é cuidadosamente fragmentada em tokens. Esta segmentação é crucial porque o execvp, utilizado para executar o comando, requer que o comando e os seus argumentos sejam fornecidos como um array de strings terminado em NULL. Esse processo de tokenização assegura que o mesmo seja interpretado e executado corretamente pelo sistema, permitindo uma gestão eficaz dos recursos do servidor e a correta execução das tarefas conforme solicitado pelo cliente. Comparativamente, a execução de processos simples é consideravelmente mais direta e menos trabalhosa do que a execução de processos compostos, uma vez que não envolve a comunicação entre processos e a gestão de pipes. Assim, esta simplicidade permite uma implementação mais rapida e menos propensa a erros.

4.1. Processos compostos

A execução de tarefas compostas é feita através da função execute_pipeline_process, que tal como a função execute_simple_process, é responsável pela execução do comando e pelo redirecionamento da saída para um ficheiro de texto. No entanto, a execução de tarefas compostas é mais complexa, uma vez que envolve a execução de vários comandos em sequência, com a saída de um comando a ser redirecionada para a entrada do comando seguinte. Todo este mecanismo foi possivel graças ao uso de pipes, que permitem a comunicação entre os processos de forma eficiente e sem erros.Na execução destes processos compostos, cada um dos comandos é executado em processos separados, cada qual tratado por uma instância de processo filho criada através da chamada ao sistema `fork()`. A comunicação entre esses processos é estabelecida através de pipes, que são elementos cruciais para a execução sequencial e interdependente dos comandos. Cada pipe é constituído por um par de `file descriptors`: um para leitura e outro para escrita. Deste modo, o stdout de um processo é ligado diretamente ao stdin do próximo processo na cadeia através do file descriptor correspondente. Ao usar esta técnica de `pipelining`, a saída de dados de um comando não precisa de ser armazenada em disco antes de ser lida pelo comando seguinte. Isso reduz a latência e o overhead associado à leitura e escrita em sistemas de armazenamento, resultando numa execução mais rápida e eficiente das tarefas, otimizando os recursos do sistema e permitindo assim um fluxo de processamento mais fluido entre os comandos envolvidos. Para garantir a integridade e a correta execução dos processos, é essencial gerir adequadamente os recursos do sistema. Isto inclui fechar apropriadamente os file descriptors dos pipes que não são mais necessários, evitando assim leaks de recursos e possíveis erros de execução. Adicionalmente, cada processo filho verifica se é o último na cadeia de comandos. Caso seja, o seu output é redirecionado para um ficheiro de saída devidamente identificado em função do identificador da tarefa, onde os resultados finais são armazenados. Finalmente, o processo pai (que neste caso também é um processo filho) aguarda o término de todos os processos filhos, utilizando `wait()`. Esta espera assegura que todos os comandos na pipeline foram executados até a conclusão e que todos os recursos associados foram corretamente libertados. Em suma, o principal interveniente na execução de tarefas compostas é o uso de pipes, que permite toda esta comunicação entre processos de forma eficiente e sem erros.

5. Escalonamento de Tarefas

Texto

5.1. Servidor

Texto

6. Titulo

Texto

6.1. Tabela

column1	column2
row1	row2

Tabela 1: Tabela 1.