TP2 Relatório - Grupo 25

Diogo Outeiro - a104092
José Soares - a103995
Nuno Melo - a104446

Maio 2024

Contents

List of Figures

1 Introdução

Este projeto tem como objetivo desenvolver um serviço de orquestração de tarefas em um sistema computacional. O propósito é permitir que os utilizadores submetam tarefas para execução num servidor, indicando a duração estimada da tarefa e o programa ou conjunto de programas a serem executados. O servidor é responsável por escalonar e executar essas tarefas de maneira eficiente, mantendo todos os registos relevantes para análises posteriores.

Neste relatório, apresentaremos de forma sucinta a solução desenvolvida pelo nosso grupo, focando especialmente na arquitetura de processos adotada e no uso dos mecanismos de comunicação. Exploraremos como esses elementos foram fundamentais para o desenvolvimento de um sistema eficiente de orquestração de tarefas. Também ao longo deste documento, iremos justificar nossas escolhas arquiteturais e demonstrar como elas contribuíram para o sucesso do projeto.

2 Resumo

Como já foi referido, o serviço consiste em um cliente, responsável por enviar solicitações de execução de tarefas, e um servidor, encarregado de receber essas solicitações, escalonar as tarefas e executá-las conforme necessário.

No nosso trabalho prático implementamos cada uma das funcionalidades do enunciado, que se encontram a seguir explicadas de forma breve. Além disso, foi pedido no enunciado a execução e avaliação de eficiência do processamento de tarefas.

2.1 Funcionalidades Básicas Implementadas:

Execução de Tarefas do Usuário:

- Desenvolvemos um cliente capaz de receber informações do utilizador, como o tempo estimado e o programa a ser executado.
- Implementamos no servidor a capacidade de receber e escalonar essas tarefas, redirecionando a saída padrão e de erro para arquivos com base no identificador da tarefa.
- Após a conclusão da tarefa, registamos o identificador da tarefa e seu tempo de execução em um arquivo persistente.

Consulta de Tarefas em Execução:

- Criamos no cliente a funcionalidade de consultar o status das tarefas em execução, em espera e concluídas.
- Implementamos no servidor o processamento dessas consultas, retornando as informações solicitadas ao cliente.

Encadeamento de Programas e Processamento Paralelo Implementados:

- Desenvolvemos no cliente a capacidade de suportar a execução encadeada de programas (pipelines), permitindo aos utilizadores encadear vários programas em uma única tarefa.
- Implementamos no servidor a execução dessas pipelines de programas conforme especificado pelos utilizadores.

Processamento de Várias Tarefas em Paralelo:

- No servidor, implementamos a capacidade de processar um número definido de tarefas em paralelo.
- O algoritmo de escalonamento foi utilizado para aumentar a velocidade de processamento das tarefas, levando em consideração o número de tarefas que podem ser executadas simultaneamente.

2.2 Testes e Desempenho

O projeto inclui testes para avaliar a eficiência da política de escalonamento implementada, comparando-a com outras políticas. Testes são realizados com diferentes configurações de paralelização do servidor para compreender seu impacto na experiência do utilizador, como o tempo médio de execução das tarefas. O relatório incluirá uma reflexão sobre os resultados dos testes realizados, bem como a utilização de scripts para automatizar os testes.

3 Arquitetura do Projeto

A arquitetura de processos desempenha um papel crucial no desenvolvimento de um sistema de orquestração de tarefas eficiente. Ao projetar a arquitetura do nosso sistema, foi fundamental considerar questões como concorrência, comunicação entre processos e escalonamento de tarefas.

Para atender aos requisitos do projeto, optamos por uma arquitetura cliente-servidor. Nessa arquitetura, o cliente é responsável por interagir com os utilizadores, enquanto o servidor é encarregado de executar e gerenciar as tarefas submetidas pelos utilizadores.

- **Cliente**: O cliente é implementado como um processo separado que recebe as solicitações dos utilizadores e as encaminha para o servidor. E este é também responsável por exibir a confirmação da tarefa pelo servidor. Esta abordagem permite uma separação clara de responsabilidades.
- **Servidor**: O servidor é o núcleo do sistema, responsável por receber, escalonar e executar as tarefas submetidas pelos utilizadores. No nosso caso também gerencia o estado das tarefas e fornece informações sobre o `status` das tarefas em execução, em espera e concluídas. O servidor é implementado como um processo separado para garantir que possa lidar com múltiplos clientes de forma concorrente.

De forma simples, a interação entre o cliente e o orquestrador ocorre por meio de comunicação via pipes com nome, onde para o nosso projeto, trabalhamos apenas com um FIFO que liga diretamente todos os clientes que aparecerem ao servidor, sendo este o único ponto de acesso ao orchestrator. No entanto, como ainda necessário passar informação por parte do servidor, para cada cliente é também criado um pipe nomeado responsável por fazer a ligação entre os dois processos.

O funcionamento do nosso programa tem por base os seguintes passsos:

1. O cliente inicia e recebe um argumento da linha de comando indicando a operação desejada (por exemplo, execute, status, exit).
2. Se a operação for executar uma nova tarefa (execute), o cliente cria uma estrutura de dados contendo informações sobre a tarefa, como o tempo estimado de execução e o programa a ser executado. Essa estrutura é então enviada para o orquestrador através de um pipe com nome.
3. O orquestrador, ao receber a solicitação do cliente, trata da receção e da decisão para saber se a tarefa deve ser executada ou posta em fila de espera.
4. Caso a tarefa efetivamente seja para executar, ou exista uma com prioridade (definida pelo método de escalonamento) pronta para execução, é criado um novo processo encarregado de controlar a tarefa, sendo que este além de criar de novo outro processo para realmente executar o comando, também faz a espera depois para enviar ao servidor a resposta de que o comando foi executado. Assim o servidor nunca entra em bloqueio, a não ser pela leitura do fifo principal, pois para cada tarefa é feito um processo á parte para o tratamento de cada tarefa em específico.
5. Na parte da execução real do comando passado pelo utilizador são feitos todos os redirecionamentos, escritas e medidas de tempo necessárias e assim é concluído o processo para o comando inicial.

Essa interação permite que o cliente envie solicitações ao orquestrador para executar tarefas ou verificar o status das tarefas em execução. O orquestrador, por sua vez, gerencia a execução das tarefas e fornece feedback ao cliente conforme necessário (identificador ou status).

Na figura seguinte temos uma representação básica do funcionamento do nosso programa, onde cada retângulo representa um processo.

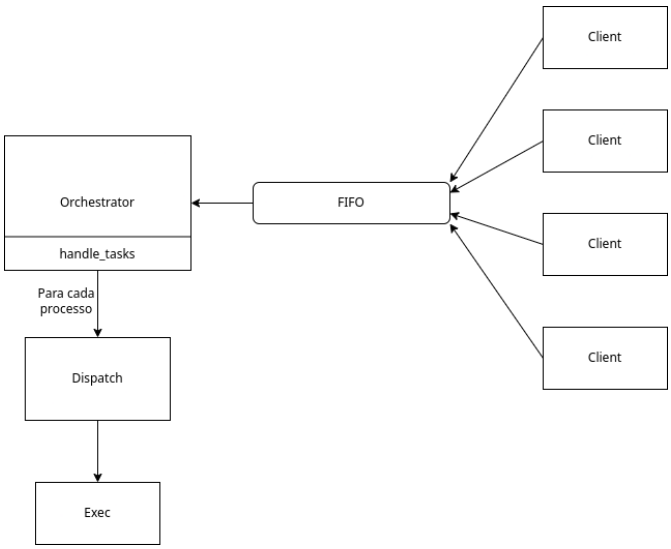

Figure 1: Arquitetura Básica

3.1 Arquitetura de Processos

Na função `dispatch`, um processo filho é criado utilizando a chamada de sistema `fork()`. Dentro deste processo filho, a função `exec_function` é chamada para executar a tarefa solicitada. Após a execução da função `exec_function`, o processo filho termina sua execução utilizando `_exit()`.

Na função `handle_tasks` do orquestrador, quando uma nova tarefa é recebida, o orquestrador verifica se o número de tarefas em execução é menor que o número máximo permitido de tarefas em paralelo. Se for o caso, o orquestrador cria um novo processo filho para executar a tarefa. Este processo filho, por sua vez, executa a função `dispatch` para executar a tarefa solicitada. Assim, o orquestrador não fica bloqueado enquanto espera pela conclusão da execução da tarefa, pois delega essa responsabilidade para os processos filhos.

Além disso, é importante observar que, após a execução da tarefa, o processo filho termina sua execução utilizando `_exit()`, garantindo que recursos do sistema sejam adequadamente libertados e que o processo pai não seja bloqueado pela execução do filho.

Dessa forma, tanto na função `dispatch` quanto na função `handle_tasks`, a construção de processos é realizada de forma a garantir que o servidor nunca fique bloqueado enquanto espera pela conclusão das tarefas executadas pelos processos filhos.

No fim da execução é enviada a estrutura de dados da `task` mas com a mensagem de `WAIT` e o servidor faz a espera para recolher o estado da tarefa iniciada em `handle_tasks`.

Na representação seguinte temos de forma simples todo o processo pensado para conceber a construção de processos.

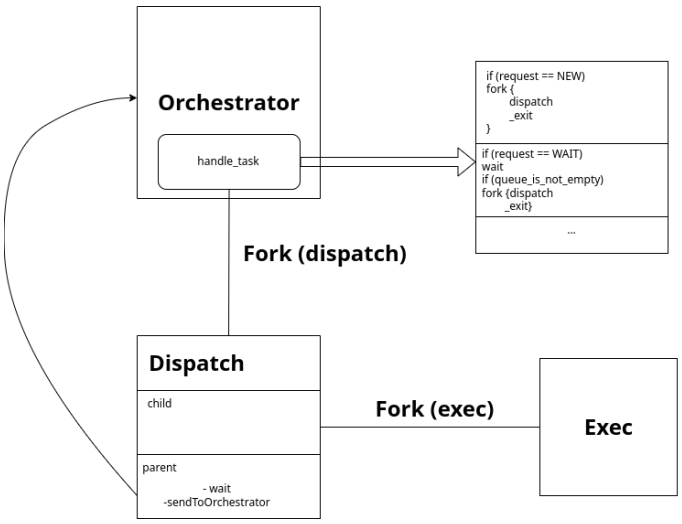

Figure 2: Arquitetura de Processos

3.2 Mecanismos de Comunicação

No âmbito da comunicação entre cliente e servidor, optamos pelo uso de pipes nomeados. Os pipes nomeados oferecem uma forma eficiente e bidirecional de troca de dados entre processos independentes. Essa escolha permite uma comunicação assíncrona e sincronizada entre o cliente e o servidor, garantindo uma interação eficiente e coordenada.

Criação dos Pipes Nomeados

No início do código do cliente e do servidor, há chamadas para criar pipes nomeados. Por exemplo, no servidor, é criado apenas um pipe nomeado chamado `MAIN_FIFO_SERVER`. Isso é feito pela função `make_fifo` nos dois processos.

Abertura dos Pipes Nomeados

O cliente e o servidor abrem os pipes nomeados para leitura e escrita. No servidor, o pipe é aberto com permissões de leitura e escrita (`O_RDONLY` e `O_WRONLY`), e isto dá se para nunca existir o problema de EOF (End of File) a quando da submissão de tarefas. No cliente, o pipe nomeado chamado `MAIN_FIFO_SERVER` é aberto principalmente para escrita (`O_WRONLY`) e depois é aberto um pipe nomeado respetivo ao seu processo `FIFO_PID` para envio do identificador de tarefa por parte do servidor.

Escrita nos Pipes Nomeados

No cliente, após a preparação dos dados da tarefa, estes são escritos no pipe nomeado através de uma estrutura de dados usando a função `write`.

Leitura nos Pipes Nomeados

No servidor, há um loop infinito que lê continuamente os dados do pipe nomeado. Quando o servidor recebe os dados da tarefa, é responsável por processar o pedido conforme necessário.

4 Avaliação e Testes de Eficiência

Nesta secção, apresentamos a avaliação de duas políticas de escalonamento implementadas no sistema: First-Come, First-Served (FCFS) e Shortest Job First (SJF). Essas políticas determinam a ordem de execução das tarefas com base em diferentes critérios, o que pode influenciar o desempenho e o tempo de resposta do sistema.

4.1 First-Come, First-Served (FCFS)

A política FCFS é uma das políticas de escalonamento mais simples, em que as tarefas são executadas na ordem em que foram recebidas pelo servidor. Isso significa que a primeira tarefa a chegar é a primeira a ser executada, independentemente da sua duração ou exigência de recursos. A simplicidade desta política pode ser uma vantagem em cenários de baixa carga, mas pode levar a longos tempos de espera para tarefas grandes ou de longa duração.

4.2 Shortest Job First (SJF)

A política SJF prioriza a execução das tarefas mais curtas primeiro. Isso significa que, entre todas as tarefas pendentes, a que possui o menor tempo de execução é selecionada para execução em seguida. Essa política visa minimizar o tempo médio de espera das tarefas na fila e pode ser especialmente eficaz em situações em que há uma grande variação no tempo de execução das tarefas.

4.3 Testes e Avaliação

Foram realizados testes para avaliar o desempenho das políticas FCFS e SJF em diferentes cenários de carga de trabalho e tipos de tarefas. A seguir, enumeramos os resultados obtidos durante os testes:

1. Tempo total de execução das tarefas em cada política.
2. Comparação do desempenho das políticas sob diferentes cargas de trabalho.

Esses resultados nos permitirão avaliar a eficácia das políticas de escalonamento implementadas e identificar possíveis áreas de melhoria no sistema de orquestração de tarefas.

4.4 Teste de Eficiência das Políticas de Escalonamento

Os testes de eficiência das políticas de escalonamento foram fundamentais para avaliar o desempenho e a capacidade de resposta do sistema em diferentes cenários de carga de trabalho. Nesses testes, buscamos compreender como o sistema reage e se adapta a diferentes condições de uso.

- `./bin/client execute 1000 -u "cat largefile.txt"`: Este comando executa a leitura do arquivo "largefile.txt" por 1000 milissegundos. Espera-se um tempo de execução próximo de 1000 ms.
- `./bin/client execute 100 -u "cat src/client.c"`: Executa a leitura do arquivo "client.c" por 100 ms. Espera-se um tempo de execução próximo de 100 ms.
- `./bin/client execute 1500 -p "cat largefile.txt | wc"`: Este comando executa a contagem de palavras do arquivo "largefile.txt" usando o comando `wc` após a leitura. Espera-se um tempo de execução próximo de 1500 ms.

- `./bin/client execute 100 -u "cat src/controler.c"`: Executa a leitura do arquivo "controler.c" por 100 ms. Espera-se um tempo de execução próximo de 100 ms.

Nota: O arquivo `largefile.txt` tem 500MB.

Estes comandos foram executados em paralelo, com um total de 4 processos simultâneos. O tempo de execução estimado para cada comando foi fornecido com base na sua natureza e duração especificadas. Os resultados reais obtidos durante os testes serão apresentados na tabela de resultados para análise comparativa entre as políticas de escalonamento SJF e FCFS.

Comando	**Tempo Médio (SJF)**	**Tempo Médio (FCFS)**
cat largefile.txt	872.5 ms	1152.5 ms
cat src/client.c	358 ms	533 ms
cat largefile.txt — wc	1384 ms	1555 ms
cat src/controler.c	379.5 ms	537 ms

Table 1: Resultados do teste de eficiência com SJF e FCFS

Esses resultados mostram o tempo médio de execução para cada comando sob as políticas de escalonamento SJF e FCFS. Como observado, o SJF geralmente apresenta um tempo médio de execução ligeiramente melhor em comparação com o FCFS em todos os casos. Isso sugere que o SJF pode ser mais eficiente em termos de tempo médio de execução das tarefas, proporcionando uma melhor experiência para os usuários em geral.

4.5 Teste de Desempenho do Servidor com Diferentes Níveis de Paralelismo

Para avaliar o desempenho do servidor em lidar com múltiplas tarefas simultaneamente, realizamos uma série de testes utilizando o comando `./test_client.sh`. O objetivo desses testes foi observar como o tempo de execução das tarefas variava conforme aumentávamos o nível de paralelismo. Neste processo o servidor seguia a política FCFS dado que o importante neste teste era avaliar o tempo médio de execução, e para isso, utilizamos sempre o valor da última tarefa a ser processada. Os testes foram realizados 7 vezes onde retiramos o melhor e o pior valor, no final foi feito uma média que usamos para construir o gŕafico.

Para cada teste, configuramos o servidor para executar 20 tarefas, cada uma com um tempo estimado de 1000 milissegundos. Essas tarefas consistiam na execução do comando `cat` em um arquivo grande (`largefile.txt`) com 500 megabytes de tamanho.

Realizamos testes com diferentes níveis de paralelismo: executando um número máximo de tarefas em paralismo de 1, 2, 3 e 5. Observamos que, à medida que aumentávamos o nível de paralelismo, o tempo total de execução das tarefas tendia a diminuir.

Essa diminuição no tempo de execução pode ser atribuída à capacidade do sistema de processar várias tarefas simultaneamente. Quando executamos tarefas em paralelo, aproveitamos melhor os recursos do sistema, como CPU e memória, resultando em uma conclusão mais rápida das tarefas.

É importante ressaltar que esses resultados podem variar dependendo do ambiente de execução, do hardware do servidor e da natureza das tarefas. No entanto, os testes fornecem insights sobre como o servidor se comporta sob diferentes cargas de trabalho e níveis de paralelismo.

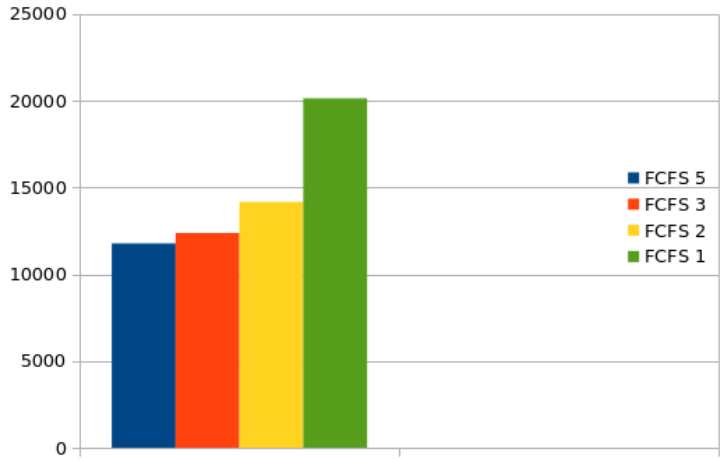

Figure 3: Representação do tempo médio de execução para cada nível de paralelismo

FCFS 5	**FCFS 3**	**FCFS 2**	**FCFS 1**
11044	12228	15941	20026
11864	12626	13646	19518
11677	12659	14154	20375
12284	12273	13349	20927
12018	12121	13773	19846
AVG 11777.4	**AVG** 12381.4	**AVG** 14172.6	**AVG** 20138.4

Figure 4: Dados de tempo médio para cada algoritmo FCFS.

5 Conclusão

O desenvolvimento deste sistema de orquestração de tarefas representou um desafio interessante para o nosso grupo, pois envolveu toda uma implementação de uma arquitetura entre cliente e servidor, bem como a adoção de políticas de escalonamento. Ao longo deste projeto, destacamos os seguintes pontos-chave:

- **Arquitetura do Sistema:** Optamos por uma abordagem cliente-servidor, que permitiu uma clara separação de responsabilidades e facilitou a comunicação assíncrona entre os diferentes componentes do sistema.
- **Implementação Funcionalidades:** Conseguimos implementar com sucesso todas as funcionalidades exigidas pelo enunciado, incluindo a execução de tarefas do usuário, a consulta do status das tarefas em execução e o encadeamento de programas.
- **Políticas de Escalonamento:** Avaliamos duas políticas de escalonamento - First-Come, First-Served (FCFS) e Shortest Job First (SJF) - e observamos que o SJF tende a oferecer um tempo médio de execução melhor, especialmente em cenários com uma grande variação no tempo de execução das tarefas.
- **Desempenho do Servidor:** Ao testar o servidor com diferentes níveis de paralelismo, constatamos uma diminuição no tempo total de execução das tarefas à medida que aumentamos o número máximo de tarefas em paralelo. Isso demonstra a capacidade do sistema de processar eficientemente múltiplas tarefas simultaneamente.

Em suma, o nosso sistema de orquestração de tarefas mostra-se promissor em termos de eficiência e desempenho. Por fim, consideramos todo o projeto uma excelente forma de aplicar os conceitos estudados nas aulas, além de expandir nosso conhecimento na área de programação. A experiência de desenvolver um sistema completo, desde a concepção da arquitetura até a implementação das funcionalidades, foi enriquecedora e proporcionou-nos uma visão mais ampla sobre o funcionamento e a complexidade de uma "parte" do que representa um sistema operativo.