Universidade do Minho
Licenciatura em Engenharia Informática

Projeto SO

Orchestrator Service

Grupo 27
Mariana Rocha - a90817
Hugo Abelheira - a95151
Ana Pires - a96060

7 de maio de 2024

Conteúdo

Lista de Figuras

1 Introdução

O presente relatório apresenta o desenvolvimento de um sistema de processamento de tarefas. Este sistema foi projetado para gerir e executar tarefas de forma eficiente e organizada, com o objetivo de automatizar processos computacionais.

A implementação do sistema foi realizada em C, utilizando conceitos avançados de programação, como manipulação de processos, comunicação entre processos através de pipes e uso de chamadas ao sistema do sistema operacional Unix/Linux.

O relatório detalha a arquitetura do sistema, destacando as decisões tomadas durante o processo de desenvolvimento. Além disso, são apresentados exemplos de código significativos para ilustrar a implementação prática do sistema.

Por fim, são discutidos os resultados obtidos, incluindo a eficiência do sistema na execução de tarefas, possíveis limitações e sugestões para melhorias.

2 Arquitetura do Projeto

A arquitetura do projeto é composta por três componentes principais: o cliente, o orquestrador e os *managers* de tarefas. O cliente é responsável por enviar tarefas para execução, enquanto o orquestrador é encarregado de distribuir essas tarefas entre os diferentes *managers* de tarefas disponíveis e adicionar as tarefas a uma fila de espera. Por sua vez, os *managers* de tarefas são responsáveis pela execução das tarefas recebidas.

No centro dessa arquitetura está o orquestrador, que atua como um intermediário entre os clientes e os executadores de tarefas. O orquestrador recebe as tarefas dos clientes, organiza-as numa fila de espera e faz a distribuição para os *managers* de tarefas disponíveis. Essa abordagem permite uma distribuição eficiente da carga de trabalho.

Os clientes interagem com o orquestrador enviando pedidos contendo as tarefas a serem executadas. Essas tarefas são transmitidas por meio do *pipe* com nome designado por `SERVER`, permitindo uma comunicação entre o cliente e o orquestrador. Uma vez que o orquestrador recebe um pedido, ele processa-o e encaminha-o para um dos *managers* de tarefas disponíveis.

Os *managers* de tarefas são implementados como processos separados e são responsáveis por executar as tarefas atribuídas pelo orquestrador. Eles comunicam com o orquestrador através de *pipes* com nomes e são escalados para lidar com um grande volume de tarefas.

Essa arquitetura modular e escalável proporciona uma maior flexibilidade e capacidade de adaptação do sistema, permitindo a fácil adição de novos clientes e *managers* de tarefas conforme necessário. Isso torna o sistema adequado para lidar com diferentes cargas de trabalho e requisitos de desempenho.

3 Comunicação Servidor Cliente

Na comunicação cliente-servidor, dois sistemas distintos, o cliente e o servidor, interagem entre si para trocar informações e realizar determinadas tarefas. Esta comunicação é geralmente baseada num modelo de solicitação onde o cliente envia pedidos para o servidor e resposta posterior para um ficheiro de *output*.

No contexto do projeto desenvolvido, a comunicação entre o cliente e o servidor é realizada por meio de *pipes* com nome (também conhecidos como *FIFOs*). No projeto, o servidor cria um *pipe* com nome - SERVER, e o cliente envia informações através desse *pipe*. O cliente envia pedidos que contêm tarefas individuais a serem executadas para o servidor através do *FIFO*. O servidor, por sua vez, lê os pedidos do *pipe*, processa as tarefas solicitadas e envia as respostas de volta para o cliente, no caso do *status*.

Segue em anexo a visualização de instruções de como iniciar o orchestrador e como enviar pedidos do cliente.

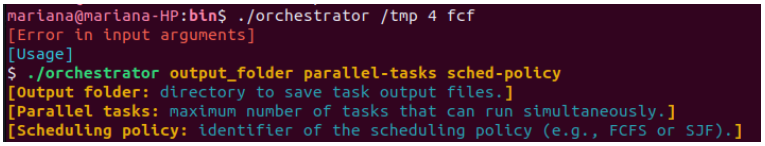

Figura 1: `./orchestrator`

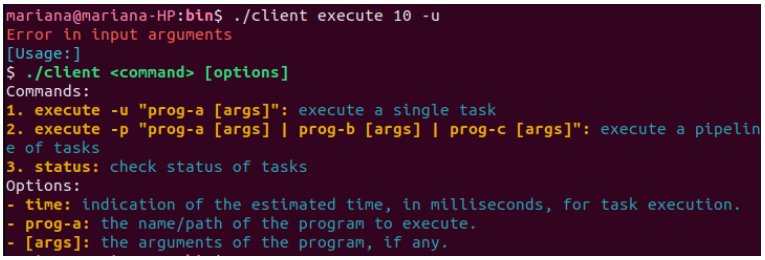

Figura 2: `./client`

4 A estrutura *Task*

A estrutura *Task* desempenha um papel fundamental no sistema de processamento de tarefas desenvolvido, pois representa uma unidade de trabalho que pode ser atribuída e executada pelo sistema. A estrutura *Task* é composta por diversos campos que contêm informações essenciais sobre a tarefa a ser executada. Entre esses campos, destacam-se o *PID* (identificador único do processo), que identifica exclusivamente cada tarefa, o tempo esperado de execução, que indica a estimativa de tempo necessário para completar a tarefa, e o comando a ser executado, que define a ação a ser realizada pela tarefa. Além disso, a estrutura *Task* também regista informações sobre o tempo de início e fim da execução da tarefa, o tempo gasto durante a execução e o identificador do *manager* responsável pela sua execução. Estes atributos permitem um acompanhamento preciso do ciclo de vida de cada tarefa e são essenciais para o funcionamento eficiente do sistema.

Esta estrutura é criada através de um *parser* quando o cliente solicita um pedido ao servidor, onde lhe é devolvido o identificador único do seu pedido. Este parser guarda toda a informação visível na imagem seguinte.

```c
typedef struct task{
    int pid; // identificador de cada processo
    struct timeval time_start; // tempo em que a tarefa comecou
    struct timeval time_end; // tempo em que a tarefa terminou
    int time_spent; // tempo que a task efetivamente demorou a executar
    int time_expected; // tempo que o utilizador pensa que a tarefa irá demorar
    int command_flag; // flag para determinar a que comando corresponde a task
    char exec_args[500]; // argumentos da task para a sua execução
    int manager_id; // identificador do manager
} Task;
```

Figura 3: Estrutura Task

4.1 Mensagens para o servidor

4.1.1 Status

A mensagem de *status* é enviada pelo cliente ao servidor para solicitar informações sobre o estado atual do sistema e das tarefas em execução. Essa mensagem é utilizada para monitorizar o progresso das tarefas e garantir o correto funcionamento do sistema como um todo. Ao receber uma mensagem de status, o servidor responde fornecendo detalhes sobre as tarefas agendadas, em execução e concluídas.

A função que executa o *status* no servidor é responsável fornecer as informações solicitadas. Esta abre um FIFO nomeado para escrita, onde enviará a saída da função para o cliente. Em seguida, redireciona a saída padrão *stdout* e a saída de erro *stderr* para esse FIFO usando a função *dup2*. Após isso, a função itera sobre a fila de tarefas, escrevendo detalhes sobre as tarefas agendadas, em execução e concluídas no FIFO. Além disso, lê do arquivo de *log* os detalhes das tarefas concluídas e escreve-os no FIFO. Finalmente, restaura a saída padrão e fecha os descritores de arquivo relevantes. As leituras e escritas são realizadas através de chamadas ao sistema, onde o orchestrador escreve através de `write`'s para o `fifo_idTask` o pedido pelo status e o cliente lê através de `read`'s o conteúdo escrito pelo servidor.

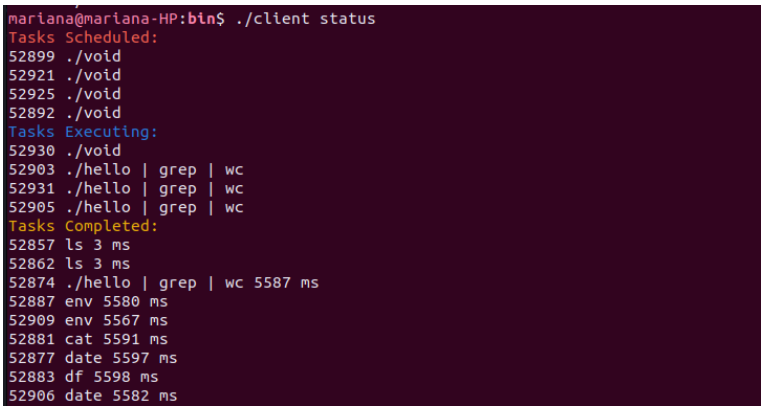

Figura 4: `./client status`

4.1.2 Execute -u

A mensagem de execução com a *flag* -u é enviada pelo cliente ao servidor para solicitar a execução de uma tarefa de forma independente. Nesse modo de execução, a tarefa é tratada como uma unidade indivisível e adicionada a uma fila de espera e quando possível é atribuída a um *manager* que se encontre disponível. Este tipo de mensagem é adequado para tarefas que não dependem de outras tarefas para serem executadas e que podem ser processadas de forma independente e paralela.

A função que executa as taks com a *flag* -u cria um processo filho utilizando `fork()` para executar o comando solicitado pela tarefa. O *standard output* e *standard error* do processo filho são redirecionados para o ficheiro de output contendo o id da task em execução e esta é executada através de `execvp`.

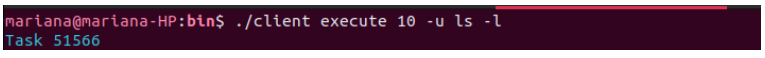

Figura 5: `./client execute 10 -u ls -l`

4.1.3 Execute -p

A mensagem de execução com a *flag* -p é enviada pelo cliente ao servidor para solicitar a execução de uma tarefa de forma concorrente, formando um pipeline de execução. No servidor, o *manager* processa essa mensagem, dividindo-a em comandos individuais e estabelecendo uma cadeia de processos para executá-los sequencialmente. Cada processo é executado através de `execvp`.

Cada comando é executado por um processo filho, que é criado utilizando `fork()`. Para coordenar a comunicação entre os comandos, são utilizados pipes anónimos, que ligam a saída de um comando à entrada do próximo. O primeiro comando no pipeline redireciona sua saída para um pipe, enquanto os comandos intermediários redirecionam sua entrada do pipe anterior e a saída para o próximo pipe. O último comando no pipeline redireciona os seus *standard output* e *standard error* para o ficheiro de output contendo o id da task em execução. Dessa forma, a execução de todo o pipeline de comandos é considerada uma única tarefa e é coordenada pelo mesmo *manager*, permitindo a execução concorrente de múltiplos comandos de forma eficiente.

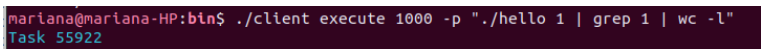

Figura 6: `./client execute 10 -p "./hello 1 | grep 1 | wc -l"`

5 Paralelismo de Tarefas

O paralelismo de tarefas é uma técnica amplamente utilizada em sistemas computacionais para melhorar a eficiência e o desempenho, permitindo que múltiplas tarefas sejam executadas simultaneamente. Em vez de aguardar a conclusão de uma tarefa antes de iniciar outra, o paralelismo permite que várias tarefas sejam executadas de forma independente e concorrente, aproveitando ao máximo os recursos disponíveis e reduzindo o tempo total de execução.

No contexto do sistema de processamento de tarefas desenvolvido neste projeto, o paralelismo de tarefas é alcançado por meio da distribuição de tarefas entre múltiplos *managers*, que podem executar as tarefas de forma concorrente e independente.

Este paralelismo oferece várias vantagens no contexto do sistema desenvolvido, incluindo a utilização eficiente dos recursos disponíveis, a redução do tempo total de execução das tarefas e a capacidade de lidar com cargas de trabalho intensivas de forma escalável e flexível. Além disso, o paralelismo de tarefas permite uma distribuição equilibrada da carga de trabalho entre os diferentes componentes do sistema, evitando *bottlenecks* de desempenho e maximizando a utilização dos recursos computacionais.

De modo que, quando cada *manager* acaba a execução de cada tarefa, este escreve no ficheiro `log.txt` através de `write`'s, no diretório indicado na inicialização do servidor, com o seu identificador da tarefa concluída, o programa que foi executado e o seu tempo de execução desde que chegou ao servidor até ter sido terminado (em milisegundos). O *standard output* e *standard error* do pedido são redirecionados para um ficheiro do tipo `output_idTask.txt`, sendo o `idTask` o identificador único da *task* em questão.

6 Políticas de Escalonamento

As políticas de escalonamento são conjuntos de regras e algoritmos utilizados para determinar a ordem de execução de tarefas ou processos em um sistema computacional. Elas desempenham um papel fundamental na gestão eficiente dos recursos do sistema, garantindo a justiça na alocação de recursos e maximizando a utilização dos recursos disponíveis.

Estas políticas servem para controlar o acesso concorrente aos recursos do sistema, otimizar o desempenho, minimizar o tempo de espera e maximizar a capacidade de resposta do sistema às solicitações dos utilizadores. Elas ajudam a equilibrar a carga de trabalho entre os diferentes componentes do sistema, garantindo que nenhum recurso seja sobrecarregado enquanto outros permanecem ocupados.

As políticas de escalonamento podem variar de acordo com os requisitos do sistema e as características das tarefas a serem executadas. Algumas políticas populares incluem o *First-Come, First-Served* (FCFS), *Shortest Job First* (SJF), *Round Robin*, entre outras. Cada política possui vantagens e limitações, e a escolha da política adequada depende do contexto e dos objetivos específicos do sistema.

6.1 FCFS

A política FCFS é uma das políticas de escalonamento mais simples e utilizadas com frequência. Nesta política, as tarefas são executadas na ordem em que são recebidas, ou seja, a primeira tarefa a chegar é a primeira a ser executada. Isto significa que as tarefas são processadas de acordo com a sua ordem de chegada, sem considerar o tempo de execução ou quaisquer prioridades.

Uma das principais vantagens da política FCFS é a sua simplicidade e facilidade de implementação. Além disso, ela garante um comportamento justo e previsível, pois todas as tarefas são tratadas igualmente, sem favorecimentos. No entanto, a política FCFS pode levar a problemas de *starvation* de tarefas de longa duração, uma vez que tarefas menores podem ser bloqueadas atrás de tarefas mais longas que chegaram primeiro, resultando em possíveis atrasos no tempo de resposta.

No nosso sistema, as tarefas foram simplesmente adicionadas à fila de espera (*queue*) por ordem de chegada, seguindo o princípio básico da política em questão.

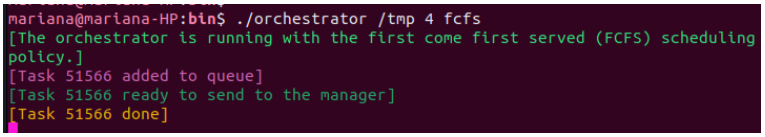

Figura 7: `./orchestrator /tmp 4 fcfs`

6.2 SJF

A política SJF é baseada no princípio de que as tarefas mais curtas devem ser executadas antes das mais longas, a fim de minimizar o tempo médio de espera e melhorar o desempenho geral do sistema. Nessa política, o sistema seleciona a próxima tarefa a ser executada com base na estimativa do tempo necessário para sua conclusão.

Uma das principais vantagens da política SJF é a eficiência na redução do tempo médio de espera e na maximização da utilização dos recursos do sistema. Ela tende a fornecer um tempo de resposta mais curto para as tarefas, especialmente para aquelas de curta duração. No entanto, a política SJF pode ser injusta para tarefas de longa duração, uma vez que elas podem ser continuamente adiadas em favor de tarefas mais curtas, resultando em possíveis problemas, onde no pior caso possível uma tarefa nunca chega a ser executada pois apenas chegam tarefas mais curtas à *queue*. Além disso, a estimativa precisa do tempo de execução das tarefas pode ser difícil em ambientes dinâmicos.

No nosso sistema, implementamos o algoritmo SJF ordenando a fila de espera (*queue*) com base no tempo esperado de execução de cada tarefa. Isso garante que as tarefas mais curtas sejam priorizadas, seguindo o princípio subjacente desta política.

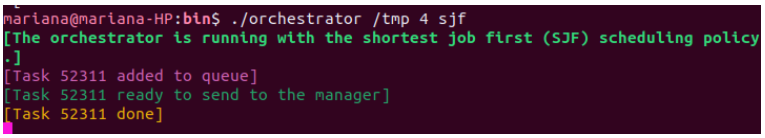

Figura 8: `./orchestrator /tmp 4 sjf`

7 Scripts e Resultados

As *scripts* desempenham um papel crucial na automatização de tarefas em sistemas computacionais. São conjuntos de comandos que simplificam processos complexos, permitindo a execução de diversas operações de forma sequencial ou condicional. Desde a manipulação de arquivos até a configuração de sistemas, as *scripts* oferecem uma maneira eficiente e padronizada de realizar uma ampla variedade de tarefas.

No contexto do projeto, as *scripts* são empregadas para definir e executar tarefas, possibilitando a interação entre o cliente e o servidor. Ao utilizar *scripts*, torna-se mais fácil e rápido enviar comandos específicos para o servidor, como a execução de programas ou a realização de operações de sistema. Isso simplifica o processo de comunicação entre os componentes do sistema, proporcionando uma forma flexível e conveniente de automatizar as operações.

Para avaliar as políticas de escalonamento FCFS e SJF, foram criadas três scripts distintas: *smallScript.sh* (com tempos de duração mais curtos por tarefa), *testScript.sh* (com tempos de duração diversos por tarefa) e *bigScript.sh* (com tempos de duração mais longos por tarefa), cada uma contendo diferentes tipos de pedidos. Cada *script* foi executada quatro vezes com números diferentes de tarefas em paralelo (1, 4, 10 e 100).

Como podemos observar pelas imagens seguintes, ambas políticas de escalonamento apresentam tempos médios de execução altos para a execução de uma tarefa em paralelo para as 3 *scripts*.

Os resultados obtidos mostram que ambas as políticas de escalonamento apresentam tempos médios de execução elevados para a execução de uma tarefa em paralelo em todas as *scripts*. Além disso, observou-se que à medida que o número de pedidos executados em paralelo aumenta, os tempos médios de execução diminuem para todas as *scripts*. No entanto, a política de escalonamento FCFS apresenta tempos mais longos para a *script* que engloba tempos de curta e longa duração alternados, confirmando o que foi dito anteriormente onde tarefas menores podem ser bloqueadas atrás de tarefas mais longas que chegaram primeiro. A política de escalonamento SJF apresenta melhores valores nesse sentido, no entanto é de relembrar que para a *bigScript.sh* apresentou valores relativamente mais altos em relação ao algoritmo FCFS, pois nas tarefas de longa duração, estas podem ser continuamente adiadas em favor de tarefas ainda mais curtas apesar de neste caso também serem de longa duração.

FCFS	Small	Regular	Big
1	27.745	97232.7288	758620.7627
4	1.762	9966.5932	120861.4746
10	2.254	4322.4746	15102.661
100	2.15	3355.8136	5084.2373

Figura 9: Tempos médios de execução por pedido na política de escalonamento FCFS

SJF	Small	Regular	Big
1	34	17392.4576	791383.322
4	5.169	14502.4915	107635.8644
10	1.83	3816.8475	49956.7288
100	1.847	2158.4746	5285.2373

Figura 10: Tempos médios de execução por pedido na política de escalonamento SJF

Portanto, ambas as políticas de escalonamento têm suas vantagens e podem ser adequadas para diferentes situações, dependendo das características dos pedidos e das necessidades do sistema.

8 Conclusão

Ao longo deste relatório, exploramos detalhadamente o sistema de processamento de tarefas desenvolvido. Desde a arquitetura até a implementação prática, examinamos cada componente e funcionalidade do sistema, destacando a sua importância e contribuição para a automação de tarefas em ambientes computacionais. A comunicação entre o cliente e o servidor foi fundamental, permitindo o envio e a execução de tarefas de forma eficiente e coordenada. O paralelismo de tarefas ofereceu uma maneira de otimizar o desempenho, lidando com múltiplas tarefas simultaneamente e minimizando *bottlenecks*. Além disso, as políticas de escalonamento implementadas, como o FCFS e o SJF, proporcionaram uma abordagem estratégica para distribuir e priorizar as tarefas, maximizando a eficiência do sistema.

No âmbito das *scripts*, destacamos a sua importância na automatização de processos, simplificando a execução de comandos e facilitando a interação com o sistema. Estas ferramentas desempenham um papel fundamental na eficiência operacional, permitindo a execução de uma ampla variedade de tarefas de forma padronizada e automatizada.

Em suma, o sistema de processamento de tarefas apresentado neste relatório representa uma solução abrangente e eficaz para a automação de tarefas em ambientes computacionais, fornecendo uma base sólida para futuros desenvolvimentos e aplicações na área de sistemas de automatização.