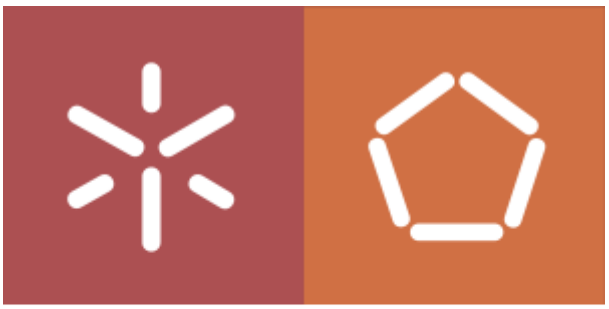

Universidade do Minho
Escola de Engenharia

Licenciatura em Engenharia Informática

Sistemas Operativos

Grupo 107

a104262 Afonso Gregório de Sousa
a104088, André Sousa Miranda
a100599, Gonçalo Antunes Corais

Trabalho prático: Orquestrador de Tarefas

Maio 2024

Índice

1. Introdução

Aproximando-se o fim do semestre, também o projeto realizado no âmbito da Unidade Curricular de Sistemas Operativos atinge o seu fim. Servirá, portanto, o presente relatório para assistir o leitor na compreensão e interpretação do trabalho que foi elaborado pelo grupo, bem como para refletir sobre o mesmo.

Tendo como objetivo a elaboração de um software de orquestração de tarefas, com recurso à linguagem C e em ambiente Linux, este projeto implicava não só a criação de um cliente e servidor, mas também o estabelecimento de comunicação entre os mesmos, em ambos os sentidos. Quanto ao servidor, intitulado de *orchestrator*, recebe informações relativas aos programas em execução e aos respectivos processos que os executam. Teria, também, após a sua execução, de guardar em ficheiro os dados recebidos e ainda, por vezes, transmitir ao cliente dados por ele requisitados. Quanto ao cliente, denominado *client*, é enviada a informação relativa aos programas executados pelo utilizador, o envio de pedidos ao orchestrator(por exemplo, um status request) e a receção, interpretação e resposta desses mesmos pedidos.

Assim, e de forma a facilitar a execução desta empreitada, foi inicialmente planeada toda a arquitetura e estrutura do projeto. Durante tal processo, foram identificadas também aquelas que seriam as maiores dificuldades: o estabelecimento da comunicação via pipes com nome entre o cliente e o servidor de modo que o servidor estivesse sempre à espera de informação, o estabelecimento da comunicação, na direção contrária, de modo a que o servidor enviasse dados que somente um processo pudesse receber e a criação de *structs*, estrutura que acreditamos que melhor se adequa para armazenar as informações de execução de processos no servidor, dadas as exigências do projeto. Tais dificuldades confirmaram-se, mas obviamente não impossibilitaram a correta e total realização deste projeto dentro do prazo, devido ao bom planeamento inicial.

Passaremos, de seguida, por clarificar a arquitetura planeada, findando com uma explicação/demonstração da mesma na prática, em cada função.

2. Arquitetura

Desde o início definimos a criação de quatro ficheiros em C, um para o cliente(*client.c*), um para o servidor(*orchestrator.c*) e os ficheiros header que lhes são associados, *client.h* e *orchestrator.h*, respetivamente, para as estruturas criadas que seriam partilhadas.

Sendo necessário definir um determinado número de funcionalidades, nomeadamente, o “execute -p”, “execute -u”, “status”, “stats-time”, “stats-command” e “stats-uniq”, após olhar cuidadosamente para o modo de implementação a utilizar, definimos também que, para cada uma das funcionalidades referidas, existiria tanto no servidor como no cliente uma função responsável por implementar o comportamento de cada uma na respetiva interface. Para além destas e da função main, cada um dos dois ficheiros possuirá também funções auxiliares ao seu funcionamento, como comparação de timestamps (*client*), escrita para ficheiro (*orchestrator*) e criação de *structs* e implementação de funções que promovem o seu funcionamento (*orchestrator*).

No que toca à comunicação entre *client* e *orchestrator*, esta será realizada via FIFOs(pipes com nome). Este método revela-se ser o mais acertado, dada a sua confiabilidade, por transferência de dados com controlo de fluxo e a sua versatilidade, sendo que pode ser usado tanto para a comunicação entre processos locais ou remotos. Pretendemos que, uma vez posto a correr o *orchestrator*, este crie um FIFO e o abra para leitura, permanecendo à espera que algum *client* seja executado e envie dados para o mesmo. De modo a manter o FIFO aberto quando um *client* termina a sua comunicação, no *orchestrator* o FIFO é aberto também para escrita, tendo sempre, portanto, um extremo de leitura e de escrita abertos. Assim, é possível executar um *client*, ele finalizar, o utilizador aguardar o tempo que bem entender e executar um outro *client*, sem que o *orchestrator* receba EoF(end of file).

Quanto à escrita de dados do *orchestrator* para o *client,* por exemplo, quando é pedido um status ou alguma estatística, iremos recorrer novamente a FIFOs. Contudo, esta situação é um pouco mais complicada que a anterior. Havendo a possibilidade de ter mais do que um *client* a pedir informação do *orchestrator* ao mesmo tempo, o FIFO utilizado não pode ser partilhado por todos, pois quem irá ler qual parte dos dados é incerto, podendo distorcer o correto funcionamento do programa. Para resolver este problema, será criado no *client* um novo FIFO sempre que existir algum pedido deste género. De modo a torná-lo privado àquela execução, o nome do FIFO será o número do processo que o *client* estiver a utilizar para enviar a informação, que será único. Sempre que for oportuno realizar comunicação entre um processo pai e respetivos processos filhos dentro de uma função, esta será implementada com pipes anónimos. Esta via de comunicação demonstra-se simples de implementar e utilizar, bastante eficiente, dado que não exige qualquer cópia de dados entre os processos, bidirecional e escalável, permitindo que vários processos comuniquem com um outro, sem interferência. Tais motivos levaram-nos a concluir que se tratava da melhor forma de comunicação a adotar nestes casos.

Antes de partir para a explicação detalhada das funções e estruturas implementadas, é ainda de mencionar que todas as leituras e escritas para FIFOs, pipes anónimos e ficheiros serão realizadas através das funções read e write e a abertura e fecho de descritores através das funções open e close. De modo a não tornar extensas as explicações individuais, deixa-se também mencionado desde já que existem em todas as chamadas destas funções as respetivas verificações de possíveis erros de abertura, escrita e leitura.

2.1 client.h

O arquivo *client.h* define protótipos de funções que implementam a lógica para enviar comandos de execução e solicitar o status das tarefas ao servidor. Estas funções facilitam a criação de uma interface de comunicação robusta e eficaz, permitindo que o cliente envie requisições especificadas através da linha de comando e receba respostas pertinentes do servidor.

2.2 orchestrator.h

No *orchestrator.h*, são definidas as estruturas e funções utilizadas pelo servidor para gerenciar as tarefas recebidas. Este arquivo inclui definições para estruturas que armazenam informações sobre tarefas em execução (ativas) e em fila de espera, e declara funções para a inicialização do servidor, escalonamento de tarefas, e gestão de tarefas ativas. Este cabeçalho é fundamental para organizar o código do servidor de maneira que suporte o escalonamento eficiente e o tratamento adequado das tarefas, baseando-se na política escolhida pela equipa de trabalho, Shortest Job First (SJF).

As estruturas definidas são:

1. **TaskStartTime:** Armazena o ID de uma tarefa e o tempo de início da tarefa.
2. **Task:** Representa uma tarefa com um ID, um comando e um tempo estimado de execução.
3. **ActiveTask:** Representa uma tarefa ativa, sendo esta pertencente à estrutura *Task,* um PID e um tempo de início.
4. **CompletedTask:** Representa uma tarefa concluída, sendo esta pertencente à estrutura *Task,* um tempo de início e um tempo de término.

2.3 client.c

O arquivo *client.c* implementa a funcionalidade do cliente, processando argumentos de linha de comando para executar tarefas e consultar o seu status. Este código faz a gestão da comunicação com o servidor através de FIFOs, enviando comandos e recebendo respostas. A função principal lê os argumentos do utilizador/cliente, como o tempo estimado e o programa a executar, e envia esses detalhes ao servidor. Respostas do servidor, como o identificador da tarefa e o status das tarefas, são recebidas e exibidas ao cliente. Este arquivo também inclui tratamento de erros para garantir que problemas na comunicação ou execução não comprometam a operacionalidade do cliente.

2.4 orchestrator.c

O *orchestrator.c* contém a lógica central do servidor, incluindo a aceitação de conexões dos clientes, o processamento dos comandos recebidos, e a gestão das tarefas conforme a política de escalonamento. Ele inicializa o servidor e entra num loop onde aguarda por requisições de clientes. Quando um comando é recebido, ele é processado para determinar se é uma solicitação de nova tarefa ou uma consulta de status. Tarefas são escalonadas e executadas com base na disponibilidade de recursos, e informações sobre tarefas ativas, pendentes e concluídas são mantidas e fornecidas conforme necessário. Este componente é crucial para assegurar que o servidor opere de maneira eficiente e estável, maximizando o throughput e minimizando o tempo de resposta.

3. Resultados dos testes efetuados

Durante a realização do trabalho foram efetuados testes para assegurar o desenvolvimento do mesmo e a eficiência de cada função de modo a conseguirmos finalizar o trabalho prático com todas as implementações pedidas. Com recurso a figuras presentes a seguir, iremos mostrar dois exemplos de testes efetuados.

3.1 Tarefas ativas e consequente passagem para Tarefas concluídas

Para começar, temos que compilar o trabalho através do comando “make run” que irá abrir um novo terminal. De seguida, utilizando o comando presente na figura 1, este inicia o servidor. Continuando no segundo terminal, iremos submeter tarefas para serem executadas. Se não houver tarefas ativas ou se o número de tarefas ativas for inferior ao valor do *parallel_tasks*(número presente nos comandos como mostra a figura 1), a tarefa começa a ser executada como mostra a figura 2. Quando acabar, essa tarefa passa para as tarefas concluídas tal como é visível na figura 3.

```
andre@andre-VirtualBox:~/Desktop/SO trabalho/SO_Project$ ./orchestrator output/
3 FIFO
Servidor iniciado.
```

Figura 1 - Comando para iniciar o servidor

```
andre@andre-VirtualBox:~/Desktop/SO trabalho/SO_Project$ ./client execute 5000 -
u "ls -l"
Task submitted successfully.
andre@andre-VirtualBox:~/Desktop/SO trabalho/SO_Project$ ./client status
Tarefas em espera, ativas e concluídas:
Tarefas em espera:

Tarefas ativas:
Task ID: 0, Command: ls -l, PID: 3057

Tarefas concluídas:
```

Figura 2 - Submissão de uma tarefa e o estado dela

```
andre@andre-VirtualBox:~/Desktop/SO trabalho/SO_Project$ ./client status
Tarefas em espera, ativas e concluídas:
Tarefas em espera:

Tarefas ativas:

Tarefas concluídas:
Task ID: 0, Command: ls -l, Execution time: 5184 ms
```

Figura 3 - Atualização do estado quando a tarefa é concluída

3.2 Tarefas ativas e Tarefas em fila de espera

Tal como no exemplo de cima, começamos por usar o comando “make run” e o da figura 1 para iniciar o servidor. Posteriormente, submetemos tarefas e quando o número de

tarefas ativas for igual ao *parallel_tasks*, as tarefas vão automaticamente para a fila de espera tal como nos mostra a figura 4.

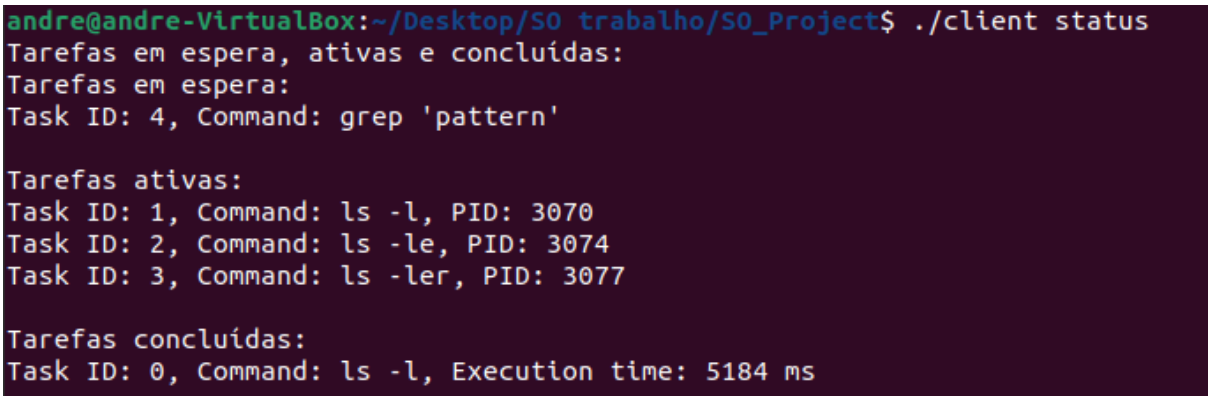

Figura 4 - Tarefas ativas e Tarefas em fila de espera

4. Conclusão

Ao concluir este projeto, é evidente que avançamos significativamente em direção à criação de um serviço de orquestração de tarefas. Durante o processo, enfrentamos desafios importantes, penosos e demorados, mas que no geral, consideramos que conseguimos superar.

O sistema de orquestração de tarefas proposto oferece uma solução eficiente e flexível para gerenciar e executar tarefas em um ambiente computacional. Composto por um cliente e um servidor desenvolvidos em C, utilizando pipes com nome para comunicação, o sistema permite a submissão e o acompanhamento de tarefas de forma transparente e eficaz.

A capacidade de encadear programas e processar múltiplas tarefas em paralelo oferece desempenho aprimorado, enquanto a escolha e avaliação de diferentes políticas de escalonamento permitem otimizar o tempo médio de espera das tarefas.

A interface intuitiva do serviço, juntamente com a flexibilidade da linha de comando, torna a interação dos usuários simples e eficaz. Em resumo, o sistema proporciona uma experiência otimizada e ágil para gerenciamento e execução de tarefas em um ambiente computacional.