Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Sistemas Operativos

Ano Letivo de 2023/2024

Orquestrador de Tarefas

António Filipe Castro Silva(a100533)
Duarte Machado Leitão(a100550)
Pedro Emanuel Organista Silva(a100745)

7 de maio de 2024

SO

1 Introdução

Neste trabalho prático, pretendia-se implementar um serviço de orquestração de forma a escalonar e executar tarefas num computador.

Os utilizadores usam o programa "client.c"para conseguir submeter ao servidor a intenção de executar uma tarefa, dando uma indicação da duração em milissegundos que acham que esta função necessita e qual a tarefa (podendo ser só um programa ou um conjunto, ou seja, uma pipeline de programas) a executar.

Cada tarefa tem associado um identificador único gerado aleatoriamente que é transmitido ao cliente mal o servidor recebe o pedido. É o servidor que é responsável por escalonar e executar as tarefas do utilizadores, escalonando-as de certa forma de acordo com a sua política. A informação produzidas pelas tarefas para o *standard output/standard error* são direcionadas pelo servidor para um ficheiro cujo nome corresponde ao utilizador único utilizado pela tarefa.

O cliente, além de poder executar tarefas, também pode consultar o servidor com um comando **status** para saber as tarefas que estão em execução, em espera para execução ou se já terminaram.

2 Arquitetura da Aplicação

Na figura abaixo estruturamos de alguma forma como funciona a arquitetura da aplicação.

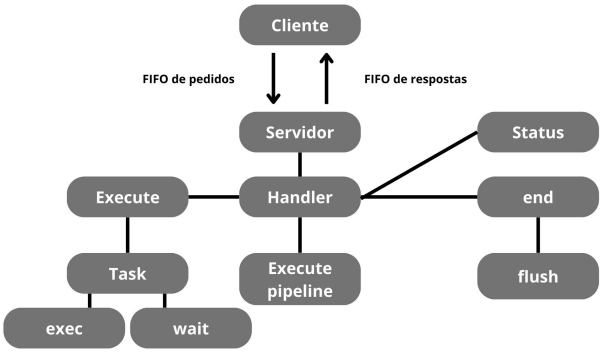

Figura 2.1: Arquitetura da Aplicação

O **cliente**, que se encontra completamente implementado no **client.c**, é de execução imediata, sendo que não espera por respostas das tarefas executadas pelo servidor (além do comando **status**). Este envia pedidos ao servidor e termina a sua execução. O resultado das tarefas é guardado e pode ser consultado na pasta **out**, tendo como nome o seu identificador único atribuído no pedido da tarefa que o cliente recebe, ou seja, **ID.txt**.

Este cliente pode pedir por uma execução de uma programa, uma pipeline de programas ou por um estado do servidor.

Um pedido de execução pode ser feito através do argumento execute:

- **Programa individual**: ./bin/client execute time -u "programa [args]"
- **Pipeline**: ./bin/client execute time -p "programa1 [args] | programa2 [args] | ..."

O **time** representa o tempo estimado de execução da tarefa e será levado em conta pelo servidor para aplicação das políticas de escalonamento, a **flag** pode ser **-u** ou **-p** dependendo se se pretende executar um programa ou uma pipeline de programas. Os programas devem ser escritos entre aspas exatamente como mostrado nos comandos. A pipeline pode executar N programas, sendo considerada como uma única tarefa aos olhos do servidor. Temos incluídos alguns programas de teste na pasta **programs** mas qualquer programa pode ser usado desde que seja enviado o seu caminho correto.

Além de pedidos de execução, um pedido de status pode ser feito para consultar o estado das tarefas enviadas para o servidor, utilizando o comando **./bin/client status**. O servidor responderá com uma lista das tarefas em execução, em espera e executadas, junto com os seus IDs e nomes.

Já o servidor é algo que fica sempre a correr em background à espera de pedidos e deve ser executado com 3 argumentos:

- ./bin/orchestrator output_folder parallel_tasks sched_policy

O **output_folder** representa o local onde os resultados das tarefas serão guardadas. As **parallel_tasks** representam o número máximo de tarefas que podem ser executadas pelo servidor ao mesmo tempo em paralelo. A **sched_policy** representa a política de escalonamento a utilizar pelo servidor para executar as tarefas.

Em termos de políticas de escalonamento, implementamos duas políticas de escalonamento que devem ser passadas como argumento.

- **FCFS** (First Come, First Serve): o servidor escolhe a tarefa no topo da fila de espera para executar, as tarefas são executadas por ordem de chegada e o seu tempo previsto não tem influência na escolha do servidor.
- **SJF** (Shortest Job First): servidor escolhe as tarefas com tempo estimado de execução menor. Não garante que as tarefas mais rápidas vão ser executadas primeiro, simplesmente confia no tempo estimado.

Quando o servidor é inicializado este cria dois **FIFOs** utilizando a função **new_fifo** que se encontra nas **utils.c**, criando um FIFO para respostas chamada **write** e um FIFO para pedidos chamado **read**. Guarda as settings num ficheiro chamado **settings.txt** e cria outro ficheiro chamado **finished.txt** onde guarda as informações de todas as tarefas terminadas: **ID programa/pipeline tempo_de_execução**. Além disso inicializa uma pequena Base de Dados como lhe escolhemos chamar onde guarda o número máximo de funções que podem estar em paralelo, mete um tamanho máximo de queue de 15 tarefas, mete um tamanho máximo de tarefas que podem estar em execução e inicializa o gerador de números aleatórios.

Após estes tratamentos o servidor realmente fica à escuta de pedidos e quando este recebe um faz o tratamento do pedido e utiliza a função **handle_command** para tratar da sua execução.

Esse **handle_command** que se encontra no **handler.c** é a função principal que trata de todos os pedidos feitos ao servidor.

Esta função inicialmente verifica qual é o tipo de pedido, pudendo ser 4 diferentes casos:

- **execute** -> este comando começa por pegar nos dados do pedido e criar a tarefa, depois verifica se há espaço para ser executada. Caso haja é adicionada à lista de tarefas em execução, faz-se um *fork* para criar um processo para executá-la e corre-se a função **addTask** que se encontra no **escalonador.c**. Esta função faz outro *fork* que trata da execução da tarefa e escreve o resultado num ficheiro. O pai espera pelo fim da execução do filho para notificar o servidor e escrever a tarefa que terminou no ficheiro **finished.txt**. Caso não haja espaço é colocada na Queue e o cliente recebe sempre uma resposta a dizer que a sua tarefa foi recebida com sucesso, recebendo dessa forma o identificador único da mesma. Ao ser colocada na Queue, o servidor verifica se a sua política de escalonamento é **SJF**, que implicaria que a queue devesse ser novamente organizada por ordem crescente de tempo, tendo em conta esta nova tarefa.

- **executep** -> funciona de forma muito parecida com o execute tendo um *parse* diferente por ser uma Pipeline e o cliente recebe uma mensagem diferente depois de recebida a dizer que foi uma pipeline. Em termos de execução, caso tenha espaço para executar, utiliza novamente a função **add_Task**, mas como verifica que é uma Pipeline acaba por utilizar a função **executePipeline**. Esta funciona de forma diferente, pois corre vários programas sequencialmente e o STD_OUT do primeiro programa torna-se o STD_IN do próximo, indo verificando sempre se há um seguinte, até executar o último programa e aí guarda esse resultado num ficheiro tal como a execução de um programa normal. Novamente guarda a informação que esta Pipeline acabou no ficheiro **finished.txt** e notifica o servidor do seu término.

- **status** -> quando o servidor recebe um pedido de verificação de *status*, este faz um fork para tratar desse pedido e utiliza a função **getStatus** para demonstrar o estado do servidor. Nesta função que também se encontra o **escalonador.c**, este trata da demonstração de todas as funções que estão em execução, tendo um tratamento para se forem programas individuais ou pipelines, todas as que estão na queue à espera de serem executadas e, por fim, vai ao ficheiro **finished.txt** a partir da função **getFinished** de forma a listar todos os programas/pipelines que foram executados até agora.

- **end** -> este é o comando que o servidor recebe quando um processo-filho termina a execução da sua tarefa. Ao receber esta informação, o servidor trata de remover esta tarefa das que estão em execução e utiliza a função **flushQueue** de forma a colocar em execução a tarefa que se encontrava no topo da Queue e atualiza a Queue de forma a que a tarefa que estava em segundo lugar, fique em primeiro, a que estava em terceiro, em segundo e assim em diante.

3 Protocolo de Comunicação

O cliente quando quer enviar um pedido de uma tarefa começa por escrever na execução em si na *bash* se é uma pipeline utilizando **-p** ou se é um programa individual utilizando **-u**.

Caso seja **-u**, envia para o servidor a informação que é um **execute** colocando um "\n"depois. Posteriormente envia-lhe o tempo previsto de execução colocando novamente um "\n"e no final envia o programa em si e todos os seus argumentos separados por espaços.

Caso seja **-p**, envia para o servidor a informação que é um **executep** colocando um "\n"depois. Posteriormente envia-lhe o tempo previsto de execução colocando novamente um "\n"e no final envia-lhe todos os programa a executar separando-os com "\n".

Ambos estes casos passam por um parser, quer de Programa Individual (parseProgram), quer de Pipeline (parsePipeline), para facilitar a escrita da mensagem.

Para todas as mensagens, no final delas é colocado um **LIMITADOR_MENSAGENS**, que simboliza "??", de forma a que o servidor, no caso de haver vários pedidos acumulados no buffer do pipe (devido a ter estado temporariamente indisponível para ler), consiga separá-las e esses pedidos não se percam. Esta funcionalidade parece funcionar devido ao facto de o servidor conseguir lidar com *overload* de pedidos no pipe, no entanto, acreditamos que precise de mais testes para confirmar a sua eficácia.

Último protocolo de mensagem que vale a pena mencionar é a forma com que, quando uma tarefa termina, o seu processo informa o servidor. O processo envia ao servidor uma mensagem que diz "end\n id_tarefa", fazendo assim que o servidor saiba que tem de remover a tarefa das que estavam em execução e dar *flush* à Queue.

4 Testes

De forma a avaliarmos as nossas duas políticas de escalonamento fizemos um teste em **shell script (.sh)** para perceber a sua eficiência e compararmos as duas. No nosso teste corremos diferentes execuções de utilizadores, sendo algumas programas individuais e outras pipelines, tendo cada uma tempos diferentes, conseguindo assim ver qual das duas políticas é melhor para a experiência dos utilizadores, isto é, qual reduz mais o tempo médio de execução das tarefas todas.

Para testar diferentes configurações de paralelização do servidor basta alterar uma variável chamada **num_execucoes** que se encontra por *default* a 1, ou seja, sem paralelização.

Em baixo apresentamos a interface que aparece depois de fazer o make quando corremos o **make test** que corre o shell script **testing.sh**:

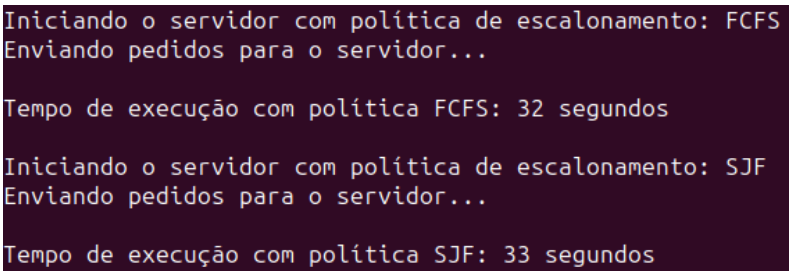

Figura 4.1: Teste sem paralelização

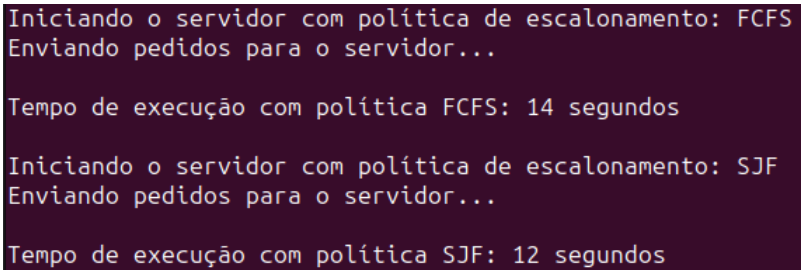

Figura 4.2: Teste com paralelização de 2 tarefas

Conseguimos ver que com paralelização o tempo de execução é muito menor por puder correr duas tarefas ao mesmo tempo.

Se fizermos **"./bin/client status"** durante a execução de cada política conseguimos ver que a queue fica deste modo:

```
Executing:
45209 ../so-orchestrator/programs/hello
6084 ../so-orchestrator/programs/hello

Scheduled:
99084 cat | grep | wc |
84045 ../so-orchestrator/programs/hello
37752 cat | grep | wc |
4418 ../so-orchestrator/programs/hello
93448 ../so-orchestrator/programs/hello | grep | wc |

Finished:
75313 cat | grep | wc | 943 ms
```

Figura 4.3: Status da Queue com paralelização do FCFS

```
Executing:
71814 ../so-orchestrator/programs/hello
20875 ../so-orchestrator/programs/hello | grep | wc |

Scheduled:
3351 ../so-orchestrator/programs/hello
26578 ../so-orchestrator/programs/hello
41644 ../so-orchestrator/programs/hello

Finished:
52503 cat | grep | wc | 199 ms
69810 cat | grep | wc | 201 ms
62040 cat | grep | wc | 204 ms
```

Figura 4.4: Status da Queue com paralelização do SJF

Como podemos ver no FCFS ele simplesmente vai seguir a ordem que é dada pelo teste, enquanto que no SJF vai fazer as tarefas mais rápidas primeiro e como ele tem duas em paralelo ele correu a primeira, a segunda era rápida, mas ao organizar colocou as mais rápidas todas por ordem e elas ficam todas logo executadas sem terem de esperar pelas tarefas mais pesadas.

5 Conclusão

Com os testes que mostramos previamente conseguimos ver que implementamos tudo o que era pedido no enunciado com sucesso. Tentamos meter o programa o mais estável possível, implementar todas as funcionalidades da forma mais correto e tratar de todos os casos que podiam causar problemas no programa.

A comunicação entre o cliente e o servidor parece estável com a utilização de FIFOs não havendo conflitos de comandos nem de concorrência.

Conseguimos respeitar os limites do sistema e a poupança de recursos, tentando minimizar o número de processos abertos ao mesmo tempo, sem comprometer a integridade do sistema.

Com a implementação de várias políticas e da paralelização, conseguimos perceber a diferença e o impacto que ambas têm na experiência do utilizador.