



Universidade do Minho
Escola de Engenharia

Sistemas Operativos
Trabalho Prático
Grupo 47

Marco Soares Gonçalves (a104614)
Leonardo Gomes Alves (a104093)

7 de junho de 2024

Índice

1	Introdução	2
2	Arquitetura da aplicação	3
3	Funcionamento da aplicação	4
3.1	Cliente	4
3.2	<i>Orchestrator</i> / Servidor	4
3.3	Execução de tarefas	5
3.4	Políticas de escalonamento	6
3.5	Processamento de várias tarefas em paralelo	7
4	Outros	8
4.1	Scripts	8
5	Conclusão	8
6	Sinónimos	9

1 Introdução

O presente relatório serve para abordar, de uma forma resumida, o nosso trabalho prático para a UC de Sistemas Operativos do 2º ano de Engenharia Informática.

Neste relatório, falaremos do processo e das dificuldades que surgiram no decorrer do seu desenvolvimento. Este projeto tem como objetivo receber pedidos de um cliente, que serão executados por um servidor/*orchestrator* que terá em conta fatores como o número máximo de tarefas em paralelo e o tipo de escalonamento para a fila de espera.

Face aos critérios definidos no enunciado do trabalho, decidimos implementar todas as funcionalidades propostas, que são as seguintes:

- **Execução de tarefas do utilizador**
- **Consulta de tarefas em execução**
- **Execução encadeada de programas**
- **Processamento de várias tarefas em paralelo**
- **Avaliação de políticas de escalonamento - FCFS e SJF**

2 Arquitetura da aplicação

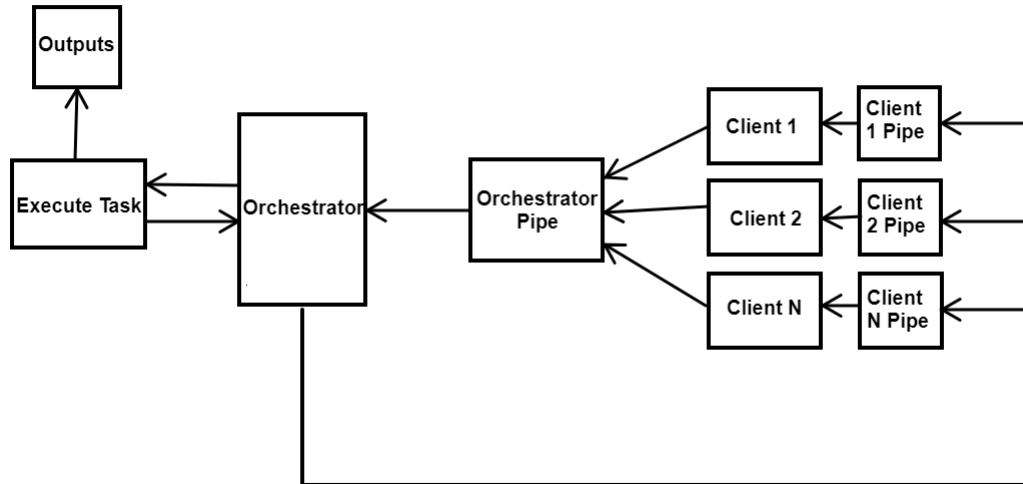


Figura 1: Arquitetura da Aplicação

Para facilitar a compreensão do nosso trabalho, decidimos fazer um pequeno desenho da arquitetura do nosso programa, **Figura 1**.

Em primeiro lugar, os clientes (Client 1, Client 2, Client N) são responsáveis por enviar os seus pedidos que pretendem executar, para o *pipe* do servidor (orchestrator). Este, por sua vez reencaminha-os para o próprio servidor, onde são inseridos numa fila de espera até serem executados.

Posto isto, como os pedidos são finalmente recebidos pelo servidor, este envia uma mensagem para um *pipe* que foi criado pelos clientes que fizeram o pedido, para informar que a tarefa foi recebida, no entanto, veremos mais à frente que nem sempre isto acontece.

Tendo assim recebido tarefas, o servidor vai ser responsável por executá-las e criar ficheiros de *output*, que poderão ser vistos pelos clientes.

3 Funcionamento da aplicação

Nos seguintes pontos, iremos falar mais detalhadamente como o nosso programa funciona.

3.1 Cliente

O nosso cliente funciona da mesma forma que nos foi pedido no enunciado. É o elemento responsável por enviar *queries* para o servidor, tanto tarefas normais como encadeadas.

Como já foi referido na arquitetura, o nosso cliente é responsável por ler o input inserido pelo utilizador (fazendo `./client execute ...`). Feito isto, caso for realmente um input válido, é criada uma estrutura de dados em que envia qual o tipo de tarefa (-p, -u, status, kill) e o input do utilizador para o servidor através do nosso *pipe* principal (*pipe* do servidor).

Para recebermos uma resposta do servidor, é também criado um *pipe* para cada cliente que necessite de executar alguma tarefa (o nome do pipe é do tipo `CLIENT_X`, onde X corresponde ao número do processo do cliente).

Feito isto, **dependendo da tarefa**, o utilizador fica em *stand-by* para receber uma resposta do servidor, caso haja. Para as opções de executar tarefas, o cliente fica à espera de receber uma mensagem vinda do servidor, a qual indica que a tarefa foi recebida e qual o número que foi atribuído a essa tarefa.

O nosso cliente possui a opção **client status**, em que o objetivo é mostrar quais são as *queries* que estão em fila, que já foram executadas ou que estão a ser executadas neste momento. Para este **client status**, o cliente fica à espera que o servidor trate de recolher estas informações e as envie pelo pipe, que, após serem recebidas, são escritas no ecrã do cliente.

Além do que nos foi pedido, decidimos também implementar uma opção **client kill** que a sua função é fechar o servidor, visto que este está a correr num loop infinito.

3.2 Orchestrator / Servidor

O servidor corresponde à parte mais importante do nosso projeto.

Como já referido anteriormente, o *orchestrator* trata de receber *queries* dos clientes e processa-as. Para tal, o nosso servidor é constituído por uma fila que trata de gerir os inputs vindos dos utilizadores (através do pipe do servidor, que é criado no próprio) e, quando tiver disponível para executar a tarefa que está na primeira posição da fila, executa-a.

Uma das dificuldades que nos surgiu, foi como seria possível manter o servidor a receber tarefas e ao mesmo tempo estar a executá-las. Após várias tentativas, chegámos à conclusão que teríamos de ter processos filhos do processo do servidor que, quando

disponíveis, executavam tarefas. Para isto, o nosso servidor receberia inputs, caso existissem, inseria-os na fila e, caso fosse possível, criava um processo filho e executava aí a tarefa, permitindo ao servidor continuar a receber tarefas.

Com o objetivo de limitar o número de processos a correr ao mesmo tempo decidimos criar ficheiros que tratariam disso. Por exemplo, caso fosse possível apenas estar uma tarefa a executar no servidor, o que nós faríamos seria criar um ficheiro para essa única "tarefa" e, o servidor iria ler quantos *bytes* estavam escritos nele. Caso não tivesse nada escrito, significava que poderia executar uma nova tarefa e, quando a fosse a executar, escreveria algo no ficheiro que indicava que uma tarefa estava a ser executada nesse "processo". Apenas no final da execução, o servidor tratava de apagar o que tinha escrito. Desta forma, seria possível manter um número máximo de processos a correr no nosso programa.

Para ser possível fornecer a opção de **client status**, decidimos manter as informações relevantes em ficheiro. Tendo um ficheiro para as tarefas em execução, outro para as tarefas que estão em fila (*Scheduled*) e outro para as tarefas completas. Por exemplo, no caso das tarefas em execução, numa situação hipotética em que o limite máximo de tarefas em paralelo é duas, existiriam dois ficheiros que tinham informação qual era a tarefa que estava a ser executada nesse "processo". Portanto, para responder a um **client status**, o servidor apenas tem que ler todos os ficheiros relevantes (que mencionámos anteriormente), e escrevê-los no *pipe* que está ligado ao cliente que fez este pedido, desta forma o cliente apenas lê o que vem do *pipe* e imprime no ecrã.

3.3 Execução de tarefas

No que toca ao processo da execução de tarefas, foi uma das partes mais rápidas do nosso trabalho pelo facto de ser bastante idêntico aos problemas propostos nas aulas TP de Sistemas Operativos.

Ambas as execuções de tarefas, execução única e execução encadeada, têm a mesma base. Começam ambas por invocar a função **gettimeofday**, afim de calcular o tempo de execução de uma tarefa e, posto isto, criam o seu respetivo ficheiro de output (do tipo `TASKx_OUTPUT`, onde x corresponde ao indicador da tarefa), para onde são redirecionados os descritores de output e de erros (`STDOUT`, `STDERR`). Por fim, executam o input inserido pelo utilizador, invocam novamente a função **gettimeofday** para ser possível calcular o tempo total de execução da tarefa e é escrito no ficheiro de tarefas completadas que essa tarefa foi executada num determinado tempo (em milissegundos).

3.4 Políticas de escalonamento

Como foi proposto no enunciado, decidimos implementar duas políticas de escalonamento, **FCFS** e **SJF**, em que na versão **FCFS** as primeiras tarefas recebidas são as primeiras a serem executadas e assim em diante, e na versão **SJF**, são executadas em primeiro lugar as tarefas que necessitam de menor tempo para executarem.

Foram relativamente fáceis de implementar, visto que são conceitos que já foram bastante vezes abordados em várias Unidades Curriculares do nosso curso. Para tal, tivemos que definir todas as funções que são necessárias para uma fila funcionar corretamente, e a única função em que tivemos que diferir foi a função *enqueue*, que o seu objetivo era inserir na nossa fila. Todas as restantes funções, como por exemplo as de tirar elementos da fila, funcionam perfeitamente para ambas as versões.

Desta forma, com as políticas de escalonamento já implementadas, decidimos fazer testes para analisar quais as diferenças entre ambas.

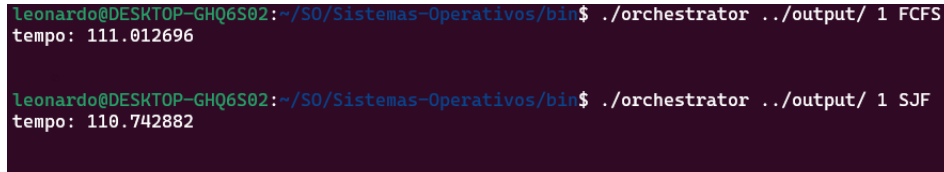
Para isto, utilizámos o *script* abaixo que possui funções com diversos tempos de execução diferentes.

```
1  #!/bin/bash
2
3  for ((i = 1; i <= 100; i++))
4  do
5      echo "./client execute 100 -u 'man ls'"
6      echo "./client execute 2 -p 'grep -v []# /etc/passwd | cut -f7 -d: | uniq | wc -l'"
7      echo "./client execute 1000 -u 'sleep 1'"
8
9      ./client execute 100 -u "man ls"
10     ./client execute 2 -p "grep -v []# /etc/passwd | cut -f7 -d: | uniq | wc -l"
11     ./client execute 1000 -u "sleep 1"
12 done
13
```

Figura 2: Script utilizado na avaliação das políticas de escalonamento

Com este *script*, bastou fazermos leves alterações no código que permitissem obtermos os tempos de espera de cada processo e também o tempo total que o programa demora a executar essas 300 funções (3 funções por iteração * 100).

Desta forma, corremos várias vezes os testes e verificámos que o tempo total de execução das 300 tarefas para ambas as políticas de escalonamento estavam sempre no intervalo [109, 113] segundos, não havendo uma significativa discrepância no tempo de ambas as versões, pelo que, o tempo total destas são bastante similares.



```
Leonardo@DESKTOP-GHQ6S02:~/S0/Sistemas-Operativos/bin$ ./orchestrator ../output/ 1 FCFS
tempo: 111.012696

Leonardo@DESKTOP-GHQ6S02:~/S0/Sistemas-Operativos/bin$ ./orchestrator ../output/ 1 SJF
tempo: 110.742882
```

Figura 3: Tempo demorado para executar todas as tarefas em ambas as versões

Por fim, restou-nos avaliar o tempo médio de execução que os processos demoravam. É de realçar que nesta métrica, é utilizado o tempo desde que o servidor recebeu uma tarefa, até a acabar de executar. Foi também esta métrica que utilizámos por exemplo no **client status**.

Para isso, decidimos fazer uma média de todos os tempos de execução. Na versão **FCFS**, obtemos uma média de **55.765 segundos** de tempo de execução, que se deve ao facto de existir a execução dum **sleep 1** a cada 3 tarefas, que logicamente aumenta o tempo de espera das tarefas, visto que ocupa o tempo do processo em execução em 1 segundo.

Já na versão **SJF**, o tempo de execução médio é de cerca de **22.666 segundos**, menos de metade do tempo da versão **FCFS**. Este tempo médio de execução é muito menor, devido ao facto dos processos que têm menor tempo de execução serem executados em primeiro lugar. Por exemplo, o comando com menor tempo esperado de execução da última iteração do ciclo do *script*, o seu tempo de espera na versão **FCFS** estaria acima dos 100 segundos e, agora na versão **SJF**, estaria abaixo de 1 segundo (o tempo da execução da tarefa com menos tempo esperado é menor que 1 ms). Este fator leva logicamente a uma redução notória do tempo médio de execução dos processos.

3.5 Processamento de várias tarefas em paralelo

Relativamente ao processamento de várias tarefas em paralelo, já falado anteriormente na parte do servidor, utilizámos sincronização através da escrita e leitura de ficheiros. Para ser possível a execução de vários processos ao mesmo tempo, teremos que criar N ficheiros para quantos N processos em simultâneo desejarmos. Desta forma, o servidor iria verificar se alguns desses N ficheiros tinha tamanho de 0 *bytes* (nada escrito), e caso fosse verdadeiro, executaria nesse "processo". Assim conseguimos garantir a limitação de processos em simultâneo e o processamento de várias tarefas em paralelo.

4 Outros

4.1 Scripts

Um dos aspetos que realmente foi útil para testar o funcionamento do nosso código foi a utilização de *scripts*.

Foi uma peça fundamental na fase em que tentávamos descobrir como manter o servidor a receber pedidos e executá-los ao mesmo tempo, pois, com o seu uso, conseguíamos ver o que se estava a passar com o programa. No entanto, os *scripts* foram úteis em todo o decorrer do projeto, visto que os usámos desde o seu início até ao seu final, incluindo claramente na avaliação das políticas de escalonamento.

5 Conclusão

Em suma, no âmbito do que nos foi pedido para desenvolvermos, achámos que conseguimos cumprir com objetivo deste trabalho prático, visto que também fomos capazes de desenvolver todas as funcionalidades.

Apesar de tudo, achámos que ficaram alguns aspetos a melhorar, nomeadamente na parte de modularidade e encapsulamento, e também em alguma repetição de código que poderia ser evitada.

Com isto, gostámos bastante de desenvolver este projeto e certamente nos trouxe conhecimentos que serão utilizados futuramente.

6 Sinónimos

Orchestrator = Servidor = Server

Tarefa = *Query*