

UNIVERSIDADE DO MINHO

LICENCIATURA EM ENGENHARIA INFORMÁTICA

---

Sistemas Operativos

**Grupo 101**

---

**RASTREAMENTO E MONITORIZAÇÃO DA EXECUÇÃO DE  
PROGRAMAS**

Carlos Fernandes (A100890)

João Serrão (A104444)

Maria Cunha (A93264)

Maio 2024

# 1 Introdução

Este projeto foi realizado no âmbito do trabalho de grupo de Sistemas Operativos. O principal objetivo com este trabalho é a criação de um sistema informático capaz de executar tarefas através de um servidor com pedidos transmitidos de um cliente. Para tal, foi definida uma estratégia e estrutura para o desenvolvimento do *cliente*, do *orchestrator* e do método de comunicação entre estes. Neste trabalho começou-se por estabelecer definições sobre diversas variáveis do sistema de modo a garantir a eficiência e consistência do mesmo. Em seguida, procedeu-se à criação de uma struct denominada *Task* destinada a guardar as informações necessárias para o armazenamento e execução das tarefas pedidas pelo cliente. Focando-nos no cliente, desenvolvemos a capacidade deste requerir ao servidor uma ou várias tarefas em simultâneo e o respetivo *status*. Após isto, implementamos duas estratégias de escalonamento, nomeadamente o *First Come First Served* (FCFS) e o *Shortest Job First* (SJF). Finalmente, produziu-se testes para comparar as diferentes teorias de escalonamento e verificar a integridade do sistema.

## 2 Estratégia e Implementação

### 2.1 Definições Auxiliares

Para facilitar a implementação e comunicação entre o *orchestrator* e o *client*, foram estabelecidas diversas definições fundamentais. Estas definições são essenciais para garantir a consistência e eficiência do sistema, fornecendo limites e parâmetros necessários para um funcionamento desejável das funções desenvolvidas. Abaixo estão algumas das principais definições utilizadas:

- `#define SERVER "server_fifo"`: Define o nome do *pipe* utilizado pelo *orchestrator* para receber mensagens dos clientes.
- `#define CLIENT "client_fifo"`: Define o nome do *pipe* utilizado pelo cliente para enviar mensagens ao *orchestrator*.
- `#define MAX_COMMAND_LENGTH 300`: Define o tamanho máximo permitido para o comprimento de um comando como 300.
- `#define MAX_NAME_LENGTH 20`: Define o comprimento máximo permitido para o nome de uma tarefa ou cliente em 20 caracteres.
- `#define MAX_ARGS 100`: Estipula o número máximo de argumentos que podem ser passados para uma tarefa, sendo este limite o valor de 100.
- `#define MAX_COMMANDS 30`: Estabelece o número máximo de comandos que uma tarefa pode conter.

- `#define MAX_COMMAND_LENGTH 300`: Define o tamanho máximo permitido para o comprimento de um comando como 300.
- `#define MAX_TASKS 1000`: Estabelece o número máximo de tarefas que podem ser geridas pelo sistema.
- `#define MAX_BUFFER_SIZE 364000`: Define o tamanho máximo do *buffer* para a comunicação entre o cliente e o *orchestrator* para o comando *status*.

## 2.2 Estrutura de Armazenamento

O grupo optou por definir uma estrutura de armazenamento para trocar informações do lado do *client* para o *orchestrator*. Assim, esta estrutura, denominada de **Task**, foi projetada para conter todos os dados relevantes necessários para a execução de uma ou várias tarefas.

A estrutura **Task** é definida da seguinte forma:

```
typedef struct task{
    int pid;
    int estimated_time;
    int real_time;
    int number_commands;
    char commands[MAX_COMMANDS][MAX_COMMAND_LENGTH];
    char name[MAX_NAME_LENGTH];
    char args[MAX_COMMAND_LENGTH];
    int num_args;
    int type;
} Task;
```

Cada campo da estrutura **Task** tem um propósito claro que passamos a explicitar:

- **pid**: Este campo armazena o identificador do processo associado à tarefa, o que é útil para referência e na gestão de processos.
- **estimated\_time**: Este campo é ocupado pelo tempo que o cliente fornece ao iniciar uma sessão, sendo este o tempo estimado necessário para a conclusão da tarefa.
- **real\_time**: Este campo será determinado depois da conclusão da tarefa em causa, determinando ao certo a duração da sua execução.
- **number\_commands**: Este campo foi concebido com o único propósito de diferenciar entre a execução de uma só tarefa ou de várias (declarado quando o cliente usa *flag -p* ou *-u* com o comando *execute*). Assim, este campo estará a 1 em caso de execução única, e indica ao *orchestrator* essa informação, pois, caso contrário, a maneira como se abordava a execução das tarefas mudava.

- **commands:** Este campo é uma matriz que só será utilizada na execução de tarefas em *pipeline*, com o uso da *flag -p*. Esta matriz é declarada estaticamente, tendo um valor máximo para o número de comandos e o tamanho dos tamanhos a serem executados.
- **name:** Este campo armazena o nome de uma tarefa, para fácil identificação.
- **args:** Argumentos associados à tarefa.
- **num\_args:** O número de argumentos fornecidos para a tarefa, permitindo uma configuração flexível e adaptável das operações a serem realizadas.
- **type:** Este campo define o tipo de tarefa a ser executado, distinguindo entre a inicialização de uma nova tarefa (com o valor 1) e a solicitação de *status* sobre as tarefas (com o valor 2).

## 2.3 Cliente

Assim, o primeiro passo tomado no desenvolvimento do projeto foi a criação de um ficheiro *.c* que constesse todas as operações relativas ao cliente. Assim sendo, fez-se uma clara divisão de tarefas que o programa teria que ser capaz de efetuar: a execução de uma tarefa única, a execução de várias tarefas em paralelo, e um comando de *status* para averiguar o tráfego existente e as tarefas previamente executadas ou em curso no *orchestrator* até aquele ponto. Para evitar conflitos tanto na receção como no envio das *Tasks* cada cliente cria um FIFO com o seu *pid* associado.

Deste modo, passamos a explicar todas as funções designadas para a implementação destes objetivos.

### 2.3.1 Execução Única de Tarefas (*Execute -U*)

Uma das primeiras funcionalidades implementadas foi a capacidade de executar uma única tarefa por vez. Para atingir esse objetivo, desenvolvemos a função *parse\_command*, que desempenha um papel crucial no processamento e interpretação dos comandos enviados pelo cliente.

Ao invés de processar os comandos diretamente na função *main*, optamos por transferir essa responsabilidade para a função *parse\_command*. Esta função recebe a *string* de comando previamente tratada pela *main* e adapta-as para a estrutura *Task* que será posteriormente enviada para o *orchestrator*.

Por exemplo, a função *main* recebe o comando '*.client execute 50 -u tarefa arg1 arg2 [...]*', extraíndo para uma *string* a secção '*50 tarefa arg1 arg2 [...]*' e oferecendo-a à função *parse\_command* que extrai estas informações como o tempo estimado para a conclusão da tarefa, o nome da tarefa e possíveis argumentos adicionais. Finalmente, como se tinha referido na secção anterior deste relatório referente aos argumentos da estrutura *Task*, passa-se o valor 1 ao campo *number\_commands*.

Assim, esta função retorna uma *Task* *t* incompletamente preenchida. No entanto, dentro da função *main*, preenche-se os campos *pid* com o identificador do processo atual, assim como o tipo da mensagem (que será 1 para execução de uma tarefa) e inicializa-se a contagem da execução real no campo *real\_time*. De seguida, envia-se a estrutura pelo *pipe* do *orchestrator* com o identificador do processo atual à frente do *pipe* do cliente para uma melhor identificação a quem pertence a tarefa a ser executada.

### 2.3.2 Execução de Várias Tarefas (*Execute -P*)

Para esta funcionalidade, também se criou uma função de *parsing* chamada *parse\_pipe*. A função *parse\_pipe* é responsável por dividir uma *string* de comandos em comandos individuais, considerando possíveis encapsulamentos entre aspas. Esta função analisa a *string* de comandos e extrai cada comando individualmente, adicionando-os à matriz *command\_list* para posterior envio ao *orchestrator*. Tal como feito para o desenvolvimento da opção *execute -u*, o tempo de início da execução da tarefa é registado usando a função *gettimeofday*, permitindo o cálculo do tempo real que a tarefa levará para ser concluída.

Na função *main*, os comandos a serem executados são passados pela linha de comando. Se esses comandos estiverem encapsulados entre aspas, são tratados como um único argumento. Caso contrário, são concatenados numa única *string* e posteriormente divididos em comandos individuais.

Assim, cria-se uma instância da estrutura *Task*, que armazenará os detalhes da tarefa a ser executada. Os comandos extraídos da matriz *command\_list* assim como o número de comandos a serem executados são copiados para essa estrutura nos campos, respetivamente, *commands* e *number\_commands*. A estrutura *Task* é então enviada para o *orchestrator* através do *pipe* desenvolvido para esse efeito, permitindo que o *orchestrator* receba e execute as tarefas.

## 2.4 Status

Para a implementação desta funcionalidade, continua-se a fazer uso da estrutura *Task*. Na parte do cliente, esta é apenas preenchida nos campos *pid* e *type* com, respetivamente, o valor do processo atual e o valor 2 para sinalizar ao *orchestrator* a natureza do comando a ser executado.

Após enviar a solicitação de comunicação com a estrutura *Task*, o cliente abre o seu *pipe* para leitura e aguarda a resposta do *orchestrator*. Este, lê o pedido e recorre à função *write\_status\_to\_buffer*, que armazena as informações da *wait\_list*, da *execution\_list* e dos *logs* num *buffer* criado para ser escrito no FIFO do cliente.

## 2.5 Orchestrator

O *orchestrator* cria o seu FIFO denominado *server\_fifo* para ler as *Task* escritas nele pelo cliente. O servidor possui duas listas principais, a *wait\_list*, que guarda as

informações das tarefas recebidas e a *execution\_list* que guarda as informações das tarefas em execução.

Quando uma tarefa é recebida pelo cliente este atribui um identificador único que é incrementado uma unidade a cada tarefa recebida. Seguidamente, escreve no FIFO do respetivo cliente a informação de que a instância da estrutura *Task* foi recebida juntamente com esse identificador. Caso o servidor seja terminado o valor da variável *count*, usada para definir o identificador, é guardado num ficheiro e acedido quando voltamos a executar o *orchestrator*. Para além disso, a partir daí o valor do campo *pid* da tarefa passa a ser o seu identificador.

Para assegurar as funcionalidades descritas no enunciado, aquando da receção das tarefas, estas são inseridas na *wait\_list*, independentemente do facto de serem tarefas únicas ou em *pipeline*. A única exceção é o *status*, que não entra na lista, pois um pedido de *execute* não deve ser bloqueado por um de *status*.

Na parte da execução, o *orchestrator* executa tarefas em paralelo num ciclo, criando processos para cada uma, até ao valor máximo definido na sua inicialização. A tarefa a executar é sempre a primeira da fila de espera, e é removida imediatamente da mesma.

A função *addToExecutionList* é responsável por adicionar uma tarefa à lista de execução, garantindo que não exceda o limite máximo de tarefas.

Após a tarefa ter sido adicionada à lista de execução, esta é processada dentro do processo filho criado. A função *processRequest* é crucial para processar solicitações de tarefas recebidas dos clientes, decidindo se a tarefa é executada individualmente ou como *pipeline*, e então chama as funções correspondentes para execução dependendo do valor *number\_commands* da instância da estrutura *Task* recebida.

A função *execute\_command* executa uma única tarefa, redirecionando o *STDOUT* ou o *STDERR* para um ficheiro com o nome do *id* da tarefa e executando o programa associado à tarefa.

Para tarefas que envolvem *pipelines* de comandos, a função *exec\_command\_pipe* executa cada comando em sequência, redirecionando a saída para o próximo comando na *pipeline*. Por sua vez, a função *execute\_pipe* coordena a execução de todos os comandos da *pipeline*, criando os *pipes* anónimos necessários e garantindo que a saída seja direcionada corretamente.

Após o processamento da tarefa, a função *writeLog* regista os resultados da tarefa num arquivo de *log*, calculando o tempo real de execução e formatando os dados para escrita. Além disso, a tarefa é removida da lista de execução, pela função *removeFromExecutionList* que recebe como argumento o *pid* da tarefa que é obtido através do *exit\_status* do processo filho.

## 2.6 Escalonamento

Para o escalonamento das tarefas utilizámos duas políticas, sendo estas o *First Come First Served* (**FCFS**) e o *Shortest Job First* (**SJF**), sendo que a diferenciação é feita quando executamos o *orchestrator*. Caso o valor do terceiro argumento seja 1, será utilizado **FCFS**, caso seja 2 será utilizado **SJF**.

Primeiramente implementámos a política **FCFS**, que simplesmente adiciona uma tarefa recebida ao fim da lista de espera e seleciona a primeira da lista para executar, removendo-a de seguida.

Seguidamente, elaborámos o escalonamento utilizando **SJF**, sendo que a única modificação foi a forma como a tarefa é inserida na lista de espera. Neste caso é feito um *insertion sort* com base no tempo estimado da tarefa.

### 3 Testes

Nesta secção apresentamos alguns dos testes efetuados e comparamos as políticas de escalonamento implementadas.

```

soul@carlos:~/Desktop/S02324/bin$ ./orchestrator "../results" 10 1
Task: 1 received.
Command execution succeeded. Result written to: ../results/1

```

Figura 1: Teste de execução de tarefa única

```

soul@carlos:~/Desktop/S02324/bin$ ./orchestrator "../results" 10 1
Task: 1 received.
Command execution succeeded. Result written to: ../results/1
Task: 2 received.
Command execution succeeded. Result written to: ../results/2

```

Figura 2: Teste de execução de tarefas encadeadas

```

soul@carlos:~/Desktop/S02324/bin$ ./orchestrator "../results" 10 1
Task: 1 received.
Command execution failed. Error written to: ../results/1
Task: 2 received.
Command execution failed. Error written to: ../results/2

```

Figura 3: Teste de erro na execução de uma tarefa

### 3.1 Teste do status

O teste foi executado tirando partido de um *script* e com o servidor configurado para 3 tarefas em paralelo e SJF.

```
s0ul@carlos:~/Desktop/S02324/bin$ ./client status
Scheduled:
12 pwd
11 pwd
10 pwd
9 pwd
8 pwd
7 pwd
6 pwd
5 pwd
4 ./void

Executing:
1 ./void
2 ./void
3 ./void
```

Figura 4: *Status* no início da execução

```
s0ul@carlos:~/Desktop/S02324/bin$ ./client status
Scheduled:

Executing:
4 ./void

Finished:
1 ./void 15 15004ms
2 ./void 15 15004ms
3 ./void 15 15004ms
13 pwd 16548ms
12 pwd 16549ms
11 pwd 16552ms
10 pwd 18558ms
9 pwd 18560ms
8 pwd 18563ms
7 pwd 20571ms
6 pwd 20574ms
5 pwd 20578ms
```

Figura 5: *Status* intermédio

```
s0ul@carlos:~/Desktop/S02324/bin$ ./client status
Scheduled:

Executing:

Finished:
1 ./void 15 15004ms
2 ./void 15 15004ms
3 ./void 15 15004ms
13 pwd 16548ms
12 pwd 16549ms
11 pwd 16552ms
10 pwd 18558ms
9 pwd 18560ms
8 pwd 18563ms
7 pwd 20571ms
6 pwd 20574ms
5 pwd 20578ms
4 ./void 15 37583ms
```

Figura 6: *Status* no fim da execução

```
for ((i = 1; i <= 4; i++)); do
|   ./client execute 15009 -u "./void 15"
done

for ((i = 1; i <= 9; i++)); do
|   ./client execute 10 -u pwd
done
```

Figura 7: *Script* de teste do status



### 3.2 Testes que tiram partido do script2.sh

39	/hello3	grp	1	wc	22239s
40	/hello3	grp	1	wc	22239s
41	/hello3	grp	1	wc	22260s
42	/hello3	grp	1	wc	22529s
43	/hello3	grp	1	wc	22529s
44	/hello3	grp	1	wc	22526s
45	/hello3	grp	1	wc	22526s
46	/hello3	grp	1	wc	22829s
47	/hello3	grp	1	wc	22829s
48	/hello3	grp	1	wc	22829s
49	/hello3	grp	1	wc	22844s
50	/hello3	grp	1	wc	22844s
51	/hello3	grp	1	wc	31273s
52	/hello3	grp	1	wc	31273s
53	/hello3	grp	1	wc	31266s
54	/hello3	grp	1	wc	31266s
55	/hello3	grp	1	wc	32293s
56	/hello3	grp	1	wc	34298s
57	/hello3	grp	1	wc	34298s
58	/hello3	grp	1	wc	34270s
59	/hello3	grp	1	wc	34270s
60	/hello3	grp	1	wc	35256s
61	/hello3	grp	1	wc	35256s
62	/hello3	grp	1	wc	37247s
63	/hello3	grp	1	wc	37247s
64	/hello3	grp	1	wc	37274s
65	/hello3	grp	1	wc	37274s
66	/hello3	grp	1	wc	38263s
67	/hello3	grp	1	wc	38263s
68	/hello3	grp	1	wc	40255s
69	/hello3	grp	1	wc	40255s
70	/hello3	grp	1	wc	40249s
71	/hello3	grp	1	wc	40249s
72	/hello3	grp	1	wc	41266s
73	/hello3	grp	1	wc	41266s
74	/hello3	grp	1	wc	43270s
75	/hello3	grp	1	wc	43270s
76	/hello3	grp	1	wc	43273s
77	/hello3	grp	1	wc	43273s
78	/hello3	grp	1	wc	46271s
79	/hello3	grp	1	wc	46271s
80	/hello3	grp	1	wc	46278s
81	/hello3	grp	1	wc	46278s
82	/hello3	grp	1	wc	51595s
83	/hello3	grp	1	wc	51595s
84	/hello3	grp	1	wc	51595s
85	/hello3	grp	1	wc	51595s
86	/hello3	grp	1	wc	51544s
87	/hello3	grp	1	wc	51511s
88	/hello3	grp	1	wc	51511s
89	/hello3	grp	1	wc	56489s
90	/hello3	grp	1	wc	56489s
91	/hello3	grp	1	wc	56489s
92	/hello3	grp	1	wc	56489s
93	/hello3	grp	1	wc	57415s
94	/hello3	grp	1	wc	61428s
95	/hello3	grp	1	wc	61428s
96	/hello3	grp	1	wc	61428s
97	/hello3	grp	1	wc	61428s
98	/hello3	grp	1	wc	62478s
99	/hello3	grp	1	wc	62478s
100	/hello3	grp	1	wc	66495s
101	/hello3	grp	1	wc	66495s
102	/hello3	grp	1	wc	66495s
103	/hello3	grp	1	wc	66495s
104	/hello3	grp	1	wc	67488s

Figura 8: Teste **FCFS** com 5 tarefas em paralelo

[illegible]

Figura 9: Teste **SJF** com 5 tarefas em paralelo

10	/void 2 200ins		
20	/void 2 200ins		
30	/void 3 300ins		
4	/void 3 300ins		
2	/void 3 300ins		
4	/void 3 300ins		
5	/void 3 300ins		
7	/void 3 300ins		
6	/void 3 300ins		
9	/void 3 300ins		
10	/void 3 300ins		
10	/void 3 grep 1	wc -l	3113ins
54	/hello 1 grep 1	wc -l	3117ins
53	/hello 1 grep 1	wc -l	3118ins
53	/hello 1 grep 1	wc -l	3121ins
60	/hello 1 grep 1	wc -l	3115ins
60	/hello 1 grep 1	wc -l	3115ins
62	/hello 1 grep 1	wc -l	3112ins
61	/hello 1 grep 1	wc -l	3112ins
51	/hello 1 grep 1	wc -l	3126ins
55	/hello 1 grep 1	wc -l	3122ins
63	/hello 1 grep 1	wc -l	3118ins
61	/hello 1 grep 1	wc -l	3115ins
56	/hello 1 grep 1	wc -l	3124ins
56	/hello 1 grep 1	wc -l	3124ins
69	/hello 1 grep 1	wc -l	4089ins
67	/hello 1 grep 1	wc -l	4089ins
68	/hello 1 grep 1	wc -l	4091ins
71	/hello 1 grep 1	wc -l	4089ins
70	/hello 1 grep 1	wc -l	4086ins
70	/hello 1 grep 1	wc -l	4094ins
78	/hello 1 grep 1	wc -l	4094ins
78	/hello 1 grep 1	wc -l	4094ins
73	/hello 1 grep 1	wc -l	4093ins
73	/hello 1 grep 1	wc -l	4093ins
79	/hello 1 grep 1	wc -l	4089ins
88	/hello 1 grep 1	wc -l	4089ins
88	/hello 1 grep 1	wc -l	4089ins
76	/hello 1 grep 1	wc -l	4095ins
76	/hello 1 grep 1	wc -l	4095ins
80	/hello 3 grep 1	wc -l	6219ins
81	/hello 3 grep 1	wc -l	6213ins
83	/hello 3 grep 1	wc -l	6209ins
83	/hello 3 grep 1	wc -l	6209ins
90	/hello 3 grep 1	wc -l	6208ins
90	/hello 3 grep 1	wc -l	6208ins
82	/hello 3 grep 1	wc -l	6215ins
89	/hello 3 grep 1	wc -l	6211ins
89	/hello 3 grep 1	wc -l	6211ins
86	/hello 3 grep 1	wc -l	6216ins
86	/hello 3 grep 1	wc -l	6216ins
92	/hello 3 grep 1	wc -l	7215ins
92	/hello 3 grep 1	wc -l	7215ins
90	/hello 3 grep 1	wc -l	7216ins
91	/hello 3 grep 1	wc -l	7220ins
91	/hello 3 grep 1	wc -l	7220ins
95	/hello 3 grep 1	wc -l	7219ins
93	/hello 3 grep 1	wc -l	7220ins
93	/hello 3 grep 1	wc -l	7220ins
99	/hello 3 grep 1	wc -l	7219ins

Figura 10: Teste **FCFS** com 50 tarefas em paralelo

```

19 /void 2 200ms
20 /void 3 300ms
21 /void 3 300ms
2 /void 3 300ms
3 /void 3 300ms
4 /void 3 300ms
5 /void 3 300ms
6 /void 3 300ms
7 /void 3 300ms
8 /void 3 300ms
10 /void 3 300ms
70 /hello 1 grep 1 wc -l 3083ms
71 /hello 1 grep 1 wc -l 3085ms
72 /hello 1 grep 1 wc -l 3087ms
73 /hello 1 grep 1 wc -l 3089ms
74 /hello 1 grep 1 wc -l 3091ms
75 /hello 1 grep 1 wc -l 3093ms
76 /hello 1 grep 1 wc -l 3095ms
77 /hello 1 grep 1 wc -l 3097ms
78 /hello 1 grep 1 wc -l 3099ms
79 /hello 1 grep 1 wc -l 4091ms
80 /hello 1 grep 1 wc -l 4093ms
81 /hello 1 grep 1 wc -l 4095ms
82 /hello 1 grep 1 wc -l 4097ms
83 /hello 1 grep 1 wc -l 4099ms
84 /hello 1 grep 1 wc -l 4101ms
85 /hello 1 grep 1 wc -l 4103ms
86 /hello 1 grep 1 wc -l 4105ms
87 /hello 1 grep 1 wc -l 4107ms
88 /hello 1 grep 1 wc -l 4109ms
89 /hello 1 grep 1 wc -l 4111ms
90 /hello 1 grep 1 wc -l 4113ms
91 /hello 1 grep 1 wc -l 4115ms
92 /hello 1 grep 1 wc -l 4117ms
93 /hello 1 grep 1 wc -l 4119ms
94 /hello 1 grep 1 wc -l 4121ms
95 /hello 1 grep 1 wc -l 4123ms
96 /hello 1 grep 1 wc -l 4125ms
97 /hello 3 grep 1 wc -l 6203ms
98 /hello 3 grep 1 wc -l 6197ms
99 /hello 3 grep 1 wc -l 6200ms
100 /hello 3 grep 1 wc -l 6205ms
94 /hello 3 grep 1 wc -l 6205ms
95 /hello 3 grep 1 wc -l 6205ms
96 /hello 3 grep 1 wc -l 6205ms
97 /hello 3 grep 1 wc -l 6205ms
98 /hello 3 grep 1 wc -l 6204ms
99 /hello 3 grep 1 wc -l 6203ms
100 /hello 3 grep 1 wc -l 6203ms
84 /hello 3 grep 1 wc -l 7213ms
85 /hello 3 grep 1 wc -l 7215ms
86 /hello 3 grep 1 wc -l 7215ms
87 /hello 3 grep 1 wc -l 7216ms
88 /hello 3 grep 1 wc -l 7215ms
89 /hello 3 grep 1 wc -l 7215ms
90 /hello 3 grep 1 wc -l 7215ms
91 /hello 3 grep 1 wc -l 7220ms
92 /hello 3 grep 1 wc -l 7220ms
93 /hello 3 grep 1 wc -l 7220ms
94 /hello 3 grep 1 wc -l 7220ms
95 /hello 3 grep 1 wc -l 7225ms
96 /hello 3 grep 1 wc -l 7225ms
97 /hello 3 grep 1 wc -l 7225ms
98 /hello 3 grep 1 wc -l 7225ms
99 /hello 3 grep 1 wc -l 7225ms
100 /hello 3 grep 1 wc -l 7225ms

```

Figura 11: Teste **SJF** com 50 tarefas em paralelo

### 3.3 Testes que tiram partido do *script.sh*

```
24 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 16048ms
26 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 16051ms
27 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 16053ms
28 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 16053ms
25 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 16055ms
29 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 18055ms
33 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 18059ms
32 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 18062ms
30 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 18064ms
31 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 18064ms
35 ./hello 5 | grep 1 | wc -l 25680ms
36 ./hello 5 | grep 1 | wc -l 25680ms
34 ./hello 5 | grep 1 | wc -l 25682ms
37 ./hello 5 | grep 1 | wc -l 25688ms
38 ./hello 5 | grep 1 | wc -l 25688ms
42 ./hello 5 | grep 1 | wc -l 32708ms
40 ./hello 5 | grep 1 | wc -l 32708ms
43 ./hello 5 | grep 1 | wc -l 32708ms
39 ./hello 5 | grep 1 | wc -l 32712ms
41 ./hello 5 | grep 1 | wc -l 32714ms
s0ul@carlos:~/Desktop/S02324/bin$
```

Figura 12: Teste **FCFS** com 5 tarefas em paralelo (*script.sh*)

```
32 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 23054ms
31 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 23058ms
30 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 25047ms
29 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 25048ms
36 ./hello 5 | grep 1 | wc -l 28562ms
28 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 27052ms
27 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 27053ms
26 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 28063ms
35 ./hello 5 | grep 1 | wc -l 28551ms
34 ./hello 5 | grep 1 | wc -l 28561ms
25 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 29067ms
24 cat ../src/test.txt | grep 1 | sort -r | head -n 5 | sed s/text/TEXT/g 29071ms
21 ls /etc | wc -l 30071ms
23 ls /etc | wc -l 30069ms
22 ls /etc | wc -l 30070ms
20 ls /etc | wc -l 31070ms
19 ls /etc | wc -l 31071ms
16 ls /etc | wc -l 32082ms
17 ls /etc | wc -l 32081ms
18 ls /etc | wc -l 32081ms
15 ls /etc | wc -l 33082ms
14 ls /etc | wc -l 33083ms
s0ul@carlos:~/Desktop/S02324/bin$
```

Figura 13: Teste **SJF** com 5 tarefas em paralelo (*script.sh*)

### 3.4 Testes que tiram partido do *script3.sh* (1000 tarefas)

```
Command execution succeeded. Result written to: ../results/994
Command execution succeeded. Result written to: ../results/996
Command execution succeeded. Result written to: ../results/999
Command execution succeeded. Result written to: ../results/995
Command execution succeeded. Result written to: ../results/998
Command execution succeeded. Result written to: ../results/1000
994 ./void 1 40113ms
999 ./void 1 40107ms
996 ./void 1 40111ms
995 ./void 1 40112ms
998 ./void 1 40110ms
1000 ./void 1 41111ms
s0ul@carlos:~/Desktop/S02324/bin$
```

Figura 14: Teste **FCFS** com 50 tarefas em paralelo (*script3.sh*)

```
Command execution succeeded. Result written to: ../results/95
Command execution succeeded. Result written to: ../results/94
Command execution succeeded. Result written to: ../results/53
Command execution succeeded. Result written to: ../results/52
Command execution succeeded. Result written to: ../results/51
Segmentation fault (core dumped)
56 ./void 1 38678ms
55 ./void 1 38680ms
54 ./void 1 38682ms
53 ./void 1 38684ms
52 ./void 1 39688ms
51 ./void 1 39689ms
s0ul@carlos:~/Desktop/S02324/bin$
```

Figura 15: Teste **SJF** com 50 tarefas em paralelo (*script3.sh*)

### 3.5 Considerações

Através dos testes apresentados e outros efetuados, verificou-se que o tempo médio de execução total das tarefas é ligeiramente superior no caso do **SJF**, no entanto este faz com que as tarefas com tempo estimado mais curto terminem bastante antes relativamente à estratégia **FCFS**.

Quanto à paralelização de tarefas, a política que beneficiou mais do aumento do número máximo de tarefas em paralelo foi a **SJF**, como era de esperar, o tempo total de execução é quase dez vez menor quando alteramos o número máximo de 5 para 50, como podemos observar nas figuras 8, 9, 10 e 11. Adicionalmente, é preciso também notar a importância do uso de estimativas de tempo realistas por parte do utilizador, caso contrário o algoritmo **SJF** será prejudicado, como podemos ver na figura 13. Por fim, quando é requisitado o número máximo de tarefas, constatamos que, ao contrário do **FCFS**, o **SJF** não consegue suportar todos os pedidos, como podemos ver na figura 15.

## 4 Conclusão

A nível geral, e tendo em conta o que foi explicado nos capítulos anteriores, podemos afirmar que temos um projeto bem conseguido. Os guiões das aulas práticas ajudaram imenso no que toca às bases do projeto. A maior dificuldade enfrentada foi a escolha de como se processaria o pedido de *status* da parte do *orchestrator*, mas consideramos que solucionamos bem o problema.

Em suma, este projeto permitiu-nos aplicar de forma prática os conhecimentos adquiridos na UC de Sistemas Operativos, tornando a aprendizagem mais interessante.