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 definido 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 [...]'*, extraindo 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.

Figura 1: Teste de execução de tarefa única

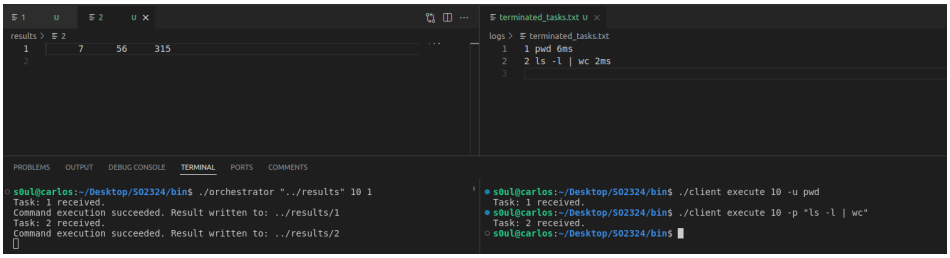

Figura 2: Teste de execução de tarefas encadeadas

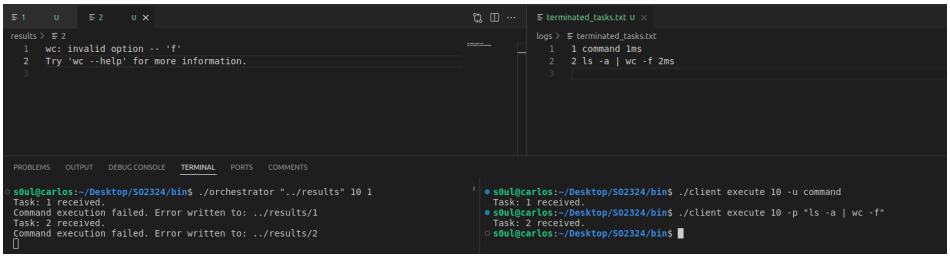

Figura 3: Teste de erro na execução de uma tarefa

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/SO2324/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/SO2324/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/SO2324/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

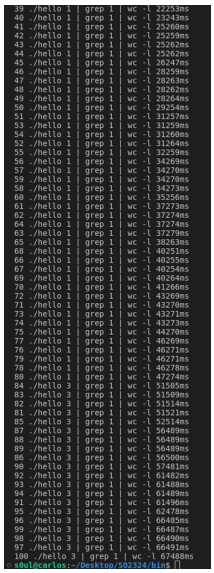

Figura 8: Teste **FCFS** com 5 tarefas em paralelo

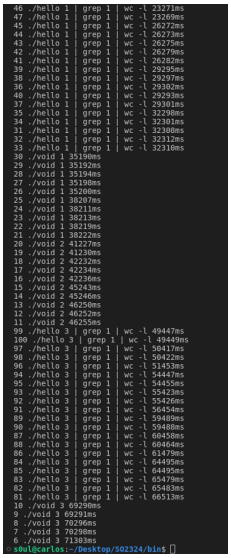

Figura 9: Teste **SJF** com 5 tarefas em paralelo

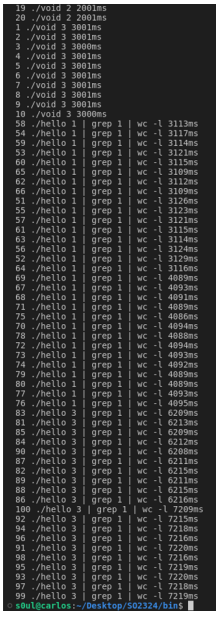

Figura 10: Teste **FCFS** com 50 tarefas em paralelo

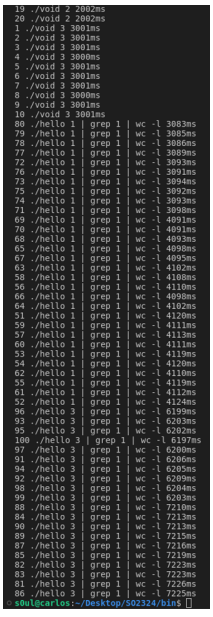

Figura 11: Teste **SJF** com 50 tarefas em paralelo

3.3 Testes que tiram partido do *script.sh*

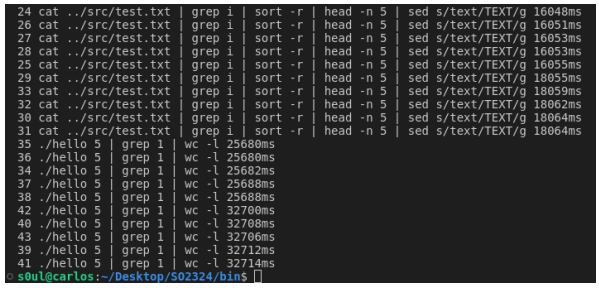

Figura 12: Teste **FCFS** com 5 tarefas em paralelo (*script.sh*)

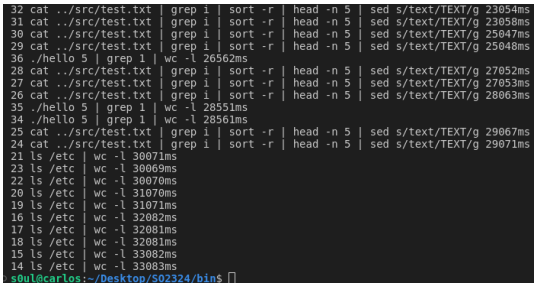

Figura 13: Teste **SJF** com 5 tarefas em paralelo (*script.sh*)

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

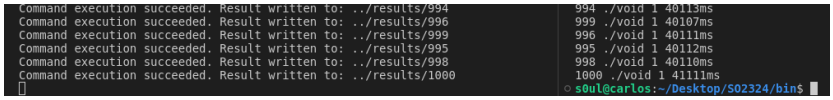

Figura 14: Teste **FCFS** com 50 tarefas em paralelo (*script3.sh*)

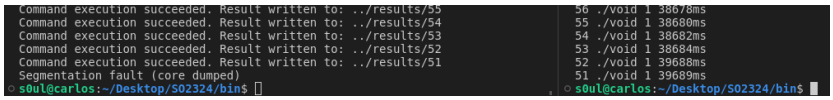

Figura 15: Teste **SJF** com 50 tarefas em paralelo (*script3.sh*)

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 requesitado 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.