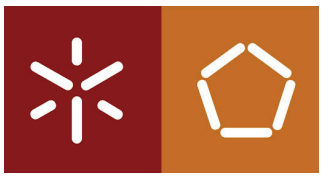

7 de maio de 2024

Relatório do Trabalho Prático de Sistemas Operativos

Lara Regina da Silva Pereira, a100556

Martim de Oliveira e Melo Ferreira, a100653

Rafael Lopes Seara, a104094

Conteúdo

1. Introdução

O presente relatório refere-se à descrição do processo de desenvolvimento do Trabalho Prático da Unidade Curricular de Sistemas Operativos. O Trabalho Prático tem como objetivo a implementação de um serviço de orquestração de tarefas, que visa coordenar a execução de tarefas submetidas por um utilizador num servidor.

A arquitetura proposta para a implementação deste serviço consiste num modelo cliente-servidor, onde o utilizador interage com o sistema através de um programa cliente, submetendo as suas solicitações de tarefas ao servidor. Cada tarefa consiste num programa ou conjunto de programas a serem executados. Uma tarefa pode ser ainda um pedido de consulta do estado do servidor. O servidor é responsável por receber, escalonar e executar as tarefas submetidas. O resultado das tarefas é redirecionado para um ficheiro.

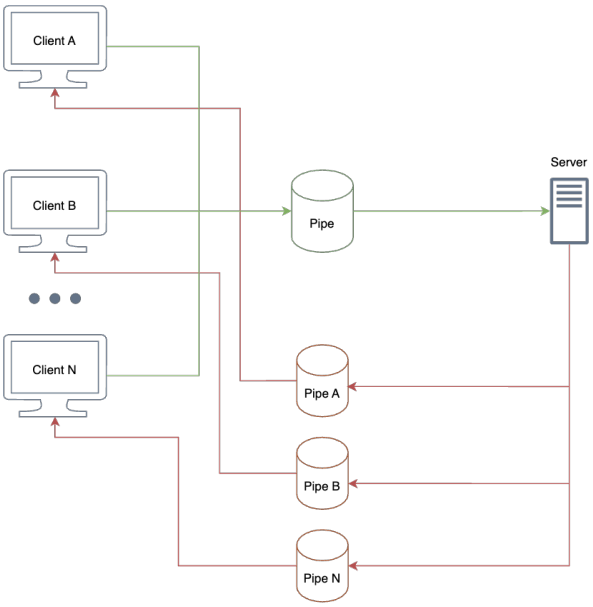

Figura 1: Arquitetura da aplicação

Desta forma, o serviço de orquestração de tarefas apresentado neste trabalho representa uma solução eficaz para a coordenação e execução de processos em ambiente computacional com maior eficiência e controlo sobre as tarefas realizadas. Esta capacidade foi possível utilizando aos conceitos explorados durante as aulas: acesso a ficheiros, processos, *forks*, *pipes*, FIFOs, entre outros.

2. Cliente (programa *client*)

Nesta secção, exploramos as funcionalidades do cliente no contexto do sistema de orquestração de tarefas que estamos a desenvolver. A principal função do cliente é a interação com os utilizadores do sistema, permitindo-lhes a submissão de tarefas para execução e a consulta do estado das suas tarefas no servidor. Estas funcionalidades são acedidas via linha de comandos. Por sua vez, para a comunicação com o servidor, foi implementada uma comunicação através de FIFOs.

2.1. Interface com o utilizador

A interface do programa *client* é via linha de comandos e deve suportar duas funcionalidades fundamentais:

1. **Envio de tarefas para execução:**

   `./client execute time -u "prog -a [args]"`

   Também é possível efetuar a execução encadeada de programas (*pipelines*) através da utilização do operador `|`.

   `./client execute time -p "prog-a [args] | prog-b [args] | prog-c [args]"`

   - `time`: tempo estimado para a execução da tarefa (em milissegundos)
   - `-u` ou `-p`: *flag* indicativa da execução de um programa individual ou de uma pipeline de programas, respetivamente
   - `prog-a`: o nome/caminho do programa a executar
   - `[args]`: os argumentos do programa, caso existam

2. **Consulta das tarefas em execução:**

   `./client status`

   Devem ser listadas as tarefas em execução, as tarefas em espera para executar e as tarefas terminadas.

2.2. Envio de tarefas para execução

Quando o cliente pretende enviar uma tarefa para execução é chamada a função `handleExecute`. O primeiro passo desta função consiste na construção e inicialização de um objeto `Task`.

```
typedef struct {
          int id;
          pid_t pid;
          char command[300];
          int flag;
          int status;
          int predict_time;
          long runtime_ms;
          int waiting_counter;
      } Task;
```

- `id`: identificador único da tarefa
- `pid`: identificador do processo da tarefa

- `command[300]`: programa(s) a executar e respetivo(s) argumento(s)
- `flag`: identificador do tipo de tarefa (0 - um só programa para execução, 1 - *pipeline* para execução, 2 - consulta do *status* do servidor)
- `status`: estado da tarefa (0 - em espera, 1 - em execução, 2 - completa)

De seguida, é aberto o FIFO do servidor para escrita. Caso seja bem sucedida, a `Task` é escrita no FIFO e enviada para o servidor para ser executada. O FIFO do servidor é fechado e, logo de seguida, é aberto o FIFO do cliente com o nome `client_PID`, em que PID é o identificador do processo cliente. Deste FIFO, o cliente lê o ID da tarefa, atribuído pelo servidor, e comunica-o ao utilizador via *standard output*. Por fim, o FIFO do cliente é fechado e removido.

2.3. Consulta do estado do servidor

Quando o cliente pretende consultar o estado das tarefas do servidor é chamada a função `handleStatus`.

Esta função começa por criar um novo objeto `Task`. À semelhança da função anterior, é aberto o FIFO do servidor para escrita, por onde é enviada a `Task` ao servidor. Após o FIFO do servidor ser fechado, é aberto o FIFO do cliente com o mesmo nome utilizado anteriormente, `client_PID`. Deste FIFO, é lida a informação enviada do servidor acerca do seu estado e, posteriormente, escrita no *standard output*. Por fim, o FIFO do cliente é fechado e removido.

3. Servidor (programa *orchestrator*)

Os principais focos do servidor neste sistema de orquestração de tarefas são o escalonamento, a execução e o registo dos resultados das tarefas submetidas pelo cliente. Para tal, o servidor deverá ter uma política de escalonamento para priorização de tarefas, bem com, garantir o paralelismo entre tarefas e a comunicação entre os vários processos.

O servidor utiliza uma fila de espera para gerir a execução das várias tarefas. Para a comunicação com os clientes são utilizados FIFOs, onde se passa uma comunicação bidirecional, em que o cliente envia tarefas e o servidor fornece as respostas.

3.1. Inicialização do servidor

```
./orchestrator output-folder parallel-tasks sched-policy
```

«- `output-folder`: pasta onde serão guardados os ficheiros com o *output* das tarefas executadas
- `parallel-tasks`: número de tarefas que podem ser executadas em paralelo
- `sched-policy`: política de escalonamento de tarefas para execução

3.2. *Overview* do servidor

Após inicialização, o servidor verifica se a pasta de *output* indicada é válida e existe. Em caso afirmativo, todos os ficheiros contidos na pasta são eliminados pela função `deleteFilesInDirectory`, caso a pasta não exista, é criada. De seguida, é criada uma pasta onde serão guardados os ficheiros com os resultados das tarefas em caso erro.

Posteriormente, é criado um FIFO que é aberto para leitura. Em caso de erro na abertura do FIFO, este é tratado pela função `handleError` (que escreve uma mensagem de erro para o *standard output*).

Neste momento, inicia um ciclo infinito que permitirá ao servidor estar à escuta do FIFO para ler as tarefas enviadas pelo cliente. Quando a leitura é bem sucedida, os dados recebidos são armazenados num objeto `Task`, idêntico ao utilizado pelo cliente. A operação a ser executada de seguida está dependente da *flag* da tarefa em questão:

- Se a *flag* for 0 ou 1, a tarefa é um programa ou um *pipeline* para execução, portanto, será adicionada a uma fila de espera através da função `addExecutionToQueue`;
- Se a *flag* for igual a 2, significa que o cliente pretende consultar o estado do servidor, utilizando para tal a função `handleStatus`;
- Se a *flag* for 3, significa que a tarefa foi concluída e o seu estado é atualizado na lista de tarefas do servidor. Também é atualizado o tempo que a tarefa demorou a executar e decrementado o valor da variável global `counter_max_parallel`, que controla o número máximo de tarefas executadas em paralelo, para que possam ser executadas mais tarefas. Com auxílio da função `wait(NULL)`, o programa do servidor faz com que o processo pai aguarde até que qualquer um dos seus filhos termine. Isto garante que a execução do programa pai é suspensa até que todas as tarefas concluídas tenham sido tratadas completamente;
- Se a *flag* não for nenhuma das mencionadas, a função `handleError` exibe uma mensagem de erro.

Terminada a preparação da fila de espera e distribuição de tarefas, a função `handleParallelTasks` é executada em ciclo até deixar de ter tarefas na fila de espera para serem executadas.

3.3. Fila de espera

A fila de espera é uma variável global que consiste numa lista de objetos do tipo `Task`. Juntamente com a fila de espera, é definida outra variável global, o `next_id`, que simultaneamente define o ID de uma nova tarefa e contabiliza o número total de tarefas.

Quando a função `addExecutionToQueue` inicia a sua execução, o ID da tarefa é definido como o `next_id` atual. O `status` da tarefa é `0`, o que significa que a tarefa encontra-se à espera para ser executada e o `waiting_counter` é inicializado a 10. O `waiting_counter` tem como objetivo controlar o tempo de espera da tarefa, de forma a que esta não fique à espera de ser executada por um período indefinido de tempo.

De seguida, a tarefa é adicionada à fila de espera, no índice dado pelo `next_id` e é exibida uma mensagem no *standard output*, indicando que a tarefa foi adicionada à fila. Mais tarde, o `next_id` é incrementado, para garantir que o ID da tarefa seguinte é único.

Neste momento, é gerado o FIFO do cliente, que servirá para o servidor enviar mensagens ao cliente que solicitou a execução da tarefa. O nome do FIFO é `client_PID`, em que PID é o nome do identificador de processo do cliente. Este FIFO é aberto para escrita, e é enviado para o cliente o ID atribuído à tarefa que este enviou para execução. Finalmente, o FIFO do cliente é fechado após a escrita.

3.4. Execução de tarefas paralelas

A execução das tarefas que se encontram na fila de espera inicia aquando da chamada da função `handleParallelTasks`.

Esta função inicia com a ordenação da fila de espera de acordo com a política de escalonamento indicada no momento da inicialização, isto é, serão colocadas no início da fila as tarefas cuja execução é prioritária. Esta ordenação é muito importante uma vez que é definido um número de tarefas paralelas limitado para o servidor. Assim, terá de ser selecionado um conjunto de tarefas para serem executadas de cada vez.

3.4.1. Política de escalonamento

Foram implementadas duas políticas de escalonamento:

- ***First Come First Serve*** (`sched_policy = 1`)

Esta política tem em consideração dois fatores: o `status` da tarefa (tarefas em espera vão para o início da fila) e o `waiting_counter` (quanto mais antiga a tarefa, mais prioridade tem a sua execução).

- ***Fast and First Come First Serve*** (`sched_policy = 0`)

Esta política de escalonamento é uma evolução da política anterior. Para além de ser considerado o `waiting_counter`, ou seja, a prioridade das tarefas que chegaram primeiro, é considerado o `predict_time`. Assim, para duas tarefas com o mesmo `waiting_counter`, o algoritmo de escalonamento considera prioritária aquela que prevê ser executada em menos tempo.

Ambas as políticas de escalonamento são implementadas com recurso à função `qsort`, resultando numa fila de espera ordenada de forma decrescente de prioridade das tarefas.

3.4.2. Paralelismo na execução

A função `handleParallelTasks` retoma a sua execução com a verificação de uma condição: se o número de tarefas a serem executadas em paralelo (`counter_max_parallel`) é menor que o número total de tarefas que podem ser executadas em paralelo (`parallelTasksNum`) e se existem mais tarefas para serem processadas (`next_id`) do que tarefas já processadas (`counter_tasks`). Caso estas condições não se verifiquem, a função termina a sua execução. Caso contrário, significa que poderão ser executadas mais tarefas em paralelo.

Neste último caso, a função começa por decrementar o `waiting_counter` a todas as tarefas à espera para serem executadas e incrementa os contadores `counter_max_parallel` e `counter_tasks` de forma a indicar que irá ser executada uma tarefa. O `status` da primeira `Task` da fila de espera (a mais prioritária) é alterado para 1, indicando que se encontra em execução.

A este ponto, o servidor cria um novo processo filho através da função `fork`. Desta forma, o servidor poderá executar paralelamente várias tarefas diferentes, sem esperar que uma delas esteja concluída para iniciar a seguinte. O processo inicia um temporizador e utiliza a função `handleExecute` para executar a `Task` que estamos a tratar.

No final da execução da tarefa, a `flag` da tarefa é alterada para 3, indicativo de uma tarefa concluída, o temporizador é interrompido e calculado o `runtime_ms`, o tempo, em milissegundos, que durou a execução da tarefa. Para além disto, o processo filho abre o FIFO do servidor para escrita, para onde envia a tarefa com a sua informação atualizada. O processo filho é, ainda, encarregue de escrever num ficheiro na pasta de *output* indicada na inicialização do servidor, onde é escrito o ID da tarefa e o seu tempo de execução.

Finalmente, o processo filho é encerrado.

3.4.3. Execução de tarefas

A função `handleExecute` é invocada pela função `handleParallelTasks` para executar uma tarefa paralela. A tarefa pode consistir num só comando, ou em vários comandos, onde o `output` de um serve como `input` do seguinte.

Esta função começa por realizar um *parsing* dos comandos da `Task`, com a função `parseCommands` separando-os numa lista de *strings*, onde cada *string* representa um único comando. Cada comando irá traduzir-se na criação de um processo único para a sua execução. De seguida, é criada uma matriz com os *pipes* necessários para a comunicação entre processos (num_processos − 1), através da função `pipe`. Nesta matriz, cada elemento é um *pipe*, cada *pipe* é representado por um *array* de dois inteiros: o descritor do ficheiro de leitura do *pipe* e o descritor do ficheiro de escrita do *pipe*.

Após a criação de todos os *pipes*, podemos proceder à execução dos vários processos. Para tal, é utilizado um ciclo para iterar sobre a lista de comandos, no qual, para cada comando, é realizado:

- A criação de um novo processo filho através de `fork`;
- O *standard input* é redirecionado para o descritor de leitura do *pipe* anterior e o *standard output* é redirecionado para o descritor de escrita do *pipe* atual. Caso seja o último *pipe*, o *standard output* é redirecionado para um ficheiro na pasta de *output* especificada ao servidor, onde será escrito o resultado da execução desta *Task*;
- Os *pipes* são fechados;
- O comando que estamos a executar é dividido nos seus vários argumentos através da função `parseSingleCommand` e executado com recurso à função `execvp`.

Após a execução de todos os processos filhos, os *pipes* são fechados. O processo pai aguarda a conclusão de todos os processos filhos, com a função `wait(NULL)`, antes da função terminar.

3.5. Consulta do estado do servidor

A função `handleStatus` é executada quando o servidor recebe um pedido do cliente para consultar o estado das tarefas por enviadas. Inicialmente, é criado um processo filho, através da função fork, para, tal como na execução de tarefas, o processo pai possa continuar a sua execução, sem esperar pela conclusão desta tarefa específica.

De seguida, é aberto um FIFO do cliente, com o nome `client_PID`, onde PID é o identificador do processo da tarefa, para que o servidor possa enviar a informação de volta ao cliente que a solicitou. Assim, o servidor procede para a reunião da informações do estado da sua fila de espera, através da iteração pela lista de tarefas e verificação dos valores do `status`:

- Tarefas em fila de espera (*Scheduled*) com `status` = 0
- Tarefas em execução (*Executing*) com `status` = 1
- Tarefas concluídas (*Completed*) com `status` = 2

A partir desta informação, é gerada uma mensagem com o ID da tarefa, a `flag`, o comando associado e, no caso de tarefas concluídas, o tempo de execução. Finalmente, as mensagens são escritas no FIFO com a função `write`.

Após todas as informações serem enviadas, o FIFO do cliente é fechado e removido, de forma a libertar recursos. Neste momento, também é encerrado o processo filho.

4. Testes

Para melhor compreender a eficiência das duas políticas de escalonamento implementadas, decidimos realizar um teste onde podemos comparar os tempos de execução para três tarefas com as duas políticas diferentes.

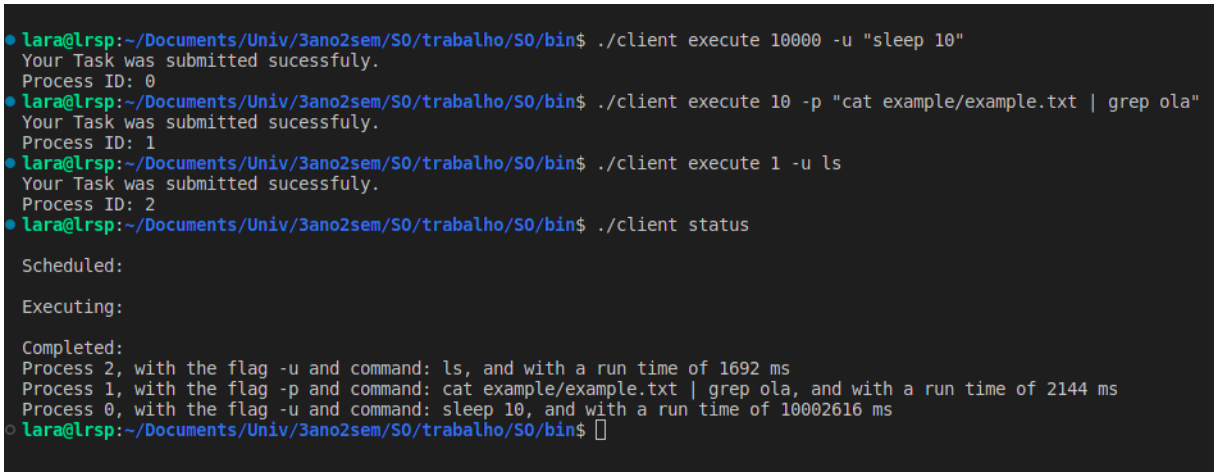

Figura 2: Execução de três tarefas com a política *Fast and First Come First Serve* (FFCFS)

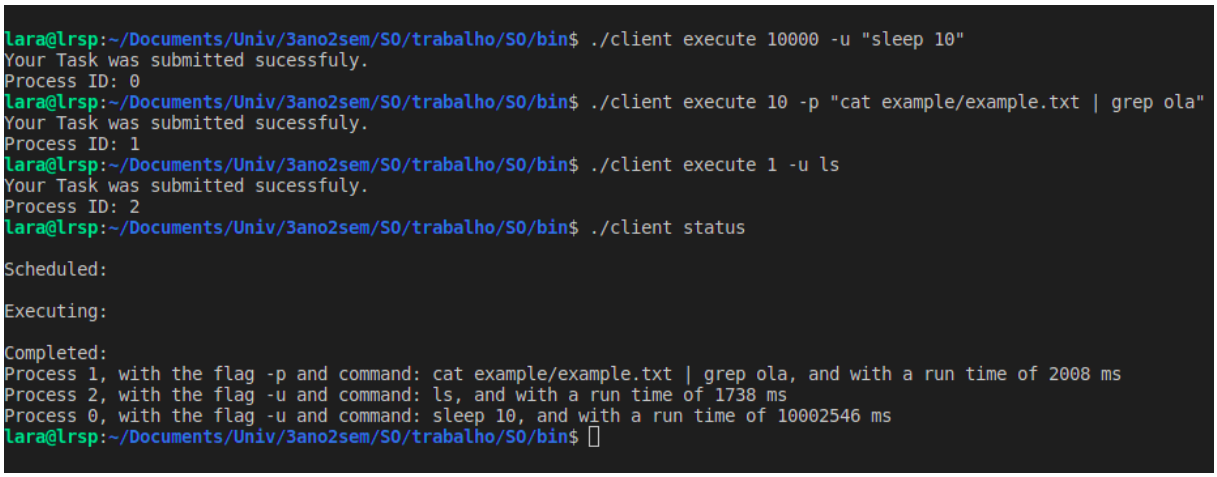

Figura 3: Execução de três tarefas com a política *First Come First Serve* (FCFS)

Observamos que as tarefas foram finalizadas por ordem diferente e também com tempos de execução diferentes. No algoritmo FFCFS, a tarefa 2, cujo tempo previsto de execução é menor, teve prioridade. Em geral, apesar de ser uma diferença pouco significativa, o tempo total de execução das três tarefas é menor com a política FCFS.

Para podermos obter resultados mais comprovados, e concluir melhor sobre a diferença das duas políticas, realizamos testes automáticos através de *scripts*, onde executamos 100 vezes cada uma das três tarefas. O tempo de execução total foi:

- **FCFS**: 10 segundo e 221 milissegundos
- **FFCFS**: 10 segundos e 212 milissegudos

Podemos concluir que, neste caso, a política implementada *Fast and First Come First Serve* foi, para as tarefas testadas, capaz de garantir um tempo de execução menor.

5. Conclusão

Para concluir, consideramos que o trabalho desenvolvido cumpre todos os objetivos inicialmente propostos para o desenvolvimento deste orquestrador de tarefas.

Como trabalho futuro, poderíamos tentar procurar estratégias para melhorar a eficiência da execução de tarefas e até desenvolver outros algoritmos de política de escalonamento.