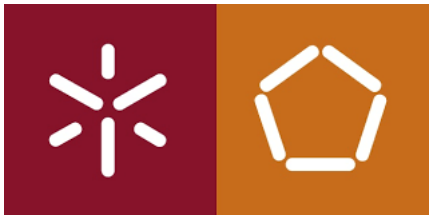


Universidade do Minho

Licenciatura em Engenharia Informática

Sistemas Operativos

Orquestrador de Tarefas

Grupo 23

Inês Ferreira (A97040) Joana Branco (A96584)
João Longo (A95536)

7 de maio de 2024

Introdução

Este projeto foi desenvolvido no âmbito da Unidade Curricular de Sistemas Operativos onde foi proposta a implementação de um serviço de orquestração de tarefas num computador. Tem como objetivo a comunicação entre um servidor e um cliente e, no qual é possível consultar a qualquer momento qual o ponto de situação do programa.

Arquitetura do sistema

Trata-se de um problema cliente/servidor, onde o cliente faz pedidos ao servidor, sendo este último que os processa.

O cliente serve como interface para o utilizador interagir com o programa via linha de comandos.

De modo a desenvolver o objetivo proposto, dividimos o projeto nos diferentes ficheiros: **orchestrator**, **client**, **parse** e **taskqueue**.

O *orchestrator* apresenta-se como o servidor deste sistema e o *client* é a interface com o qual o utilizador irá comunicar. O *parse* permite a identificação dos dados que estão a ser recebidos, tanto no utilizador como no cliente, e a executar funções consoantes os mesmos. Já o *taskqueue* é responsável pelo armazenamento de dados numa fila. Corresponde a uma estrutura de dados que é capaz de guardar as tarefas de forma sequencial.

Adicionalmente foram criados ficheiros para permitir a automatização dos testes na avaliação de políticas de escalonamento, **scripts**, que poderão ser vistos no capítulo mais à frente.

De modo a existir comunicação entre o *orchestrator* e o *client* foram criados pipes com nome. Também podem ser interpretados como FIFOS (*First In First Out*) e tem um nome associado ao pid de cada cliente.

Estruturas de dados

Houve a necessidade da criação de várias estruturas de dados, capazes de armazenar informação sobre as **tasks**, **tipo de tarefas**, **tipo de política de escalonamento**, **nodo singular da fila de espera** e **fila de espera**. Todas estas *structs* estão guardadas no ficheiro **taskqueue.h**.

Na *struct* que representa uma *task*, **TASKS**, temos um inteiro, **fd**, onde é guardado o **pid**

que vai ser o id único do cliente. No **time** guardamos o tempo que o utilizador estima para a realização da tarefa. Já o **type** é representado por um *enum* no qual armazenamos o tipo de pedido (simple, pipelined ou status). Finalmente no *array* **argument** é guardado o 4º argumento dado pelo *input*, estando limitado a 300 bytes.

A **SCHEDPOL** está associada à política de escalonamento que é um argumento que vai ser passado como argumento na inicialização do servidor. No contexto do trabalho temos as opções de **FCFS** (*First Come First Served*) e **SJF** (*Shortest Job First*).

Por último, a *struct* relativa à fila de espera, **QUEUE** conta com um apontador para o nodo do início da fila e um para o último elemento da fila. Desta maneira é possível delimitar as margens da *queue*. Cada um dos nodos é caracterizado por ter informação sobre a *task* e qual o nodo seguinte a si.

Servidor

O servidor requer do seguinte comando para inicializar a sua execução: **./orchestrator output_folder parallel-tasks sched-policy**. Neste caso o *output_folder* corresponde aos nossos ficheiros de **logs** onde são guardadas as respostas às tarefas recebidas pelo cliente. O número de tarefas que podem ser executadas em paralelo, *parallel-tasks*, é opcional. Quanto à política de escalonamento optamos pelas seguintes opções: **FCFS** (*First Come First Served*) e **SJF** (*Shortest Job First*). O primeiro diz respeito a tarefas que são executadas pela ordem que chegam à fila. Quanto ao SJF, a tarefa que demore menos tempo a ser executada é a primeira a ser escolhida.

Apesar de apenas termos optado pelos algoritmos anteriormente mencionados, da maneira que o nosso código está implementado, torna-se fácil o suporte com outras políticas.

Implementação

Numa fase inicial são criados a quantidade de *pipes* previamente dadas pelo utilizador na instrução de execução, que vão corresponder a cada um dos processos-filho existentes. De modo a saber que *pipe* deve usar, o processo-pai é que encaminha as tarefas para o processo-filho livre, onde tem um *pipe* disponível.

Após a criação de uma tarefa, é feita a espera até que um processo-filho comunique ao servidor de que pode receber uma nova tarefa. É garantido também a espera de uma nova tarefa por parte do servidor e de que o processo-filho não executa a mesma tarefa mais que uma vez. Para cada uma das tarefas é analisado o seu tipo, **simple**, **pipelined** ou **status**.

Tipos de Tarefa

Como mencionado anteriormente, após a leitura de uma tarefa, o processo filho analisa qual o tipo da mesma:

* **Simple**

Neste caso, recorre-se ao *parse* implementado de modo a analisar os argumentos dados pela tarefa.

Após a criação de um ficheiro de saída, é necessário o redirecionamento da saída padrão. A este ponto já é possível desenvolver a parte de execução da tarefa, substituindo o que estava no processo-filho com o que é passado como argumento.

Aqui também é registado o tempo de execução de cada uma das tarefas, para este caso fizemos uso da biblioteca *time.h*.

Há que ter em atenção o fecho adequado de cada um dos descritores e o restauro do *Standard Output* para o que estava inicialmente.

Por fim, os resultados são guardados no ficheiro passado inicialmente.

* **Pipeline**

Neste cenário, é necessário guardar o *output* de cada uma das instruções para um ficheiro temporário na sub-pasta *pipes*. Este ficheiro será passado como o último parâmetro da instrução seguinte. A última instrução irá escrever para a pasta de *output* indicada inicialmente ao servidor.

Ao contrário da execução de instruções simples, é necessário manipular os argumentos passados para cada instrução. O valor que indica o próximo separador é temporariamente mudado para indicar *NULL* para a primeira instrução. Todas as seguintes, a *string* com o separador é mudada para indicar o caminho para o ficheiro temporário, com a *string* seguinte sendo *NULL*.

Os valores são restaurados aos seus valores originais imediatamente após a execução do segmento.

No segmento final, é necessário redimensionar temporariamente o *array* de argumentos para adicionar um apontador *NULL*.

* **Status**

Para que um pedido de *status* não monopolize o servidor, um pedido *status* é tratado como uma tarefa, com prioridade máxima.

Neste caso, é aberto o *pipe* de resposta para escrita que tinha sido previamente criado pelo cliente. É desta forma que vão ser comunicadas as informações de *status* da *queue* de tarefas, estrutura de dados presente no nosso programa.

Primeiramente, é comunicado ao servidor-filho a *Queue*. O mesmo *pipe* utilizado para a comunicação de tarefas entre servidor-pai e servidor-filho é reutilizado para comunicar esta.

O envio de informação sobre a fila de tarefas passa pela descrição do local onde as tarefas já concluídas são armazenadas, estas últimas respeitando a formatação que inclui o id único da tarefa e o seu respetivo argumento.

Aqui é feito, também, o restauro do *Standard Output* para o que estava inicialmente.

Tivemos atenção, também, à necessidade de libertar a memória previamente alocada com o recurso da função *free*.

Posteriormente, há a necessidade de inicializar a fila de tarefas onde vão ser guardadas as tarefas provenientes da comunicação com o cliente.

Quando a comunicação é feita com o cliente e é recebida uma nova *task* pelo servidor, é enviada uma resposta de volta ao mesmo cliente que a enviou (através do identificador único no descritor) e a tarefa em questão é adicionada à *queue*.

O *flow* da *queue* baseia-se no *pop* das tarefas que já foram executadas e a seguinte, que deverá ser processada, é enviada a um dos processo-filho disponíveis para executar no momento.

Cliente

O cliente requer do seguinte comando para inicializar a sua execução: **./client execute time x "prog-a [args]"**. Neste caso, *time* faz referência a uma indicação de tempo estimado, em milissegundos, que o utilizador indica, para a execução da tarefa. A *flag* x é uma indicação para o tipo execução: **-u** para programa indivual, **-p** para ser executada em formato *pipeline* e **status** no caso de se pretender saber quais os programas em execução. O parâmetro *prog-a* indica o caminho do programa a executar e *[args]* os argumentos desse mesmo programa. Dependendo do valor da *flag* x vamos ter ou não os restantes argumentos seguintes. Por exemplo, para quando x é *status*, não existe qualquer outro argumento a seguir a si.

Implementação

No que diz respeito ao lado do cliente começamos por verificar se a informação que o cliente recebe do *input* é válida, no caso de a informação ser válida guardamos a informação numa *struct*.

A *struct* mencionada, que guarda a informação recebida, é do tipo **TASKS**.

Analisando os parâmetros da *struct* **TASKS** no inteiro **fd** guardamos o **pid** que vai ser o id único do cliente. No **time** guardamos o tempo que o utilizador estima para a realização da tarefa. O **type** especifica-se como um **enum**, no qual armazenamos o tipo de pedido. No caso do argumento na posição 2 ser -u ou -p, o valor de **type** é **simple** ou **pipelined** respetivamente.

Finalmente no *array* **argument** guardamos o 4º argumento do nosso *input* sendo que este *array* tem um comprimento de 300 caracteres para que os argumentos passados à função *execute* não excedam os 300 bytes dado que um carácter ocupa o espaço de um byte. O *parse* dos argumentos passados à função *execute* guardados em **argument** é só depois feito no servidor.

Abrimos um **FIFO** em modo de escrita com diretoria **temp/contoserver** que é criado do lado do servidor. Para além disso, criamos um **FIFO** com diretoria **temp/valor_do_pid** do lado do cliente.

Posteriormente, procedemos à escrita da *struct* na qual armazenamos a informação do *input* para o **FIFO** que abrimos em modo de escrita, fechando-o de seguida. Por fim, abrimos em modo de leitura o **FIFO** que criamos anteriormente no cliente e mantemos o cliente à espera.

De notar que criamos o **FIFO** do lado do cliente antes do servidor o tentar abrir e só abrimos o lado de leitura no cliente depois de abrir o lado de escrita no servidor para que o cliente não fica-se bloqueado.

Parser

Houve a necessidade de fazer a análise e manipulação do *input* recebido no ficheiro **parse.c**.

No que diz respeito ao *parse* dos argumentos passados à opção **execute**, estes são interpretados como sendo um *array* de caracteres (uma *string*).

Começou-se por criar uma função **contTitleEArgument** que conta o número de palavras no *array*, esta conta é feita contando o número de caracteres de espaço mais um.

Depois numa outra função **storeTitleAndArguments** guardamos os argumentos que antes estavam num *array* de caracteres(*string*) num *array* de *strings* em que cada posição do *array* terá uma palavra para isso tivemos de realizar um *malloc* para o *array* de **strings** com o espaço igual ao número de palavras. Isto por si só permite-nos fazer o **parse** para os casos em que temos apenas de realizar uma tarefa.

No caso em que temos de realizar mais do que uma tarefa o servidor encarrega-se de fazer o *parse* a partir do *array* de **strings**. O servidor faz o **parse** na função **execute_pipeline**. Nesta função o *array* de *strings* resultante do *parse* é percorrido de maneira a que seja passado à função **execute_task** um *array* de *strings* em que todas as *strings* desse *array* correspondem a uma tarefa.

Cliente-Servidor

De modo a permitir uma melhor compreensão da comunicação entre o cliente e o servidor, elaborou-se a seguinte figura.

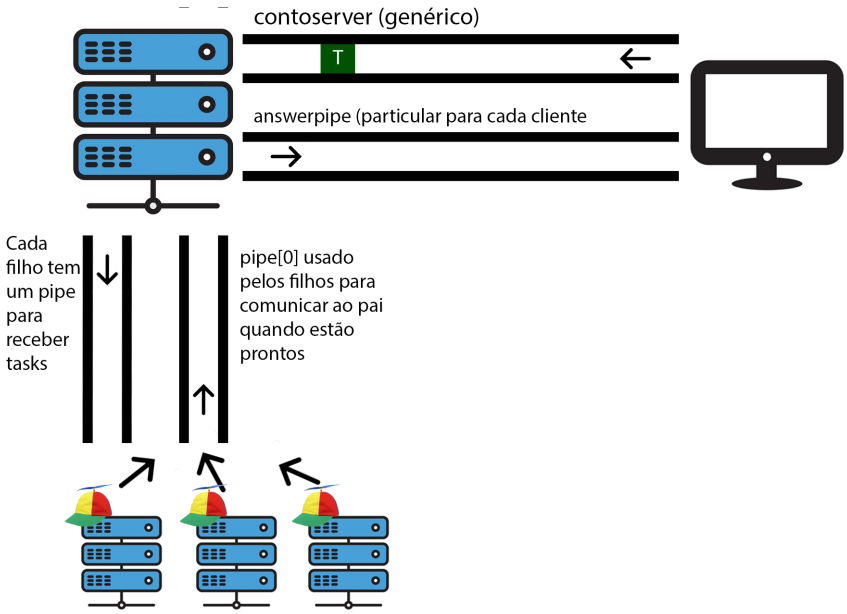


Figura 1: Comunicação cliente-servidor

O esboço elaborado descreve a comunicação entre Cliente-Servidor. Entre o cliente (especificado com a imagem de um computador) e o servidor (corresponde ao processo-pai) existem dois *pipes* que permitem o envio e recebimento de informação.

O *pipe* **contoserver** denota o *pipe* com a orientação cliente para servidor e são aqui que são comunicadas as tarefas que o servidor deve executar. Na mesma mensagem que a tarefa a ser executada, é enviada o pipe a que o servidor deve escrever para comunicar com o respetivo cliente (representado na imagem como **answerpipe**).

Por fim, os **pipes** verticais representam os *pipes* que o servidor principal usa para comunicar com os seus filhos, responsáveis por completar as tarefas armazenadas. Um *pipe* é utilizado para os filhos comunicarem quando estão disponíveis.

Cada filho também tem associado a ele mesmo um *pipe* que o servidor usa para comunicar a tarefa a executar. Este *pipe* é reutilizado para comunicar a *queue* caso o filho precise desta para responder a um pedido *status*.

Exemplos de Execução

Foram desenvolvidas **scripts** em *bash* de modo a automatizar os testes a serem realizados para avaliação das políticas de escalonamento. Deste modo é possível perceber, por exemplo, qual a eficiência de cada uma das políticas escolhidas.

No nosso caso, apenas desenvolvemos o **FCFS** (*First Come First Served*) e **SJF** (*Shortest Job First*).

FCFS vs SJF

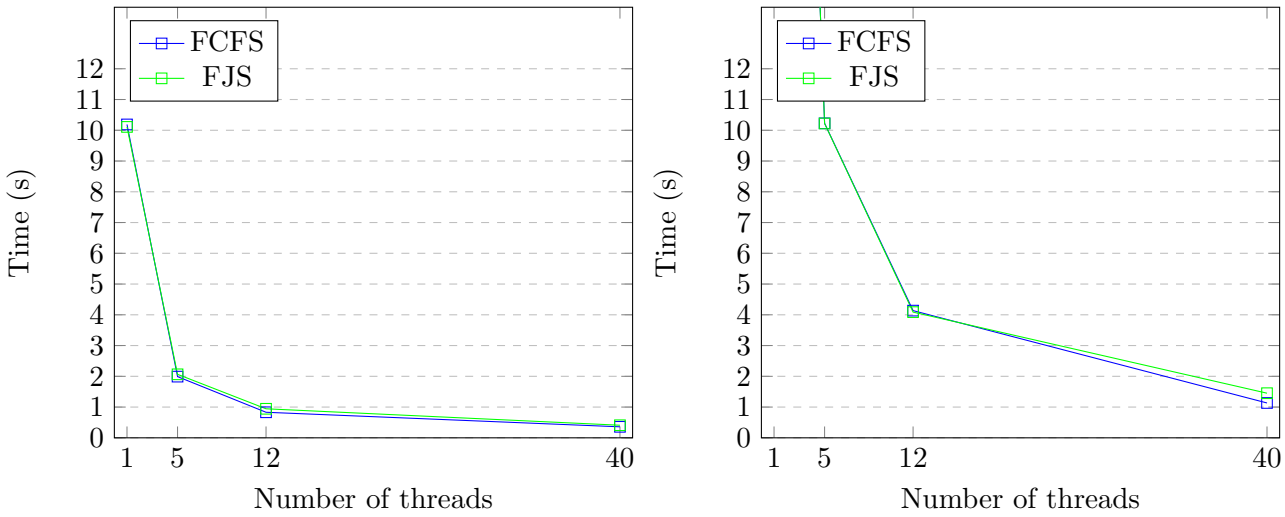
O **FCFS** pode se traduzir em longos tempos de espera. Contrariamente, o **SJF** é baseado na minimização do tempo médio de espera mas aqui, tarefas de longa duração podem ficar à espera demasiado tempo por outras tarefas mais curtas.

Decidimos usar um **script bash** que inicie um cliente com a instrução *sleep*. Isto irá simular pedidos reais com tempos variados.

O grupo decidiu correr cada experiência com cada uma das estratégias e com 1, 5, 12 e 40 pedidos em paralelo máximos. Cada experiência consistia em 100 pedidos.

O grupo iniciou uma primeira experiência onde todos os pedidos demoram uma quantidade fixa de tempo, neste caso, 0.1 segundos.

Uma segunda experiência foi feita com tempos variados, onde o tempo de execução era 10 milissegundos por cada instrução prévia (primeira tarefa 10ms, segunda tarefa 20ms, e assim sucessivamente). A experiência foi repetida com ordens de inserção variadas e também foi monitorizado o tempo necessário para executar um total de 80% das instruções.[1]



Analisando os resultados obtidos, podemos concluir que a estratégia **SJF** é mais eficaz para quantidades baixas de paralelismo, estimando-se que com servidores com menos de 8 *threads* dedicadas a executar pedidos, esta seja mais rápida. Suspeita-se que este resultado é devido à natureza dos testes, que medem o tempo para o cliente receber a confirmação de envio do servidor. Assim, haverá um segmento de tempo onde o servidor ainda está a executar as ultimas instruções, mas já enviou confirmação de receção destas.

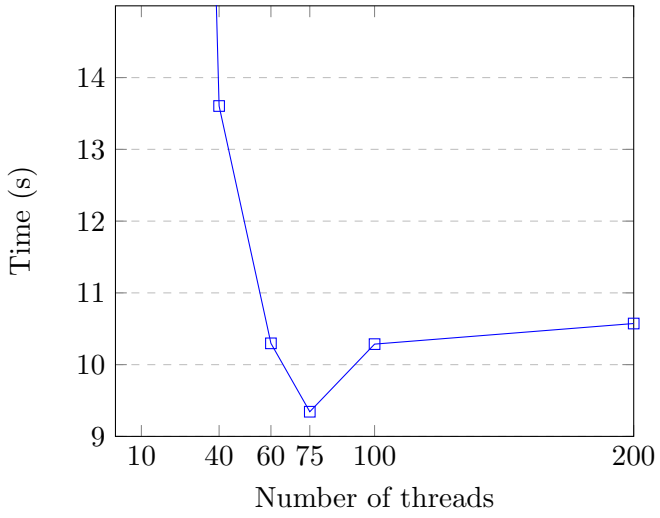
Com a estratégia **SJF**, pode-se garantir que as últimas tarefas são as mais demoradas. Este pequeno aumento de velocidade é, no entanto, mitigado e até ultrapassado pelo tempo de cálculo necessário para o servidor encontrar o pedido mais curto para executar. Isto confirma as propostas iniciais do grupo.

[1]Todos os testes foram corridos na mesma máquina, com um CPU com velocidade base de 3.6GHz e 4 processadores lógicos, e 16GB de memória DDR4, com velocidade de 2400MHz, a correr em dual-channel

Também se esperava que a estratégia **SJF** fosse consideravelmente superior à estratégia **FCFS** no teste de conclusão parcial. No entanto, o teste mostra uma quantidade igual de tempo para ambas as estratégias concluírem 80% das tarefas. O teste foi feito usando o relógio do sistema, que não consegue medir com precisão elevada, mas mesmo levando os parâmetros aos extremos máximos práticos, ambas as estratégias tiveram os mesmos resultados.

Diferentes configurações de paralelização do servidor

Foi realizado um último teste para medir qual a ideal quantidade de processos a realizar pedidos no servidor. Foram corridos mil pedidos com tempos variados entre os 10 e 1000 milissegundos. Todas as experiências foram corridas com a estratégia **FCFS**. [2]



Como se pode observar, o programa aumenta a sua velocidade imensamente até as 75 threads. A partir deste número ideal, a velocidade diminui lentamente.

[2]Ter em conta que os testes incluíam escritas para debug. Os tempos reais seriam mais rápidos, estes números devem ser usados apenas para comparação com outros resultados dos mesmos testes.

Conclusão

O grupo sente-se bastante satisfeito com o produto final, e sente que cumpriu todos os objetivos apresentados inicialmente. Embora não se tenham realizado funcionalidades extra, como mais políticas de organização de pedidos, o código criado é estável e modular o suficiente para a fácil adição de novas funcionalidades, políticas de escalonamento e novos tipos de pedidos.

No entanto, são reconhecidos certos aspetos em que se poderia melhorar, como na eficiência do comando *status*. No planeamento de funcionalidades, também houve uma preocupação desnecessária em reduzir a quantidade de memória usada pelo servidor, por vezes sacrificando assim velocidade. Um exemplo disto seria no comando *status*, onde tarefas concluídas são acedidas em disco e não são carregadas para memória previamente.