Sistemas Operativos

Relatório do Projeto MP2 Parte A - Application Server

André Lino dos Santos (up201907879)
João André Silva Roleira Marinho (up201905952)
Miguel Boaventura Rodrigues (up201906042)
Tiago Caldas da Silva (up201906045)

2020/2021

Conteúdo

1 Introdução

No âmbito do segundo projeto da cadeira de Sistemas Operativos, desenvolvemos uma aplicação do tipo cliente-servidor. Numa fase incial, o foco foi o tratamento de pedidos por parte do cliente e posteriormente procurámos responder mais concretamente ao problema da produção de respostas do servidor. O sistema funciona à base da geração de pedidos *multithread* para um servidor, que por sua vez, recorre a uma biblioteca externa para processar as mensagens do cliente e enviá-las de seguida, para o local de origem, onde a informação é recebida. Os grandes desafios da implementação giram em torno do paralelismo e da concorrência entre *threads*, dada a propensão deste tipo de programas a situações de *deadlock* ou *busy waiting*, que poderão ser resolvidas com recurso a primitivas de sincronização como os *mutex*.

2 *Application Server*

2.1 Cliente

Como referido previamente, a primeira parte do projeto teve como foco principal as ações do cliente, nomeadamente o envio de pedidos e a receção das respetivas respostas após processamento do lado do servidor. Assim, a implementação pode ser segmentada em diferentes funções.

2.1.1 Execução e Geração de *Threads*

Inicialmente, o programa é responsável pela validação do comando submetido, ao qual sucede a verificação da existência do canal de comunicação público - um ficheiro do tipo **FIFO**, por onde serão efetuados os pedidos. O servidor estará responsável pela criação do canal, ou seja, caso a execução do cliente inicie primeiro, este ficará em espera ativa, aguardando pela criação do **FIFO**.

Após a criação de condições para a comunicação inicial entre cliente e servidor, a componente multithread entra então em ação. Nesta fase, o cliente cria sequencialmente as threads, em intervalos pseudo-aleatórios, cada uma responsável por enviar o seu pedido pelo **FIFO** público e pela criação e, posterior eliminação, do canal privado por onde receberá a resposta do servidor à sua mensagem. As funções responsáveis por todo este processo, desde a criação da ligação à receção, ou não da resposta, fica a cargo do módulo ***Request***.

Em relação aos pedidos, é importante o envio de um conjunto informações. De entre essa informações existem vários identificadores, aqui em particular - o identificador único do pedido, o **pid** (*process id*) e o **tid** (*thread id*). São ainda contemplados a carga associada à execução da tarefa (**task_load**), valor entre **1** e **9**, e o resultado da tarefa (**task_res**), com o valor respetivo no caso do servidor e **-1** do lado do cliente.

2.1.2 Receção de mensagens e registos

Inicialmente, com o intuito de gerir a receção das mensagens após a espera no buffer do servidor, ponderámos a utilização da função ***select***, que permite ao programa gerir múltiplos file descriptors. Após a experiência, chegámos à conclusão que neste contexto não teria grande utilidade, em grande parte causadas pelas debilidades desta função - mais informação em man select(2). Assim, a *thread* lê diretamente a resposta, a partir do momento em que o servidor a processa e envia pelo canal de ligação com o cliente. Todas estas operações são acompanhadas pela submissão de mensagens descritivas para a saída padrão ***stdout*** e estão a cargo do módulo ***Logs***.

O intervalo temporal submetido no comando determina quando é que o sistema suspende o envio e a receção dos pedidos. Assim, o fim do programa pode ocorrer de 2 modos. O primeiro acontece quando o cliente termina o envio de pedidos que ainda aguardavam por uma resposta. Neste caso, as respostas que o servidor ainda tinha por transmitir são canceladas e acompanhadas pelo envio de uma mensagem ***FAILD*** para o registo. Do lado do cliente, as threads devem

desistir da espera e registar uma mensagem do tipo ***GAVUP*** para a saída padrão. A outra possibilidade ocorre quando o tempo definido pelo servidor termina, enviando a mensagem ***2LATE*** para os pedidos em espera. Por parte do cliente, é enviada uma mensagem ***CLOSD*** para o registo a informar que o servidor se encontra fechado. Toda a gestão que este término e junção de threads implica, obrigou-nos a efetuar algum trabalho de pesquisa e de experimentação, que será explicado de forma pormenorizada mais à frente.

2.1.3 Término de Threads

Um dos grandes obstáculos desta matéria debateu-se sobre o facto do fecho do servidor, após o término do tempo estipulado para geração de pedidos no cliente, criar situações em que certas threads ficariam à espera da escrita do outro lado. A solução deste problema tem diferentes abordagens possíveis. A primeira opção a ser idealizada consistia em fechar os **FIFO**'s privados a partir da thread principal, desbloqueando todas as situações de *busy waiting*, concluindo o programa. Esta proposta era eficaz e obtia sucesso ao desbloquear as threads, no entanto, não conseguia enviar corretamente as mensagens de ***GAVUP*** (thread desiste de esperar pelo resultado, após o fim do tempo). Após alguma pesquisa, optámos por usar a função ***pthread_cancel***, capaz de as cancelar aquelas threads que ainda estavam bloqueadas. Esta alternativa permitiu até distinguir quais threads que tinham sido, efetivamente, canceladas tornado possível o processamento de todas as mensagens necessárias para os registos.

2.2 Servidor

Após a implementação do cliente, procurámos então resolver a segunda parte do problema, relativa ao processamento dos pedidos efetuados no servidor. De igual modo, procurámos segmentar o código em diferentes funções, descrevendo-as de seguida. Uma nota importante a tomar relativamente a esta parte do projeto é a sua semelhança com o famoso problema do produtor e consumidor.

2.2.1 Receção de pedidos

As receções dos pedidos são feitas pela *thread* principal do programa. Num primeiro instante, é por aqui que se abre o canal público de comunicação - um ficheiro especial do tipo **FIFO**. De seguida, esta *thread* entra num ciclo onde, conforme a chegada de pedidos irá gerar, para cada um dos pedidos uma nova *thread* reponsável por tratar desse pedido; por outras palavras, o tratamento do pedido nada mais é do que uma simples chamada à biblioteca fornecida e que varia de acordo com o valor do campo *message.task_load*, que segundo o enunciado será um valor inteiro - gerado de forma pseudo-aleatória - entre **'1'** e **'9'**.

2.2.2 Produção da Resposta

Como foi referido acima, cada pedido recebido fica a cargo de uma *thread* que por sua vez irá processá-lo de acordo com os dados enviados da parte do cliente. É aqui que surgem os primeiros desafios relativos à sincronização entre o acesso a dados em memória - que podem gerar condições de corrida indesejáveis e até, no pior dos casos, *deadlocks* (impasses em português). Voltando agora à semelhança com o problema do produtor e do consumidor, este conjunto de *threads* que manipula os pedidos são os produtores. Assim, estas threads após efeturem a chamada à biblioteca necessitam de colocar essa nova informação numa zona de memória para que mais tarde possa ser encaminhada de volta ao cliente.

No nosso caso em particular, esta zona de memória, é uma fila baseada numa lista ligada - de modo a tornar possível o reencaminhamento das mensagens por ordem cronológica de chegada.[1] Não obstante, o facto dessa zona de memória ser limitada obriga a que, de forma a não se

[1] A implementação desta fila - também ela do tipo FIFO - está presente no ficheiro ***queue.c***, tendo como base aquela demostrada neste vídeo: `https://bit.ly/3bBVLGp`.

perderem mensagens, seja necessário obrigar determinadas *threads* a pararem pelo simples facto de não haver mais espaço disponível nesse armazém. Para isto utilizamos semáforos, uma primitiva de sincronização que nos dá o número de blocos livres nessa zona de memória e que podem sem preenchidos com a informação já processada. Contudo este é um "trabalho de equipa"na medida em que, como se verá mais à frente, é necessário alertar o consumidor de que já algo pronto a ser consumido - o que obriga à utilização de dois semáforos (um para informar se o *buffer* está cheio e, em contrapartida, outro para informar se o *buffer* está vazio).

2.2.3 Devolução da Resposta ao Cliente

A resposta ao cliente tem uma metodologia idêntica à utilizada anteriormente. Após o acesso à biblioteca, a *thread* começa por aceder ao *buffer*, recolhendo a mensagem a enviar para o cliente, que aguarda então pela resposta. Este acesso, só pode ser efetuado sempre que o *buffer* tem alguma informação pronta a ser lida. Isto implica o uso dos dois semáforos mencionados anteriormente, na medida em que é necessário alertar os produtores de que se a zona de memória partilhada estava cheia, agora já não deverá estar. Posto isto, a *thread* em questão conecta-se ao **FIFO** privado, para onde acaba por enviar a mensagem final. Caso tudo se tenha processado corretamente, aquando do acesso à biblioteca, o conteúdo da mensagem será o resultado em questão, acompanhado do registo **TSKDN**, indicando que a tarefa requerida foi efetuada com sucesso. No entanto, se o limite de tempo for ultrapassado, e, por consequência, a biblioteca tiver negado o pedido, a mensagem enviada terá o valor de **'-1'**, com um registo **2LATE**, representando a chegada com atraso do pedido do cliente. Por fim, há ainda outra possibilidade, que diz respeito ao limite temporal do cliente. Quando este termina a geração de pedidos, as *threads* que se encontravam em espera param de esperar resposta. Desse modo, quando o servidor tenta estabelecer ligação através do **FIFO** privado, não vai obter sucesso, enviando então para o registo **FAILD**.

3 Conclusão

Apesar da complexidade de certas particularidades do projeto, de modo geral, podemos considerar que quer a implementação do cliente, quer a do servidor, foram bem sucedidas. Em todas as fases de desenvolvimento, tentámos optar sempre por funções *thread safe* e pelas alternativas que provassem ser mais eficientes e confiáveis. Consideramos também que aplicámos grande parte dos conceitos trabalhados em aula e que este projeto contribuiu ativamente para a melhor compreensão das caraterísticas de um programa *multithreading* e dos conceitos de concorrência entre os processos do sistema.

Por fim, tendo em conta a dedicação e contribuição equitativa de todos os elementos do grupo, consideramos justa uma divisão de 25% para cada um.

There are lots of Linux users who don't care how the kernel works,
but only want to use it. That is a tribute to how good Linux is.
Linus Trovalds.

4 Apêndice

```
inst ; i ; t ; pid ; tid ; res ; oper
1619742520 ; 1 ; 2 ; 14737 ; 140060128683776 ; -1 ; IWANT
1619742520 ; 2 ; 4 ; 14737 ; 140060120291072 ; -1 ; IWANT
1619742521 ; 3 ; 3 ; 14737 ; 140060111898368 ; -1 ; IWANT
1619742521 ; 4 ; 3 ; 14737 ; 140060103505664 ; -1 ; IWANT
1619742521 ; 5 ; 2 ; 14737 ; 140060024370944 ; -1 ; IWANT
1619742521 ; 1 ; 2 ; 14736 ; 140687176709888 ; 10 ; GOTRS
1619742521 ; 6 ; 4 ; 14737 ; 140060015978240 ; -1 ; IWANT
1619742521 ; 7 ; 2 ; 14737 ; 140060007585536 ; -1 ; IWANT
1619742521 ; 8 ; 9 ; 14737 ; 140059999192832 ; -1 ; IWANT
1619742521 ; 5 ; 2 ; 14736 ; 140687176709888 ; 20 ; GOTRS
1619742521 ; 3 ; 3 ; 14736 ; 140687176709888 ; 30 ; GOTRS
1619742521 ; 4 ; 3 ; 14736 ; 140687176709888 ; 40 ; GOTRS
1619742521 ; 9 ; 5 ; 14737 ; 140059990800128 ; -1 ; IWANT
1619742521 ; 2 ; 4 ; 14736 ; 140687176709888 ; 50 ; GOTRS
1619742521 ; 10 ; 9 ; 14737 ; 140059982407424 ; -1 ; IWANT
1619742521 ; 7 ; 2 ; 14736 ; 140687176709888 ; 60 ; GOTRS
1619742521 ; 11 ; 6 ; 14737 ; 140059974014720 ; -1 ; IWANT
1619742521 ; 12 ; 3 ; 14737 ; 140059965622016 ; -1 ; IWANT
1619742521 ; 6 ; 4 ; 14736 ; 140687176709888 ; 70 ; GOTRS
```

Exemplo do registo de operações do cliente.

```
inst ; i ; t ; pid ; tid ; res ; oper
1621381195 ; 0 ; 2 ; 75106 ; 139855702861632 ; -1 ; RECVD
[lib] a 2 task is starting (with 0 delay)
1621381195 ; 1 ; 7 ; 75106 ; 139855702861632 ; -1 ; RECVD
[lib] a 7 task is starting (with 0 delay)
[lib] a 2 task has finished
1621381195 ; 0 ; 2 ; 75106 ; 139855694464768 ; 10 ; TSKEX
1621381195 ; 0 ; 2 ; 75106 ; 139855702857472 ; 10 ; TSKDN
1621381195 ; 2 ; 4 ; 75106 ; 139855702861632 ; -1 ; RECVD
[lib] a 4 task is starting (with 0 delay)
1621381195 ; 3 ; 7 ; 75106 ; 139855702861632 ; -1 ; RECVD
[lib] a 7 task is starting (with 0 delay)
1621381195 ; 4 ; 5 ; 75106 ; 139855702861632 ; -1 ; RECVD
[lib] a 5 task is starting (with 0 delay)
[lib] a 4 task has finished
1621381195 ; 2 ; 4 ; 75106 ; 139855677568768 ; 20 ; TSKEX
1621381195 ; 2 ; 4 ; 75106 ; 139855702857472 ; 20 ; TSKDN
1621381195 ; 5 ; 6 ; 75106 ; 139855702861632 ; -1 ; RECVD
[lib] a 6 task is starting (with 0 delay)
[lib] a 7 task has finished
1621381195 ; 1 ; 7 ; 75106 ; 139855686072064 ; 30 ; TSKEX
1621381195 ; 1 ; 7 ; 75106 ; 139855702857472 ; 30 ; TSKDN
1621381195 ; 6 ; 9 ; 75106 ; 139855702861632 ; -1 ; RECVD
[lib] a 9 task is starting (with 0 delay)
1621381195 ; 7 ; 5 ; 75106 ; 139855702861632 ; -1 ; RECVD
[lib] a 5 task is starting (with 0 delay)
[lib] a 5 task has finished
1621381195 ; 4 ; 5 ; 75106 ; 139855660783360 ; 40 ; TSKEX
1621381195 ; 4 ; 5 ; 75106 ; 139855702857472 ; 40 ; TSKDN
```

Exemplo do registo de operações do servidor.