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 parte de uma aplicação do tipo cliente-servidor, mais concretamente no que diz respeito ao tratamento de pedidos por parte do cliente. 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, esta 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

A implementação da parte B do projeto, que diz respeito ao servidor, será projetada e aprofundada posteriormente.

3 Conclusão

Apesar da complexidade de certas particularidades do projeto, de modo geral, podemos considerar que a implementação do cliente foi bem sucedida. 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.