

SOPE 2018/2019

TRABALHO PRÁTICO Nº 2

Simulação de um sistema de *home banking*

Implementação de uma arquitetura cliente/servidor baseada em *FIFO*s

**Trabalho Realizado por:** José Guerra. up20170621@fe.up.pt

Manuel Coutinho up201704211@fe.up.pt

Mário Gil up201705723@fe.up.pt

**Unidade Curricular:** SOPE

**Ano Letivo:** 2018/2019

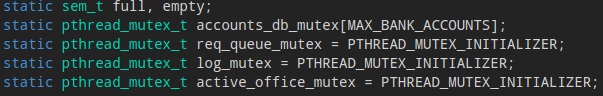
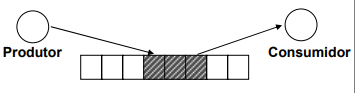
**Regente:** Jorge Alves da Silva

**Professor das aulas práticas:** Pedro Miguel Moreira da Silva

**Data de Entrega:** 2019/05/17.

Mecanismos de sincronização

Neste trabalho, tal como requisitado no guião, o atendimento de pedidos no sistema de *home banking* foi feito de acordo com o problema do produtor-consumidor. Para este efeito, foram necessários dois semáforos *full* e *empty*, um array de mutex’s *account\_db\_mutex[MAX\_BANK\_ACCOUNTS]* e três outros mutex’s auxiliares: *req\_queue\_mutex* (para proteger a fila de pedidos), *log\_mutex* (para proteger o log do server) e ainda *active\_office\_mutex* (para proteger o contador do número de balcões ativos).



Antes de proceder à explicação do funcionamento de cada mecanismo, considera-se relevante primeiro clarificar qual o papel assumido por cada parte do programa face a este problema. O **produtor** é o main thread do server, responsável por, quando um pedido chega de um qualquer user, recebê-lo e colocá-lo na queue de atendimento; o **consumidor** será um do threads ativos (balcões) a que o server deu origem no início da sua execução, encarregues de retirar e atender os pedidos presentes na queue.

Os semáforos ***full*** e ***empty***, bem como o mutex ***req\_queue\_mutex***, são mecanismos comuns de sincronização quando estamos perante um problema do produtor-consumidor. Os semáforos servem para sincronizar o main thread do server (produtor) e os balcões (consumidores). O full, embora se relacione com a quantidade de pedidos na queue, não significa que a mesma está cheia de pedidos, mas sim que possui pelo menos 1 pedido para ser processado. O empty regista informação acerca de quantos balcões estão livres naquele momento para poderem atender um pedido (devido a esta definição pensou-se em obter o número de balcões ativos pela diferença entre o número total destes e o valor lido deste semáforo. Contudo foi nos aconselhado “simplificar” e adotar a estratégia do contador). O mutex *req\_queue\_mutex* garante a exclusão mútua dos balcões quando estes tentam adicionar ou retirar à queue um pedido, de modo a evitar que, por exemplo, 2 balcões diferentes consigam ler e atender o mesmo pedido. O full é inicializado com 0 por não haver nenhum pedido na queue inicialmente, o empty é inicializado com o número de balcões(threads) que o server criou.

Os restantes mecanismos de sincronização, o array de mutex’s, o mutex *log\_mutex* e o *active\_office\_mutex*, embora não relacionados com o problema do produtor-consumidor, são de extrema importância para o sistema de *home banking*. O **array de mutex’s** garante que a mesma conta bancária não é acedida ao mesmo tempo por dois balcões diferentes, permitindo assim que não haja inconsistências quando múltiplas operações são requisitadas simultaneamente sobre as mesmas contas (nas operações de transferência, o primeiro *lock* é sempre no menor dos dois ids, de modo a evitar possíveis deadlocks). Poderia ter sido utilizado 1 único mutex para todo o array de contas, mas essa implementação introduziria esperas desnecessárias no sistema, visto que não há problema em 2 operações sobre contas diferentes serem realizadas em simultâneo. O mutex ***log\_mutex*** é utilizado para que, sempre que haja uma escrita para o log, não aconteça a interrupção da escrita para o mesmo log de um outro thread, evitando assim situações em que o registo teria conteúdo de 2 operações misturado. Finalmente, o ***active\_office\_mutex*** é usado para evitar que dois threads alterem o contador de balcões ativos ao mesmo tempo.

Inicialmente, o main thread do server irá ficar à espera de receber um pedido de um user, sendo esta a fase de “**produção**”.

No entanto, logo de seguida é feito um sem\_wait(empty) para garantir que não se insere nenhum pedido do user na queue caso não haja nenhum balcão disponível para atender o mesmo. Quando empty for maior que 0, significa que há pelo menos um balcão para atender o pedido e este pode então ser adicionado à queue. Após a inserção na fila do pedido, é feito um sem\_post(full) para sinalizar aos balcões que existe um pedido na queue que poderão atender.

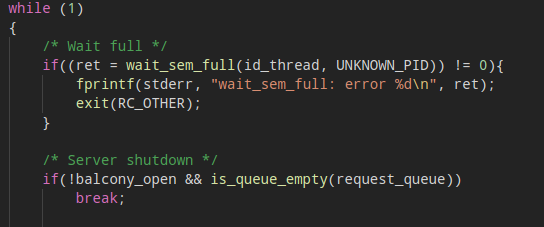
Do lado dos balcões é feito um sem\_wait(full) para esperar que haja algum pedido na queue. Caso haja pelo menos um, um dos threads será desbloqueado, irá retirar o pedido da queue e processá-lo, “**consumindo**”. No final dará a indicação de que já se encontra livre novamente para processar um próximo pedido com um sem\_post(empty). No processamento do pedido por parte de um balcão, quando este quer aceder à queue, é necessário fazer o lock do mutex req\_queue\_mutex, de forma a garantir que, quando este está a tentar aceder a queue, não existe mais nenhum balcão a querer fazer o mesmo. Após retirar o pedido da queue, o mesmo balcão fará o unlock do mutex para que outro possa aceder.

Estrutura das Mensagens

A estrutura de mensagens trocadas, em ambas direções, foi a proposta pelos professores e presente no código fornecido. A **escrita** das mensagens é “atómica” (escreve-a toda de uma só vez), evitando assim incongruências e possíveis misturas de dados no FIFO. A **leitura** é tripartida: primeiro lê o op\_type, seguido do length, e, com este último parâmetro, conseguimos ler apenas o tamanho exato ocupado pelo resto da mensagem (que será variável conforme os dados enviados, devido ao uso de *unions* nas structs utilizadas). Este protocolo permite que apenas sejam escritos e lidos os bytes necessários, otimizando a comunicação entre o servidor e os seus utilizadores (para ambos os lados).

Encerramento do server

De modo a conseguir controlar o encerramento do server, definimos uma variável global *balcony\_open* que indica se os balcões devem ser encerrados ou não. Além disso, aquando da abertura do *FIFO* seguro (em modo leitura), abrimo-lo também em modo de escrita separadamente. Esta técnica irá garantir que o *FIFO* em modo read se mantenha aberto por tempo indefinido, sendo possível o server mais facilmente fechá-lo ao encerrar, bastando fechar o descritor “dummy”.

Aquando do encerramento do server, após a receção e validação do pedido, além de fechar o descritor “dummy”, também as permissões do *FIFO* do server são alteradas para apenas de leitura, evitando que os users consigam enviar mais pedidos. Além disto, a variável global que controla o estado (aberto ou fechado) dos balcões também é mudada para 0, permitindo não só cada balcão sair do seu ciclo, como também o main thread. Contudo, destaca-se que os respetivos ciclos apenas serão deixados se não existir mais informação para ler no *FIFO* e a queue estiver vazia, de modo a garantir que efetivamente todos os pedidos obtêm resposta.

Com a finalidade de destravar todos os balcões que se encontravam retidos no primeiro wait\_sem\_full, o main thread, quando abandona o seu ciclo, envia tantos “sinais” quanto o número de threads, permitindo a terminação de todos os balcões. De seguida, procede a recolhê-los, por ordem. Estes ciclos devem ser separados, pois enquanto a junção dos threads é por ordem crescente de índice, os post\_sem\_full() realizados irão acordar os thread por uma ordem não definida, que poderá não corresponder à anterior, causando um impasse, pois se ficaria à espera de um thread que não iria terminar.

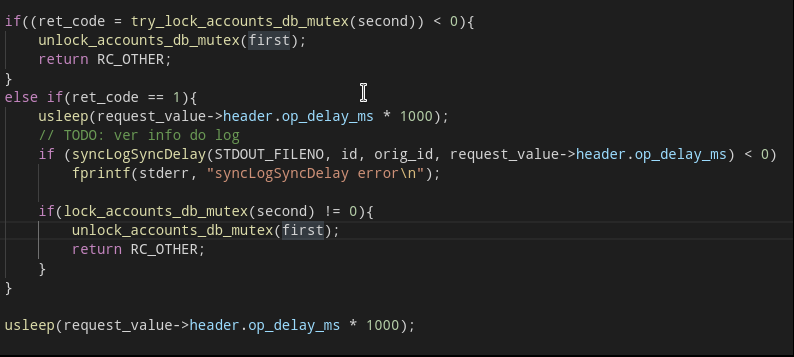
Extras

**Pode não se por isto.**

Este capítulo adicional serve apenas ressaltar dois pontos que julgamos ser de importante discussão.

O primeiro deles diz respeito a quando o enunciado nos pede para introduzir apenas um atraso quando o acesso a ambas as contas é feito em simultâneo aquando de uma transferência. Ficámos na dúvida se isto se referia a caso o nosso mecanismo de sincronização fosse apenas um mutex para todo o array de contas e não individualizado, ou caso requeria uma implementação com trylock para a segunda conta (segunda secção crítica) e caso este falha-se aí sim era posto um segundo atraso.

Na dúvida, apesar de entregue a situação que achamos correta (um atraso por acesso a secção crítica, já que o nosso modo de sincronização permite), temos um branch disponível no github com a solução alternativa sob o nome de transfer-2.

O segundo ponto a mencionar refere-se a, à semelhança do que foi feito para o slog (server do log), para evitar concorrência e possível falha na escrita de mensagens para o log do user, foi implementado um semáforo com nome que é aberto pelo server e ao qual todos os users têm acesso. Este procedimento tem alguns inconvenientes, uma vez que não consegue garantir sincronização na ausênica de um servidor e, por esta razão, também esta implementação se encontra num branch (named-sem) cuja utilização, aquando da situação descrita, escreve uma mensagem de warning ao utilizador, avisando-o que a sincronização nesta circunstância é impossível, e este se encontra por sua conta - nada de muito grave, pois é o que acontece na nossa situação default da solução entregue. Contudo, como o objetivo da feature se encontra parcialmente incompleto, optamos por deixá-lo de fora (poderia acontecer o acaso das mensagens interferirem também com a correção automática).