

SOPE 2018/2019

TRABALHO PRÁTICO Nº 2

Simulação de um sistema de home banking

Implementação de uma arquitetura cliente/servidor baseada em FIFOs

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

Manuel Coutinho up @fe.up.pt

Mário Gil [up @fe.up.pt](mailto:up@fe.up.pt)

**Unidade Curricular:** SOPE

**Ano Letivo:** 2018/2019

**Regente:** **Jorge Alves da Silva**

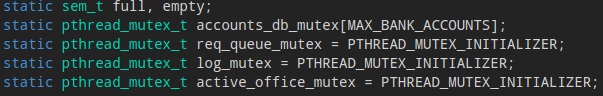
**Data de Entrega:** 2019/05/17.

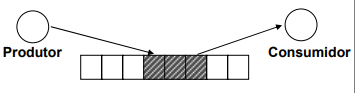
Estrutura das mensagens trocadas entre clientes e servidor (e vice-versa)

Mecanismo de sincronização

Neste trabalho, tal como foi requisitado no guião, o atendimento de pedidos no sistema de Home banking foi feito de acordo com o problema do produtor-consumidor. Para tal efeito foi necessário o uso de dois semáforos sem\_t full e sem\_t empty, um array de mutex’s pthread\_mutex\_t 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).

// TODO: por e fala do sem com nome??



 Antes de se passar para à explicação do funcionamento de cada mecanismo é necessário primeiro definir bem quem é que o produtor e o consumidor deste problema. O produtor é o main thread do server: quando um pedido chega de um qualquer user, é a função do server colocar este numa queue para que possa ser atendido por um consumidor que será um dos threads ativos (balcões) a que o server no início da sua execução deu origem.

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 pelo menos tem 1 pedido para ser processado. O empty dá 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 fazendo a diferença entre o número total destes e o valor lido deste semáforo, contudo foi nos aconselhado a “simplificar” e adotar a estratégia do contador). O mutex req\_queue\_mutex garante a exclusão mútua dos balcões quando estes tentam retirar da queue um pedido. O full é inicializado com 0 por não haver nenhum pedido na queue inicialmente e 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 (nas operações de transferência o primeiro lock é sempre no menor dos dois ids para evitar deadlocks). O mutex log\_mutex serve para que sempre que haja uma escrita para o log não aconteça a interrupção da escrita para o mesmo log de um outro processo, causando assim erros de escrita no ficheiro e o active\_office\_mutex para evitar que dois threads alterem o contador ao mesmo tempo.

No início do server, 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, se empty for maior que 0 significa que há pelo menos um balcão para atender o pedido. Após a inserção na fila do pedido faz-se um sem\_post(full) para sinalizar aos balcões que existe um pedido na queue.

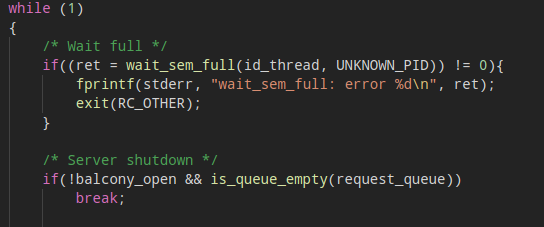
Do lado dos balcões é feito um sem\_wait(full) para esperar que haja algum pedido na queue, caso haja pelo menos um , um dos threads vai conseguir entrar, processar o pedido e no final dá a indicação que já se encontra livre para processar um outro pedido com um sem\_post(empty). No processamento do pedido por parte de um balcão , quando este quer aceder a queue é forçado a fazer o lock do mutex req\_queue\_mutex de forma a garantir que quando este esta a tentar aceder a queue não existe mais nenhum balcão a querer fazer o mesmo, depois de retirar o pedido da queue, o mesmo balcão da unlock do mutex para que outro posso aceder a queue.

Estrutura de Mensagem

A estrutura de mensagens trocadas foi a proposta pelos professores e presente no código fornecido, sendo a comunicação de escrita de mensagens “atómica” (escreve-a toda de uma só vez), evitando assim incongruências nos dados no FIFO, e a leitura tripartida: primeiro lê o op\_type, do qual se conhece o tamanho, depois o length, e com este parâmetro conseguimos ler apenas os tamanho correto do resto da mensagem, prevenindo leituras indevidas e escritas de tamanho desnecessário.

Encerramento do server

Para se proceder ao 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. 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 apenas se a queue estiver vazia e não houver mais informação para ler do fifo, é que os ciclos correspondentes são deixados.

 Com a finalidade de destravar todos os balcões que se encontravam retidos no primeiro wait\_sem\_full, o main thread quando acaba o seu ciclo envia tantos “sinais” quanto o número de threads, permitindo as suas terminações e recolhendo-os, por ordem, num ciclo separado (estes ciclos devem ser separados, pois enquanto a junção dos threads é por ordem crescente, qualquer um destes pode apanhar o sinal lançado pelo post\_sem\_full(), causando um impasse, pois se fica à espera de um thread que não irá terminar).

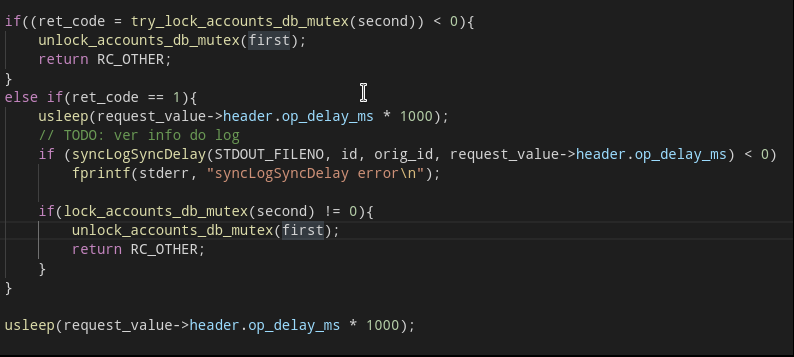
Extras

Pode não se por isto.

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

Primeiro deles é 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.

 Segundo ponto, foi à semelhança do 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 situção 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 correação automática).