Relatório Trabalho Prático Nº 2

“Home Banking”

Turma 6 Grupo 2

19 de maio de 2019

**Trabalho realizado por:**

Diogo Oliveira Reis, up201405015

Martim de Carvalho e Sousa Pinto da Silva, up201705205

Rúben Filipe Seabra de Almeida, up201704618

Estrutura das mensagens trocadas entre clientes e servidor

As mensagens trocadas entre o programa cliente e o programa servidor seguem o formato TLV (Tipo, Tamanho e Valor) canalizadas através de PIPES com nome (FIFOS). O server-side decompõe-se em duas grandes tarefas. A Main Thread que tem como função receber os pedidos definidos pelo user, tendo por isso acesso direto a um FIFO de uso comum entre os diferentes clientes e o server (FIFO\_SRV) criado e administrado pelo server, e as threads “atendimento” que simulam “balcões” cuja função é processar as demandas dos clientes.

As mensagens TLV que circulam em FIFO\_SRV, possuem tamanho variável, dependendo do tipo de pedido efetuado pelo cliente. Não obstante, todas as mensagens independentemente do tipo de operação, possuem um cabeçalho (“header”) comum, dos quais fazem parte o identificador do processo, o destino da resposta, o ID da conta, a senha associada e o atrasado da operação.

A resposta aos pedidos é feita sob a alçada do user-side. A solução apresentada executa em em ato contínuo ao envio sob a forma e formato descrito nos parágrafos antecedentes, a criação de um FIFO de resposta de nome FIFO\_XXXXX, em que XXXXX é uma representação genérica do Process ID (PID). O FIFO de resposta é identificado pelo PID do requerente. De referir ainda que, as respostas aos pedidos por parte do Server seguem também o formato de mensagens TLV.

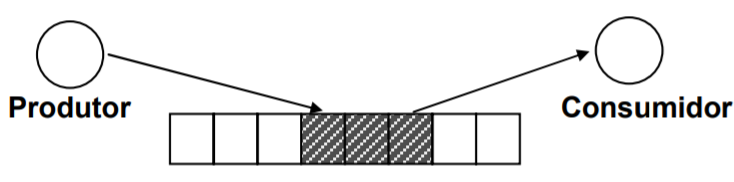
Mecanismos de sincronização utilizados

Este segmento pode ser dividido em três grandes secções:

1. Sincronia no processamento dos pedidos;
2. Sincronia no acesso ao array de contas
3. Escrita para Server-Log.

**Nota: Problemas de sincronização ocorrem, assim, sobretudo no server-side.**

Para o ponto 1: **Era imposto pelo enunciado,** que os pedidos fossem colocados numa fila de pedidos, de modo a garantir que eram processados por ordem de chegada. E que fosse implementado um sistema de processamento baseado no **problema do produtor de consumidor.**



Problema Produtor-Consumidor

No contexto do problema apresentado, considera-se o produtor a MAIN\_THREAD, aquele que deposita pedidos no buffer(fila). E os consumidores, as diferentes threads que processam pedidos. Para poder alcançar esta exigência implementou-se:

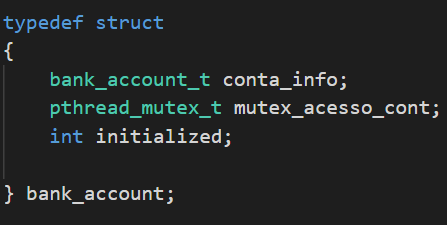
* **Semáforos:** 2 Semáforos: **sem\_consumer\_balcoes e sem\_producter\_pedidos.** Consumer\_balcoes, inicializado ao número de balcões introduzidas como parâmetro aquando na inicialização do servidor. Este semáforo será sincronizado ao longo da execução com o número de balcões disponíveis em cada momento para atender um pedido. Producter\_pedidos, inicializado a 0, segue ao longo da execução, o número de pedidos que se encontram em espera no buffer para serem processados.
* **Mutexes:** Tal como no problema do produtor consumidor, **pthread\_mutex\_t acesso\_fila,** é usado para garantir que as instruções de acesso à fila de pedidos sejam feitas de forma atómica e mutuamente exclusiva, erradicando potenciais erros de integridade dos dados que possam advir da execução concorrencial.

**A manipulação da fila de pedidos é considerada uma secção crítica.**

Para o Ponto 2: Neste capítulo, dividiremos os detalhes da operação em três grandes conjuntos: **A operação não é uma transferência, nem um pedido de encerramento. A operação é uma transferência. A operação é um pedido de encerramento (Próximo capítulo). Esta últimas duas são as que exigem da nossa parte especial atenção, dada as precauções adicionais que requerem.**

De referir que a necessidade de sincronização neste ponto está relacionada com a necessidade de garantir, que no momento de uma qualquer operação, o saldo da conta não possa ser alterado pela execução concorrencial do programa.

**A secção crítica no acesso ao Array de Contas está relacionada com o acesso ao saldo e da necessidade de garantir a integridade deste dado.**

A solução por nós apresentada tem por base a **incorporação de um mutex associado a cada conta**. Esta solução permite maximizar a multiprogramação, pois assegura que possam estar, em teoria, operações a serem processadas nas N contas em paralelo, com atomicidade de acesso aos dados garantida.

**Sincronização na operação de transferência:** No caso de uma transferência como estão envolvidas duas contas, ou seja, dois mutexes a serem necessários de bloquear para garantir acesso exclusivo aos dados da conta. Este modelo poderia causar um teórico caso de deadlock, por todos os meios indesejável. **Com o recurso aos dois mutexes, numa transferência simultânea de um cliente A para um cliente B, com outra do cliente B para o Cliente A, em simultâneo, teríamos uma situação de deadlock.**

Para resolver esta questão desenhamos um sistema misto que introduz outro mecanismo de sincronização utilizado na nossa implementação, **Condition Variables, pthread\_cond\_t condition\_var\_transfer, mais concretamente.**

Ao ser requisitada uma transferência, após a respetiva validação dos dados, é necessário projetar uma secção crítica a fim de garantir a preservação da integridade dos dados, é necessário garantir a atomicidade das operações associadas a cada conta.

1. O mutex associado à conta de origem é bloqueado
2. Caso o mutex associado à conta destino, esteja bloqueado (**Recurso à função trylock)**, estabelece-se uma variável de condição só desbloqueada quando o referido mutex estiver desbloqueado
3. O mutex associado à conta de destino é bloqueado
4. Após o processamento, para além de desbloquear os mutexes, somos obrigados, com esta solução, efetuar uma chamada **pthread\_cond\_signal**, para sinalizar o fim de uma operação de transferência. **De notar que esta estratégia mais elaborada só “entra em cena” quando o deadlock é iminente, no caso mais natural, tudo se resume a um mero bloqueio de mutexes.**

Realização do encerramento do servidor

O encerramento do servidor é uma operação que necessita de dois processamentos distintos na thread Main (produtor), e nas threads dos consumidores(balcões).

Em Main quando é recebido um packet TLV de tipo OP\_Shutdown é ativa um booleano especialmente inicializada para o efeito, que em conjunto com a ação da mudança das permissões para read-only do FIFO\_SRV, no user-side, garante que todos os pedidos pendentes quer no FIFO, quer na fila de pedidos são processados por ordem de cheagada. Esta implementação têm o seu culminar quando a fila de pedidos fica com tamanho 0 (não existem mais pedidos pendentes), então os procedimentos de shutdown são ativos dentre lado: **O semáforo de pedidos é colocado (usando sem\_post) ao número de balcões inicialmente definidos para garantir que qualquer thread bloqueada no semáforo esperando que um pedido seja depositado, possa ser desbloqueada para entrarem, também elas no mecanismo de shutdown definido para os balcões** **definido a seguir.**

Nos threads que simulam os balcões: Nas threads há a distinguir dois casos, **o do thread a que é destacado o thread de shutdown e as restantes threads que, mesmo não recebendo o pedido de shutdown, necessitam, também elas de desencadearem o processo de encerramento.**

Estes mecanismos de encerramento dos balcões só podem ser iniciados após o processamento de todos os pedidos que no momento da receção do pedido de encerramento estejam quer na fila de pedidos, quer no FIFO. Por outras palavras quando o thread main, encarregado pela produção de pedidos sinalizar o fim de execução, isto é possível, numa forma que evite espera ativa com o recurso a uma variável de condição, mais concretamente, **pthread\_cond\_t condition\_var\_shutdown.** Esta variável de condição é desbloqueada no thread Main quando inicia os seus mecanismos de shutdown, quando o tamanho da fila é, ou seja, todos os pedidos pendentes são processados. Quando a variável de condição é desbloqueada, é ativada uma flag booleana, que em conjunto com o desbloqueio forçado efetuado pelo post do semáforo do produtor referido acima, invoca a função **pthread\_exit,** recebidono main através de **pthread\_join**, retornando assim o servidor.