**Relatório de Implementação do Trabalho Prático - Processamento de Transações**

**Lucas L. Mentz1**

1Sistemas Operacionais - Bacharelado em Ciência da Computação - Centro de Ciências Tecnológicas (CCT) - Universidade do Estado de Santa Catarina (UDESC)

lucaslittermentz@gmail.com

1. **Introdução**

Este documento relata a implementação de uma solução para o trabalho proposto para a matéria Sistemas Operacionais - SOP0001 - no segundo semestre de 2017. O trabalho se trata de um programa em linguagem C, uma variação de produtor-consumidor em que há um produtor - trecho da função main - e alguns consumidores - threads que executam a função run\_transaction. O contexto do programa é uma aplicação de execução de transações bancárias em que há um produtor que enfileira transações bancárias lidas de um arquivo de transações e vários consumidores que executam as transações enfileiradas de forma paralela.

A seção 2 foca no funcionamento básico da solução proposta, as estruturas de dados utilizadas no armazenamento das contas de banco e transações além de identificar as possíveis condições de disputa quando usa-se paralelismo com threads. Na seção 3 são apresentadas os mecanismos utilizados na solução dos problemas de condições de disputa.

1. **Funcionamento Básico**

O programa desenvolvido é compilado em um executável tbanc. Seu uso é descrito quando usa-se o programa seguido de argumento --help, ou quando os argumentos informados não seguem o padrão necessário. Para sua execução são necessários um arquivo descrevendo as contas bancárias envolvidas e um arquivo descrevendo as transações bancárias.

A função main inicia e verifica se os argumentos passados na execução estão corretos ($ tbanc <arqcontas> <arqtrans> <numthreads>) e então faz a abertura dos arquivos e definição do número de *threads* segundo estes argumentos. A próxima operação é a leitura das contas contidas no arquivo <arqcontas> - uma conta por linha, no formato “<conta> <saldo>”, <conta> é um inteiro sem sinal e <saldo> é um número real (double) - que vão sendo guardados em uma lista dinâmica simplesmente encadeada. Em seguida são inicializados os semáforos e mutexes utilizados. A criação das *threads* é o próximo passo - <numthreads>, um inteiro positivo, dita o número de *threads* que serão usados. Após a criação das *threads* a *main* entra no loop de leitura e enfileiramento das transações que são lidas do arquivo <arqtrans> e postas em uma fila dinâmica simplesmente encadeada. Feito isso, o restante da *main* trata da apresentação de estatísticas e posterior limpeza das estruturas utilizadas para preparar o encerramento do programa.

As *threads* executam a subrotina run\_transactions. Essa subrotina tem um loop infinito com a condição de parada no seu interior - pára quando a fila de transações está sem elementos e a *flag* de encerramento foi ativada - seguida da operação de consumo da próxima transação na fila - faz a remoção da transação da fila e devolve uma cópia da transação para a *thread* chamadora -, recupera o endereço das duas contas envolvidas, verifica se a transação é válida (conta de origem e de destino existem e a conta de origem tem o saldo necessário) e então efetua a retirada do valor na conta de origem e o depósito na conta de destino.

As condições de disputa são as seguintes:

1. *thread* tenta remover uma transação ao mesmo tempo que a *main* tenta enfileirar uma. Isso pode causar a inserção de uma transação que terá seu ponteiro *next* apontando para uma transação que já não existe mais, consumida por uma *thread*.
2. Duas ou mais *threads* alteram o saldo de uma conta simultaneamente. Ao menos uma das operações será perdida imprevisivelmente.
3. Duas ou mais *threads* consomem transações ao mesmo tempo. Uma das *threads* irá falhar em consumir transação ou irá consumí-la não por inteiro e irá dar free() numa área de memória já desalocada.
4. Uma *thread* consome transações quando não há nenhuma na fila. Pode causar erros de acesso à endereços de memória inválidos ou impróprios.

As estruturas de dados utilizadas para armazenamento das contas de banco e transações foram filas dinâmicas simplesmente encadeadas - o primeiro elemento aponta para o próximo, que aponta para o próximo, etc., e o último elemento aponta para nulo, identificando o fim da fila. A inserção de contas e de transações foi feita no final da fila, modificando o ponteiro do último elemento para apontar para o novo inserido. O consumo de transações é feita no começo da fila, de forma que não seja necessário modificar nenhum dos elementos que permanecem.

Na próxima seção apresento as soluções utilizadas para prevenir as condições de disputa listados.

1. **Tratamento de Condições de Disputa**

O programa desenvolvido faz uso de semáforos da API POSIX e *mutexes* e *threads* da API pthread.

Foi feito o uso de um semáforo iniciado em 0 (zero) para controlar o acesso à fila dinâmica de transações - prevenindo a condição de disputa de número 4 - de forma que se uma *thread* tenta acessar a área crítica (consumo de transação) ela chama sem\_wait(&sem\_transacao) e é parada se não houver nenhuma transação enfileirada e a *main* é responsável por sinalizar (sem\_post(&sem\_transacao)) a inserção de transações. Essa implementação causa um problema, o de uma thread ficar presa esperando a sinalização quando a *main* já tiver lido todas as transações. Esse problema foi solucionado com a adição de uma sinalização adicional na saída do loop de leitura de transações efetivamente acordando a *thread* para que possa verificar que a condição de parada está ativa, e antes de encerrar seu próprio loop a *thread* que entrou na condição de parada também sinaliza o semáforo para acordar outras *threads* que possam também estar esperando.

A solução da primeira e terceira condição de disputa foram feitos com uso de um *mutex* guardando acesso à fila de transações. Quando um *thread* vai consumir uma transação ele solicita trava do *mutex* (pthread\_mutex\_lock(&mut\_transacao)) e quando ganha acesso faz seu uso da fila de transações e imediatamente libera a trava da fila de transações (pthread\_mutex\_unlock(&mut\_transacao)). A mesma coisa é feita pela *main* quando está inserindo transações.

A segunda condição de disputa foi solucionada com o uso de um *mutex* para cada conta, de forma que a *struct* que representa uma conta no banco inclui além do número da conta e do saldo uma instância própria do tipo *pthread\_mutex\_t*. Antes de fazer a transação as *threads* verificam se a transação é válida (contas de origem e destino existem e se há saldo suficiente na conta de origem) e como não é feito nenhuma alteração neste momento não foi usado *mutex* nesse momento. Quando uma *thread* vai alterar o saldo da conta de origem ele solicita trava (pthread\_mutex\_lock(&(c\_origem->mutex))) e imediatamente após já libera a trava dessa conta (pthread\_mutex\_unlock(&(c\_origem->mutex))), e faz o mesmo para a conta de destino.