Trabalho Prático 1 - Programação Concorrente

Brendon Vicente Rocha Silva
Graduando em Ciências da Computação
Universidade Federal de Santa Catarina - UFSC
Florianópolis, SC, Brasil
brendon.vicente@grad.ufsc.br

1 DISTRIBUIÇÃO DO TRABALHO

O trabalho foi realizado de forma individual, sendo inteiramente feito pelo autor deste arquivo.

2 ESTRUTURAS DE SINCRONIZAÇÃO

Para a solução do problema, optou-se pela utilização de seis estruturas de sincronização:

```
// Semáforo de senhas
sem_t sem_clerk_ps;

// Semáforos individuais
sem_t* sem_client;
sem_t* sem_clerk;

// Mutex de operações na fila
pthread_mutex_t mutex_queue;

// Mutex de operações no Ticket Caller
pthread_mutex_t mutex_client, mutex_clerk;
```

Figura 1 - Estruturas de sincronização

- ***sem_clerk_ps*:** semáforo responsável por garantir que os atendentes só iniciarão o trabalho quando *tickets* já tenham sido pegos. Os atendentes esperam a liberação do semáforo para começar a processar e os clientes, após retirarem uma senha, incrementam o semáforo, liberando, desta forma, os atendentes.
- ***sem_client*:** é um semáforo individual, de cada cliente e, de forma análoga ao anterior, garante que a *thread* cliente só encerre seu processamento quando o pedido está pronto. O cozinheiro é responsável por liberar este semáforo, possibilitando que o cliente finalize sua computação.

- ***sem_clerk*:** também é uma estrutura criada individualmente para cada atendente. Este semáforo, por sua vez, é utilizado para que a *thread clerk* aguarde até que o processamento feito em ***client_inform_order( )*** seja concluído.
- ***mutex_queue*:** *mutex* utilizado para sincronização de operações na fila de pedidos. Garante que duas ou mais *threads* não escrevam na fila, ao mesmo tempo.
- ***mutex_client*:** *mutex* que assegura que somente um cliente por vez tenha acesso a ***get_unique_ticket( )***.
- ***mutex_clerk***: análogo a ***mutex_client***. Garante que só um atendente tenha acesso a ***get_retrieved_ticket( )***.

3 CONCLUSÕES

É necessário o uso de mecanismos de sincronização para a solução do problema proposto, de forma concorrente, visto que múltiplas *threads* acessam a mesma região de memória compartilhada, em tempo de execução, podendo ocorrer inconsistências. Além disso, o uso de estruturas de sincronização é útil para situações em que uma *thread* deve aguardar a finalização do processamento de outra, como visto anteriormente, na resolução abordada neste trabalho.