Universidade Federal de São Carlos
Campus São Carlos

Problema Hilzer's Babershop
Problema da Barbearia de Hilzer

Alunos:
Amanda Lima Ribeiro 743504
Jean Araujo 620394
Rodrigo Sato Gomes 619809
Vinicius de Oliveira Peixoto 628263

Sistemas Operacionais

Professora:
Kellen Vivaldini

São Carlos, SP
2019

1. Introdução

Em uma barbearia existem 3 cadeiras, 3 barbeiros e uma área de espera em que pode acomodar 4 clientes no sofá, e existe uma área de espera para clientes futuros. O limite total de clientes na barbearia é de 20. Os consumidores não podem entrar dentro da loja se ela está lotada com outros clientes. Uma vez dentro o cliente senta-se no sofá, ou fica em pé caso esteja cheio. Quando um barbeiro está livre, o consumidor que está no primeiro lugar do sofá é atendido. Se tem algum cliente em pé, o que estiver mais tempo na loja senta no sofá. Quando o barbeiro terminar de cortar o cabelo de um cliente qualquer barbeiro pode aceitar o pagamento, mas apenas um pode registrar no caixa, e é aceito um pagamento de cliente por vez. Os barbeiros dividem seu tempo cortando cabelo, aceitando pagamento e dormindo enquanto espera um cliente.

2. Possíveis Problemas

Este problema, embora pareça simples e trivial, apresenta um elevado grau de complexidade na hora de transformá-lo em um algoritmo. Isso se dá pelo fato de que nenhuma atividade dele tem um tempo exato de execução, o que faz com que haja casos em que uma atividade seja executado mais rápida que outra, e a segunda atividade dependa de um dado gerado pela primeira atividade, mas a primeira esteja em um estado que dependa de um dado gerado pela segunda, que seria obtido mais adiante. Isso caracteriza uma situação de *deadlock*.

Outra situação que pode ocorrer é de uma atividade estar esperando um recurso para continuar, porém este recurso está sendo usado por outra atividade. Todas as atividades acabam usando o recurso e a primeira atividade nunca tem a chance de usar o recurso. Isso caracteriza uma situação de *starvation*.

3. Solução do Problema

O algoritmo usado trata o problema de forma bem linear, isto é, uma thread só avança sua execução uma vez que sua tarefa atual foi concluída. Isso é garantido através de uso de semáforos, apresentados na Figura 3-1. Os semáforos travam a execução, mantendo a ordem desejada.

```
sem_t barbeiro, corte, cadeira, caixa;          // Ações da barbearia
sem_t cliente, fazPagamento, recebeTroco;       // Ações dos clientes
sem_t imprime, bloqueio;                        // Ações do sistema
```

Figura 3-1: Os semáforos do algoritmo

Para as salas de espera e sofá, foram criadas filas. Filas são estruturas de dados muito úteis para este problema, uma vez que sua interação com as demais estruturas só se dá através do

primeiro elemento e do último, a cabeça e a cauda (Figura 3-2). Como a sala de espera e o sofá se comportam como filas de atendimento, somente o primeiro é atendido, e novos clientes são os últimos a serem atendidos, exatamente como na estrutura de dados.

```
typedef struct{
    sem_t cabeca;
    sem_t cauda;
} fila;
```

Figura 3-2: A estrutura FILA

Outra atividade que fez uso da estrutura fila foi a impressão na tela. Isso foi necessário pois todas as outras atividades estão sendo executadas simultaneamente, então a ordem de impressão pode ser diferente. Com a fila, cada impressão é colocada em sequência, e a primeira frase a pedir impressão é a primeira a ser impressa.

As filas avançam quando o elemento que está na frente, a cabeça. é usado, então o segundo requisita seu lugar como primeiro. Como um lugar foi liberado, a cauda pode receber mais um novo elemento. Esses procedimentos são vistos na Figura 3-3.

```
void avancaFila(fila *p_fila){
    sem_wait(&(p_fila->cabeca));
    sem_post(&(p_fila->cauda));
}
```

Figura 3-3: O método avancaFila

Os clientes e barbeiros são threads nesse algoritmo, uma vez que as ações dos clientes de dos barbeiros acontecem ao mesmo tempo.

3.1 As ações do cliente

O cliente, ao chegar na barbearia, tenta entrar. Caso todos os espaços estejam sendo usados, isto é, sala de espera e sofá cheios, o cliente vai embora. Se o cliente conseguir entrar, então o número de clientes dentro da barbearia aumenta em 1.

Como o cliente acabou de entrar, ele é colocado na sala de espera. Isso significa que ele é colocado na fila que representa a sala de espera. Ele será o último da fila. Ele, então, fica sempre checando o semáforo da posição a sua frente para saber se pode seguir em frente. Caso ele já seja o primeiro, ele irá verificar se ele pode ir para o sofá.

No sofá o sistema é exatamente o mesmo, mas com um número reduzido. Se o cliente teve sucesso em ir para o sofá, ele chama o método avancaFila() para que os outros semáforos da fila sejam atualizados e os clientes movam uma posição para frente.

Então, seguindo a mesma ideia da sala de espera, o cliente fica olhando a posição à sua frente para ver se pode avançar. Caso ele for o primeiro, ele sinaliza aos barbeiros que "há um cliente esperando atendimento". Isso é feito através de um dos semáforos, o semáforo **cliente**.

Com essa sinalização, ele aguarda uma cadeira de barbeiro ficar livre. O semáforo **cadeira** é usado para esse pedaço. Com isso, ele é impedido de seguir adiante enquanto não houver uma cadeira disponível. Porém, essa condição não é suficiente para garantir seu atendimento. Falta também um barbeiro estar disponível. Então, é usado o semáforo **barbeiro** para solicitar o atendimento de um barbeiro. Novamente, o cliente não avança enquanto não houver um barbeiro livre.

Com as duas condições cumpridas, o cliente finalmente começa a ser atendido, o que significa que ele sai do sofá e solicita para que a fila que representa o sofá avance através do método avancaFila(). Nesse momento, seu cabelo começa a ser cortado, e é indicado pelo semáforo **corte**.

Ao terminar de cortar o cabelo, ele libera o barbeiro para outros afazeres. Porém, suas atividades ainda não acabaram. Ele precisa fazer o pagamento, o que é indicado pelo semáforo **fazPagamento**. O pagamento deve ser feito para um barbeiro disponível, que num cenário provável será o mesmo que estava o atendendo anteriormente, já que ele acabara de se tornar disponível. Até lá, ele fica esperando seu troco com o semáforo **recebeTroco**. Após receber o troco, o cliente finalmente deixa a barbearia, diminuindo o número de clientes.

Como fica evidenciado aqui, as ações são sempre bloqueadas por vários semáforos, sendo impossível que um cliente avance as etapas sem que uma anterior a ela seja cumprida, mesmo que essas etapas sejam dependentes do barbeiro.

3.2 As ações do barbeiro

A primeira diferença gritante entre o barbeiro e o cliente é que a execução do cliente é única, enquanto o barbeiro precisa executar suas atividades sempre, pois ele nunca vai embora da barbearia.

Ao iniciar, o barbeiro está disponível para qualquer tarefa, o que ele indica pelo semáforo **barbeiro**. Se não houver clientes no sofá e ele estiver sem clientes precisando de serviço, ele vai dormir. Isso o caracteriza como um barbeiro ocioso. Nesse ponto ele está esperando um cliente, através do semáforo **cliente** e tirá-lo do estado ocioso.

Se um cliente pediu seus serviços, então ele começa a cortar o cabelo desse cliente, chamando o semáforo **corte**. O semáforo **corte** só é ativado, pelo cliente, depois dele sair do sofá e sentar na cadeira, o que estabelece a ordem do fluxo de atividades.

Depois de terminar de cortar o cabelo, ele verifica se há pagamentos para ser realizados através do semáforo **fazPagamento**, não necessariamente pelo cliente que ele acabou de atender. Ele recebe o pagamento, mas deve depositá-lo na caixa registradora. Como há só uma caixa registradora, o acesso é bloqueado pelo semáforo **caixa**, e o barbeiro aguarda até a caixa ficar disponível para então usá-la.

Após usar a caixa registradora, ele passa o troco para o cliente através do semáforo **recebeTroco**, e então fica livre para outros afazeres.

3.3 O sistema principal e condições de corrida

Como evidenciado nas seções anteriores, os semáforos ditam qual é o próximo passo a ser executado. Isso limita as possibilidades de erros como *deadlocks* e *starvation.*

Para o sistema principal, que gerencia as threads, há uma verificação de quantidade de clientes dentro da barbearia e de barbeiros ociosos. Caso não haja clientes e todos os barbeiros estão ociosos, então o programa chegou ao seu final.

Essa condição gera um problema, já que as chamadas de clientes e barbeiros ocorrem simultâneamente. Caso as threads dos barbeiros sejam executadas antes de um cliente, a condição será cumprida e o programa é finalizado prematuramente. Para isso, uma pausa foi introduzida com a finalidade de dar tempo de um cliente entrar na barbearia e quebrar a condição, segundo a Figura 3-4.

```
sleep(1);   // Fornece um tempo para que o primeiro cliente

/*
    O programa finaliza quando todos os clientes forem atend
*/
while(nroClientes != 0 || barbeiroOcioso != NRO_BARBEIROS);
```

Figura 3-4: Condição de finalização do programa

4. Conclusão

Trabalhar com threads e sistemas paralelos em escala maior se provou ser uma tarefa complexa. Não tratar os casos na sua devida ordem gera resultados inconsistentes, e nesses projetos maiores há muitos casos a serem tratados. A atenção e os cuidados para se evitar problemas de sincronização e bloqueio involuntário de recursos deve ser sempre alta.

No mais, foi verificado que não há uma única forma de se resolver o Problema da Barbearia de Hilzer. Outras ideias, como bloqueio usando mutex e abordagens parecidas com o sistema de produtor-consumidor, se mostram eficazes também, uma vez que só é necessário garantir a exclusão mútua, a ausência de *deadlocks* e de *starvation.*

Referências

http://web.cecs.pdx.edu/~harry/Blitz/OSProject/p3/SleepingBarberProblem.pdf (Último acesso: Maio de 2019)

https://www.researchcollection.ethz.ch/bitstream/handle/20.500.11850/69214/eth-5148-01.pdf?sequence=1&isAllowed=y (Último acesso: Maio de 2019)