Universidade de Brasília
Departamento de Ciência da Computação

Relatório do Trabalho - Programação Concorrente

Nome: Ian Nery Bandeira
Matrícula: 17/0144739

Resumo: Esse trabalho consiste na criação de uma situação hipotética de um mercado contendo algumas condições de corrida, e na implementação de um algoritmo que simula tal situação e procura solucionar tais condições de corrida.

1 - Introdução

Uma condição de corrida surge no software quando um programa de computador, para funcionar correctamente, depende da sequência ou do tempo dos processos ou threads do programa. As condições críticas de corrida normalmente causam uma execução inválida e bugs de software. Elas acontecem frequentemente quando os processos ou os threads dependem de algum estado partilhado entre eles, e que está ocorrendo de forma arbitrária entre as threads. A forma de solucionar o problema ainda utilizando corretamente variáveis compartilhadas é criando zonas críticas, que devem ser mutuamente exclusivas, enquanto o não cumprimento dessa solução pode gerar corrompimento do estado partilhado e a execução errônea do programa.

Uma condição de corrida pode ser difícil de reproduzir e debugar, já que o resultado final é não determinístico e depende do tempo relativo entre as threads presentes no programa. Problemas desta natureza podem, portanto, desaparecer quando se roda o programa em modo de debug ou adiciona um *printf* ou um *sleep*, porém, não é correto ou uma boa prática de programação solucionar as condições de corrida desta forma, já que o problema em si não foi solucionado, e sim mascarado.

Esse relatório é organizado como segue. Na seção 2, será feita a formalização do problema idealizado, bem como a descrição do que deveriam ser as variáveis compartilhadas que gerarão as condições de corrida. Na seção 3 será descrito como foi escrito o algoritmo para a solução do problema proposto na seção anterior, junto de como as condições de corrida foram solucionadas, e na seção 4, teremos as considerações finais acerca do projeto.

2 - Formalização do Problema

A princípio, o enunciado do problema tem como título **Mercadinhoinho**. O motivo para tal fora a idéia de criar um modelo de mercado, mas em uma escala reduzida, para que não polua muito o terminal com diversas threads sendo executadas, nem que exista a necessidade de adicionar uma complexidade exacerbada na resolução do problema. O enunciado do problema será descrito a seguir.

2.1 - Mercadinhoinho

No subúrbio de uma cidade, existe um mercadinhoinho. Esse mercadinhoinho é muito pequeno, mas com a dedicação de seus funcionários, todos os clientes conseguem comprar suas necessidades de forma rápida. O mercadinho possui apenas uma prateleira, com capacidade para 40 produtos, e 3 caixas rápidos, com preferência para clientes preferenciais (idosos, gestantes, etc). O diferencial deste mercadinhoinho, é que logo que um cliente tira um produto da prateleira para levar ao caixa, um funcionário logo irá repor o lugar vazio na prateleira com outro (ou o mesmo) produto. Por restrições impostas pelas directivas da OMS para o combate à propagação do vírus SARS-COV 19, é permitida a permanência de apenas 6 clientes simultaneamente dentro do mercadinhoinho.

2.1 - Explicação das Entidades

Neste problema teremos algumas entidades passivas e ativas. As entidades ativas, ou seja, os atores da simulação, são coincidentemente as pessoas envolvidas no problema, como os funcionários, clientes comuns e preferenciais. Estes são chamados assim pois necessitam de efetivamente executar uma tarefa nas entidades passivas presentes no mercado, como colocar/retirar um produto da prateleira, pagar algo no caixa ou esperar na fila. As entidades passivas são as que são sujeitas às condições de corrida baseadas nas ações das entidades ativas, ou seja, há de ocorrer uma exclusão mútua quando as ações são executadas, para que não hajam problemas, como a adição de dois ou mais produtos à prateleira, quando esta tem apenas uma posição livre; a retirada de dois ou mais produtos da prateleira, quando só há um produto nela. Analogamente, isso vale para a lotação do mercadinhoinho, e para a lotação e fila dos caixas de pagamento.

Na figura abaixo, têm-se um diagrama com o comportamento esperado para cada uma das entidades ativas dentro do mercadinhoinho, e como se trata de uma simulação, ambas as sequências de ações das entidades são repetidas indefinidamente.

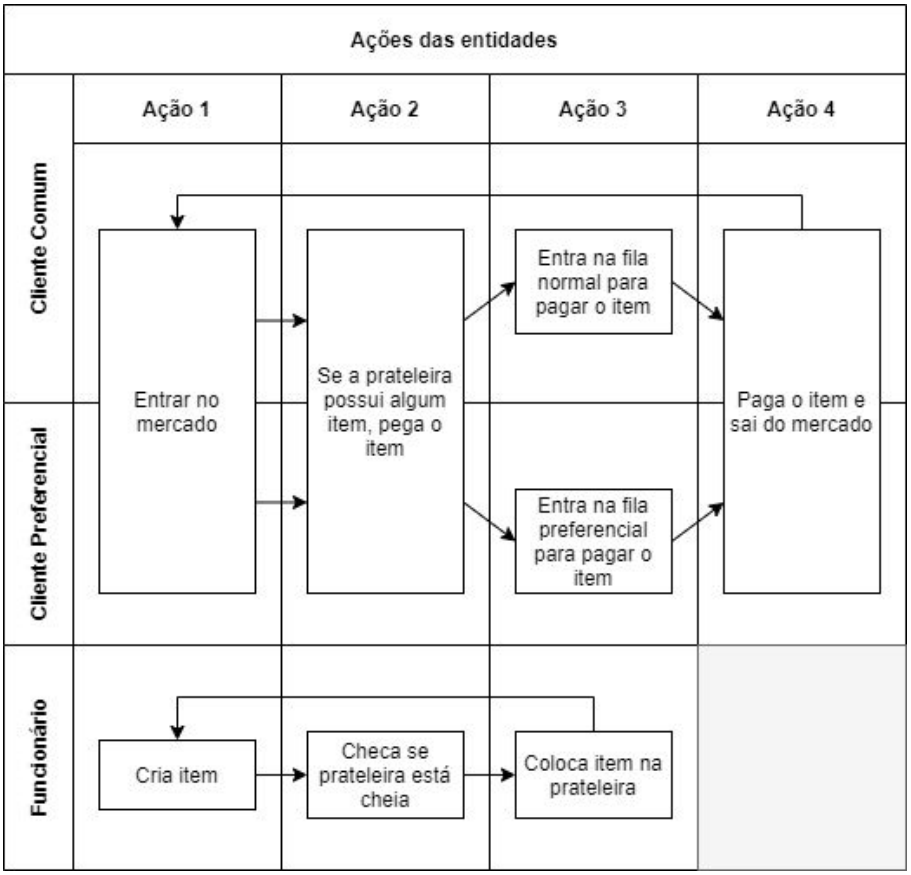

Figura 1 – Diagrama de ações executadas pelas entidades ativas da simulação.

3 - Implementação do Algoritmo

Como temos 3 entidades ativas no projeto, as ações das entidades de Funcionário, Cliente comum e Cliente preferencial serão trabalhadas em funções separadas para cada uma das respectivas entidades, para que exista um número arbitrário de atores que executam as ações de cada entidade de forma concorrente. Ao decorrer desta seção, será abordado como foram solucionadas as condições de corrida de cada ação, quando ocorrerem.

3.1 - Lotação do Mercadinhoinho

A primeira condição de corrida aparente é a de lotação do mercadinhoinho, que checa quantos clientes estão presentes no mercadinho para executar as outras ações, e não deixa que outros clientes entrem antes que outro cliente tenha saído do mercadinhoinho com seu produto e libere o índice de lotação. Para esta condição de corrida, fora criado um semáforo *lotacao*, declarado com a capacidade de clientes de 6; e este funciona de forma com que para que a entidade de Cliente comum ou Cliente preferencial execute outras ações, ela precisa primeiro passar por um *trywait(&lotacao) == 0,* que determina quando há espaço no mercadinhoinho. Quando um cliente sai do mercadinhoinho, ele executa um *sem_post(&lotacao)* para liberar um espaço no semáforo. Para determinar a quantidade de pessoas no mercadinhoinho, foi utilizada a função *sem_getvalue(&lotacao, &qtd_clientes)*, que coloca na variável *qtd_clientes* o valor desejado.

3.2 - Prateleira

No caso da entidade passiva de prateleira, todas as entidades passivas tem alguma ação para executar nela. Então, é necessário primeiro determinar a barreira que determina a capacidade de produtos que cabem nela, e isso é feito por dois semáforos, um que determina se existem posições livres na prateleira, e outro que determina se existem posições ocupadas na prateleira. Para todas as entidades, é necessário também um lock de exclusão mútua para com a prateleira. As threads da entidade Funcionário esperam por uma posição livre na prateleira, se existe, coloca um produto na prateleira (que é um vetor de inteiros) e dá um sinal para o semáforo de posições ocupadas, enquanto as threads das entidades de Clientes trabalham de maneira análoga, esperando para existir posições ocupadas na prateleira. Se existir, retira um produto da prateleira e dá sinal para o semáforo de posições livres.

3.3 - Fila do Caixa

A fila do caixa funciona de maneira análoga à questão 5 da prova 1, onde foi necessário criar um estacionamento onde threads de professores, funcionários e alunos "competiam" por vagas no estacionamento, e esperam em uma fila com prioridade para professores e funcionários, caso não existam vagas disponíveis.

Primeiro, é necessário um lock de exclusão mútua para garantir que não hajam condições de corrida entre os clientes que querem executar a ação de "pagar" seus produtos no caixa. Enquanto há caixas vazios, ou seja, se a variável *caixas_ocupados* não é igual a 3, deixa qualquer cliente pagar seu produto. Se a variável é igual a 3, ou seja, todos os caixas estão ocupados, cria-se uma fila preferencial e uma fila comum com a variável de condição para cada uma. No caso da fila preferencial, ela libera uma thread de cliente preferencial caso libere algum caixa, mas no caso da fila comum, a thread de cliente comum só é liberada caso libere algum caixa E se não existirem clientes preferenciais na fila, para garantir que exista a preferência entre clientes. Após executar essa ação, dentro do lock de exclusão mútua dos caixas, o cliente paga seu produto, libera um caixa para uso, dá um sinal para ambas as variáveis de condição, e libera um espaço no mercadinho.

4 - Considerações Finais/Conclusões

Considerando que o projeto executado teve como objetivo principal consolidar os conhecimentos de programação concorrente e aplicá-los em um contexto arbitrário, o maior desafio para mim neste projeto foi a elaboração de um enredo que fizesse sentido e que eu conseguisse aplicar os conceitos aplicados em sala. Apesar dessas dificuldades, foi muito proveitoso tanto para o uso da imaginação quanto para a consolidação do conteúdo.

Referências Bibliográficas

Ben-Ari, M., Principles of Concurrent and Distributed Programming, Prentice Hall, 2a ed., 2006.

Eduardo Adilio Pelinson Alchieri, Slides e Vídeo-Aulas disponíveis no moodle da disciplina de programação concorrente.