Trabalho de Programação Concorrente
Problema dos Caixas de Supermercado

Vinícius Caixeta de Souza, 18/0132199

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0202 - Programação Concorrente

1. Introdução

A programação concorrente foca na execução de programas com partes do código sendo executadas simultaneamente ao invés de sequencialmente, para isso são criadas threads que podem executar uma mesma função ao mesmo tempo. Apesar de ter benefícios como tempo de execução mais rápido e custo menor para implementar, o fato de várias threads poderem acessar dados ao mesmo tempo dificulta a correção de erros de programas e requer a criação de mecanismos que controlem o acesso à área de exclusão mútua, como locks e semáforos.

Este trabalho tem como objetivo criar um algoritmo que use threads e outros mecanismos estudados na matéria para resolver um problema complexo criado pelo aluno em que seja vantajoso utilizar programação concorrente. O problema desenvolvido foi sobre caixas normais e rápidos em um supermercado e para criar o algoritmo utilizou-se a linguagem de programação C usando a biblioteca POSIX Pthreads.

2. Formalização do Problema Proposto

O problema do supermercardo consiste em distribuir caixas rápidos e normais de forma que consiga atender a maior quantidade de clientes possível sem que hajam caixas inativos. Assim, o supermercado começa abrindo somente uma quantidade pequena de caixas rápidos e normais e após algum tempo os clientes começam a entrar em tempo aleatório para simular um cenário real. A quantidade total de clientes e caixas pode ser modificada para determinar o melhor cenário.

Cada cliente ao entrar no mercado decide pegar uma quantidade entre 5 até 60 itens, quanto mais ele pegar mais ele demora para ir ao caixa, caso tenha menos de 20 itens irá ao caixa rápido, caso contrário ao normal. Para escolher o caixa o cliente observa qual das filas possui o menor tamanho, caso nenhuma delas tenha um tamanho menor que 10 o cliente devolve os produtos nas prateleiras e sai do supermercado.

Os caixas atendem somente uma pessoa por vez, sendo que existe somente uma usando a esteira, caixas normais demoram mais para liberar a esteira do que os rápidos. Caso existem 5 pessoas em uma fila é acordado um novo caixa de acordo com o seu tipo, contanto que ainda exista um dormindo.

No problema original os caixas poderiam voltar a dormir caso nenhum cliente aparecesse, levando em conta que uma pequena quantidade não poderia parar de funcionar. Porém não foram encontradas alternativas para poder desenvolver está parte no algoritmo, explicação na próxima seção.

3. Descrição do Algoritmo Desenvolvido para Solução do Problema Proposto

Para começar o algoritmo são criadas as variáveis relacionadas a programação concorrente:

1. Threads
    - cliente[]: vetor aonde cada elemento representa um cliente, ao ser criado realiza a função f_cliente;
    - caixa_rapido[]: vetor aonde cada elemento representa um caixa rápido, ao ser criado realiza a função f_caixa_rapido;
    - caixa_normal[]: vetor aonde cada elemento representa um caixa normal, ao ser criado realiza a função f_caixa_normal.
2. Recursos compartilhados
    - filas[]: vetor de inteiros aonde cada elemento representa a quantidade de pessoas na fila de determinado caixa, quando o caixa não está aberto o valor é -1.
3. Locks
    - caixa_mutex: lock usado para garantir que o somente um cliente utilize o caixa;
    - posicoes_mutex: lock que garante acesso exclusivo ao vetor filas;
    - prioridade_mutex: lock que garante prioridade ao acessar o vetor filas.
4. Semáforos
    - sem_caixa_rapido: determina a quantidade de caixas rápidos que tem permissão para abrir;
    - sem_caixa_normal: determina a quantidade de caixas normais que tem permissão para abrir;
    - sem_tamanho_fila: determina a quantidade de pessoas que podem entrar na fila de determinado caixa;
    - sem_caixa_atendendo: determina se o caixa está esperando ou atendendo um cliente;
    - sem_cliente_no_caixa: determina se o cliente terminou de ser atendido ou está no processo.

Com as variáveis criadas os semáforos sem_caixa_rapido e sem_caixa_normal são iniciados com número de permissões menor que o número total de caixas de cada tipo para que somente alguns abram inicialmente. Em seguida é colocado o valor -1 para todas as posições no vetor filas para sinalizar que determinado caixa está fechado, além disso os semáforos sem_caixa_atendendo e sem_cliente_no_caixa são inicializados com 0 permissões.

Para sinalizar o começo do programa é printado na tela afirmando que o supermercado está abrindo, com isso as primeiras threads a serem criadas são as caixa_rapido e caixa_normal, na qual realizam as funções f_caixa_rapido e f_caixa_normal, respectivamente, com a variável id para representar o número de cada caixa. Após 0,1 segundos as threads dos clientes são criadas para realizar a função f_cliente também com a variável id, a diferença de tempo é implementada considerando que antes do supermercado deixar clientes entrarem já haverá caixas abertos.

As funções f_caixa_rapido e f_caixa_normal começam com o caixa verificando se tem permisão para passar no semáforo sem_caixa_rapido e sem_caixa_normal, respectivamente, se tiver ele printa no programa que está aberto, troca o valor -1 para 0 no vetor

de filas e inicia o semáforo sem_tamanho_fila com 10 permissões de acordo com seu id. Em seguida o caixa fica em um loop infinito (while(1)) e fica esperando abrir uma permissão no semáforo sem_caixa_atendendo, no qual significa que um cliente está querendo ser atendido.

Caso o caixa seja acordado por algum cliente (sem_post(&sem_caixa_atendendo)) ele demora 1,5 ou 0,6 segundos, dependendo do seu tipo, e por fim sinaliza ao cliente que terminou de passar os produtos e pode sair do caixa (sem_post(&sem_cliente_no_caixa)). Como já mencionado na seção anterior, no problema original o caixa poderia parar de funcionar caso não existisse ninguém na fila, porém ao tentar algumas algumas alternativas não foi possível implementar este objetivo, tais tentativas foram:

- Fechar o caixa caso ninguém estivesse no vetor filas: caso isto ocorresse o caixa quando fosse aberto iria imediatamente fechar, já que no início não haveria ninguém na fila;
- Esperar determinada quantidade de tempo (sleep()) para ver se alguém aparece na fila: embora funcionasse, poderia ocorrer de dentre a espera várias pessoas entrarem na fila e nenhuma ser atendida pelo caixa;
- Criar um break para sair do loop: neste caso o caixa iria em seguida fechar permanentemente, sendo que a ideia seria poder voltar a funionar caso o número de clientes nas filas voltasse a ser grande;
- Incluir mais um loop no semáforo sem_caixa_rapido ou sem_caixa_normal: ao invés de fechar permanentemente o caixa iria funcionar indefinidamente, sendo que o ideal seria após todos os clientes saírem eles fechassem.

Em relação a função f_cliente realizada pelas threads cliente, primeiro antes de entrar no supermercado os clientes escolhem um número aleatório entre 5 até 60 para representar a quantidade de produtos que irão comprar, em seguida é escolhido outro número aleatório para determinar em que horário o cliente entrará, levando em conta que em um cenário real os clientes não entrariam todos ao mesmo tempo.

Dentro do supermercado os clientes gastam uma determinada quantidade de tempo para pegar os itens, quantos mais produtos pegarem maior o tempo para irem ao caixa. Ao irem nos caixas eles escolhem a menor fila, para isso precisam de acesso exclusivo ao vetor filas para encontrar o menor valor, logo utiliza-se os locks posicoes_mutex e prioridade_mutex.

Ao ter acesso exclusivo o cliente determina em qual fila entrará utilizando a função menor_fila, na qual retorná o caixa com a menor fila dependendo da quantidade de produtos que o cliente possui. Em seguida é printado no programa aonde o cliente irá, quantos produtos possui e a quantidade de pessoas que já estão na fila, além de adicionar mais um na posição do vetor filas. Caso ao entrar na fila o tamanho dela vire 5 o cliente tenta acordar um caixa rápido ou normal, dependendo na quantidade de itens, finalizando o acesso exclusivo e desbloqueando os locks posicoes_mutex e prioridade_mutex.

Sabendo qual caixa tem menor fila ele verifica se tem permissão para entrar no semáforo do caixa (sem_trywait(&sem_tamanho_fila[fila_caixa])), caso tenha ele primeiro bloqueia o acesso para os outros clientes com o lock caixa_mutex[fila_caixa], habilita uma permissão pro tamanho da fila do caixa sem_tamanho_fila[fila_caixa] e diminui o tamanho do vetor filas usando o lock prioridade_mutex, ele utiliza somente este lock para garantir prioridade em relação aos clientes que ainda estão entrando na fila.

Em seguida o cliente acorda o caixa usando o semáforo sem_caixa_atendendo e espera ele processar o pedido usando o semáforo sem_cliente_no_caixa, quando acabar o cliente desbloqueia o lock caixa_mutex[fila_mutex] e printa no programa em verde avisando que conseguiu fazer as compras e está saindo do supermercado.

No caso em que o cliente não possui permissão para entrar no semáforo sem_tamanho_fila[fila_caixa] ele simplesmente aviso no programa em vermelho que não conseguiu encontrar uma fila de tamanho menor que 10 e então resolveu sair do supermercado sem comprar os itens, além de diminuir o tamanho do vetor filas em um usando o lock prioridade_mutex. Ao final das duas alternativas a função f_cliente é então finalizada e o programa termina as threads de clientes com o pthread_join.

- Para compilar o programa basta digitar em um terminal gcc -pthread -o trabalho supermercado_caixas.c e executar com ./trabalho
- A execução do programa pode ser vista no seguinte link: https://youtu.be/6CHTVZY4VGY

4. Conclusão

Neste trabalho foi realizado a criação de um problema que pudesse mostrar as vantagens da programação concorrente e os vários métodos disponiveis para realizá-la. Ao final da implementação pode-se imaginar o aumento da duração do código caso ele fosse executado sequencialmente com um número grande de clientes, notando que caso existissem mais caixas não haveria diferença.

Também foi aprendido a dificuldade de reparar o código pois além de possuir erros normais de programação existiam erros relacionados ao compartilhamento de dados e distribuição para seu acesso exclusivo. Apesar de não ter implementado todos os objetivos do problema ainda foi possível aprender a escolher os melhores recursos para realizar as tarefas executadas.

References