RELATÓRIO DO TRABALHO FINAL

Marcella Pantarotto (13/0143880)

Resumo: Trabalho final da disciplina de Programação Concorrente da Universidade de Brasília. O projeto proposto foi o desenvolvimento de um sistema com threads concorrentes e para isto, foi implementado um simulador de uma fazenda que produz leite e queijos, com threads de vacas leiteiras, bezerros, funcionários e um vendedor de queijos. Foi utilizado um lock e variáveis condicionais para o processo de retirada do leite e um semáforo para a produção e venda de queijos.

Palavras-chave: Programação Concorrente; Condição de Corrida; Variáveis Condicionais; Locks; Semáforos.

1 Introdução

O problema desenvolvido neste projeto foi o de um simulador de uma fazenda, utilizando programação concorrente e seus paradigmas, sendo threads, locks, variáveis condicionais e semáforos os principais conceitos utilizados.

O funcionamento da fazenda ocorre com threads e, para isto, primeiramente, foram criadas threads para as vacas leiteiras, para seus respectivos bezerros, para os funcionários que trabalham na fazenda e também para o vendedor da loja de queijos. A primeira região crítica é a do leite, que é representado por um array, em que cada posição representa a quantidade de leite produzido por cada vaca. Essa região crítica é acessada por meio do lock, que representa o úbere da vaca. Os bezerros bebem desse leite quando querem mamar, decrementando a quantidade de leite, e o funcionários, ao ordenhar o leite das vacas, também decrementam a quantidade de leite.

O bezerro e o funcionário disputam uma condição de corrida para acessar o leite, porém, como a vaca possui instinto materno e quer garantir a sobrevivência de seu filhote, o bezerro possui preferência ao leite em relação os funcionários. Portanto, foi utilizada uma variável global para indicar essa preferência e quando ocorrer uma competição pelo lock do úbere da vaca, o bezerro sempre ganha.

Em seguida, foi utilizado um semáforo para controlar a produção e a venda de queijos. Após os funcionários retirarem a quantidade de leite necessária para a fabricação de um queijo, incrementam o semáforo do queijo da segunda região crítica do projeto, isto serve para indicar que um queijo está pronto e disponível à venda na loja. O processo da venda é realizado pela thread do vendedor que, por sua vez, decrementa o semáforo do queijo, indicando que uma venda foi realizada com sucesso.

2 Fundamentação Teórica

Para o desenvolvimento e implementação deste projeto, os principais conceitos utilizados foram:

2.1 Programação concorrente

Na programação concorrente instruções são executadas por diversos processos ou threads simultâneos que cooperam entre si para a realização de uma tarefa, usando entrelaçamento de ações e um processamento simultâneo lógico. Um processo é dito cooperante quando é capaz de afetar ou ser afetado (mudado), pela execução de outro processo. Os processos se relacionam através de troca de mensagens e áreas de memória (variáveis) compartilhadas. Também podem ser chamados de fluxos de execução.

2.2 Condição de Corrida

Condições de corrida são situações onde dois ou mais processos estão acessando dados ou recursos compartilhados e o resultado final do processamento depende de quem executa e quando é executado.

2.3 Threads

A diferença básica entre threads e processos é que um processo não compartilha um mesmo recurso do computador simultaneamente com outro processo, enquanto que uma thread pode compartilhar um mesmo recurso simultaneamente com outras threads (dentro do mesmo processo). Dessa forma, threads são segmentos independentes de um processo em execução e também são as entidades escalonadas para usar a CPU.

2.4 Locks

É um mecanismo de sincronização de processos ou threads, em que processos ou threads devem ser programados de modo que seus efeitos sobre os dados compartilhados sejam equivalentes serialmente, ou seja, é uma ferramenta utilizada para acessar uma variável compartilhada, uma região crítica de memória compartilhada.

2.6 Variável Condicional

O processo de utilização de uma variável condicional consiste na criação de uma variável que é utilizada para sinalizar o acesso e liberação de uma thread sobre uma região crítica, ela pode ser usada para indicar uma certa preferência de umas threads em relação a outras.

2.6 Semáforos

Semáforos são muito similares aos locks, a diferença está no fato que se forem dados vários unlocks em um mutex e, em sequência, for dado um lock, ele fecha. Já com o semáforo, se der vários *up's*, ele vai contando, aumentando as permissões e, depois, quando der um *down*, só retira uma permissão. Se for utilizado um semáforo de apenas uma permissão e sempre der *down* e em seguida der *up*, funcionará igual a um lock.

3 Funcionamento

São criadas *N* threads de vacas, bezerros e funcionários e as threads são separadas em grupos de tal forma que existem *N* grupos, cada um formado por uma vaca, um bezerro e um funcionário. Para armazenar o leite, foi criado um array chamado **`leite[N]`** onde cada posição deste array representa a região crítica de cada grupo.

Também foram criadas variáveis condicionais para as vacas, os bezerros e os funcionários e um lock chamado **`úbere`** para controlar o acesso à região crítica. Portanto, se a posição do array do grupo em análise for menor que 5, a variável condicional da vaca é ativada, acordando a thread da vaca para que o leite seja produzido. Esse valor foi escolhido por ser a quantidade de litros que o funcionário retira ao ordenhar a vaca e por ser menor que a quantidade de litros que o bezerro mama, que é igual a 10.

Assim que o valor do leite for alterado para o valor máximo de leite produzido pela vaca, as variáveis condicionais do bezerro e do funcionário são acionadas para acordarem ambas estas threads. Ao competirem pelo lock do **`úbere`**, o funcionário sempre verifica a variável global de preferência do bezerro e caso ela esteja ativada e o funcionário estiver em posse do lock, soltará o lock para que o bezerro possa concorrer novamente e pegar o lock.

Uma vez que o funcionário conseguir retirar 15 litros de leite, será capaz de produzir um queijo e para isto, foi criado um semáforo chamado **`queijos`** que é incrementado toda vez que o funcionário conseguir atingir esta meta de leite. Por fim, a última thread criada é a que representa o vendedor, que fica adormecida enquanto o semáforo estiver zerado. Mas, assim que houver permissões, o vendedor decrementa o semáforo dos queijos para indicar que uma compra foi realizada com sucesso.

Algumas variáveis foram adicionadas para acompanhar o progresso do sistema, portanto pode-se acompanhar a quantidade total de leite bebido pelos bezerros, a quantidade total de leite retirado pelos funcionários, antes da fabricação do queijo, pois quando o queijo é produzido a quantidade de leite necessária para a produção é decrementada e, por fim, pode-se acompanhar também a quantidade total de queijos vendidos e quantos permanecem em estoque.

4 Análise de Resultados

Para o melhor entendimento do funcionamento do sistema, são impressas na tela de saída as ações realizadas pelas threads e cada uma delas será explicada a seguir, para isto, foi utilizado o grupo 0 como exemplo.

1. **`Grupo 0 - Vaca 0 criada`**

   Indica a criação da thread da vaca do grupo 0.

2. **`Grupo 0 - Bezerro 0 criado`**

   Indica a criação da thread do bezerro do grupo 0.

3. **`Grupo 0 - Funcionario 0 criado`**

   Indica a criação da thread do funcionário do grupo 0.

4. **`Vendedor criado!`**

   Indica a criação da thread do vendedor.

5. **`-- Vaca 0 pastando`**

   Indica que a thread da vaca foi ativada com sua variável condicional, pois a quantidade de leite é baixa ou nula. E também que a thread conseguiu pegar o lock do úbere.

6. **`Grupo 0 - Vaca 0 terminou de pastar`**
   `Total de leite no grupo: 30`

   Indica que a thread da vaca irá incrementar a quantidade de leite do grupo em 30 litros, mudando o valor dentro da posição 0 do array. Também significa que as variáveis condicionais do bezerro e do funcionário são ativadas, acordando ambas estas threads.

7. **`Grupo 0 - Bezerro 0 acabou de mamar`**
   `Leite remanescente no grupo: 20`
   `Total de leite bebido: 10`

   Indica que a thread do bezerro irá decrementar a quantidade de leite do grupo em 10 litros, mudando o valor dentro da posição 0 do array. Como indica, a quantidade de leite remanescente no grupo foi decrementado e a quantidade total de leite bebido foi incrementada.

8. **`Grupo 0 - Funcionário 0 acabou de ordenhar`**
   `Leite remanescente no grupo: 15`
   `Total de leite retirado: 5`

Indica que a thread do funcionário irá decrementar a quantidade de leite do grupo em 5 litros, mudando o valor dentro da posição 0 do array. Como indica, a quantidade de leite remanescente no grupo foi decrementado e a quantidade total de leite retirado foi incrementada.

9. *** Grupo 0 - Fazendo QUEIJO!**
   Leite remanescente no grupo: 0
   Total de queijo: 1
   Indica que a thread do funcionário conseguiu retirar 15 litros de leite e irá fabricar um queijo. Para isso, a quantidade total de leite retirado é decrementada por 15, quantidade necessária para a fabricação do queijo, e o semáforo de queijos é incrementado, como indicado na saída.

10. **** VENDA FEITA!**
    Queijos remanescentes: 2
    Total de queijos vendidos: 1
    Indica que a thread do vendedor irá decrementar o semáforo de queijos. Como indica na saída, pode-se ver quantos queijos ainda permanecem à venda e quantos já foram vendidos ao total.

11. **Não há leite suficiente! Bezerro 0 quer mamar!**
    Indica que a quantidade de leite presente na posição 0 do array é inferior a 10 litros, valor estabelecido para o consumo do bezerro. Também indica que a variável condicional do bezerro foi ativada e sua thread será adormecida.

12. **Não há leite suficiente! Funcionário 0 precisa ordenhar!**
    Indica que a quantidade de leite presente na posição 0 do array é inferior a 5 litros, valor estabelecido em cada ordenha do funcionário. Também indica que a variável condicional do bezerro foi ativada e sua thread será adormecida.

13. **Bezerro 0 quer mamar! Funcionário 0 terá que esperar!**
    Indica que a thread do funcionário pegou o lock do úbere, porém a thread do bezerro deseja a posse do lock para mamar, então o funcionário solta o lock do úbere.

A partir disso é possível entender a saída do sistema que é apresentada a seguir. Para tal, foi utilizado como exemplo o valor de $N = 1$, portanto existem apenas 1 threads para vacas, 1 thread para bezerros, 1 thread para funcionários e 1 thread para o vendedor, mas este valor pode ser alterado, atribuindo um novo valor para N.

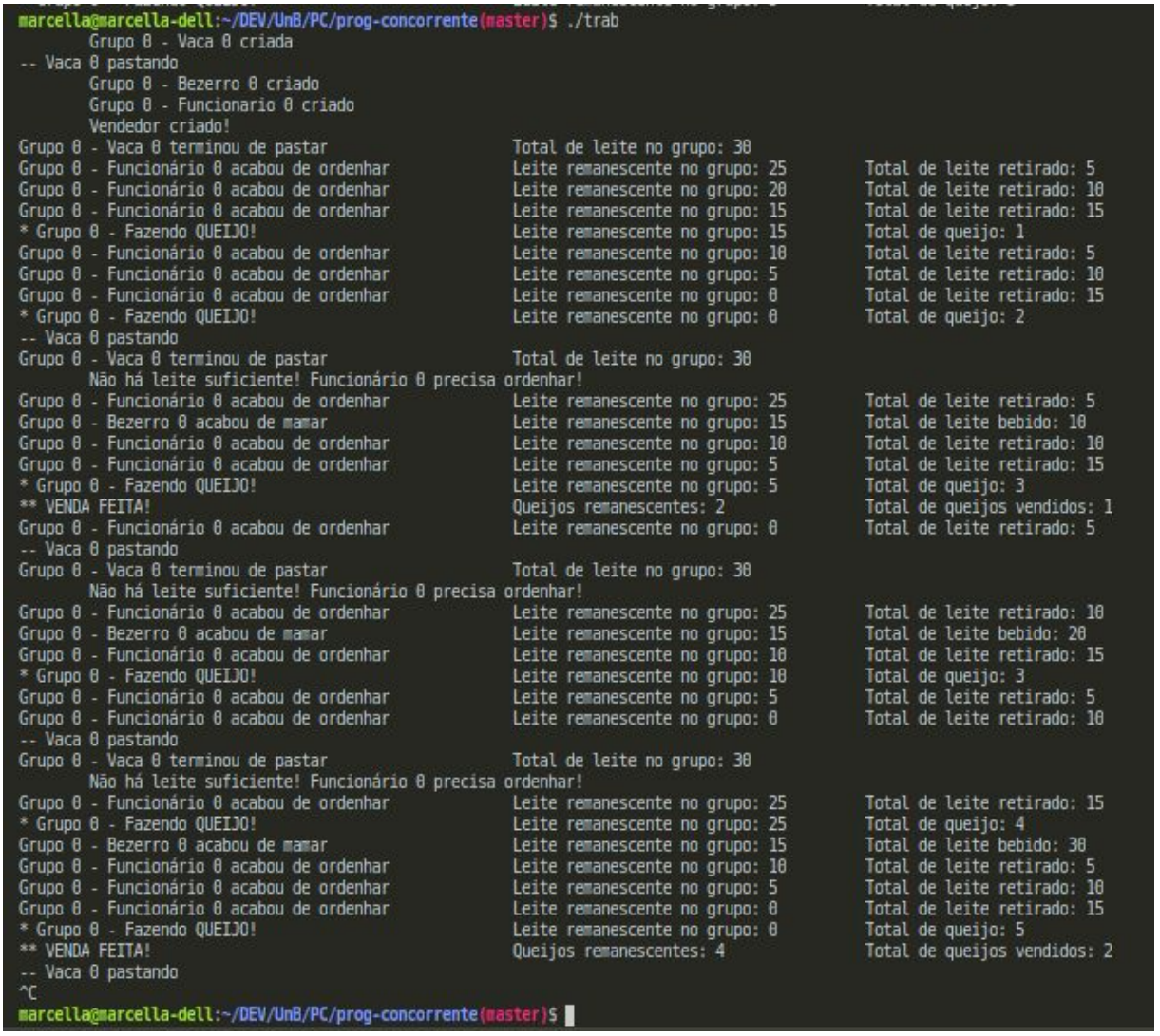

Figura 1 – Saída do sistema para $N = 1$.
Fonte: Ministério da Saúde, Governo Federal do Brasil.

Como pode ser percebido na Figura 1, todas as threads trabalham de forma simultânea e harmoniosa, não havendo sinais de *deadlock* ou *starvation*. A thread da vaca produz leite quando é necessário e as threads do bezerro e funcionário consomem este leite, sendo esta última thread também responsável pela produção do queijo, enquanto a thread do vendedor realiza as vendas sem problema algum.

5 Considerações Finais/Conclusões

Após a conclusão deste projeto, conclui-se que o sistema desenvolvido cumpre o objetivo proposto com sucesso. Através do uso de lock, variáveis condicionais e semáforos pôde-se trabalhar com threads e garantir o acesso e a modificação de regiões críticas de forma concorrente, sem nenhum sinal de *deadlock* ou *starvation*. Também não ocorre espera ocupada, pois a thread da vaca adormece depois de produzir o leite e acorda as threads do bezerro e do funcionário, estas, por sua vez, adormecem quando a quantidade de leite é

inferior àquelas que eles precisam e acordam a thread da vaca. O mesmo acontece com o uso do semáforo dos queijos, enquanto a thread do funcionário não incrementar o semáforo, a thread do vendedor ficará adormecida.

Referências Bibliográficas

1. Clay BRESHEARS, C. The Art of Concurrency, 1a ed. O'Reilly Media, Inc. Estados Unidos da América, 2009.
2. ANDREWS, G. R. Concurrent programming: principles and practice. The Benjamin/Cummings Publishing Company, Inc. Estados Unidos da América 1991.
3. BEN-ARI, M. Principles of Concurrent and Distributed Programming, 2a ed. Addison-Wesley. Estados Unidos da América, 24 fev. 2006
4. HERLIHY, M., SHAVIT, N. The Art of Multiprocessor Programming, 1a ed. Elsevier, Inc. Estados Unidos da América, 29 fev. 2008.