Programação Concorrente - Trabalho 1
Sincronização entre Processos

Lucas Dalle Rocha, 17/0016641

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0202 - Programação Concorrente - Turma 01A

1. Introdução

O artigo apresentado aborda a formalização do problema proposto, isto é, a simulação de um sistema de compra de placas de vídeo por diversos processos e a descrição do algoritmo obtido, haja vista especificação do escopo do problema, com utilização da biblioteca *POSIX Pthreads* em linguagem C.

Com base nisso, o intuito do artigo é detalhar as principais informações acerca do problema, além de destacar como paradigmas e aplicações da disciplina puderam contribuir para resolução do mesmo, com a utilização dos especificados mecanismos de sincronização (*locks*, variáveis de condição e semáforos).

Assim, o artigo será dividido de forma a contemplar tópicos relevantes no contexto de programação concorrente, haja vista uma melhor compreensão das ideias propostas. Portanto, a convenção utilizada baseia-se em: introdução, formalização do problema proposto, descrição do algoritmo desenvolvido para solucionar o problema proposto, conclusão e referenciais bibliográficos.

2. Formalização do Problema Proposto

Dessa forma, é importante esquematizar o escopo do nosso problema, de modo a facilitar a compreensão acerca da implementação do algoritmo. O problema apresentado é dividido em quatro classes de atores: o produtor de placas de vídeo; o usuário com finalidades não relacionadas a lucro monetário (considerado usuário comum); o minerador, que utiliza a placa comercializada com o intuito de produzir dinheiro; e o construtor de centros de mineração.

Assim, deve-se existir um controle sobre a divisão de recursos entre os processos e resolução das condições de corrida, principalmente sobre o estoque de placas de vídeo, que deve ter seu acesso restrito à região crítica haja vista a interação das três primeiras classes, e sobre o acesso e manipulação monetária, através de memória compartilhada entre os processos.

Dito isso, podemos esquematizar a descrição e o funcionamento desejado do nosso programa, bem como as restrições entre *threads*. Inicialmente, existe a presença de um único produtor de placas de vídeo, que tem por função fabricar determinado número de placas de vídeo quando o estoque estiver vazio, ou seja, periodicamente realiza a renovação da quantidade total de placas no estoque.

Ademais, existem múltiplos usuários e mineradores, não necessariamente iguais numericamente, que realizam compras das placas de vídeo diretamente do estoque do produtor. O usuário realiza a compra de uma única unidade da placa de vídeo, enquanto o

minerador realiza mais de uma compra, em uma mesma iteração, de modo a não realizar compra alguma caso não haja a quantidade mínima para seu consumo. Apesar disso, o usuário paga um preço superior ao do minerador, por unidade de placa, de modo que a prioridade de venda do produtor deve ser dada ao usuário, embora minerador consuma em maiores quantidades do produto. Já em relação à necessidade de reposição do estoque, ambas *threads* devem sinalizar ao produtor quanto a falta do produto em estoque, de modo respectivo ao seu consumo (usuário sinaliza quando não há placa de vídeo alguma e minerador sinaliza quando não há placas de vídeo o suficiente para o consumo de cada iteração).

Por fim, temos a relação entre minerador e construtor (único, igualmente ao produtor). Após diversas iterações de consumo por parte do minerador, caso atinja uma certa quantidade de placas de vídeo em sua posse, ocorre a contratação de um construtor, que realiza a construção de centros de mineração para o respectivo minerador, de modo que a partir do funcionamento dos centros de mineração, o minerador possa vir a lucrar. Outrossim, deve-se ressaltar que um usuário comum não realiza interação alguma com o construtor, visto que não possui intenção monetária com a utilização das placas de vídeo.

3. Descrição do Algoritmo

Inicialmente há definição de macros constantes no algoritmo, tais como número de usuários, número de mineradores, número de placas produzidas pelo produtor (a cada iteração do mesmo), número de placas consumidas pelo minerador (a cada iteração do mesmo), quantidade de placas necessárias possuídas pelo minerador para criação de centro de mineração, preço pago por cada unidade do produto pelo usuário e, de modo igual, pelo minerador, e finalmente, o dinheiro gerado pelo minerador por cada centro de mineração (a cada iteração do mesmo).

Assim, temos a criação da *thread* produtor, bem como das *threads* usuários e mineradores (baseado na quantidade definida na macro dos mesmos), e da *thread* construtor. Dessa forma, o armazenamento da quantidade de placas de vídeo disponíveis no estoque e do dinheiro relativo ao produtor é feito por variáveis compartilhadas, que se trata da região crítica entre produtor e os demais processos, com exceção do construtor. Além disso, foram utilizadas algumas outras variáveis relativas as *threads* mineradores (vetores, por sua vez), como a quantidade de placas de vídeo e centros de mineração possuídos por cada minerador, e seu respectivo lucro, baseado nos centros de mineração.

Já em relação aos mecanismos de sincronização entre os processos, foi utilizado um *lock* para controlar o acesso ao estoque das placas de vídeo disponíveis, visto que esse é compartilhado entre as *threads*. Ademais, três variáveis condicionais foram utilizadas para controlar o fluxo entre as *threads* produtor (adormece enquanto há placas disponíveis no estoque e acorda as demais *threads* quando produz novas placas), usuários (sinaliza o produtor e adormece quando não há placas disponíveis) e mineradores (sinaliza o produtor e adormece quando não há placas o suficiente para consumo, ou sinaliza os usuários e adormece quando os mesmos desejam adquirir alguma placa de vídeo, pois possuem prioridade). Por fim, houve a utilização de um semáforo para controlar a quantidade de usuários que desejam adquirir uma nova placa de vídeo (sinaliza desejo em consumir uma nova placa e mantém prioridade dos usuários, de modo a decrementar semáforo após consumo da placa), e um outro semáforo respectivo ao construtor, que espera permissão

(quantidade de placas de vídeo atingida por algum minerador) para construção de um novo centro de mineração, e ambos semáforos são inicializados com o valor zero (permissões são garantidas ao passo que usuário procura adquirir placa ou minerador possua condição de sinalizar construtor para construção de centro de mineração).

Acerca do dinheiro arrecadado pelo produtor, é importante observar que o usuário realiza incremento fixo a cada unidade de placa de vídeo adquirida, enquanto que o minerador realiza incremento baseado na quantidade de placas de vídeo consumidas por iteração da *thread*, bem como seu valor total da quantidade de placas possuídas. Dessa forma, caso o minerador possua quantidade suficiente para criação do centro de mineração, a *thread* construtor começa a atuar, pelo *post* dado no semáforo pelo minerador, e com a aquisição do centro, o lucro do respectivo minerador é baseado na quantidade total de centros possuídos (armazenado em sua posição do vetor) e o dinheiro gerado por cada centro.

Destarte, para análise dos resultados obtidos, mensagens de exibição são impressas no terminal para uma compreensão, em completude, do funcionamento do sistema como um todo, das quais podem ou não ser executadas. Dentre elas, estão expressas, a título de exemplo:

1. `Produtor: acumulei (int) reais no total.`

   Imprime, no início de cada iteração da *thread* produtor, o quantidade total adquirida pela venda das placas de vídeo para as *threads* usuários e mineradores, apenas como um informativo.

2. `Produtor: estou fabricando (int) placas de vídeo!`

   Imprime, em cada iteração da *thread* produtor, a quantidade de placas de vídeo a serem fabricadas, respectiva a macro constante inicialmente definida.

3. `Produtor: acabei de fabricar as placas de vídeo.`

   Imprime, ao final da atribuição de novas placas de vídeo ao estoque, respectiva a *thread* produtor.

4. `Usuario (id): quero placa de vídeo!`

   Imprime mensagem do respectivo usuário, ao sair do *sleep* (que simula usuário sem desejo de adquirir uma nova placa de vídeo), desejo de aquisição de uma nova placa.

5. `Usuario (id): não há placas de vídeo disponíveis, esperarei pelo produtor...`

   Imprime mensagem apenas quando o estoque de placas de vídeo encontra-se vazio, para em seguida adormecer a *thread* e acordar o produtor, a fim de que haja a

fabricação de uma nova remessa de placas.

6. `Usuario (id): consegui uma placa de vídeo.`

   Imprime mensagem após decrementar quantidade de placas de vídeo disponíveis no estoque e incrementar dinheiro acumulado pelo produtor, de modo a simular pagamento do usuário pelo produto.

7. `Minerador (id): já ganhei (int) reais com meu(s) centro(s) de mineração.`

   Imprime mensagem, no início de cada iteração da *thread* minerador, apenas se possuir pelo menos um centro de mineração, de modo a indicar o lucro obtido em relação a quantidade de centros de mineração.

8. `Minerador (id): algum usuário necessita da placa de video, vou esperar...`

   Imprime mensagem apenas quando o valor do semáforo utilizado para contabilizar a quantidade de usuários que desejam realizar a compra de uma nova placa é maior do que zero, de modo a acordar a *thread* usuário logo em seguida.

9. `Minerador (id): não há placas de vídeo suficientes no estoque, esperarei pelo produtor...`

   Imprime mensagem apenas quando a quantidade de placas de vídeo no estoque não for o suficiente para a aquisição por iteração dos mineradores (depende da macro constante definida no início do programa), de modo a acordar a *thread* produtor logo em seguida.

10. `Minerador (id): consegui (int) placa(s) de vídeo`

    Imprime mensagem após decrementar quantidade de placas de vídeo disponíveis no estoque (baseado no consumo por iteração dos mineradores) e incrementar quantidade de placas por minerador e dinheiro do produtor.

11. `Minerador (id): possuo o suficiente para criar um centro de mineração, chamando construtor...`

    Imprime mensagem após minerador possuir quantidade de placas de vídeo igual ou superior ao necessário para criar um centro de mineração, para em seguida decrementar a quantidade possuída em relação a utilizada para o centro e dar permissão (através do semáforo) para a *thread* construtor.

12. `Construtor: preciso de (int) placas de vídeo para criação de centro de mineração.`

Imprime mensagem anteriormente a espera por permissão para criação de um novo centro de mineração, apenas para sinalização da quantidade necessária expressa pela macro constante.

13. `Construtor: preciso de (int) placas de vídeo para criação de centro de mineração.`

    Imprime mensagem anteriormente a espera por permissão para criação de um novo centro de mineração, apenas para sinalização da quantidade necessária expressa pela macro constante.

14. `Construtor: vou construir o centro de mineração!`

    Imprime mensagem quando construtor recebe permissão, através do semáforo, e inicia período de construção de um novo centro (simulado com um *sleep*), de modo a incrementar número total de centros construídos, em seguida.

15. `Construtor: acabei de fazer um centro de mineração. Já construí (int) no total.`

    Imprime mensagem quando construtor finaliza a construção do centro de mineração, apenas para sinalização do término.

Além disso, mensagens de erro também estão estabelecidas no programa, tais como possíveis erros durante a criação de *threads*:

- `Erro ao criar thread (nome)! O programa será encerrado.`

E, por fim, mensagens de erro relativas ao funcionamento normal do programa, como definição incorreta das macros constantes que ignoram especificações que devem ser levadas em conta e foram apresentadas na seção de formalização do problema proposto, tais como:

- `Número de placas consumidas pelos mineradores maior do que o número de placas produzidas! O programa sera encerrado.`

- `Usuário deve pagar pela placa de vídeo mais caro do que o minerador, para que tenha prioridade! O programa sera encerrado.`

4. Conclusão

Destarte, haja vista a formalização do problema proposto e a descrição do algoritmo, é inegável a necessidade de aplicação de paradigmas de programação concorrente

e aplicações dos mecanismos de sincronização, apresentados durante o semestre. Isso se deve ao fato de que, com a utilização dos *locks*, variáveis de condição e semáforos, é possível garantir o acesso exclusivo por cada uma das *threads* em uma região de memória compartilhada, para que o funcionamento do sistema aconteça com interação de processos em um mesmo tempo lógico, sem que haja problemas de bloqueio eterno entre processos ou inanição dos mesmos. Assim, foi possível aplicar, de maneira prática, os conceitos ensinados, de modo a implementar lógicas aprendidas através dos problemas clássicos vistos em aulas e, consequentemente, aprofundar a compreensão em problemas diários de concorrência.

Referências

[1] G.R. Andrews e S. Elliott. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, 1991.

[2] Blaise Barney. *POSIX Threads Programming*. `https://hpc-tutorials.llnl.gov/posix/`. [Online; accessed 09-May-2021].

[3] M. Ben-Ari. *Principles of Concurrent and Distributed Programming, 2nd edition*. Prentice Hall international series in computer science. 2006.

[4] C. Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, 2009.