Sincronização de *threads* movimentando em uma grade
Exercício Prático - Fundamentos de Sistemas Paralelos e Distribuídos

Vinicius Silva Gomes {2021421869}[1]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

{vinicius.gomes}@dcc.ufmg.br

1. Introdução

A programação com múltiplas *threads* surgiu da necessidade de melhorar o desempenho e a eficiência de sistemas computacionais, especialmente em ambientes com múltiplos processadores. A ideia por trás é que várias tarefas sejam executadas simultaneamente em diferentes fluxos de execução, em um mesmo computador.

Isso trouxe inúmeros benefícios do ponto de vista de desempenho, concorrência/paralelismo e responsividade de sistemas, no entanto, também trouxe diversos desafios. O compartilhamento de recursos pode levar a condições de corrida e *deadlocks*, o acesso a recursos compartilhados deve ser muito bem gerenciado para que não hajam problemas de consistência entre os acessos, entre outros. Portanto, o desenvolvimento nesse modelo vem acompanhado de preocupações que não existem em um modelo sequencial.

Nesse sentido, a proposta desse exercício prático é implementar um sistema de sincronização entre *threads* que se assemelha a um sistema de controle de movimentação para um jogo digital ou para o controle de robôs. A ideia é que através desse sistema, todas as vantagens e desafios por trás do desenvolvimento *multithreading* sejam exploradas, principalmente quanto a sincronização de acesso a estruturas compartilhadas.

2. Estruturas, variáveis e funções

Para o desenvolvimento do programa, três *structs* principais foram utilizados: `position_t`, `thread_t` e `cell_t`. Além disso, as variáveis globais `lock`, `condition` e `occupied` e as funções `move`, `enter`, `leave` e `passa_tempo` são responsáveis por manipular os dados da entrada e realizar o movimento das *threads* ao longo do *grid*.

2.1. Estruturas

2.1.1. `position_t`

Um *struct* simples que contém três inteiros: `x`, `y` e `wait_time`. É uma estrutura instanciada dentro do *struct* da *thread* que armazena as coordenadas de uma posição (x, y) pela qual a *thread* deve passar e o tempo que essa *thread* deve ficar nessa posição.

2.1.2. `thread_t`

É o *struct* que armazena as informações sobre a *thread* que são lidas da entrada. Ele armazena o *id* e o grupo da *thread*, o número de posições que serão percorridas por ela e

um vetor de tipo estruturado `position_t`, que define as posições presentes no caminho que ela irá percorrer no *grid*, além do tempo que ela deve passar em cada posição.

Uma variável com esse tipo é instanciada antes de cada *thread* ser criada e é passada como parâmetro para a função que determina o comportamento que cada *thread* irá executar.

2.1.3. `cell_t`

Um *struct* auxiliar que armazena um inteiro (que funciona como um *booleano*) para indicar se a vaga na posição está disponível ou não e os inteiros referentes ao *id* e ao grupo de uma *thread*, caso exista alguma ocupando aquela vaga. Ele é usado como tipo estruturado para o tensor `occupied` (que será melhor destrinchado na próxima subseção).

Sua finalidade é facilitar o processo de verificar se existe alguma *thread* ocupando aquela vaga e, caso existir, armazenar e identificar as informações sobre essa *thread* que são pertinentes para os métodos de entrada e saída de *threads* em posições do *grid*.

2.2. Variáveis

Todas as variáveis listadas aqui são variáveis globais e, portanto, são acessíveis em todos os pontos do programa. As variáveis locais usadas pelas funções foram omitidas.

2.2.1. `lock` e `condition`

São matrizes de tipo `pthread_mutex_t` e `pthread_cond_t`, respectivamente. A primeira armazena os *mutexes* e a segunda as variáveis de condição associadas a cada posição no *grid*. Essas duas estruturas em conjunto permitirão o acesso à seção crítica de maneira controlada e que as *threads* possam entrar em espera e serem acordadas de acordo com o status da posição que desejam acessar.

Em especial, as travas e o sinal `wait` possibilitarão que as *threads* aguardem até que possam acessar uma nova posição do *grid* sem violar as restrições estabelecidas. Além disso, o sinal `signal` irá permitir que uma *thread*, ao deixar uma posição, acorde as outras *threads* que estão esperando pelo acesso a essa posição que ela deixou. Isso irá possibilitar que o programa, de modo geral, execute sem que um estado onde não é possível realizar progresso seja atingido (irá evitar que aconteça um estado onde uma *thread* tem a posse da trava de uma posição e não pode progredir enquanto uma *thread* deseja liberar uma vaga daquela posição mas não consegue a trava dela para fazer isso).

Essas estruturas foram modeladas como matrizes porque a definição do problema beneficiava essa modelagem. Como nenhum tipo de varredura é feita na estrutura com a posição das *threads* no *grid*, apenas acessos pontuais à posições são feitos, modelar os *mutexes* a nível de posição possibilita que *threads* que querem acessar posições diferentes executem de forma independente, sem a necessidade de travar a estrutura como um todo e impedir que as outras *threads* progridam.

O mesmo vale para as variáveis de condição. A modelagem delas a nível de posição permite que somente as *threads* que se interessam por uma posição entrem em espera

ou sejam acordadas. Assim, *threads* que não estão em um trecho do caminho onde há disputa pelo acesso a uma posição não são impactadas e progridem de forma independente. Modelagens diferentes poderiam fazer com que conjuntos maiores de *threads* entrassem em espera sem necessidade ou fossem acordadas sem que houvesse alguma mudança no status das *threads* que estão executando na posição que as interessa. Esses cenários prejudicariam a execução do programa, que não aconteceria da melhor forma possível.

2.2.2. `occupied`

É um tensor de tipo `cell_t`. Ele é responsável pelo controle interno do programa com relação à posição de cada *thread* durante a execução. Sempre que uma *thread* entrar em uma nova posição ou deixar a posição atual, `occupied` será atualizado para refletir essas mudanças. Assim, esse tensor é a região de memória compartilhada pelas *threads* e, portanto, seu acesso e manipulação representam seções críticas que devem ser protegidas pelos *mutexes*, para que não hajam inconsistências.

As primeiras duas dimensões do tensor indexam uma posição no *grid* e terceira dimensão armazena duas posições (ou vagas, como eu chamei durante a maior parte desse relatório). A modelagem é essa pois no máximo duas *threads* podem estar na mesma posição, em um dado momento, caso ambas façam parte de grupos diferentes. Assim, cada *thread* pode preencher uma das duas vagas com seu *id* e grupo, indicando que ela está ocupando aquela vaga na posição (x, y).

2.3. Funções

2.3.1. `enter`

Função chamada pela *thread* quando deseja acessar uma posição (x, y) do *grid*. Ela recebe como parâmetro os inteiros referentes a posição e o *id* e grupo da *thread*. Inicialmente, a função trava o acesso à posição (x, y) usando o *mutex* associado a essa posição e verifica as condições para que a *thread* possa entrar na posição:

- As duas vagas estão disponíveis;
- Apenas uma vaga está ocupada por uma *thread* de um grupo diferente do grupo da *thread* atual.

Caso alguma das condições não seja satisfeita, a *thread* entra em espera no corpo do *loop*, através do sinal `wait` associado à variável de condição e ao *mutex* da posição (x, y) que é disparado. Essa *thread* sairá dessa espera quando alguma outra *thread* enviar um `signal` na variável de condição associado a mesma posição.

Quando a *thread* sai do *loop*, temos que ambas as vagas estão vazias ou apenas uma delas está ocupada por uma *thread* com grupo diferente da *thread* atual. Sendo assim, a função encontra a vaga disponível, marca ela como ocupada e a preenche com o *id* e o grupo da *thread* atual. Por fim, a função libera o *lock* da posição (x, y) e termina.

2.3.2. `leave`

Função executada quando uma *thread* deseja liberar o acesso de uma posição que estava anteriormente. Ela recebe como parâmetros as coordenadas da posição e o *id* da *thread*

que está saindo.

Inicialmente, a função trava o acesso a posição (x, y) informada e verifica em qual das duas vagas dessa posição os dados da *thread* estão. Quando a posição é encontrada, *id* e grupo são definidos como -1 e a vaga é marcada como disponível novamente.

Após isso, a função dispara um `signal` que acorda todas as *threads* que estavam esperando por um sinal para acessar a mesma posição (x, y). Por fim, a função libera o *lock* dessa posição e termina.

2.3.3. `move`

Função executada por cada *thread* quando é criada. Ela recebe como parâmetro um ponteiro para um tipo estruturado `thread_t`, que armazena as informações da *thread* que foram lidas na entrada.

Inicialmente, a *thread* entra na primeira posição do seu caminho (que sempre estará livre já que por definição toda *thread* começa em uma posição diferente) e executa `passa_tempo` nessa posição. Em seguida, para todas as posições restantes no caminho, a *thread* tenta entrar na próxima posição com a função `enter`. Quando o acesso a essa posição for concedido, a *thread* sai da posição antiga com a função `leave` e executa a função `passa_tempo` na nova posição.

Essa sequência é repetida até que a *thread* passe por todas as posições no caminho. Quando isso acontecer, ela deixa a última posição que acessou (pode ser que alguma outra *thread* do mesmo grupo precise passar por essa posição ainda e, caso ela não se retire, essa passagem seria impedida) e a função desaloca o ponteiro com as informações lidas na entrada.

2.3.4. `passa_tempo`

Função definida e disponibilizada na especificação do trabalho para simular uma tarefa de computação intensiva. Ela também realiza as impressões no terminal para que o fluxo do programa e de movimentação das *threads* sejam avaliados.

3. Sincronização entre as *threads*

Assim como foi citado anteriormente, as matrizes de *mutexes* e variáveis de condição juntamente das funções `enter` e `leave` são as estruturas responsáveis por realizar a sincronização das *threads*.

O controle do acesso a cada posição do status atual do *grid*, contido no tensor `occupied`, é realizado pelo *mutex* associado a essa posição, por isso essa estrutura foi modelada como uma matriz. Isso possibilita um controle de acesso a nível de célula, o que permite que as *threads* executem de forma independente, caso não estejam disputando pelo acesso a uma mesma posição.

Da mesma forma, as variáveis de condição também foram modeladas em uma matriz para que as *threads* possam entrar em espera e serem acordadas de acordo com a posição que desejam acessar. Essa situação faz com que o *lock* de uma posição seja

liberado temporariamente apenas, o que assegura que o programa possa continuar executando sem estados de impedimento nos lugares do *grid* onde há disputa por posição e permite que *threads* que não estão na mesma região do *grid* executem de forma independente. Assim, essa modelagem impede que *threads* entrem em espera sem necessidade e que *threads* acordem sem que haja alguma mudança no status da posição para a qual elas desejam se mover.

Com a modelagem dessas estruturas seguindo essa ideia, as funções `enter` e `leave` deverão apenas controlar as mudanças feitas no tensor `occupied` para atualizar a posição das *threads*. A função `enter`, primeiramente, adquire a trava da posição e verifica as condições para que a *thread* possa ser inserida em uma vaga dela, acessando o tensor `occupied`. Essa verificação é feita em um `while` e não em um `if-else` porque, uma vez que uma *thread* entrou em espera com o sinal `wait`, é possível que, após ser acordada por um `signal` na posição que ela está esperando, outra *thread* tenha sido escalonada primeiro e as duas vagas estejam ocupadas novamente ou que a *thread* que ainda permanece executando seja do mesmo grupo que a *thread* atual. Assim, o uso do `while` nesse caso garante que as condições de acesso sejam checadas novamente após a *thread* ser acordada, assegurando que as restrições do problema não sejam violadas em nenhum momento da execução.

Quando a *thread* entra em espera, o *lock* da posição é liberado temporariamente. Isso permite que as outras *threads* continuem executando utilizando o *lock* daquela posição, fazendo com que não seja possível atingir um estado onde uma *thread* contém a trava da posição mas não consegue progredir e outra *thread* quer a trava para deixar a posição e liberar uma vaga mas não pode porque a trava está de posse da *thread* bloqueada.

Uma vez que o fluxo de execução saia do *loop*, a posse do *lock* na posição volta para a *thread* e sabemos que existem condições para que ela se mova para a posição que deseja. Assim, o tensor `occupied` é atualizado com a inserção da *thread* em uma das posições vagas e a trava da posição é liberada.

Após conseguir se mover para a posição atual, a *thread* deve deixar a posição que estava anteriormente. Para tanto, a função `leave` é chamada. Inicialmente, ela trava o acesso a posição que vai ser liberada usando o *mutex* dessa posição. Após isso, a função procura em qual das vagas a *thread* está armazenada, define a vaga como disponível e preenche com o valor -1 o *id* e o grupo da *thread*.

Em seguida, a função dispara um `signal` associado a variável de condição da posição liberada. Esse sinal irá acordar todas as *threads* que estão em espera pelo acesso à essa posição. Fazendo isso, as *threads* recém-acordadas passarão pelo teste de condição do *loop* novamente e, de acordo com as condições no momento em que cada uma for escalonada, poderão prosseguir com sua execução ou irão entrar em espera novamente. Por fim, a função `leave` libera a trava da posição e termina sua execução.

Essa estratégia implementada garante o acesso consistente à seção crítica e implementa a sincronização de forma adequada entre as *threads*, utilizando as variáveis de condição para controlar o estado das *threads* e liberar o *lock* de uma posição temporariamente. Isso impede que hajam estados que bloqueiem a execução de alguma *thread* por causa de espera com posse do *lock* e assegura o desempenho adequado para o programa, garantindo que as *threads* não entrem em espera sem necessidade e também não sejam

acordadas quando alguma mudança que não as interessa acontece.

4. Conclusão

Com o desenvolvimento do programa, foi possível compreender melhor como o desenvolvimento com múltiplas *threads* é feito. Em especial, foi possível observar na prática as diferenças com relação ao desenvolvimento sequencial e como *multithreading* é capaz de proporcionar maior responsividade e desempenho aos programas.

Além disso, foi possível observar o impacto que as diferentes formas de se modelar o acesso exclusivo às estruturas compartilhadas causam no programa. Mais que isso, a modelagem proposta se beneficiou da característica dos acessos que o programa deveria realizar para propor uma estrutura de *mutexes* e variáveis de condição que permitissem que o controle de acesso fosse feito a nível de posição no *grid*.

Isso deu mais independência para as *threads*, o que propiciou um desempenho maior para o programa e um entendimento mais direto da dinâmica de controle de acesso às posições e de como as *threads* entram em espera e acordam a partir dos sinais.