Computação Concorrente (MAB-117)
Comunicação entre threads via memória compartilhada e mecanismos de sincronização (parte I)

Prof. Silvana Rossetto

[1]Departamento de Ciência da Computação (DCC)
Instituto de Matemática (IM)
Universidade Federal do Rio de Janeiro (UFRJ)
Setembro de 2011

1. Comunicação e sincronização entre threads via memória compartilhada

A facilidade provida por um espaço de endereçamento compartilhado é normalmente usada para implementar a **comunicação** entre as threads de uma aplicação. Quando uma thread precisa trocar/compartilhar uma informação com outra thread, ela o faz alterando o valor de uma variável comum para as duas threads, ou seja, uma variável que está armazenada em um endereço de memória que as duas threads compartilham (conhecem e podem acessar).

O modelo de organização tipicamente adotato nos computadores paralelos multiprocessadores é baseado no uso de memória compartilhada, i.e., um espaço de endereçamento físico único é compartilhado entre todos os processadores. Esse modelo de organização facilita o desenvolvimento de aplicações para essas arquiteturas, pois é uma extensão direta do modelo de acesso à memória usado nos computadores uniprocessadores: diferentes linhas de execução (threads) podem acessar variáveis de um programa que estão em endereços de memória igualmente acessíveis por outras threads.

Quando duas ou mais threads de um programa precisam trocar informações entre si, o programador define variáveis de acesso compartilhado entre elas. Desse modo, quando uma thread tem um valor novo que deve ser comunicado para as demais threads, ela simplesmente escreve esse valor na variável compartilhada, independente das demais threads estarem esperando por esse valor naquele momento. Quando outra thread precisar saber qual é o valor atual dessa informação, ela simplesmente lê o conteúdo atual da variável compartilhada.

Assim, toda a comunicação entre as threads se dá de forma **assíncrona**, i.e., as threads escrevem/lêem valores nas variáveis compartilhadas a qualquer tempo. Para garantir que a comunicação ocorra de forma correta (por exemplo, que o valor da variável não seja sobreescrito antes que as outras threads leiam o valor anterior), a interação entre threads via memória compartilhada gera a necessidade de **sincronização**. Neste texto são discutidos aspectos relacionados com a **comunicação** e **sincronização** entre linhas de execução distintas dentro de uma mesma aplicação, usando o modelo de **memória compartilhada**.

1.1. Seções críticas e ações atômicas

Quando escrevemos programas concorrentes com compartilhamento de variáveis entre threads, normalmente esperamos que a execução das expressões aritméticas ou sentenças de atribuição da linguagem de programação sejam feitas de forma *atômica*, i.e., que

uma vez iniciada a ação por uma thread ela seja executada por completo antes que outra thread opere sobre a mesma variável. Entretanto, as expressões aritméticas e sentenças de atribuição das linguagens de alto nível (C, Java, etc.) são normalmente compiladas em mais de uma instrução de máquina. Por conta disso, durante a execução das sentenças ou expressões da linguagem de alto nível por threads distintas, pode ocorrer entrelaçamentos das instruções de máquina correspondentes, dando possibilidade de ocorrência de resultados inesperados.

Como exemplo, considere duas threads, T_1 e T_2, associadas aos seguintes trechos de código (as variáveis y e z são inicialmente iguais a 0):

```
T1:                          T2:
x = y + z;                   y = 1;
                             z = 2;
```

Se assumirmos (incorretamente) que a execução de cada sentença de atribuição é uma ação atômica, o valor final de x, computado pela thread T_1, poderá ser 0, 1 ou 3, representando as somas $0 + 0$, $1 + 0$ e $1 + 2$, respectivamente. Entretanto, as instruções de máquina para as threads T_1 e T_2 poderão ser compiladas da seguinte forma:

```
T1:                          T2:
(1) mov y, %eax              (4) mov 1, y
(2) add z, %eax              (5) mov 2, z
(3) mov %eax, x
```

Desse modo, os seguintes entrelaçamentos de instruções de máquina poderão ocorrer:

```
(1), (2), (3), (4), (5) -> x=0
(4), (1), (2), (3), (5) -> x=1
***(1), (4), (5), (2), (3) -> x=2 ***
(4), (5), (1), (2), (3) -> x=3
```

Como o exemplo ilustra, uma vez que as instruções de máquina é que são ações atômicas (e não as expressões e sentenças da linguagem de alto nível), o entrelaçamento da execução das instruções de máquina é que determina o resultado da computação.

Toda referência a uma variável que pode ser acessada/modificada por outra thread é uma **referência crítica**. Para impedir a ocorrência de resultados indesejáveis para uma determinada computação, os trechos de código que contêm referências críticas **devem ser executados de forma atômica**. Esses trechos de código são denominados de **seção crítica** do código. Identificar os trechos de seção crítica e coodenar a sua execução é um dos problemas fundamentais da programação concorrente.

1.2. Seções de entrada e saída da seção crítica

Mecanismos de sincronização de códigos permitem **transformar uma seção crítica em uma ação atômica** usando dois outros trechos de código especiais chamados **seção de entrada** e **seção de saída**. Essas duas seções adicionais de código tem por finalidade "cercar" a entrada e saída da seção crítica, garantindo que os requisitos de execução atômica sejam atendidos. O pseudocódigo abaixo ilustra essa estratégia:

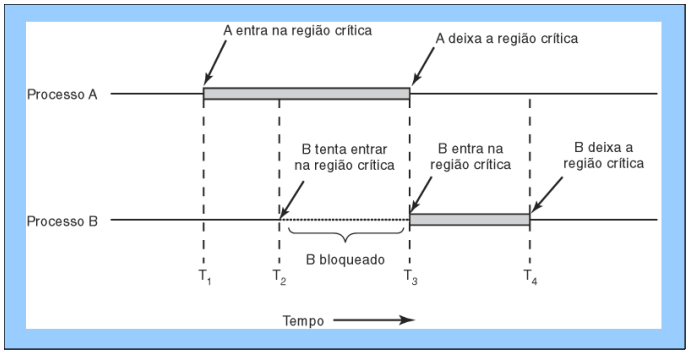


Figura 1. Exemplo de entrada e saída na seção crítica por dois processos (fonte: Pearson).

```
while(true) {
  executa fora da seção crítica (...)
  requisita a entrada na seção crítica //seção de entrada
  executa a seção crítica (...)        //seção crítica
  sai da seção crítica                 //seção de saída
}
```

A Figura 1 mostra um exemplo de execução de dois processos que entram e saem das suas seções críticas (pedaço do código onde acessam um objeto compartilhado pelos dois processos).

O **problema de exclusão mútua consiste em escrever o código das seções de entrada e saída para a seção crítica** de modo que os seguintes requisitos sejam atendidos [1, 2]:

1. *Exclusão mútua*: quando uma thread está executando a sua seção crítica nenhuma outra thread pode executar a seção crítica relacionada com o mesmo objeto, nesse caso devem esperar na seção de entrada (se a thread na seção crítica perde a CPU, ela permanece com a posse da seção crítica) (propriedade safety).
2. *Ausência de deadlock*: alguma thread em algum momento entra na seção crítica (*propriedade liveness*).
3. *Ausência de starvation*: se uma thread está tentando entrar na seção crítica, então em algum momento ela deverá conseguir entrar (*propriedade liveness*).
4. *Independência do restante do código*: uma thread que termina sua execução fora da sua seção crítica (ou entra em loop infinito fora da seção crítica) não deve interferir nas seções de entrada e saída das threads restantes, e quando nenhuma thread estiver na seção crítica e há threads que desejam executá-la, apenas as threads que estão executando as seções de entrada e saída podem participar da decisão de qual delas ganhará o acesso à seção crítica.
5. *Garantia de entrada*: quando nenhuma thread está na seção crítica e uma thread deseja executar a seção crítica, ela deve passar pela seção de entrada imediatamente, e quando uma thread executa a seção de saída e há outras threads esperando na seção de entrada uma delas deverá ser escolhida para entrar na seção crítica.

Para que as threads possam executar corretamente e chegar ao final das suas execuções, é necessário que toda thread que entre na seção crítica saia dela em algum momento, i.e., *uma thread não deve executar um loop infinito ou terminar a sua execução dentro de uma seção crítica*.

Uma questão básica para o funcionamente desse tipo de mecanismo é que as **seções de entrada e de saída devem ser ações atômicas** (caso contrário a sincronização por exclusão mútua não será garantida).

1.3. Formas de sincronização

Sincronização refere-se a qualquer mecanismo que permite ao programador controlar a ordem relativa na qual as operações ocorrem em diferentes threads. Duas formas de sincronização são necessárias em programas concorrentes de memória compartilhada: **exclusão mútua** e **condicional** [3].

Sincronização por exclusão mútua Visa garantir que *os trechos de código em cada thread que acessam objetos compartilhados não sejam executados ao mesmo tempo*, ou que uma vez iniciados sejam executados até o fim sem que outra thread inicie a execução do trecho equivalente. Essa restrição é necessária para lidar com a possibilidade de inconsistência dos valores das variáveis compartilhadas.

Como exemplo, considere o caso de uma operação que incrementa o valor de uma variável global (*ex., s++*). Para realizar o incremento, três instruções de nível mais baixo (linguagem de máquina) são normalmente necessárias: (i) ler o valor atual da variável; (ii) incrementar esse valor; e (iii) escrever o novo valor na variável global. Se duas threads executam essa operação concorrentemente, as duas poderão ler o mesmo valor inicial de *s* e realizar o incremento sobre esse valor. Ao final, embora duas operações de incremento tenham sido realizadas, o valor em *s* irá refletir apenas um incremento.

A solução para a **exclusão mútua** é definida agrupando *sequências contínuas de ações atômicas de hardware em seções críticas de software*. As **seções críticas** (trechos de código que acessam objetos compartilhados) devem ser transformadas em **ações atômicas**, de forma que a sua execução não possa ocorrer concorrentemente com outra seção crítica que referencia a mesma variável.

Sincronização por condição Visa garantir que **uma thread seja retardada enquanto uma determinada condição lógica da aplicação não for satisfeita**. Como exemplo, considere o problema *produtor/consumidor* (com threads que geram produtos e outras threads que consomem os produtos gerados). Sempre que o buffer de armazenamento dos itens criados estiver cheio, as threads que geram novos itens (produtores) devem ser retardadas para que não ocorra sobreposição de itens no buffer. Da mesma forma sempre que o buffer estiver vazio as threads que consomem os itens gerados (consumidores) devem ser retardadas para que não tentem consumir itens inválidos. A solução para a sincronização por condição é implementada retardando a execução de uma thread até que o estado da aplicação seja correto para a sua execução.

1.4. Mecanismos para sincronização entre threads

Existem diferentes mecanismos para implementar as duas formas de sincronização entre threads. Qualquer solução proposta deve prover a sincronização necessária para eliminar as **condições de corrida indesejáveis**, sem restringir as oportunidades de paralelismo na execução da aplicação. Há duas abordagens básicas para implementar a sincronização [4]:

1. sincronização por espera ocupada;

2. sincronização por escalonamento.

A **sincronização por espera ocupada** faz com que a thread fique continuamente testando o valor de uma determinada variável até que esse valor lhe permita executar a sua seção crítica com exclusividade ou continuar a sua execução. Para implementar esse mecanismo de sincronização é necessário dispor de instruções de máquina que permitam ler e escrever em localizações da memória de forma atômica.

O principal problema da solução por espera ocupada é que ela gasta ciclos de CPU enquanto espera autorização para seguir com o seu fluxo de execução normal. A "espera ocupada" só faz sentido nos seguintes casos:

- não há nada melhor para a CPU fazer enquanto espera;
- o tempo de espera é menor que o tempo requerido para a troca de contexto entre threads.

Os mecanismos de **sincronização por escalonamento** são a alternativa mais usual. As três formas mais comuns são: **semáforos**, **monitores** e **regiões críticas condicionais**. Nesse caso, quando uma thread não consegue entrar na seção crítica ela é bloqueada e a CPU pode ser cedida a outra thread, evitando assim o desperdício de ciclos de CPU.

2. Algoritmo de sincronização com espera ocupada

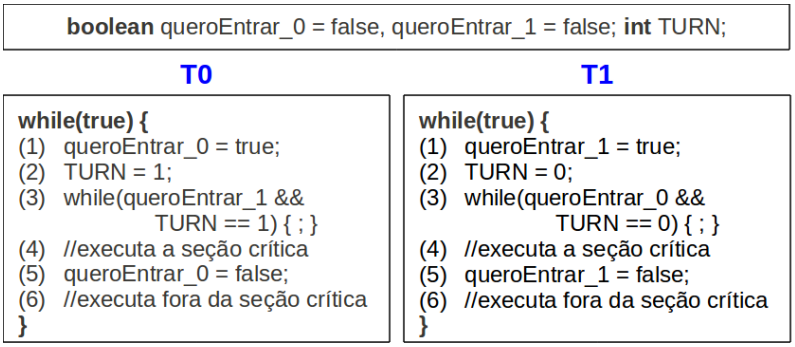
A Figura 2 mostra a solução de Peterson para o problema de sincronização por exclusão mútua com espera ocupada. O algoritmo atende a todos os requisitos para implementação das seções de entrada e saída da seção crítica (como discutido na seção 1.2).

Entretanto, a solução de Peterson pode não funcionar na presença de certas otimizações de compilação e de hardware [1]. Por exemplo, para otimizar a velocidade de execução do programa, o compilador pode permitir que cada thread mantenha cópias privadas das variáveis compartilhadas *queroEntrar_0* , *queroEntrar_1* e *turn*. Se essas otimizações são adotadas, alterações feitas nessas variáveis por uma thread serão feitas na cópia privada da variável e então não serão visíveis para a outra thread. Uma forma de solucionar esse problema (nas linguagens C e Java) é declarar as variáveis compartilhadas como `volatile`. O qualificador `volatile` informa que a variável pode ser alterada por outras linhas de execução, e por isso as opções de otimização devem ser usadas de forma diferenciada/reduzida (por exemplo, não manter cópias privadas da variável).

O uso de níveis dintintos de memória cache pelo hardware da máquina também pode fazer a solução de Peterson falhar. Mesmo com o uso do qualificador `volatile`, cópias temporárias da variável compartilhada podem ser mantidas em caches distintas, fazendo com que alterações no seu valor por uma thread não sejam vistas imediatamente por outras threads.

3. Exercícios

1. O que é *seção crítica* do código?
2. O que significa uma operação ser atômica?
3. Qual é a diferença entre sincronização por exclusão mútua e sincronização por condição?
4. O que é *espera ocupada*, qual sua principal desvantagem e em quais situações se aplica?



Referências

[1] Richard H. Carver and Kuo-Chung Tai. *Modern multithreading*. Wiley, 2006.

[2] Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Prentice-Hall, 2006.

[3] G. Andrews. *Concurrent Programming — Principles and Practice*. Addison-Wesley, 1991.

[4] M. L. Scott. *Programming Language Pragmatics*. Morgan-Kaufmann, 2 edition, 2006.