Programação Concorrente (ICP-361)
Cap. 4: Sincronização por condição

Prof. Silvana Rossetto

[1]Instituto de Computação (IC)
Universidade Federal do Rio de Janeiro (UFRJ)
Maio de 2025
silvana@ic.ufrj.br

1. Introdução

Como vimos, sincronização refere-se a qualquer mecanismo que permite ao programador controlar a ordem na qual as operações ocorrem em diferentes threads. Duas formas de sincronização são necessárias em programas concorrentes de memória compartilhada: **exclusão mútua** e **por condição** [1].

A **sincronização por condição** visa garantir que **uma thread seja retardada** enquanto uma determinada **condição lógica da aplicação** não for satisfeita. A solução para a sincronização por condição é implementada **bloqueando a execução de uma thread até que o estado da aplicação seja correto para a sua execução**. Em resumo, para coordenar a execução de um algoritmo concorrente é necessário que os fluxos de execução troquem informações entre si sobre seus estados de execução e possam suspender ou retomar suas execuções dependendo do estado de execução dos demais fluxos ou do estado global da aplicação.

2. Sincronização por condição com variáveis de condição

Uma forma de implementação de sincronização por condição é por meio de variáveis especiais — chamadas **variáveis de condição** — que permitem que as threads esperem (*bloqueando-se*) até que sejam sinalizadas (*avisadas*) por outra thread que a condição lógica foi atendida. Associado a essas variáveis de condição, temos normalmente três operações principais:

- **WAIT(condvar)**: bloqueia a thread na fila da variável de condição;
- **SIGNAL(condvar)**: desbloqueia uma thread na fila da variável de condição;
- **BROADCAST(condvar)**: desbloqueia todas as threads na fila da variável de condição.

Uma **variável de condição** é sempre usada em conjunto com uma **variável de lock**. A thread usa o **bloco de lock para checar a condição lógica da aplicação e decidir** por WAIT ou SIGNAL O lock é *implicitamente liberado* quando a thread é bloqueada e é *implicitamente devolvido* quando a thread retoma a execução do ponto de bloqueio. A Figura 1 ilustra o funcionamento das variáveis de condição.

Variáveis de condição na biblioteca PThread A biblioteca PThreads define um tipo especial chamado **pthread_cond_t** com as seguintes rotinas:

- **pthread_cond_wait (condvar, mutex)**: bloqueia a thread na condição (*condvar*) (deve ser chamada com *mutex* locado para a thread e depois de finalizado deve desalocar *mutex*);

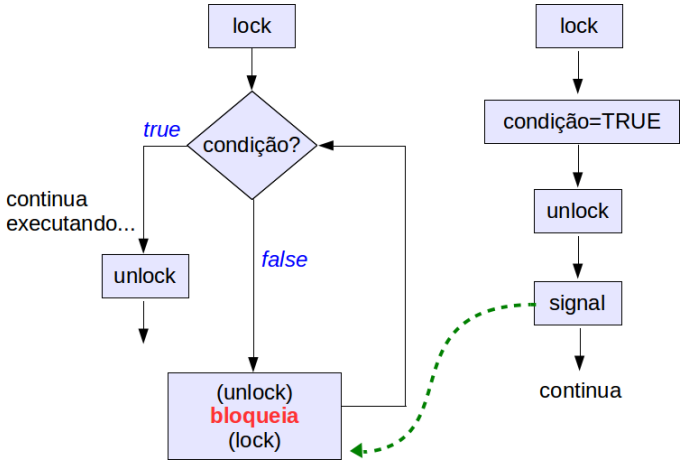

Figure 1. Um fluxo avalia o estado da aplicação, se a condição lógica estiver satisfeita, ele segue sem bloqueio, caso contrário, ele se bloqueia na fila da variável de condição. Um outro fluxo da aplicação deverá sinalizá-lo quando a condição lógica se tornar verdadeira.

- **pthread_cond_signal (condvar)**: desbloqueia uma thread esperando pela condição (*condvar*);
- **pthread_cond_broadcast (condvar)**: usado no lugar de SIGNAL quando todas as threads na fila da condição podem ser desbloqueadas;
- **pthread_cond_init (condvar, attr)**: inicializa a variável;
- **pthread_cond_destroy (condvar)**: libera a variável.

Sincronização por barreira Um tipo particular de sincronização por condição aparece em problemas que requerem que todos os fluxos de execução (ou um grupo deles) sincronizem suas ações em determinados pontos da execução do algoritmo. Essa sincronização é chamada de **sincronização por barreira** porque funciona exatamente como uma "barreira" (ou bloqueio coletivo) que faz com que todos os fluxos de execução aguardem pelos demais até que todos tenham chegado a esse ponto (ou passo) do algoritmo. A Figura 2 ilustra essa situação.

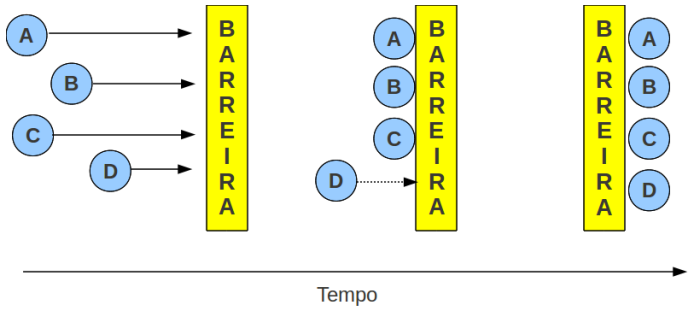

Figure 2. Ilustração do mecanismo de sincronização por barreira. Todos os fluxos de execução (A, B, C e D) devem chegar ao ponto da execução onde está a barreira para então poderem continuar suas execuções.

O código abaixo mostra uma possível implementação de uma função `barreira` que implementa a sincronização coletiva de um número dado de threads. O código faz uso de **lock** e **variável de condição**. Um **contador** (variável de escopo global que registra

o estado da aplicação, neste caso, quantas threads já chegaram na barreira) é inicializado com o valor 0. Cada thread (com exceção da última a chegar) incrementa o contador após alcançar a barreira e então **se bloqueia**. A última thread a chegar **reinicia o contador** e **desbloqueia as demais threads**.

```
/* Variaveis globais inicializadas na main */
pthread_mutex_t mutex;
pthread_cond_t cond;

//funcao barreira
void barreira(int nthreads) {
    static int bloqueadas = 0;
    pthread_mutex_lock(&mutex); //inicio secao critica
    if (bloqueadas == (nthreads-1)) {
      //ultima thread a chegar na barreira
      bloqueadas=0;
      pthread_cond_broadcast(&cond);
    } else {
      bloqueadas++;
      pthread_cond_wait(&cond, &mutex);
    }
    pthread_mutex_unlock(&mutex); //fim secao critica
}
```

Os métodos iterativos para solução de equações lineares são exemplos de problemas que quando paralelizados normalmente requerem o uso da sincronização por barreira. Esses problemas possuem passos ou etapas bem definidas onde ocorre uma verificação para decidir se o programa deve continuar executando ou se a solução encontrada até esse passo já é satisfatória. Quando paralelizamos esse tipo de problema, os pontos de verificação que antecedem uma nova etapa do algoritmo precisam ser precedidos por uma barreira para garantir que todos os fluxos de execução cheguem a esse ponto do código antes de realizar a próxima verificação, de forma que os resultados dos processamentos realizados na etapa atual sejam corretamente avaliados.

Tomemos o método iterativo de Jacobi como exemplo. Uma alternativa simples para particionar esse problema em subproblemas que podem ser executados ao mesmo tempo é atribuir para cada fluxo de execução o cálculo de uma das incógnitas do sistema. Como a cada passo é necessário garantir que todas as incógnitas foram calculadas para então iniciar o próximo passo, os fluxos de execução precisam aguardar o resultado dessa computação para então executarem a próxima iteração ou finalizarem.

Exercícios

1. O que é sincronização por condição?
2. Como funciona o mecanismo de sincronização por condição com variáveis condicionais na biblioteca PThread?
3. Por que o código da função `barreira` precisou fazer uso de uma variável de `lock`?

References

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