A sincronização refere-se a qualquer mecanismo que permite ao programador controlar a ordem relativa na qual as operações ocorrem em diferentes threads.

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

No contexto de computação concorrente, uma aplicação é constituída por mais de uma linha ou contexto de execução que podem executar simultaneamente. Quando duas ou mais linhas de execução (threads) acessam um mesmo recurso computacional ou uma mesma posição de memória (para leitura e escrita, ou escrita), surge a necessidade de usar mecanismos de sincronização entre as threads para ordenar o acesso ao recurso ou memória compartilhada — chamada sincronização por exclusão mútua. Se isso não for feito, o estado da aplicação pode ficar inconsistente ou a aplicação pode gerar resultados não esperados. Os mecanismos de sincronização por exclusão mútua permitem criar seções de código atômicas, i.e., trechos de código onde apenas uma thread da aplicação pode estar executando em cada instante de tempo, garantindo assim que a operação é completamente concluída antes que outra thread possa acessar o recurso ou espaço de memória compartilhado. A exclusão mútua pode ser implementada com “espera ocupada” (a thread permanece ocupando a CPU enquanto aguarda pela condição de acesso à seção atômica), ou com mecanismos de “bloqueio/reativação (nesse caso, a thread é bloqueada quando não pode acessar a seção atômica, e deve ser desbloqueada posteriormente quando a seção atômica é liberada). As soluções com espera ocupada podem levar as threads a situações de “livelock” (espera infinita por uma condição que não será atendida). Já as soluções com bloqueio/desbloqueio podem levar as threads a situações de “deadlock” (bloqueio indefinido). Além desses dois problemas, as soluções de sincronização podem causar ainda situações de “starvation” (as threads são seguidamente impedidas de acessar a seção atômica do código, retardando a sua execução).

**A sincronização por condição** visa garantir que uma thread seja retardada enquanto uma determinada condição lógica da aplicação não seja satisfeita. Como exemplo considere o problema produtor/consumidor. Sempre que o buffer de armazenamento dos itens criados estiver cheio, as threads que geram novos itens (produtores) devem ser retardadas para que não haja 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 definida retardando a execução de uma thread até que o estado da aplicação seja correto para a sua execução.

**Sincronização por exclusão VS Sincronização por Condição**

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. Sincronização por condição: visa garantir que uma thread seja retardada/bloqueada enquanto uma determinada condição ̃lógica da aplicação não for satisfeita.

**Seção crítica:** trechos de código em cada thread (linha de execução independente) que acessam objetos compartilhados.

**Condição de corrida** ocorre quando duas ou mais linhas de execução independentes “correm” para um ponto no código no qual ambas acessam um objeto comum, e o comportamento do sistema (ou aplicação que) depende de qual linha de execução chega primeiro. Para garantir o funcionamento correto é necessário sincronizar as ac ̧oes das linhas de execução, i.e., seguir passos explícitos para controlar a ordem na qual as suas ações ocorrem. Nem toda condição de corrida e “ruim”, i.e., algumas vezes qualquer saída do programa e aceitável. A sincronização é usada para resolver condições de corrida “ruins”: aquelas que fazem o programa produzir resultados incorretos.

**Lock Recursivo VS Semáforo Binário**

(i) Com semáforo binário, se duas chamadas para a operação P() são feitas sem uma chamada intermediária para a operação V(), a segunda chamada ir ̃a bloquear a thread. Com lock recursivo, se a thread que está de posse do lock o requisita novamente, essa thread não ̃é bloqueada. (ii) Com semáforo binário, chamadas sucessivas das operac ̧oes P() e V() podem ser feitas por diferentes threads. Com lock recursivo, chamadas sucessivas das operac ̧oes de lock() e unlock() devem ser feitas pela thread proprietária do lock.

**Deadlock** ocorre quando duas ou mais threads bloqueiam indefinidamente esperando um sinal de outra thread também bloqueada.

**Starvation** ocorre quando há uma contenção longa para a execução de um determinado trecho de código.

**Livelock** ocorre quando uma thread fica em espera ocupada indefinidamente.

**Monitor:** construção de Linguagem de Programação que provê funcionalidade equivalente a semáforos e pode ser definido como um módulo de software que consiste de um ou mais procedimentos, uma sequência de inicialização, variáveis de condição e estado interno. Uma das características que distingue monitores de outras estruturas de programação é que apenas uma operação interna pode estar ativa a cada instante de tempo. Se uma thread chama uma operação do monitor e ele encontra-se ocupado, a thread é automaticamente bloqueada até o monitor ficar livre. Essa característica permite implementar a sincronização por exclusão mútua. Outra característica particular de monitores é que qualquer operação pode suspender a si mesma colocando-se em espera por uma variável de condição. Essa característica permite implementar a sincronização por condição. Uma operação pode também sinalizar uma variável de condição fazendo com que uma thread bloqueada nessa variável seja “acordada” (desbloqueada).