**Atividades Concorrentes**

Execuções que ocorrem “simultaneamente” ou de maneira sobreposta no tempo => multithreading.

**Multithreading**

Conjunto de threads a serem executadas de forma intercalada/em paralelo, podendo compartilhar recursos e o mesmo CPU ou diferentes CPU (se for multicore)

**Vantagens**

* Melhor desempenho
* Melhor utilização dos recursos do hardware (CPU)
* Cuidado com corridas e deadlocks

**Exemplo**

Se duas threads tentam incrementar um contador sem coordenação, no final possivelmente vamos obter um valor incorreto.

* É preciso sincronização
* É preciso garantir exclusão mutua (usar locks e unlocks) nas secções críticas

**Corridas na leitura:**

Quando uma thread tenta ler um uma variável enquanto outra thread está a alterar esse mesmo valor.

* Usar locks para leitura-escrita

**Espera ativa**

Quando uma thread aguarda por um evento (como a libertação de um lock) executanto constantemente um loop de verificação ao invés de se bloquear.

**Desvantagens:**

* Desperdício de ciclos do CPU

**Exclusão Mutua**

Garante que uma e apenas uma thread acessam a uma secção crítica de cada vez.

**Secção crítica:** zona onde decorre acesso e/ou leitura de varíaveis.

**Vantagens**:

* Integridade de dados
* Evita corridas

**Desvantagens:**

* Pode reduzir o paralelismo

**Lock**

Mecanismo que assegura a exclusão mutua, funciona como a espécie de um cadeado, ou seja, uma thread está a executar e mal é dado um lock, mais nenhuma thread pode passar daí a não ser aquela que lhe foi permitida o acesso à secção crítica. Após a sua saída é dado um unlock para que uma outra thread possa executar esse mesmo excerto de código.

**Sintaxe:**

Mutex.lock();

Mutex.unlock();

**Algoritmo tipo**

Lock();

Try {

...

}

Finally {

Unlock();

}

* Threads podem ficar indefinidamente à espera para poderem entrar na secção crítica caso não seja utilizado um unlock por parte da ultima thread que entrou nesta mesma secção.

**Multiple locks**

Para aumentar a eficiencia do nosso programa na situação em que temos várias threads para desempenharem diversos papeis, devemos dar lock de todas as threads antes do try e unlock de todas no finally.

**Sincronização de Relógios:**

Em sistemas distribuídos, cada nó mantém o seu próprio relógio. Devido a diferenças de hardware e latências de rede, estes relógios acabam por se desencontrar. A sincronização de relógios é o processo de alinhar o tempo lógico ou físico entre os diversos nós. Protocolos como NTP (Network Time Protocol) alinham relógios físicos, enquanto os relógios lógicos (Lamport, Vector Clocks) asseguram uma noção de ordem parcial dos eventos, fundamental para raciocinar sobre a causalidade sem depender de tempo físico.

**Exclusão Mútua Distribuída Simples:**

A exclusão mútua distribuída garante que apenas um processo aceda a um recurso crítico de cada vez. Abordagens simples incluem o algoritmo de Lamport, em que processos solicitam acesso a um recurso enviando mensagens a todos os outros, ordenando os pedidos através de timestamps lógicos. Estes métodos geralmente assumem redes estáveis, sem grande preocupação com o tempo real, sendo baseados em comunicação fiável e no consenso entre todos os participantes.

**Exclusão Mútua Distribuída com Tempo Real:**

Nesta abordagem, não basta apenas a ordem dos pedidos – a dimensão temporal real é considerada. Os processos necessitam sincronizar os seus relógios físicos e garantir limites máximos de atraso para acesso ao recurso. Impõe-se requisitos temporais rigorosos, como em sistemas de controlo industrial, onde o acesso a um recurso partilhado deve ser estritamente temporizado para garantir corretude e segurança.

**Causalidade e Relógios Lógicos:**

Causalidade refere-se ao encadeamento de eventos em que um evento influencia outro. Num sistema distribuído, é difícil estabelecer se um evento ocorreu antes de outro sem um relógio global. Os relógios lógicos (como os relógios de Lamport e vetoriais) oferecem uma forma de capturar a relação causal entre eventos sem sincronização física. Se o timestamp lógico de um evento A for menor que o de um evento B, é possível determinar se há uma relação causal (A potencialmente causou B).

**Difusão Fiável (Reliable Broadcast):**

Difusão fiável garante que, se um processo envia uma mensagem a um conjunto de destinatários, ou todos a recebem ou nenhum a recebe, mesmo na presença de falhas (excepto falhas bizantinas). Mecanismos de ACKs e retransmissão garantem que a mensagem é entregue a todos os processos corretos.

**Difusão Epidémica (Gossip):**

Na difusão epidémica, a informação propaga-se entre nós de forma probabilística, em modo “boato”: cada nó contacta periodicamente outros nós aleatórios, partilhando o que sabe. Ao longo do tempo, a informação dissemina-se por toda a rede, de forma robusta e escalável, embora sem garantias determinísticas de entrega simultânea.

**Transações Distribuídas (2PC e TRPC):**

Transações distribuídas envolvem operações atómicas em múltiplos nós. O protocolo 2PC (Two-Phase Commit) coordena commits e rollbacks, assegurando atomicidade: ou todos confirmam a transação (commit) ou todos a anulam (rollback). Já o TRPC (Three-Phase Commit) adiciona uma terceira fase para melhorar a resiliência a falhas, tentando evitar bloqueios definitivos em caso de quedas de coordenadores, reduzindo a possibilidade de impasses. Este tipo de protocolos é essencial para manter a consistência global de dados distribuídos.

**Algoritmo de Peterson:**

O algoritmo de Peterson é um dos primeiros e mais simples protocolos de exclusão mútua a dois processos. Ele garante que, se dois processos querem entrar na sua seção crítica, um deles acabará por entrar primeiro de forma justa, evitando tanto deadlocks quanto esperas ativas desnecessárias. O algoritmo baseia-se em duas variáveis de estado (um array para indicar que um processo está interessado em entrar, e uma variável que indica a preferência sobre quem entra primeiro) e uma lógica simples para assegurar que apenas um processo entre na seção crítica de cada vez, sem favorecimentos permanentes.

Considere dois processos, P0 e P1, que querem aceder a uma seção crítica. Suponha que ambos começam ao mesmo tempo:

• Cada um define a sua variável “querEntrar” para indicar que quer entrar na seção crítica e ajusta a variável “turn” (prioridade) para si próprio.

• Antes de entrar na seção crítica, cada um verifica se o outro processo também quer entrar e se possui a vez (turn).

• Se o outro processo tiver prioridade, o processo em questão espera.

• Quando o processo com a prioridade terminar, libera a seção crítica, permitindo que o outro entre.

**Cenário:**

1. P0 quer entrar: querEntrar[0] = true; turn = 0;

2. P1 quer entrar quase ao mesmo tempo: querEntrar[1] = true; turn = 1;

3. Agora, P0 vê que querEntrar[1] é true e que turn = 1, logo P0 espera.

4. P1 vê que querEntrar[0] é true, mas também vê que turn = 1 (P1 tem prioridade), então P1 avança e entra na seção crítica.

5. Após P1 sair da seção crítica, querEntrar[1] volta a false, permitindo que P0 veja que já pode avançar, pois já não há concorrência a bloqueá-lo.

Neste exemplo, ambos os processos esperam respeitosamente pela sua vez, garantindo exclusão mútua, não havendo condições de corrida nem deadlocks.

**Algoritmo da Padaria (Bakery Algorithm) de Lamport:**

Inspirado no sistema de senhas de uma padaria (onde cada cliente recebe um número de ingresso para ser atendido por ordem), o Bakery Algorithm estende a ideia de Peterson para N processos. Cada processo, ao querer entrar na seção crítica, obtém um número maior que todos os já obtidos anteriormente. Assim, os processos são ordenados pelo valor do seu “ticket” e, em caso de empates, pelo seu identificador. O resultado é que todos os processos obtêm acesso à seção crítica por uma ordem globalmente consistente, garantindo exclusão mútua, ausência de deadlock e justiça, pois nenhum processo é constantemente ultrapassado pelo mesmo concorrente.

Imagine que temos 3 processos: P0, P1 e P2, que querem acessar a seção crítica.

• Quando um processo quer entrar, ele tira uma espécie de “ticket”. Este número de ticket é sempre maior do que qualquer número que já tenha usado antes, garantindo uma ordenação global.

• Antes de entrar na seção crítica, o processo verifica os tickets dos outros processos. Ele só entra quando o seu ticket for o mais baixo ou se tiver precedência (caso haja empate no número do ticket, desempata-se pelo ID do processo).

**Cenário:**

1. P0 quer entrar. Não há tickets emitidos ainda, então P0 obtém o ticket nº 1.

2. P1 quer entrar depois: verifica o ticket de P0 (1), obtém um ticket maior, por exemplo, o nº 2.

3. P2 quer entrar ainda depois: agora há tickets 1 e 2; P2 obtém o ticket nº 3.

A ordem lógica de entrada é P0 (ticket 1), depois P1 (ticket 2), depois P2 (ticket 3).

• P0 entra na seção crítica primeiro porque tem o menor número.

• Assim que P0 terminar, libera o seu ticket (pode voltar a zero ou tomar um valor que signifique que não quer mais entrar).

• Agora, P1 vê que P0 já saiu. O ticket de P1 (2) é agora o menor dentre os processos interessados, então P1 entra.

• Por fim, quando P1 terminar, P2 acede à seção crítica.

Este exemplo mostra que, independentemente da ordem real de chegada dos pedidos, a ordem de atendimento é determinada pelos tickets, garantindo que nenhum processo é continuamente ultrapassado pelo mesmo concorrente (justiça), mantendo exclusão mútua e evitando deadlocks.