REAL BETIS

Daniel Nogueira
Felipe Moreira
João Pedro Silva Franco
Lucas Bernardes

- Perguntas

2 - A figura demonstra dois processos que estão rodando simultaneamente em um sistema e que em determinado momento tentam acessar o mesmo recurso compartilhado. Para que não ocorra uma inconsistência de dados ou que se inicie uma condição de corrida, o sistema inicia o processo de exclusão mútua. No caso da imagem, em T1, o processo A entra na região crítica, e começa a realizar sua atividade. Contudo, B, em T2, tenta entrar na região crítica antes de A terminar o processamento e fica bloqueado esperando o processo A finalizar. Assim que o processo A não estiver mais acessando o recurso, em T3, o processo B pode entrar na região crítica e executar seu trabalho, que finaliza em T4.

Exclusão mútua: É o conceito de garantir que apenas um processo ou thread acesse uma região de recurso compartilhado (região crítica) por vez;
Inconsistência de dados: Ocorre quando um processo lê informações que já foram modificadas por outro processo, porém essas modificações ainda não foram refletidas no sistema. Quando o primeiro processo modifica a informação novamente ele não leva em consideração a outra alteração e pode deixar os dados incorretos;
Região crítica: Área do código que acessa um recurso que não deve ser acessado concorrentemente;
Processo: É uma instância em execução do programa rodando no computador.

3 - Prós: A solução funciona nos casos felizes e, de forma simples, executa seu objetivo de implementar a exclusão mútua.

Contras:

Consumo Excessivo de Recursos: A espera ocupada geralmente envolve ciclos de CPU ativos, nos quais a thread verifica repetidamente uma condição. Isso pode resultar em um consumo excessivo de recursos de processamento, especialmente em sistemas com muitas threads ou em sistemas com recursos limitados.

Potencial para Deadlocks: Se não for cuidadosamente implementada, a espera ocupada pode aumentar o risco de deadlocks, fazendo com que duas ou mais threads fiquem em espera indefinidamente.

Starvation: Se um processo nunca sair da região crítica, outro processo pode ficar esperando indefinidamente e não executando nunca seu trabalho por não ter acesso à região crítica

Outras Alternativas:

Semáforos: Mecanismo de sincronização que permite controlar o acesso concorrente a recursos compartilhados. Eles podem ser usados para implementar a sincronização entre threads, permitindo que as threads aguardem a liberação de recursos antes de prosseguir.

Monitores: Abstração de alto nível que combina dados e operações em uma única unidade. Eles são úteis para implementar exclusão mútua e sincronização entre threads, garantindo que apenas uma thread possa acessar o monitor por vez.

4 -

Processo:

- Um processo é uma instância de um programa em execução
- Cada processo possui seu próprio espaço de memória
- Os processos são independentes uns dos outros, o que significa que cada um é isolado e protegido dos outros

Thread:

- Uma thread é uma unidade de execução dentro de um processo. Um processo pode ter uma ou várias threads
- As threads compartilham o mesmo espaço de memória do processo ao qual pertencem
- As threads normalmente compartilham recursos do processo pai

5 - Safety: Garante que o sistema nunca entre em um estado inconsistente ou incorreto, ou seja, algo ruim nunca aconteceria. Um exemplo bem conhecido seria: em um sistema bancário, uma propriedade de safety seria que o saldo de uma conta nunca deve se tornar negativo devido a transações concorrentes.

Liveness: Garante que o sistema em algum momento irá entrar em um estado desejado ou terá sua tarefa completada. Por exemplo, nesse caso, em um sistema de filas, uma propriedade de liveness seria que todas as solicitações eventualmente seriam atendidas.

6 -

a) Sim, deve ser tratada para garantir que a interface permaneça responsiva durante o download, o processo responsável por ele deve ser executado em uma thread separada, enquanto a thread principal deve atualizar a tela com o progresso do download sem bloquear a interface.
b) Sim, para que seja possível atender múltiplas requisições simultâneas de maneira eficiente, o servidor pode utilizar técnicas de concorrência, como múltiplas threads ou processos. Isso permite que o servidor responda a várias requisições ao mesmo tempo, maximizando a performance.
c) Deve sim ser tratada, um exemplo desse caso são os sistemas operacionais para dispositivos móveis, pois realizar operações de rede na thread principal pode bloquear a interface do usuário, tornando a aplicação não responsiva. Portanto, é

uma prática comum fazer essas requisições em threads separadas para garantir que a UI permaneça fluida e responsiva enquanto as operações de rede são executadas em paralelo.

7 -

a) **Resposta correta!**
b) Este é o conceito de exclusão mútua.
c) Não tem nada a ver com concorrência. A concorrência pode existir entre processos diferentes e completamente independentes desde que eles estejam tentando utilizar um recurso compartilhado.
d) Coisas em paralelo não necessariamente são concorrentes, dois processos podem rodar em paralelo (executar ao mesmo tempo) mas não serem concorrentes, por não disputarem um recurso

8 -

a) Programação concorrente tem seu custo, como inicializar uma thread, a gestão de recursos, como o tempo de processamento que cada tarefa vai ter, timeslicing. Dessa forma, uma implementação de forma concorrente, que se comporta como uma implementação single thread tem uma pior performance que uma implementação de fato single thread, por exemplo.
b) Muda, o projeto da solução tem que ser feito considerando os problemas de concorrência, disputa de recurso.
c) Concorrência envolve diversos conceitos específicos do domínio e problemas específicos do domínio também, considerando que fazer softwares de um mínimo de requisitos de boa qualidade

9 -

a) Analisando a chamada do método por duas threads simultaneamente podemos definir que:
    i) Situação 1: X = 10 -> Ambas as threads leem *lastIdUsed* como 10, incrementam *lastIdUsed* para 11 e retornam 11.
    ii) Situação 2: X = 11 -> Ambas as threads leem *lastIdUsed* como 11, incrementam *lastIdUsed* para 12 e retornam 12.
    iii) Situação 3: X = 12 -> Ambas as threads leem *lastIdUsed* como 12, incrementam *lastIdUsed* para 13 e retornam 13.

b) Olhando o código, a única alteração seria adicionar a sincronização no método *getNextId,* para que apenas 1 thread possa executar esse método por vez. Resultado:

```
public class X {
  private int lastIdUsed;
  public synchronized int getNextId() {
        return ++lastIdUsed;
  }
}
```

i) Situação 1: X = 10
   1) Thread A lê *lastIdUsed* como 10, incrementa para 11 e retorna 11.
   2) Thread B lê *lastIdUsed* como 11, incrementa para 12 e retorna 12.

ii) Situação 2: X = 11
   1) Thread A lê *lastIdUsed* como 11, incrementa para 12 e retorna 12.
   2) Thread B lê *lastIdUsed* como 12, incrementa para 13 e retorna 13.

iii) Situação 3: X = 12
   1) Thread A lê *lastIdUsed* como 12, incrementa para 13 e retorna 13.
   2) Thread B lê *lastIdUsed* como 13, incrementa para 14 e retorna 14.