Artistic Trivia

Filipe Paco
Alany Gabriely
André Lopes
Cauã dos Santos Rebelo
Artur Rocha Lapot
Tayna Crisllen*
Bruno Milioli

Questões Parte 1

a)

Questões Parte 2

2 - A figura demonstra o funcionamento do uso de *Threads* usando a exclusão mútua. As *threads* tentam acessar a região crítica, que é onde ela pode ser processada mas somente uma quantidade limitada pode ser usada ao mesmo tempo, por isso o nome região crítica.
No momento T1, o processo A acessa a região crítica. No momento T2, durante o acesso da região crítica do processo A, o processo B tenta acessar a região crítica, porém como a região crítica só pode tratar 1 processo por vez, o processo B é bloqueado. No momento T3, o processo A termina de usar a região crítica e libera seu acesso, nesse momento o processo B entra na região crítica. E por último, no momento T4, o processo B termina de usar a região crítica e libera ela. O processo todo acontece com o tempo e é dividido em 4 eventos (T1, T2, T3, T4), demonstrado na parte inferior da figura.

3 - Esse código representa uma solução para a exclusão mútua baseada em espera ocupada. Essa solução é uma solução clássica para o problema da região crítica. Em um cenário onde exista uma região crítica no sistema, onde somente um processo possa entrar por vez para evitar inconsistências no sistema, a solução da exclusão mútua (representada no código) resolve esse problema da seguinte maneira: é definida uma variável de “turno” (*turn*, no código representado), cada processo fica “travado” em espera ocupada enquanto o turno não é deles (representado pelo *while* no código, o processo não sai desse *while* enquanto não for

o seu turno), quando for o seu turno, o processo entra na região crítica, faz oque quer que tenha que ser feito nela e ao sair da região crítica define um novo valor na variável de turno para representar que o turno é agora do outro processo. Esse processo todo fica se repetindo infinitamente até os processos acabarem. O problema dessa solução é que cada processo está “ocupado” enquanto espera, por mais leve que possa ser, esse *while* que fica rodando enquanto não é o turno do processo consome processamento da máquina, em um cenário com muitos processos todos esses processos consumindo poder da máquina podem causar lentidão na mesma, e isso é um recurso sendo gasto desnecessariamente pois, dentro desse *while* não é feito nada , é usado somente para travar o processo no lugar. Uma solução que também resolve o problema da região crítica sem gastar esse recurso mantendo os processos fazendo algo infinitamente para esperar é a solução de “*Sleep* e *Wakeup*”, nessa solução, enquanto um processo precisar entrar na região crítica mas a mesma está bloqueada, o processo “dorme” , ou seja, ele é suspenso até que receba um sinal para “acordar”, após o processo que estava na região crítica sair, ele manda um sinal de “*wakeup*” para o processo que está suspenso para o mesmo voltar a rodar e entrar na região crítica. Nessa solução os processos não ficam consumindo poder de processamento da máquina enquanto esperam.

4 - (Colocar em termos de programação concorrente) Enquanto *threads* funcionam como uma sequência de instruções e disputam um recurso no mesmo espaço de memória de um processo, processos são a execução de um programa de um computador através de um conjunto de threads e não disputam o mesmo espaço de memória com outros processos.

5 - O *Safety* é o conceito que um programa nunca chegará num estado inconsciente, o que é muito importante em programação concorrente, pois o mesmo processo não pode acessar a região crítica enquanto ela está sendo usada por outro processo sem criar inconsistências, já o *Liveness* quer dizer que eventualmente um programa chegará num estado de consistência, o que quer dizer que eventualmente todos os processos vão chegar num consenso ao utilizar o processador e chegar na resposta consistente. Um exemplo do uso dos dois seria de um banco, um usuário com 50 reais na conta tenta sacar 30 reais, e logo após deposita 10 reais, vamos imaginar

que esses processo demoraram para ser processados e não houve uma boa *Safety* no programa, e o processo de sacar foi feito mas demorou para debitar da conta, mas o depósito tentou acessar o total da conta ao mesmo tempo e substituiu o débito que teria sido feito pelo processo de sacar, o que resultou com o usuário ficando com 60 reais no final, pois o ato de depositar simplesmente incrementou o total da conta, antes do processo de sacar ter debitado no processador.Usando o mesmo exemplo só que para *Liveness* seria se isso funcionasse corretamente, o processo de sacar tirou 30 reais e antes de sair debitado da conta total 30 reais, e ficou com 20 reais na conta, logo depois o outro processo depositou 10 e somou do novo valor total dado pelo outro processo, ficando com 30 reais no total, isso quer dizer que eventualmente o resultado dos dois ficou consistente, mesmo tendo sido processado em momentos diferentes.

6 - a) Sim, há a necessidade pois há atualização do arquivo sendo baixado em seus metadados enquanto há também a consulta do navegador do status de *download*, sendo assim o navegador e o processo que está baixando.

b) Sim, há concorrência pois um mesmo arquivo está sendo baixado por diversos usuários diferentes, ou seja, um recurso está sendo disputado pelas máquinas desses usuários quando há a leitura desse arquivo, pois todos querem ler o arquivo.

c) Há a necessidade de concorrência pois com a necessidade de edição de dados não se pode permitir que mais de uma thread tente atualizar o mesmo recurso ao mesmo tempo.

7) b. Apenas a B está correta pois se há um processo consumindo um recurso e um aguardando para poder acessá-lo, isso significa que ao menos nesse momento há disputa entre dois processos por um recurso escasso.

8) A. A concorrência se mal aplicada pode implicar em um consumo excessivo e até mesmo travamento da memória caso um processo demore para liberar ou não libere o recurso que está consumindo.

B. O projeto de software (Documento que será como um guia para o programador na hora de implementar) muda quando se opta por utilizar concorrência pois será necessário o estudo e utilização de técnicas de concorrência pelo programador.

C. A concorrência(Disputa por recursos escassos) não é simples de conseguir pois a chance de dois processos disputarem EXATAMENTE ao mesmo tempo por um mesmo recurso é uma situação um tanto difícil de ocorrer(Algo que diminui quando aumentamos os processos e diminuímos os recursos) entretanto, é importante implementar técnicas de concorrência pois em algum momento irá ocorrer.

9) A- Os resultados possíveis obtidos pelas threads ao invocar o método getNextId() dependem da ordem em que as threads acessam e modificam o valor de lastIdUsed. Em todas as situações, cada thread obterá um valor exclusivo de ID, pois a operação de incremento é atômica (não divisível). No entanto, a ordem exata em que essas IDs são atribuídas pode variar dependendo da concorrência entre as threads.

Situação 1: Valor inicial de lastIdUsed é 10

Se o valor inicial de lastIdUsed é 10, isso significa que a próxima ID retornada será 11.

Possíveis Resultados:

1-Thread A invoca getNextId() primeiro, seguida pela Thread B:

A Thread A incrementa lastIdUsed de 10 para 11 e retorna 11. Em seguida, a Thread B incrementa lastIdUsed de 11 para 12 e retorna 12.

2-Thread B invoca getNextId() primeiro, seguida pela Thread A:

A Thread B incrementa lastIdUsed de 10 para 11 e retorna 11. Em seguida, a Thread A incrementa lastIdUsed de 11 para 12 e retorna 12.

Situação 2: Valor inicial de lastIdUsed é 11

Se o valor inicial de lastIdUsed é 11, isso significa que a próxima ID retornada será 12.

Possíveis Resultados:

1-Thread A invoca getNextId() primeiro, seguida pela Thread B:

A Thread A incrementa lastIdUsed de 11 para 12 e retorna 12. Logo, a Thread B incrementa lastIdUsed de 12 para 13 e retorna 13.

2-Thread B invoca getNextId() primeiro, seguida pela Thread A:

A Thread B incrementa lastIdUsed de 11 para 12 e retorna 12. Depois, a Thread A incrementa lastIdUsed de 12 para 13 e retorna 13.

Situação 3: Valor inicial de lastIdUsed é 12

Se o valor inicial de lastIdUsed é 12, isso significa que a próxima ID retornada será 13.

Possíveis Resultados:

1-Thread A invoca getNextId() primeiro, seguida pela Thread B:

A Thread A incrementa lastIdUsed de 12 para 13 e retorna 13. Em seguida, a Thread B incrementa lastIdUsed de 13 para 14 e retorna 14.

2- Thread B invoca getNextId() primeiro, seguida pela Thread A:

A Thread B incrementa lastIdUsed de 12 para 13 e retorna 13. Depois, a Thread A incrementa lastIdUsed de 13 para 14 e retorna 14.

9) B- Uma solução para o problema seria usar mecanismos de sincronização, como o synchronized em Java. Isso garantirá que apenas uma thread por vez possa executar o método getNextId().

```
public class X {

        private int lastIdUsed;
```

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

Situação 1: Valor inicial de lastIdUsed é 10

Thread A invoca getNextId(): lastIdUsed passa de 10 para 11.

Thread B invoca getNextId(): lastIdUsed passa de 11 para 12.

Situação 2: Valor inicial de lastIdUsed é 11

Thread A invoca getNextId(): lastIdUsed passa de 11 para 12.

Thread B invoca getNextId(): lastIdUsed passa de 12 para 13.

Situação 3: Valor inicial de lastIdUsed é 12

Thread A invoca getNextId(): lastIdUsed passa de 12 para 13.

Thread B invoca getNextId(): lastIdUsed passa de 13 para 14.

2024-06-07-Atividade-SDC-ES.pdf