Exemplo de Concorrência

Grupo Art Attack

Contextualização do Problema

No arquivo *"XNaoSincronizado.java"*, temos o método `getNextId`, descrito abaixo, que é responsável por retornar o próximo ID disponível, e duas threads simulando o uso compartilhado do objeto `X` e chamadas concorrentes ao método `getNextId`.

No entanto, como não há sincronização neste método, é possível que duas threads acessem e atualizem o valor do ID ao mesmo tempo. Isso pode resultar em uma condição de corrida, onde ambas as threads competem para atualizar o ID, levando a resultados imprevisíveis e possivelmente inconsistentes.

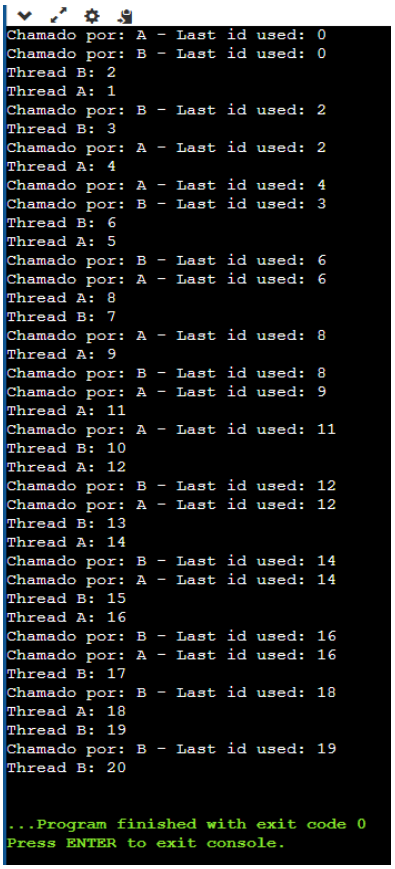
Durante a execução do programa, as threads podem intercalar suas execuções de forma imprevisível, o que pode levar a uma mistura de valores durante a execução. Por exemplo, uma thread pode ler um valor desatualizado de `lastIdUsed`, enquanto outra thread já o atualizou, levando a resultados inesperados e possivelmente incorretos.

```java
class X {
    private int lastIdUsed;

    public int getNextId(String t) {
        System.out.println("Chamado por: " + t + " - Last id used: " + lastIdUsed);
        return ++lastIdUsed;
    }
}
```

Obs: Foi acrescentado o parâmetro `'t'` para representar a thread que está fazendo a chamada ao método.

Exemplo de saída da execução do método:



A análise da saída fornecida revela um padrão de comportamento interessante entre as threads A e B ao chamarem o método `getNextId()` simultaneamente. Vamos detalhar essa análise:

1. No início, as threads chamam o método `getNextId()` simultaneamente. Como nenhuma outra thread chamou o método ainda, ambas veem `lastIdUsed` como 0.
2. A Thread A recebe o ID 1, e o `lastIdUsed` é atualizado para 1 antes que a Thread B possa obtê-lo.
3. Assim, a Thread B obtém o ID 2, pois `lastIdUsed` já foi atualizado pela Thread A.
4. Quando a Thread A chama `getNextId()` novamente, `lastIdUsed` é 2 (atualizado pela chamada anterior da Thread B). Entretanto, a Thread B recebe o ID 3 e a Thread A recebe o ID 4, pois a chamada da Thread B foi realizada em concorrência com a Thread A, e a atualização de `lastIdUsed` ocorreu após a obtenção do ID pela Thread B.
5. A Thread B chama `getNextId()` novamente e recebe o ID 6, pois a Thread A já havia recebido os IDs 4 e 5 antes de B concluir a chamada.

Logo, mesmo que o valor de `lastIdUsed` seja compartilhado entre as threads, elas não estão sincronizadas, o que leva a resultados imprevisíveis. A saída mostra que as threads não esperam uma pela outra, resultando em valores inconsistentes para `lastIdUsed` e IDs atribuídos.

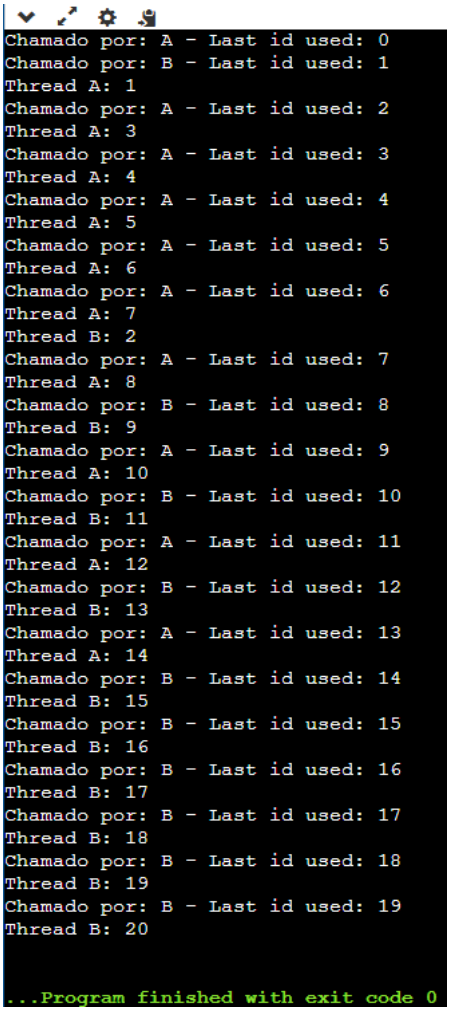
Possível solução em Java

```
9  class X {
10     private int lastIdUsed;
11
12     // Método synchronized para garantir acesso exclusivo
13     public synchronized int getNextId(String t) {
14         System.out.println("Chamado por: " + t + " - Last id used: " + lastIdUsed);
15         return ++lastIdUsed;
16     }
17 }
```

Ao modificar o método `getNextId()` para ser sincronizado, utilizando a palavra-chave `synchronized`, garantimos que apenas uma thread por vez poderá executar esse método em uma instância específica de `X`. Isso significa que quando uma thread estiver executando o método `getNextId()`, todas as outras threads que tentarem acessá-lo serão bloqueadas até que a primeira thread termine sua execução.

Com essa modificação, o código garante que os resultados serão consistentes e que as threads não entrarão em conflito ao acessar o método `getNextId()`. Isso elimina a possibilidade de condições de corrida, onde duas ou mais threads competem para atualizar o mesmo recurso compartilhado simultaneamente, resultando em resultados imprevisíveis e possivelmente incorretos.

Exemplo de saída da execução do método, que pode ser encontrado no arquivo "*XSincronizado.java*":



Analisando essa saída, podemos perceber que, embora a saída dos IDs retornados pelas threads possa estar fora de ordem, o valor do `lastIdUsed` está sempre atualizado e não há discrepância entre o ID retornado e o valor atualizado do `lastIdUsed`.

Por exemplo, quando a Thread B é chamada pela primeira vez, o `lastIdUsed` é 1 e o ID retornado é 2, o que indica que a atualização do `lastIdUsed` ocorreu corretamente. Da mesma forma, em todas as outras chamadas subsequentes do método `getNextId()`, o valor do `lastIdUsed` corresponde ao ID retornado pela respectiva thread.

Isso confirma que a sincronização adequada do método `getNextId()` usando `synchronized` garante que as atualizações do `lastIdUsed` sejam consistentes e que não ocorram condições de corrida ou valores desatualizados.

Ainda, podemos chamar o método `getNextId()` dentro de um bloco `synchronized`, garantindo que apenas uma thread por vez possa executar este bloco de código. Isso evita que as threads entrem em conflito ao acessar e atualizar o valor do `lastIdUsed`.

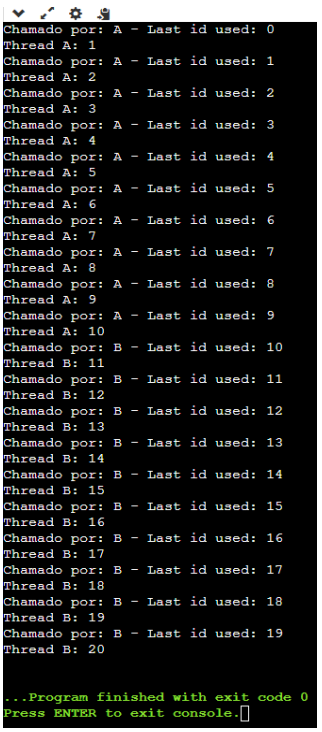
Trecho do código:

```java
Thread threadA = new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            synchronized (obj){
                int nextId = obj.getNextId("A");
                System.out.println("Thread A: " + nextId);
            }
        }
    }
});

// Criando e iniciando a segunda thread
Thread threadB = new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            synchronized (obj){
                int nextId = obj.getNextId("B");
                System.out.println("Thread B: " + nextId);
            }
        }
    }
});
```

Quando uma thread executa o bloco `synchronized`, ela adquire o bloqueio para o objeto `obj`, o que significa que outras threads que tentarem executar o mesmo bloco serão bloqueadas até que o bloqueio seja liberado pela thread atual.

Exemplo de saída:



Possível solução em Python

Na solução em Python, presente no arquivo "*main.py*", usamos o conceito de Lock para garantir um acesso seguro e evitar condições de corrida no método `getNextId`. O Lock atua como um mecanismo de exclusão mútua para garantir a sincronização entre threads que acessam o método.

Dessa forma, o método é encapsulado em uma instrução `with lock:` para garantir que apenas uma thread execute este bloco de código por vez. Essa abordagem assegura a consistência e integridade dos dados em um ambiente de execução concorrente.

Exemplo de saída:

```
Chamado por: A - Last id used: 0
Thread A: 1
Chamado por: B - Last id used: 1
Thread B: 2
Chamado por: A - Last id used: 2
Thread A: 3
Chamado por: B - Last id used: 3
Thread B: 4
Chamado por: A - Last id used: 4
Thread A: 5
Chamado por: B - Last id used: 5
Thread B: 6
Chamado por: A - Last id used: 6
Thread A: 7
Chamado por: B - Last id used: 7
Thread B: 8
Chamado por: A - Last id used: 8
Thread A: 9
Chamado por: A - Last id used: 9
Thread A: 10
Chamado por: A - Last id used: 10
Thread A: 11
Chamado por: A - Last id used: 11
Thread A: 12
Chamado por: B - Last id used: 12
Thread B: 13
Chamado por: A - Last id used: 13
Thread A: 14
Chamado por: B - Last id used: 14
Thread B: 15
Chamado por: A - Last id used: 15
Thread A: 16
Chamado por: B - Last id used: 16
Thread B: 17
Chamado por: B - Last id used: 17
Thread B: 18
Chamado por: B - Last id used: 18
Thread B: 19
Chamado por: B - Last id used: 19
Thread B: 20


...Program finished with exit code 0
Press ENTER to exit console.
```