# Programação_Concorrente-Studies/ Aula12: ProdutoXConsumidor explicando condições de corrida, DeadLocks, Mutex Exclusion, e usando algumas ferramentas da linguagem para resolver essa problemática

- Primeiramente vamos abordar sobre as condições de corrida e deadlock por meio de um exemplo incorreto de uma execução de um problema de ProdutoXConsumidor:

```java
public class ProdutorConsumidor_1 {

  private static final List<Integer> LISTA = new
  ArrayList<>(initialCapacity:5);
  private static boolean produzindo = true;
  private static boolean consumindo = true;

  Run | Debug
  public static void main(String[] args) {

    Thread produtor = new Thread(() -> {
      while (true) {
        try {
          simulaProcessamento();
          if (produzindo) {
            System.out.println(x:"Produzindo");
            int numero = new Random().nextInt
            (bound:10000);
            LISTA.add(numero);
            if (LISTA.size() == 5) {
              System.out.println(x:"Pausando
              produtor.");
              produzindo = false;
            }
            if (LISTA.size() == 1) {
              System.out.println(x:"Iniciando
              consumidor.");
              consumindo = true;
            }
```

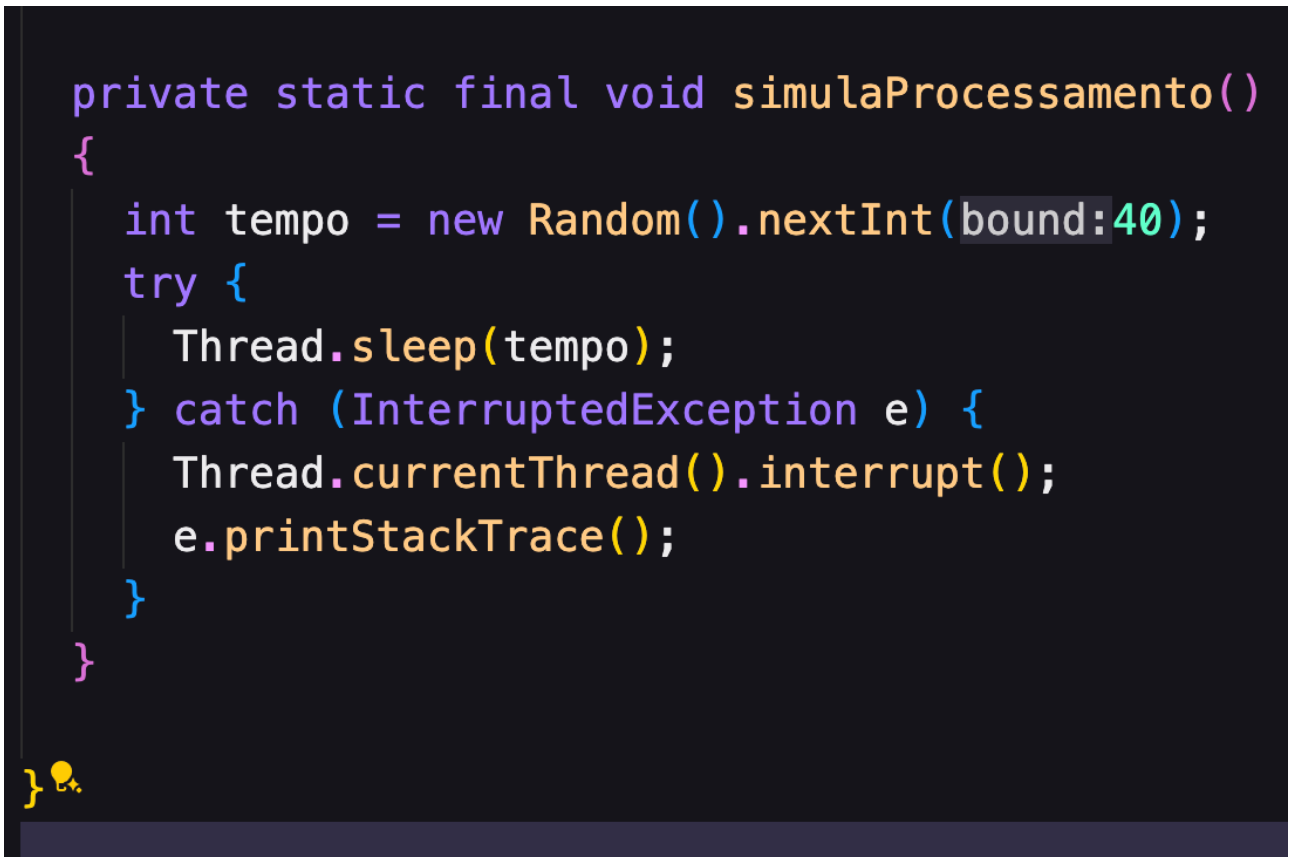
```
        } else {
          System.out.println(x:"!!! Produtor
          dormindo!");
        }
      } catch (Exception e) {
        System.out.println(e.getMessage());
      }
    }
});

Thread consumidor = new Thread(() -> {
  while (true) {
    try {
      simulaProcessamento();
      if (consumindo) {
        System.out.println(x:"Consumindo");
        Optional<Integer> numero = LISTA.
        stream().findFirst();
        numero.ifPresent(n -> {
          LISTA.remove(n);
        });
        if (LISTA.size() == 0) {
          System.out.println(x:"Pausando
          consumidor.");
          consumindo = false;
        }
```

```
            if (LISTA.size() == 4) {
              System.out.println(x:"Iniciando
              produtor.");
              produzindo = true;
            }
          } else {
            System.out.println(x:"??? Consumidor
            dormindo!");
          }
        } catch (Exception e) {
          System.out.println(e.getMessage());
        }
      }
    });

    Janelas.monitore(() -> String.valueOf(LISTA.
    size()));

    produtor.start();
    consumidor.start();
  }
```




- Primeiramente na estrutura do meu programa eu crio duas Threads, sendo elas a produtor e conaumidor, que ambas tem um loop que repete o processo de consumir ou produzir, junto a isso temos um método simula Processo que simulamos um processo por meio de um Thread.sleep() passando um tempo aleatório como sleep.
- Junto a isso estamos utilizando uma lista que será quem vai armazenar a produção e que vai ser consumida pelo consumidor, logo o produtor nesse caso ta adicionando um numero nessa lista e o consumidor consumindo os numeros dessa lista.
- utilizamos o método simulaProcesso para dar tempo entre as execuções.
- Limitamos as Threads de acordo sua definição, utilizando as variáveis produzindo e consumindo que são booleans para controle de ambas as threads.
- Analisando melhor o produtor nós criamos métodos de checar se a lista está cheia para parar o consumidor, e por último se a lista possui algo, caso possua o consumidor começa a execução.
- Agora analisando o consumidor temos métodos de controle parecidos, porém se a lista estiver vazia ele para o consumidor pois não há mais nada para consumir, e junto a isso uma checagem que vê se a lista tem 4 locais ocupados, caso tenha ele inicia o produtor.
- Porém quando utilizamos esse método de implementação ocorre um problema durante a execução do código travando o produtor ou o consumidor por conta de uma condição de corrida pois pode ocorrer uma chamada do produtor quando a lista estiver cheia de definir o consumidor como true, porém quando o consumidor volta a executar ele estava parado antes uma linha antes da linha que declara o consumidor como false, quando isso ocorre nosso produtor acha que acordou o consumidor, mas na verdade ele está dormindo, sendo assim ele para de produzir pois a fila está cheia e o consumidor para de consumir pois passaram para ele que ele deveria iniciar mas ele volta a atuar exatamente no momento em que é feita a chamada que seta ele como false.
- Nesse caso além das condições de corrida causadas, acaba ocorrendo um deadlock, em que ambas as Threads ficam esperando um recurso uma da outra e esse recurso nunca chega a ser liberado por nenhuma das duas, dessa forma ocorrendo um travamento.
- No caso a condição de corrida foi a ação de uma Thread modificar o valor de uma variável e a outra assim que chamada modifica novamente esse mesmo valor, porém para um valor que não é o correto, já o deadlock nesse caso é uma consequência dessas condições de corrida, pois ambas as Threads param esperando uma pela outra, e oq ocorre no final é que nenhuma faz nada e vão ficar aguardando uma pela outra e vice-versa, infinitamente.

- Agora vamos analisar outro exemplo do produtorXConsumidor e por meio desse exemplo aprender sobre a condição crítica e o mutex:

```
*/
public class ProdutorConsumidor_2 {

    private static final BlockingQueue<Integer>
    FILA =
        new LinkedBlockingDeque<>(capacity:5);
    private static volatile boolean produzindo =
    true;
    private static volatile boolean consumindo =
    true;
    private static final Lock LOCK = new
    ReentrantLock();

    Run | Debug
    public static void main(String[] args) {

        Thread produtor = new Thread(() -> {
            while (true) {
                try {
                    simulaProcessamento();
                    if (produzindo) {
                        LOCK.lock();
                        System.out.println(x:"Produzindo");
                        int numero = new Random().nextInt
                        (bound:10000);
                        FILA.add(numero);
```

```
            if (FILA.size() == 5) {
                System.out.println(x:"Pausando
                produtor.");
                produzindo = false;
            }
            if (FILA.size() == 1) {
                System.out.println(x:"Iniciando
                consumidor.");
                consumindo = true;
            }
            LOCK.unlock();
        } else {
            System.out.println(x:"!!! Produtor
            dormindo!");
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
  }
});
```

```
Thread consumidor = new Thread(() -> {
  while (true) {
    try {
      simulaProcessamento();
      if (consumindo) {
        LOCK.lock();
        System.out.println(x:"Consumindo");
        Optional<Integer> numero = FILA.stream
        ().findFirst();
        numero.ifPresent(n -> {
          FILA.remove(n);
        });
        if (FILA.size() == 0) {
          System.out.println(x:"Pausando
          consumidor.");
          consumindo = false;
        }
        if (FILA.size() == 4) {
          System.out.println(x:"Iniciando
          produtor.");
          produzindo = true;
        }
        LOCK.unlock();
```

```
            } else {
              System.out.println(x:"??? Consumidor
              dormindo!");
            }
          } catch (Exception e) {
            System.out.println(e.getMessage());
          }
        }
      });

      Janelas.monitore(() -> String.valueOf(FILA.
      size()));

      produtor.start();
      consumidor.start();
    }
```

```
  private static final void simulaProcessamento()
  {
    int tempo = new Random().nextInt(bound:40);
    try {
      Thread.sleep(tempo);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      e.printStackTrace();
    }
  }

}
```

- A primeira mudança clara que temos nesse algoritmo em relação ao primeiro é o uso de LinkedBlockquingDequeue, o uso do volatile nas variáveis boolean e junto a isso o uso de lock em ambas as Threads.

- O primeiro conceito que devemos entender nesse momento é o de região crítica , A região crítica é a região em que ocorre a concorrência em si, em que uma Thread concorre com outra por um recurso ou espaço e etc, nesse caso a região crítica do nosso código é a região das nossas Threads que estão acessando e modificando os recursos, nesse caso os recursos são a Lista e as variáveis boolenas, por tal motivo devemos mudar e adicionar a elas a palavra reservade volatile e o uso da classe atômica LinkedBlockingDeque.
- E como podemos resolver tal problema?, utilizando outro conceito da concorrência, sendo assim usando o Mutex ou Exclusão mútua, a exclusão mútua no caso ela faz com que a primeira Thread que acessa os nossos recursos pertencentes a região critica obtenham esse acesso enquanto a outra não consegue acessar, e quando ele para o acesso a outra consegue acessar e a primeira Thread deve esperar por isso esse nome de exclusão mútua pois só uma pode acessar o recurso por vez, dessa forma evitando assim as condições de corrida do código que podem ocasionar um deadlock.
- No caso acima modificamos nossas variáveis para classes Atômicas, e além disso adicionamos o Lock nos nossos processos e recursos que estão na região crítica.

- Mesmo com o código acima funcionando e acabando com os problemas re condições de corrida e deadlock, temos ferramentas mais ideias para a resolução dessas problemática:
- **OBS: o método put e take não serve apenas para essa tipo de fila em especifico, ele serve também para outros como SincronousQueue, e provavelmente serve para todos os tipos de Classes Atômicas, e mantendo o mesmo funcionamento de “compartilhamento” de recursos entre Threads, evitando a ocorrência de condições de corrida e de deadlock’s.**

```java
public class ProdutorConsumidor_3 {

    private static final BlockingQueue<Integer> FILA = new LinkedBlockingDeque<>(capacity:5);

    Run | Debug
    public static void main(String[] args) {

        Runnable produtor = () -> {
            simulaProcessamentoLento();
            System.out.println(x:"Produzindo");
            int numero = new Random().nextInt(bound:10000);
            try {
                FILA.put(numero);
                System.out.println(numero);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                e.printStackTrace();
            }
        };

        Runnable consumidor = () -> {
            simulaProcessamentoLento();
            System.out.println(x:"Consumindo");
            try {
                Integer take = FILA.take();
                System.out.println(take);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                e.printStackTrace();
            }
```

```
    Janelas.monitore(() -> String.valueOf(FILA.
    size()));

    ScheduledExecutorService executor =
        Executors.newScheduledThreadPool
        (corePoolSize:2);
    executor.scheduleWithFixedDelay(produtor,
    initialDelay:0, delay:10, TimeUnit.
    MILLISECONDS);
    executor.scheduleWithFixedDelay(consumidor,
    initialDelay:0, delay:10, TimeUnit.
    MILLISECONDS);
}

  private static final void
  simulaProcessamentoLento() {
    int tempo = new Random().nextInt(bound:400);
    try {
      Thread.sleep(tempo);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      e.printStackTrace();
    }
  }
```

- A primeira mudança é a ausência das variáveis booleanas que controlam o processo de produção e consumir, continuamod usando uma classe atômica, sendo ela a LinkedBlokingDeque, porém a maior mudança é que agora usamos Runnables e passamos ela para um executor, sendo esse Executor um Scheduled com uma Thread pool fixa de 2 Threads, logo usaremos 2 executores agendados que realizam tarefas diferentes, sendo seus parâmetros a Runnable que irão executar, o initial delay que começa nesse caso com 0, e após isso declaramos o delay que será de 10 em 10 ms após a primeira execução.
- Mas o que faz com que esse código ocorra sem a ocorrência de condições de corrida é o fato do uso da classe Atômica LinkedBlockngDeque , porém dessa vez com os métodos corretos para adicionar e remover elementos da fila que está na região crítica. trocamos o add e remove respectivamente pelos métodos put e take. O método put() tenta colocar elementos na lista

caso haja espaço e tempo de processamento para isso, e caso ele não possa colocar aquele elemento, a Thread simplemente aguarda até que possa realizar tal operação, já o método take(), tenta retirar um elemento na lista caso ela tenha elementos e ele tenha tempo de processamento disponível para isso, além de retornar qual foi o elemento que ele retirou da lista.

- Dessa forma utilizando esses métodos conseguimos resolver o problema do ProdutorXConsumidor quebrando todas as problemáticas de “condições de corrida” e de DeadLock que podem ocorrer utilizando tanto exclusão mútua nas regiões críticas, como usando métodos mais adequados aproveitando dos benefícios das classes Atômicas.

#Concorrent