# Programação_Concorrente-Studies/ Aula07: Sincronizando threads com o uso da Cyclic Barrier

- Pra que serve a cyclicBarrier: Quando tiver várias threads executando em paralelo e em algum momento vc necessita que uma thread espere a execução da outra.(Usaremos isso na problemática dos Trens com concorrência).

- Nosso primeiro exemplo será dividir as nossas Threads em 3 para realizar uma operação matemática complicada.
- Primeiramente vamos criar nossas threads por executores com uma pool de threads fixas e criar as tarefas que cada uma dessas 3 threads vão realizar:

```
11
12    // (432*3) + (3^14) + (45*127/12) = ?
13    public static void main(String[] args) {
14      ExecutorService executor = Executors.newFixedThreadPool(3);
15
16      Runnable r1 = () -> {
17        System.out.println(432d*3d);
18      };
19      Runnable r2 = () -> {
20        System.out.println(Math.pow(3, 14));
21      };
22      Runnable r3 = () -> {
23        System.out.println(45d*127d/12d);
```

- Porém quando executamos a nossa aplicação ele nos retorna os resultados de forma desorganizada, por isso agora usaremos a nossa CyclicBarrier

- Criando a nossa CyclicBarrier:

```
CyclicBarrier cyclicBarrier = new CyclicBarrier(3);
```

- Aqui declaramos nossa cyclic barrier e determinamos o numero de participantes dessa barrier

- Logo após em cada uma das tarefas chamamos o método await(cyclicBarrier)

```
37
38    private static void await(CyclicBarrier cyclicBarrier) {
39      try {
40        cyclicBarrier.await();
41      } catch (InterruptedException | BrokenBarrierException e) {
42        Thread.currentThread().interrupt();
43        e.printStackTrace();
44      }
45    }
  await(cyclicBarrier);
};
```

- Esse método faz com que quando a Thead chegue nele, ela pare o processamento dela e aguarde até que as outras Threds que fazem parte dessa cyclic Barrier cheguem nessa

chamada de método também, dessa forma após todas as threads chegarem nessa linha de código, elas são liberadas para continuarem o trabalho.

- Logo esse primeiro conceito e método simples do cyclic barrier faz com que criemos um ponto de sincronia entre Threads.

- Agora vamos criar uma cyclicBarrier que após as Threads aguardarem o processamento de todas chegarem na cyclicBarrier, ela incia o processamento de outra Thead, dessa forma conseguindo fazer um processamento de um resultado ou uma ação após a sincronia das Threads

```
14  public class CyclicBarrier_2 {
16      private static BlockingQueue<Double> resultados
        =
17          new LinkedBlockingQueue<>();
18
        Run | Debug
19      public static void main(String[] args) {
20          Runnable finalizacao = () -> {
21              System.out.println(x:"Somando tudo.");
22              double resultadoFinal = 0;
23              resultadoFinal += resultados.poll();
24              resultadoFinal += resultados.poll();
25              resultadoFinal += resultados.poll();
26              System.out.println("Processamento
                finalizado. "
27                      + "Resultado final: " + resultadoFinal
28                      + ". Inscreva-se no canal!");
29          };
30          CyclicBarrier cyclicBarrier = new
            CyclicBarrier(parties:3, finalizacao);
31
32          ExecutorService executor = Executors.
            newFixedThreadPool(nThreads:3);
33
34          Runnable r1 = () -> {
35  //              System.out.println(Thread.currentThread().
    getName());
36              resultados.add(432d*3d);
37              await(cyclicBarrier);
38  //              System.out.println(Thread.currentThread()
```

- Para isso utilizaremos uma BlockingQueue como visto em aulas anteriores para evitar “condições de corrida”, e a unica alteração que faremos é na hora de declarar nossa Cyclic Barrier, pois nós estamos declarando uma Thread que será chamada quando as 3 Threads chegarem no cyclicBarrier.await();
- E esse Thread será responsável por realizar o processamento do resultado final somando os valores que as outras Threads fizeram o calculo, e só após a realização dessa Thread de finalização que as outras 3 threads voltam com seu processamento.
- Então basicamente oq fizemos foi que as 3 threads realizam os cálculos até chegarem no await, quando chegam lá elas param e o cyclicBarrier inicia a thread finalização, a thread finalização faz seu processamento e após ela fazer seu processamento, as threads que estavam no await voltam a funcionar em “paralelo”.

- Agora vamos fazer um processamento ciclico com a nossa CyclicBarrier:

```
private static void restart() {
    sleep();
    executor.submit(r1);
    executor.submit(r2);
    executor.submit(r3);
}

private static void sleep() {
    try {
        Thread.sleep(millis:1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```
private static double resultadoFinal = 0;

Run | Debug
public static void main(String[] args) {
  Runnable sumarizacao = () -> {
    System.out.println(x:"Somando tudo.");
    resultadoFinal += resultados.poll();
    resultadoFinal += resultados.poll();
    resultadoFinal += resultados.poll();
    System.out.println("Processamento
    finalizado. "
        + "Resultado final: " + resultadoFinal
        + ". Próximo processamento em alguns
        segundos.");
    System.out.println(x:"------------");
//      restart();
  };
  CyclicBarrier cyclicBarrier = new
  CyclicBarrier(parties:3, sumarizacao);

  executor = Executors.newFixedThreadPool
  (nThreads:3);
  r1 = () -> {
    while (true) {
      resultados.add(432d*3d);
      await(cyclicBarrier);
      sleep();
    }
```

- Para isso teremos que fazer diversas mudanças em nosso código, sendo uma delas a criação do método restart que chama o executor para realizar as tarefas das nossas Threads e devemos modificar os nossos Runnables de forma a modificar a forma como chamamos o await.

- Utilizando as Threads com o while(true), temos um método de restart próprio da thread, logo, elas estão em um loop de se re-executar sempre que terminamos o await e fazemos o método de sumarização, para isso ser visto de forma mais clara no console criamos o método sleep que faz as Threads pararem por 2 segundos.
- Porém caso não desejarmos o uso do loop próprio de cada thread podemos chamar o método restart dentro do próprio método final de sumarização, que chamaria novamente as Threads para uma nova execução assim que acabasse o seu processamento, também mantendo nosso processo em um loop ciclico.

- Portanto vimos aqui sobre as cyclicBarriers que servem para sincronizar threads e permitindo criar barreiras de esperas que podem ou não ser ciclicas além criar processamentos enquanto as Threads estão no estado de await dessas barreiras.

#Concorrent