Programação_Concorrente-Studies/ Aula09: Semaphores , Limitando a execução de várias Threads em um mesmo trecho de código com semáforos.

- A classe de semáforos serve para controlar a quantidade de threads que pode executar um trecho de código ao mesmo tempo, como um semáforo mesmo, que permite ou não que uma thread continue o processamento, e ela faz isso dependendo da quantidade de threads que o programador permite que acesse aquele espaço de código.

- Vamos ver um primeiro exemplo, em que para ele usaremos um executor e junto a isso um semaphoro e uma tarefa Runnable

```java
 * @author MarcusCSPereira
 */
public class Semaphore_1 {

  private static final Semaphore SEMAFORO = new
  Semaphore(permits:3);

  Run | Debug
  public static void main(String[] args) {
    ExecutorService executor = Executors.
    newCachedThreadPool();

    Runnable r1 = () -> {
      String name = Thread.currentThread().getName
      ();
      int usuario = new Random().nextInt
      (bound:10000);

      acquire();
      System.out.println("Usuário " + usuario
          + " Processou usando a thread " + name
          + "\n");
      sleep();
      SEMAFORO.release();
    };

    for (int i = 0; i < 500; i++) {
      executor.execute(r1);
    }
```

```
    executor.shutdown();
}

private static void sleep() {
    // espera de 1 a 6 segundos
    try {
        int tempoEspera = new Random().nextIn
        (bound:6);
        tempoEspera++;
        Thread.sleep(1000 * tempoEspera);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        e.printStackTrace();
    }
}

private static void acquire() {
    try {
        SEMAFORO.acquire();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        e.printStackTrace();
    }
}
```

- Simulamos uma tarefa de um usuário realizando uma ação, e por meio de um for que vai de 0 a 500, ele vai realizar as tarefas no máximo de threads possivel, pois estamos utilizando um executor cached.

- Porém quando criamos o semaphoro e declaramos como parâmetro o número 3, significa que ele permitirá que apenas 3 threds realizem esse processamento ao mesmo tempo.
- Com o método acquire eu faço as theads “resevarem” seu espaço no semaphoro, e para liberar o processamento é necessário o uso do método release(), que libera aquela posição do semaphoro e dessa forma simbolizando o fim do seu processamento e “liberando um vaga” para que outra thread possa ocupar.
- Logo nessa execução, primeiro 3 theads pegam o tempo de processamento e espaço no semaphoro, e depois como elas dormem e executam por um tempo um pouco diferente, elas acabam soltando e liberando a vaga no semaphoro em tempos diferentes, e assim que uma vaga é aberta outra Thead pega essa vaga e realiza o seu processamento, então na saida teremos a saida de 3 Theads praticamente juntas e depois a saida de 1 thread por vez

- Agora vamos entender sobre o método tryAcquire dos semaphores e como retornar o número de threads esperando vaga para processamento:

```java
 *
 * @author MarcusCSPereira
 */
public class Semaphore_2 {

  private static final Semaphore SEMAFORO = new
  Semaphore(permits:100);

  private static final AtomicInteger QTD = new
  AtomicInteger(initialValue:0);

  Run | Debug
  public static void main(String[] args) {
    ScheduledExecutorService executor =
        Executors.newScheduledThreadPool
        (corePoolSize:501);

    Runnable r1 = () -> {
      String name = Thread.currentThread().getName
      ();
      int usuario = new Random().nextInt
      (bound:10000);

      boolean conseguiu = false;
      QTD.incrementAndGet();
      while (!conseguiu) {
        conseguiu = tryAcquire();
      }
      QTD.decrementAndGet();

```

```
        System.out.println("Usuário " + usuario
                + " se inscreveu no canal usando a
                thread " + name + "\n");
        sleep();
        SEMAFORO.release();
    };

    Janelas.Mensagem janela = Janelas.criaJanela
    (textoInicial:"QTD");
    Runnable r2 = () -> {
        int qtd = QTD.get();
        janela.setText(qtd + " usuários esperando
        para se cadastrar!");
    };

    for (int i = 0; i < 500; i++) {
        executor.execute(r1);
    }
    executor.scheduleWithFixedDelay(r2,
    initialDelay:0, delay:100, TimeUnit.
    MILLISECONDS);
}
```

```java
    private static void sleep() {
        // espera de 1 a 6 segundos
        try {
            int tempoEspera = new Random().nextInt
            (bound:6);
            tempoEspera++;
            Thread.sleep(1000 * tempoEspera);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }

    private static boolean tryAcquire() {
        try {
            return SEMAFORO.tryAcquire(timeout:1,
            TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
            return false;
        }
    }

}
```

- o método tryAcquire diferente do acquire precisa receber 2 parâmetros que vão representar quanto tempo a nossa Thread fica esperando uma vaga no semaforo, caso ele não consiga ele retorna false e caso consiga ele retorna true.

```
boolean conseguiu = false;
QTD.incrementAndGet();
while (!conseguiu) {
  conseguiu = tryAcquire();
}
QTD.decrementAndGet();
```

- Como o tryAcquire tem um retorno sendo true o false, podemos trabalhar de acordo com o retorno do método, podemos ter um bloco de código que necessita do semáforo como nesse caso, e que caso ele consiga a vaga ele execute, porém caso ele não consiga ele não execute apenas esse trecho de código mas pode prosseguir com os outros processamentos em que é responsável por realizar.
- Utilizando o AtomicInteger podemos analisar e receber quantas Threads estão solicitando o acesso a vaga no semaforo e quantas threads conseguiram o seu tempo de processamento e passaram pela vaga realizando o seu processamento.
- Nesse caso é importante ressaltar que usamos uma janela gráfica que realiza apenas a saida da tarefa da Runnable 2 para demonstrar quantas threads estão esperando por uma vaga no semaforo, dai vemos um claro exemplo de multiThread, afinal temos 2 execuções ao mesmo tempo, uma Thread que realiza a tarefa, e uma thread demonstrando quantas Threads estão esperando para realizar aquela mesma tarefa.

#Concorrent