Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente, Verão de 2024/2025

Laboratório #1

---

1. Escreva um programa em Kotlin com a constante `NTHREADS = 1` e a função `makeToast()` que executa um ciclo infinito obsessivo. A função `main()` cria `NTHREADS` instâncias da classe `Thread` para executar a função `makeToast()`. Execute o programa com diferentes valores de `NTHREADS`, sempre **menores** do que o número de CPUs disponíveis na máquina e observe a taxa de ocupação dos CPUs (*Task Manager* no Windows ou programa `top` em Linux).

2. Observe o código da função `thread` disponível na biblioteca Kotlin:
   https://github.com/JetBrains/kotlin/blob/master/libraries/stdlib/jvm/src/kotlin/concurrent/Thread.kt

   a. Modifique o exemplo do exercício 1 para usar esta função em vez do construtor de `Thread` e confirme que se mantém o comportamento.

   b. O que acontece se passar o parâmetro `isDaemon` com o valor `true`? Porquê?

3. Escreva um programa em Kotlin com uma variável global `counter`, iniciada com o valor 0. Na função `main()`, lance 10 *threads*, cada uma a incrementar o valor de `counter` `TCOUNT` vezes, e verifique se o valor final do contador é 10 * `TCOUNT`.

   a. Experimente o programa com diferentes valores de `TCOUNT`, entre 10 e 100000000.

   b. Consulte a documentação da classe `AtomicInteger`:
   https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html
   Corrija o programa usando o tipo `AtomicInteger` e confirme que se obtém o resultado esperado.

4. Escreva um programa em que múltiplas *threads* inserem valores inteiros sequenciais (`0 until NITERATIONS`) numa instância de `java.util.LinkedList<Int>`.

   a. Quando todas as inserções tiverem terminado, calcule a soma de todos os elementos, usando o método de extensão `sum()`. Compare com o valor esperado.

   b. Qual é o tamanho real e o tamanho esperado da lista no final das inserções.

   c. Substitua `LinkedList<Int>()` por `Collections.synchronizedList(LinkedList<Int>())`.
   https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collections.html#synchronizedList(java.util.List)
   Qual é o tamanho da lista e o resultado de `sum()` após as inserções com esta modificação?

   d. Remova a utilização de `Collections.synchronizedList` e crie uma instância de `ReentrantLock` junto à lista.
   https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html
   Altere o ciclo de inserção de elementos para que **cada** uma das inserções (mas não o ciclo!) decorra com o *lock* adquirido. Pode usar a função de extensão `withLock` disponibilizada pela biblioteca Kotlin.
   https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.concurrent/java.util.concurrent.locks.-lock/with-lock.html

5. Escreva um programa com duas listas pré-preenchidas, uma com os números de 1 a 1000 e outra com os números de 10000 a 11000. Cada uma das listas será protegida com um `ReentrantLock` próprio, de que existirão duas instâncias. A função `move1to2` adquire o *lock* da lista 1, depois o da lista 2 e, de seguida, remove um elemento da lista 1 e adiciona-o à lista 2, libertando os *locks* no final. A função `move2to1` adquire o *lock* 1, depois o 2 e move um elemento da lista 2 para a 1. Coloque 16 *threads* a executar NMOVES chamadas a `move1to2` e outras 16 a executar NMOVES chamadas a `move2to1`. Confirme que ambas as listas têm 1000 elementos no final.

(continua)

6. Considere o exemplo `EchoServerThreadPerConnection.kt` disponível em:
   https://gist.github.com/jtrindade/680ce1d4985976ef78a9e3e91eafe1de

   a. Execute o servidor e estabeleça múltiplas ligações em simultâneo, confirmando que todas podem ser utilizadas.

   b. Modifique a implementação para usar um número fixo de *threads* (por exemplo, 4) para atendimento de clientes. Sempre que é pedida uma ligação, é imediatamente atribuído o identificador de cliente, mas o atendimento efetivo do cliente só se inicia quando uma das *threads* atendentes estiver disponível. Para tal, coloque as sessões numa instância de `BlockingQueue`. As *threads* atendentes removem uma sessão da fila, atendem o respetivo cliente e, no final, regressam à fila para retirar mais um elemento.
   https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/BlockingQueue.html

7. Considere o programa `FiveFetchers.kt`, que simula cinco operações demoradas de obtenção de dados a decorrerem em simultâneo: https://gist.github.com/jtrindade/53f315b1587ad2e12b9a410f1e822ab3
   Escreva um programa semelhante em JavaScript sobre Node.js, utilizando a função `fetchData` indicada abaixo. Pode colocar o código equivalente ao da função `main` numa função `async` e utilizar `await` sobre o resultado de um `Promise.all` em substituição do `forEach` que realiza os `join`.

```
function fetchData(id) {
  const sleepTime = Math.floor(Math.random() * 5000) + 3000; // Entre 3s e 8s
  console.log(
    `A buscar dados para ID ${id} (espera de ${sleepTime / 1000} segundos)...`
  );
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(`Dados para ID ${id}`);
    }, sleepTime);
  });
}
```

Bom trabalho!

ISEL, 25 de fevereiro de 2025