Łukasz Jaworski 155 NCI

Lab 04

Wykorzystanie zamków w programach wątkowych

W tym labolatorium chciałem sprawdzić działanie **zamków**, które podobnie jak funkcja **synchronized** organizuje pracę wątku, jednak to wprowadzenie zamków daje większą elastyczność przy określaniu warunków. Mimo, że synchronizacja jest dużo prostsza we wprowadzeniu i utrzymaniu, to właśnie klasa **Locks** umożliwia nam ustawienie większej ilości parametrów.

Stworzyłem krótki program **Locks,** tworzący 4 wątki konkurujące ze sobą o dostęp do głównego zasobu, co zostało zapisane jako „ próba wypowiedzenia się". W pierwszej kolejności zaprogramowałem całą logikę bez wykorzystania zamków, by sprawdzić zachowanie aplikacji.

```
As 3 spoke threadSpeaker1
As 4 spoke threadSpeaker4
As 4 spoke threadSpeaker2
As 3 spoke threadSpeaker3
As 5 spoke threadSpeaker2
As 6 spoke threadSpeaker3
As 5 spoke threadSpeaker1
As 5 spoke threadSpeaker4
As 7 spoke threadSpeaker1
As 7 spoke threadSpeaker2
As 7 spoke threadSpeaker3
As 7 spoke threadSpeaker4
```

W programie została ustawiona kolejka pokazująca kolejność wypowiedzi każdego wątku. Mimo, że ilość wypowiedzi została ustalona na 8, to program w tej postaci nie respektuje tego ograniczenia. Wątki próbują wykonać swoją akcję jak najszybciej, co często skutkuje 'wejściem komuś w słowo' co widać po tym, że w jednej iteracji porafią się wypowiedzieć trzy wątki. Jak widać, kolejność zabierania głosów przez wątki jest losowa.

```
As 1 spoke threadSpeaker1
As 2 spoke threadSpeaker4
As 3 spoke threadSpeaker2
As 4 spoke threadSpeaker3
As 5 spoke threadSpeaker1
As 6 spoke threadSpeaker4
As 7 spoke threadSpeaker2
As 8 spoke threadSpeaker3
```

Po zmodyfikowaniu funkcji przypisującej kolejność do wątka, rozszerzając ją o istnienie zamka, a dokładniej **ReentrantLock**. Jest to jeden z najbardziej podstawowych zamków, bardzo przypominając działanie słowa **synchronized**, jednak należy pamiętać o wprowadzeniu zabezpieczenia w funkcji **try** oraz uwolnieniu zamka poprzez **lock.unlock().**

W tym przypadku zamek spełnił swoje zadanie, nie tylko organizując poprawność wejścia do funkcji (co widać po poprawnej kolejności wypowiadania się), ale również zadbał o równy dostęp do metody – Po pierwszej, losowej iteracji między czterema wątkami druga ma tą samą kolejność, aby w pierwszej kolejności dostęp uzyskały najdłużej czekające wątki.

```
try {
    lock.tryLock( time: 10, TimeUnit.SECONDS);
    i++;
    list.add(o);
    System.out.println("As "+i+" spoke thread"+Thread.currentThread().getName());
    Thread.sleep( millis: 50);
} catch (InterruptedException e) {
    e.printStackTrace();
} finally {

    lock.unlock();
}
```

Na zrzucie powyżej widać kod funkcji wykorzystującej zamek. Skorzystałem z metody **lock.tryLock()** zamiast **lock.lock()**, co zmienia sposób zgłaszania się wątków. W normalnej sytuacji po zamknięciu zamka wszystkie próby zdobycia go są nieudane, jednak dzięki funkcji tryLock możemy określić przez jaki czas wątek ma podejmować próbę uzyskania zamka. Po minięciu wskazanego czasu próba jest automatycznie odrzucana.