7. Blokady

Blokady są narzędziem do kontrolowania sekcji krytycznych aplikacji. Oferują taką samą funkcjonalność jak bloki synchronizowane (`synchronized`) lecz zapewniają lepszą wydajność. Dodatkowo umożliwiają kontrolę nad typem dostępu do sekcji krytycznej (odczyt i zapis) oraz wprowadzają ograniczenia czasowe związane z uzyskaniem dostępu do sekcji krytycznej. Możliwe jest również sprawdzenie czy blokada jest zajęta i jeśli tak wykonywanie innych czynności. Dostępne jest również przerwanie wątków, które zajęły blokadę. Inną różnicą jest możliwość zajęcia zasobu w jednej metodzie, a zwolnienie w zupełnie innej (podobieństwo do semaforów). Blokady często nazywane są również ryglami.

Interfejsy i klasy implementujące

Blokady dostępne w Javie zgromadzone są w pakiecie `java.util.concurrent.locks` i opierają się na dwóch interfejsach:

- interfejs `Lock` - prosta blokada na zasadzie zajmij i zwolnij,
- interfejs `ReadWriteLock` – blokada umożliwiająca obsługę różnych typów dostępu do zasobów (odczyt i zapis)

Klasy implementujące powyższe interfejsy to:

- `ReentrantLock` – implementacja interfejsu `Lock`,
- `ReentrantReadWriteLock` – implementacja interfejsu `ReadWriteLock`

Podstawowe metody związane z blokadami (interfejs `Lock`):

- `lock()` – metoda zajmująca blokadę, jeśli blokada nie jest dostępna wątek zostaje wstrzymany (nie można przerwać jego działania),
- lockInterruptibly() – jak wyżej, ale wątek wstrzymany może zostać przerwany,
- `tryLock()` – zajmuje blokadę tylko wtedy jeśli jest wolna,
- `tryLock(time)` – metoda próbująca zająć blokadę (w przypadku braku możliwości zajęcia blokady czeka określony czas),
- `unlock()` – zwolnienie blokady.

Dodatkowe metody (interfejs `ReadWriteLock`):

- `readLock()` – zwraca blokadę odczytu,
- `writeLock()` – zwraca blokadę zapisu.

W odróżnieniu od bloków synchronizowanych, które same dbają o zwolnienie blokady, istotne jest prawidłowe wywołanie metody zwalniającej blokadę (sekcja `finally`).

Poniżej sposób wywołania blokad oraz ich odpowiednik synchronizowany. Najpierw wersja wykorzystująca `synchronized`.

```
class Synchronizacja {

    private Object obiekt = new Object();

    public void dodaj() {

        synchronized(obiekt) {
            //sekcja krytyczna
        }
```

```
    }
}
```

Wersja wykorzystująca blokadę.

```
class Blokada{

    private Lock obiekt = new ReenterantLock();

    public void dodaj() {

        obiekt.lock();

        try {
            //sekcja krytyczna
        } catch(Exception e) {
           //obsługa błędów
        } finally {
            obiekt.unlock();
        }
    }
}
```

Blokady umożliwiają kolejkowanie wątków oczekujących na dostęp do zasobu z uwzględnieniem wątków najdłużej oczekujących (blokady uczciwe, obsługa na zasadzie kolejki FIFO). Inną możliwością jest sprawdzenie czy uzyskamy dostęp do blokady i jeśli nie ma takiej możliwości możemy wykonać inne czynności. Przykładowy fragment kodu:

```
Lock lock;

boolean test = lock.tryLock();
if (test) {
   try {
     // wykonywane są instrukcje sekcji krytycznej
   } finally {
      lock.unlock();
   }
} else {
    // inne operacje
}
```

Przykładowa aplikacja – klasa ReentrantLock

Poniżej prosta aplikacja tworząca dziesięć wątków wykorzystujących blokadę.

```
import java.util.concurrent.locks.ReentrantLock;

class Watek implements Runnable {
    private String name;
    private ReentrantLock rl;

    public Watek(String name, ReentrantLock rl) {
        this.name = name;
        this.rl = rl;
    }

    public void run() {
        rl.lock();
        try {
            System.out.println("Wątek: " + name + " jest w sekcji krytycznej");

            try {
                Thread.sleep((int)(Math.random()*5000));
            } catch (InterruptedException e) {
                System.out.println("Błąd");
```

```
            }

            System.out.println("Wątek: " + name + " opuszcza sekcję krytyczną");
        } finally {
            rl.unlock();
        }
    }
}

public class TestBlokad {
   public static void main(String[] args) {

      final ReentrantLock rl = new ReentrantLock();

       for (int i = 1; i <= 10; i++) {
        new Thread(new Watek("W"+i , rl)).start();
       }
   }
}
```

Wyniki działania aplikacji:

```
Wątek: W1 jest w sekcji krytycznej
Wątek: W1 opuszcza sekcję krytyczną
Wątek: W2 jest w sekcji krytycznej
Wątek: W2 opuszcza sekcję krytyczną
Wątek: W3 jest w sekcji krytycznej
Wątek: W3 opuszcza sekcję krytyczną
Wątek: W5 jest w sekcji krytycznej
Wątek: W5 opuszcza sekcję krytyczną
Wątek: W7 jest w sekcji krytycznej
Wątek: W7 opuszcza sekcję krytyczną
Wątek: W9 jest w sekcji krytycznej
Wątek: W9 opuszcza sekcję krytyczną
Wątek: W4 jest w sekcji krytycznej
Wątek: W4 opuszcza sekcję krytyczną
Wątek: W6 jest w sekcji krytycznej
Wątek: W6 opuszcza sekcję krytyczną
Wątek: W8 jest w sekcji krytycznej
Wątek: W8 opuszcza sekcję krytyczną
Wątek: W10 jest w sekcji krytycznej
Wątek: W10 opuszcza sekcję krytyczną
```

Przykładowa aplikacja – klasa ReentrantReadWriteLock

Klasa `ReentrantReadWriteLock` umożliwia utworzenie osobnych blokad dostępnych dla operacji odczytu jak i zapisu. Standardowo wykonywane są następujące operacje:

- utworzenie obiektu blokady,
- pobranie blokady odczytu i zapisu (metody `readLock()` i `writeLock()`),
- użycie blokady odczytu dla obiektów odczytujących,
- użycie blokady zapisu dla obiektów zapisujących.

Poniżej przykładowa aplikacja wykorzystująca tę możliwość.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class StacjaPogodowa extends Thread {
    private Dane dane;

    public StacjaPogodowa(Dane dane) {
        this.dane = dane;
    }
```

```
    public void run() {
        while(true) {

            dane.odczyt();

            try {
                Thread.sleep((int)(Math.random()*2000));
            } catch (InterruptedException e) {
                System.out.println("Błąd");
            }
        }
    }
}

class Sensor extends Thread {
    private Dane dane;

    public Sensor(Dane dane) {
        this.dane = dane;
    }

    public void run() {
        while(true) {

            dane.zapis(Math.random()*100);

            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                System.out.println("Błąd");
            }
        }
    }
}

class Dane {
    private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private Lock rl = rwl.readLock();
    private Lock wl = rwl.writeLock();

    private double temperatura;

    public void zapis(double temperatura) {
        wl.lock();
        try {

            System.out.println("Zapisuje nową temperaturę: " + temperatura);

            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                System.out.println("Błąd");
            }

            this.temperatura = temperatura;
         } finally {
           wl.unlock();
         }
    }

    public double odczyt() {
        rl.lock();
        try{
           System.out.println("Odczyt temperatury: " + temperatura);
           return temperatura;
        } finally {
           rl.unlock();
```

```
        }
    }
}

public class TestBlokadRW {

    public static void main(String[] args) {
        Dane dane = new Dane();

        new Sensor(dane).start();
        new StacjaPogodowa(dane).start();
    }

}
```

Wyniki działania aplikacji:

```
Zapisuje nową temperaturę: 60.476318086738814
Odczyt temperatury: 60.476318086738814
Odczyt temperatury: 60.476318086738814
Odczyt temperatury: 60.476318086738814
Odczyt temperatury: 60.476318086738814
Odczyt temperatury: 60.476318086738814
Odczyt temperatury: 60.476318086738814
Odczyt temperatury: 60.476318086738814
Odczyt temperatury: 60.476318086738814
Odczyt temperatury: 60.476318086738814
Zapisuje nową temperaturę: 74.24879102766423
Odczyt temperatury: 74.24879102766423
Odczyt temperatury: 74.24879102766423
Odczyt temperatury: 74.24879102766423
Odczyt temperatury: 74.24879102766423
Odczyt temperatury: 74.24879102766423
Odczyt temperatury: 74.24879102766423
```

Zadania

1. Zmodyfikuj aplikację Producent-Konsument tak, aby wykorzystywała zamiast metod synchronizowanych blokady.

2. Zmodyfikuj aplikację rozwiązującą problem ucztujących filozofów tak, aby wykorzystywała zamiast semaforów blokady (jedna blokada to jeden widelec).

3. Wykorzystując blokady typu odczyt-zapis rozwiąż problem czytelników i pisarzy. Wspólny zasób **Czytelnia** jest dzielony pomiędzy dwie grupy wątków:

- **Czytelnicy** – wszystkie wątki niedokonujące zmian w zasobie,
- **Pisarze** – pozostałe wątki (dokonujące zmian w zasobie).

Jednoczesny dostęp do zasobu może uzyskać dowolna liczba czytelników. Pisarz może otrzymać tylko dostęp wyłączny. Równocześnie z pisarzem dostępu do zasobu nie może otrzymać ani inny pisarz, ani czytelnik, gdyż mogłoby to spowodować błędy.