Teoria współbieżności

Laboratorium 2
Monitory w Javie

Albert Gierlach

14.10.2020

1. Zadanie 1

Zaimplementować semafor binarny za pomocą metod wait i notify, użyć go do synchronizacji programu Wyscig.

2. Koncept rozwiązania

Korzystając ze szkieletu semafora zaimplementowałem powyższe polecenie oraz przeprowadziłem stosowne testy.

3. Implementacja oraz wyniki

Dla uproszczenia pominąłem klasy zaimplementowane na poprzednich zajęciach.

```
class BinarySemaphore {
    private boolean available;
    private int waitingThreads;

    public BinarySemaphore() {
        this.available = true;
        this.waitingThreads = 0;
    }

    public synchronized void P() { //acquire
        this.waitingThreads++;
        while (!this.available){
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.waitingThreads--;
        this.available = false;
    }

    public synchronized void V() { //release
        if(this.waitingThreads > 0){
            this.notify();
        }
        this.available = true;
```

```
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        var results = new ArrayList<Integer>();
        for (int i = 0; i < 100; i++) {
            Counter cnt = new Counter(0);
            var ti = new IThread(cnt);
            var td = new DThread(cnt);

            ti.start();
            td.start();

            ti.join();
            td.join();

            results.add(cnt.value());
        }

        System.out.println(results.stream().allMatch(v -> v == 0));
    }
}
```

Wynik uruchomienia programu:

```
true
```

4. Wnioski

Jak można zauważyć - semafor działa poprawnie, a dostęp do zasobu jakim jest licznik został poprawnie zsynchronizowany między wątkami.

5. Zadanie 2

Pokazac, ze do implementacji semafora za pomoca metod wait i notify nie wystarczy instrukcja if tylko potrzeba uzyc while. Wyjasnic teoretycznie dlaczego i potwierdzic eksperymentem w praktyce.

6. Rozwiązanie

Do implementacji semafora za pomocą metod *.wait()* i *.notify()* nie wystarczy instrukcja if, ponieważ będzie dochodzić do sytuacji wyścigu. Załóżmy, że w programie uruchomione są trzy wątki, w następujących stanach:

- Wątek A: Uśpiony w metodzie *P()*
- Wątek B: Ma zamiar wejść do sekcji krytycznej (czyli spróbować opuścić semafor)
- Wątek C: Aktualnie wykonuje swoją sekcję krytyczną

Warto zaznaczyć, iż wątek uśpiony zwalnia mutex, który odpowiada za wejście do synchronizowanej metody *P()*. Poniżej przedstawię teoretyczną listę kroków, która przedstawi sytuację wyścigu.

1. Wątek B wchodzi do metody *P()* (zajmując tym samym mutex kolejki monitora), ponieważ wątek A jest uśpiony.
2. Wątek C opuszcza swoją sekcję krytyczną i wywołuję metodę *.notify()*, semafor zostaje podniesiony.
3. Wątek A budzi się, ale czeka na zwolnienie mutexu związanego z monitorem (Wątek B aktualnie zajął mutex).
4. Wątek B sprawdza warunek mówiący o tym czy semafor jest podniesiony, a skoro semafor jest podniesiony to opuszcza go i zaczyna wykonywać swoją sekcję krytyczną.
5. Wątek A kontynuuje wykonanie, ale nie sprawdza warunku zamieszczonego w instrukcji if - w rezultacie opuszcza semafor (który jest już opuszczony przez wątek B) i wykonuje swoją sekcję krytyczną.
6. W efekcie wątek A oraz B nie synchronizują między sobą dostępu do współdzielonego zasobu.

7. Implementacja oraz wyniki

Klasa *BinarySemaphore* została zmodyfikowana, aby używać instrukcji *if* zamiast *while*. Kod metody testującej został jedynie wzbogacony o końcowe wypisanie zebranych wartości licznika.

```
[-10, -1, 3, 0, -1, 130, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 1, -1, 0, 0,
↪  -3, 0, -1, 0, -1, -1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, -2, 0, 0, 1, 0, 0,
↪  -2, -2, 1, 2, 1, 0, 0, -1, 0, -5, -1, -1, 1, 0, 1, -1, 0, 0, 1, 0, 0, 0,
↪  0, -1, 0, -1, 0, 0, 0, 0]
false
```

8. Wnioski

Przedstawiona teoria wydaje się sprawdzać w praktyce, a przedstawione wyniki jeszcze bardziej ją potwierdzają. Dodatkowo w dokumentacji metody *.wait()* znalazłem zapis mówiący o tym, że wątek może zostać spontaniczne wybudzony, co w zasadzie wymusza na programiście użycia instrukcji *while*.

9. Zadanie 3

Zaimplementowac semafor licznikowy (ogolny) za pomoca semaforow binarnych. Czy semafor binarny jest szczegolnym przypadkiem semafora ogolnego?

10. Koncept rozwiązania

Rozwiązanie będzie polegać na dwóch semaforach binarnych oraz zmiennej, która będzie przechowywać ilość współdzielonych zasobów. Jeden z semaforów będzie synchronizował dostęp do zmiennej z liczbą zasobów tak, aby tylko jeden wątek ją zmieniał w danym momencie, a drugi semafor będzie informował o tym, czy jest dostępny chociaż jeden zasób. Test będzie polegał na stworzeniu zasobu (np. pula kluczy do jakiegoś pomieszczenia) oraz wątków, które chcą ten zasób uzyskać (np. ludzie chcący wejść do tego pomieszczenia).

11. Implementacja oraz wyniki

```
class CountingSemaphore {
    private int count;
    private final BinarySemaphore countAccess;
    private final BinarySemaphore canDecrease;

    public CountingSemaphore(int count) {
        this.countAccess = new BinarySemaphore();
        this.canDecrease = new BinarySemaphore();
        this.count = count;
    }

    public void P() { //acquire
        this.canDecrease.P();

        this.countAccess.P();

        this.count--;
        if(this.count > 0){
            this.canDecrease.V();
        }

        this.countAccess.V();
    }

    public void V() { //release
        this.countAccess.P();

        this.count++;
        if(this.count >= 1){
            this.canDecrease.V();
        }

        this.countAccess.V();
    }
}

class KeyThread extends Thread {
    private final SharedResource sr;

    KeyThread(SharedResource sr) {
```

```
        this.sr = sr;
    }

    public void run() {
        sr.getKey();
    }
}

class Main2 {
    public static void main(String[] args) {
        int keysNum = 3;
        SharedResource sr = new SharedResource(keysNum);

        var threadList = new ArrayList<KeyThread>();
        IntStream.range(0, 10).forEach(x -> threadList.add(new KeyThread(sr)));
        threadList.forEach(Thread::start);

        threadList.forEach(t -> {
            try{
                t.join();
            }catch (Exception e){
                e.printStackTrace();
            }
        });
    }
}
```

Wynik wykonania programu:

```
Thread 18 acquired key.
Thread 19 acquired key.
Thread 15 acquired key.
Thread 15 returned key.
Thread 16 acquired key.
Thread 19 returned key.
Thread 17 acquired key.
Thread 18 returned key.
Thread 20 acquired key.
Thread 16 returned key.
Thread 21 acquired key.
Thread 17 returned key.
Thread 24 acquired key.
Thread 21 returned key.
```

```
Thread 22 acquired key.
Thread 20 returned key.
Thread 23 acquired key.
Thread 24 returned key.
Thread 23 returned key.
Thread 22 returned key.
```

Powyższy listing pokazuje, że mając zasób w ilości trzech kluczy, każdy wątek czeka na zwrot klucza jeśli aktualnie nie ma żadnego dostępnego klucza.

Semafor binarny jest specjalnym przypadkiem semafora licznikowego, gdyż gwarantuje on, że tylko jeden wątek ma dostęp do zasobu, czyli liczba zasobów jest równa jeden. W związku z tym, semafor licznikowy może zostać użyty jako semafor binarny.