6. Semafory

Semafory to w uproszczeniu liczniki umożliwiające dostęp do sekcji krytycznej aplikacji jednemu lub wielu wątkom. Zasada działania semafora jest następująca:

- jeśli wartość licznika jest większa od zera wątek może uzyskać dostęp do sekcji krytycznej (licznik jest zmniejszany o jeden)
- jeśli wartość licznika jest równa zero wątek czeka na zwiększenie się jego wartości (jest to realizowane poprzez zwiększenie wartości licznika przez wątek opuszczający sekcję krytyczną).

Główne zastosowanie semaforów to:

- ograniczenie liczby odwołań do pamięci masowych,
- ograniczenie połączeń z serwerami baz danych,
- ograniczenie połączeń sieciowych,
- ograniczenie dostępu do wspólnych zasobów wykorzystywanych przez aplikację.

Ideę semaforów można porównać z semaforami występującymi na kolei:

- wartość zerowa semafora oznacza brak możliwości przejazdu (semafor opuszczony – czekanie na pozwolenie),
- wartości większe od zera oznaczają możliwość przejazdu (semafor podniesiony).

Instrukcje atomowe

W aplikacjach współbieżnych dość często wykorzystuje się instrukcje zwiększające bądź też zmniejszające wartość zmiennych. Instrukcje takie nie zapewniają jednak bezpieczeństwa wykonywanych operacji. Przykładowa klasa `Licznik` wykorzystuje do zwiększenia wartości zmiennej instrukcję `wartosc++`. Operacja ta składa się z kilku kroków, odczytu aktualnej wartości, zwiększenia jej oraz zapisu. W przypadku dostępu do niej poprzez wielu wątków istnieje niebezpieczeństwo błędu.

```
class Licznik {
    private int wartosc;
    public int pobierzWartosc() {
        return wartosc;
    }
    public int zwiekszWartosc() {
        return wartosc++;
    }
}
```

W języku Java wprowadzono instrukcje atomowe. Pierwszym sposobem na wykonywanie operacji na zmiennych w sposób bezpieczny było użycie słowa kluczowego `volatile`. Niestety „atomowość" polega tylko na operacji odczytu lub zapisu, co nie chroni przed błędami. Od wersji 1.5 wprowadzone zostały już pełnoprawne instrukcje atomowe. Pozwalają one na wykonywanie prostych operacji na zmiennych bez obawy o ich nieprawidłowe rezultaty. Wszystkie instrukcje atomowe zgrupowano w pakiecie: `java.util.concurrent.atomic`. W skład pakietu wchodzą takie klasy jak:

- `AtomicInteger`
- `AtomicLong`
- `AtomicBoolean`
- `AtomicReference`

Podstawowe operacje, jakie można przeprowadzać na zmiennych to:

- addAndGet()
- incrementAndGet()
- decrementAndGet()
- get()
- getAndSet()

Poniżej poprawiona klasa Licznik wykorzystująca metody z klasy AtomicInteger.

```
import java.util.concurrent.atomic.AtomicInteger;

class Licznik {
    private AtomicInteger wartosc = new AtomicInteger(0);

    public int pobierzWartosc() {
       return wartosc.get();
    }
    public int zwiekszWartosc() {
        return wartosc.incrementAndGet();
    }
}
```

Semafory

Wcześniejsze wersje Javy nie posiadały gotowego mechanizmu semaforów. Ich realizacja była możliwa za pomocą standardowych opcji dostępnych w aplikacjach wielowątkowych (metody wait(), notify() czy też notifyAll()). Od wersji 1.5 Java umożliwia wykorzystanie klasy Semaphore będącej częścią pakietu java.util.concurrent. Podstawowe konstruktory klasy Semaphore umożliwiają utworzenie semaforów o z góry zdefiniowanej liczbie pozwoleń. Pozwolenia to nic innego jak wartość licznika wykorzystywanego do pracy z wątkami (analogia do kolei). Utworzenie semafora wygląda w następujący sposób:

```
Semaphore semafor = new Semaphore(1);
```

Wartość jeden oznacza, iż semafor jest podniesiony. Wątek, który będzie chciał uzyskać dostęp do sekcji krytycznej aplikacji zostanie zaakceptowany. Wejście do kodu krytycznego odbywa się poprzez metodę acquire().

```
semafor.acquire();
```

Powoduje ona zmniejszenie wartości licznika semafora (w naszym przypadku semafor będzie miał wartość zero (semafor opuszczony). Jeśli nie ma możliwości uzyskania pozwolenia (semafor opuszczony), wątek wstrzymuje swoje działanie i czeka na pozwolenie. Opuszczając sekcję krytyczną wątek musi zwolnić semafor (podnieść go). Odbywa się to za pomocą metody release()(zwiększenie wartości licznika semafora o jeden) .

```
semafor.release();
```

Inne przydatne metody to między innymi:

- tryAcquire() – próbuje uzyskać pozwolenie, jeśli nie to zwraca fałsz,
- acquire(int) – próbuje uzyskać określoną liczbę pozowoleń,
- availablePermits() – zwraca aktualną liczbę pozwoleń,
- getQueueLength() – zwraca liczbę wątków czekających na pozwolenie.

Poniżej przykładowy kod wykorzystujący semafory.

```
import java.util.concurrent.Semaphore;

public class TestSemaforow {

    public static void main(String[] args) {

        for(int i = 1; i <= 10; i++) {
            new Goscie("G"+i).start();
            try {
                Thread.sleep(100);
            } catch(InterruptedException e) {
                System.out.println("Błąd");
            }
        }

    }
}

class Goscie extends Thread {

    private static final Semaphore semafor = new Semaphore(2, true);
    private String nazwa;

    public Goscie(String nazwa) {
        this.nazwa = nazwa;
        System.out.println("Gość " + nazwa + " przybył do hotelu. Aktualna dostępność
pokoi: " + semafor.availablePermits() + " Aktualna kolejka: " +
semafor.getQueueLength());
    }

    @Override
    public void run() {
            try {
                semafor.acquire();
                System.out.println("Gość: " + nazwa + " zajął pokój");
                sleep((long)(Math.random()*10000));

            } catch (InterruptedException e) {
                System.out.println("Błąd");
            } finally {
                System.out.println("Gość: " + nazwa + " opuścił pokój");
                semafor.release();
            }
    }
}
```

Dodatkowy parametr w konstruktorze umożliwia zakolejkowanie wszystkich oczekujących wątków do kolejki FIFO. Rezultat działania aplikacji poniżej.

```
Gość G1 przybył do hotelu. Aktualna dostępność pokoi: 2 Aktualna kolejka: 0
Gość: G1 zajął pokój
Gość G2 przybył do hotelu. Aktualna dostępność pokoi: 1 Aktualna kolejka: 0
Gość: G2 zajął pokój
Gość G3 przybył do hotelu. Aktualna dostępność pokoi: 0 Aktualna kolejka: 0
Gość G4 przybył do hotelu. Aktualna dostępność pokoi: 0 Aktualna kolejka: 1
Gość G5 przybył do hotelu. Aktualna dostępność pokoi: 0 Aktualna kolejka: 2
Gość G6 przybył do hotelu. Aktualna dostępność pokoi: 0 Aktualna kolejka: 3
Gość G7 przybył do hotelu. Aktualna dostępność pokoi: 0 Aktualna kolejka: 4
Gość G8 przybył do hotelu. Aktualna dostępność pokoi: 0 Aktualna kolejka: 5
Gość G9 przybył do hotelu. Aktualna dostępność pokoi: 0 Aktualna kolejka: 6
Gość G10 przybył do hotelu. Aktualna dostępność pokoi: 0 Aktualna kolejka: 7
Gość: G2 opuścił pokój
Gość: G3 zajął pokój
Gość: G1 opuścił pokój
Gość: G4 zajął pokój
Gość: G4 opuścił pokój
Gość: G5 zajął pokój
```

```
Gość: G5 opuścił pokój
Gość: G6 zajął pokój
Gość: G6 opuścił pokój
Gość: G7 zajął pokój
Gość: G3 opuścił pokój
Gość: G8 zajął pokój
Gość: G8 opuścił pokój
Gość: G9 zajął pokój
Gość: G7 opuścił pokój
Gość: G10 zajął pokój
Gość: G9 opuścił pokój
Gość: G10 opuścił pokój
```

Zadania

1. Adam i Ania mieszkają obok siebie i mają dostęp do wspólnego ogrodu. Adam jest właścicielem psa, a Ania kota. Zwierzęta nie przepadają za sobą. Napisz, z wykorzystaniem semaforów, aplikację, która utworzy dwa wątki Adama i Ani oraz umożliwi dostęp do ogrodu ich zwierzakom. Uwzględnij następujące założenia:

   - jeśli ogród jest pusty to każde ze zwierząt może zostać wyprowadzone na spacer,
   - jeśli w ogrodzie jest kot, to pies nie może do niego wejść,
   - jeśli w ogrodzie jest pies, to kot nie może do niego wejść,
   - każde ze zwierząt przebywa w ogrodzie przez określony czas (wartość losowa).

   Wykorzystaj semafor o liczbie pozwoleń równej jeden.

2. Zasymuluj parking samochodowy. Parking udostępnia 20 miejsc parkingowych, samochody przyjeżdżają na parking co jakiś czas (wartość losowa), jeśli są wolne miejsca wjeżdżają na parking i przebywają na nim określoną ilość czasu (wartość losowa). Jeśli miejsc nie ma, czekają w kolejce, aż zwolni się jakieś miejsce. Utwórz 100 samochodów (100 wątków) oraz sam parking. Wykorzystaj semafory. Dodatkowo wątek samochodu ma decydować czy będzie czekał w kolejce czy jedzie na inny parking (prawdopodobieństwo pozostania w kolejce ustaw na 75%). Wykorzystaj metodę `tryAcquire()` zwracającą wartość logiczną w zależności od dostępności pozwoleń w semaforze. Utwórz również wątek demona, który co 5 sekund będzie podawał przydatne informacje dla kierowców (liczba wolnych miejsc/liczba samochodów w kolejce).

3. Mamy trzy wątki, których zadaniem jest wyświetlanie liter.

   - wątek A wyświetla, co jakiś czas (wartość losowa) literę A,
   - wątek B wyświetla, co jakiś czas (wartość losowa) literę B,
   - wątek C wyświetla, co jakiś czas (wartość losowa) literę C.

   Wykorzystując trzy semafory doprowadź do takiej sytuacji by pojawiające się litery były zawsze we właściwej kolejności ABC.

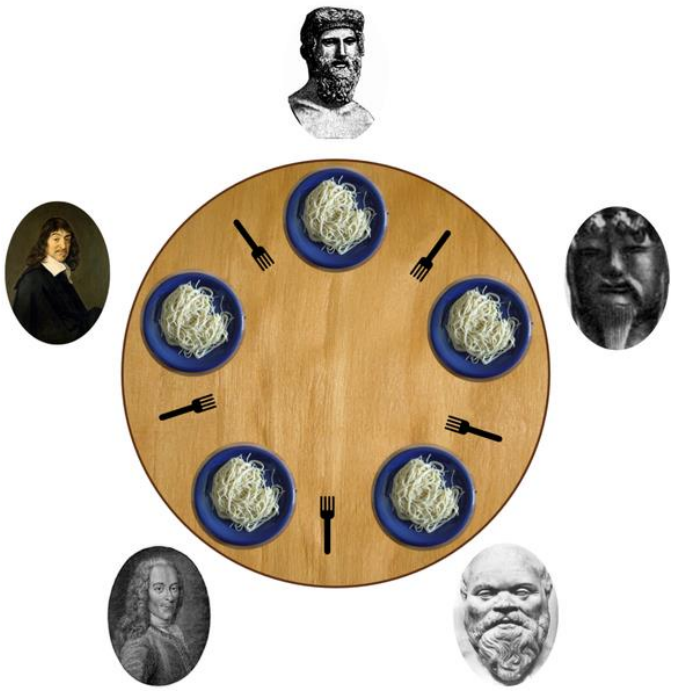
4. *„Pięciu filozofów siedzi przy stole i każdy wykonuje jedną z dwóch czynności – albo je, albo rozmyśla. Stół jest okrągły, przed każdym z nich znajduje się miska ze spaghetti, a pomiędzy każdą sąsiadującą parą filozofów leży widelec, a więc każda osoba ma przy sobie dwie sztuki - po swojej lewej i prawej stronie. Ponieważ jedzenie potrawy jest trudne przy użyciu jednego widelca, zakłada się, że każdy filozof korzysta z dwóch. Dodatkowo nie ma możliwości skorzystania z widelca, który nie znajduje się bezpośrednio przed daną osobą."* (źródło: http://pl.wikipedia.org/wiki/Problem_ucztujących_filozofów)

Zapoznaj się z pełnym opisem powyższego zadania oraz problemami, które są z nim związane:

- wzajemne wykluczanie (dwóch filozofów nie może podnieść tego samego widelca),
- zakleszczenie (każdy z filozofów podnosi widelec po lewej stronie).

Dodatkowo żaden filozof nie powinien zostać zagłodzony.

Napisz z wykorzystaniem semaforów aplikację rozwiązującą problem ucztujących filozofów. Sprawdź jej poprawność.



Źródło: Benjamin D. Esham / Wikimedia Commons [CC-BY-SA-3.0]