

Laboratorium II

Wyścig - wyjaśnienie

Dominik Marek

14 października 2024



**AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE**

1. Zadania

1. Zaimplementować semafor binarny za pomocą metod wait i notify, użyć go do synchronizacji programu Wyścig
2. Pokazać, że do implementacji semafora za pomocą metod wait i notify nie wystarczy instrukcja if tylko potrzeba użyć while. Wyjaśnić teoretycznie dlaczego i potwierdzić eksperymentem w praktyce. (wskazówka: rozważyć dwie kolejki: czekająca na wejście do monitora obiektu oraz kolejkę związana z instrukcją wait, rozważyć kto kiedy jest budzony i kiedy następuje wyścig).
3. Zaimplementować semafor licznikowy (ogólny) za pomocą semaforów binarnych. Czy semafor binarny jest szczególnym przypadkiem semafora ogólnego ?

2. Rozwiązania

1.

Implementację semafora binarnego rozpocząłem od stworzenia atrybutu `semaphoreState`, ustawianego w konstruktorze klasy, który reprezentuje stan semafora. Jeśli jest ustawiony na `true` oznacza to, iż semafor jest dostępny a jeśli na `false` to, że jest zablokowany. Następnie w metodzie `P()` odpowiedzialnej za operację blokującą dopóki semafor jest zablokowany wywoływana jest metoda `wait()`, która wprowadza wątek w stan oczekiwania (wątek pozostanie w stanie oczekiwania do momentu aż inny wątek wywoła metodę `notify()`). Gdy semafor jest dostępny (czyli `semaphoreState == true`), stan semafora zostaje ustawiony na `false`, co oznacza, że semafor zostaje zablokowany przez bieżący wątek. Przechodząc do metody `V()`, odpowiadającej za zwolnienie semafora, ustawiam w niej `semaphoreState` na `true` co oznacza, że semafor jest dostępny a następnie za pomocą metody `notify()` wybudzam jeden z wątków oczekujących na dostęp do semafora. W klasach `IThread` i `DThread` dodaje jako atrybut wyżej opisany semafor binarny który ustawiany jest w konstruktorach klas. Kolejno w obu klasach w metodzie `run` przed wywołanie metody inkrementacji bądź dekrementacji licznika wołam metodę `P()` na semaforze, a po nich metodę `V()`.

```
class BinarySemaphore {
    private boolean semaphoreState;

    public BinarySemaphore(boolean state) {
        this.semaphoreState = state;
    }

    public synchronized void P() {
        while(!semaphoreState) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        semaphoreState = false;
    }

    public synchronized void V() {
        semaphoreState = true;
        notify();
    }
}
```

```
class IThread extends Thread {
    private Counter _cnt;
    private BinarySemaphore binSem;
    public IThread(Counter c, BinarySemaphore binSem) {
        this._cnt = c;
        this.binSem = binSem;
    }
    public void run() {
        for (int i = 0; i < 100_000; ++i) {
            try { this.sleep(50); }
            catch (Exception e) {}
            this.binSem.P();
        }
    }
}
```

```

        _cnt.inc();
        this.binSem.V();
    }
}

class DThread extends Thread {
    private Counter _cnt;
    private BinarySemaphore binSem;

    public DThread(Counter c, BinarySemaphore binSem) {
        this._cnt = c;
        this.binSem = binSem;
    }

    public void run() {
        for (int i = 0; i < 100_000; ++i) {
            this.binSem.P();
            _cnt.dec();
            this.binSem.V();
        }
        try { this.sleep(1); }
        catch (Exception e) {}
    }
}

class Race2 {
    public static void main(String[] args) {
        Counter cnt = new Counter(0);
        BinarySemaphore binSem = new BinarySemaphore(true);

        IThread it = new IThread(cnt, binSem);
        DThread dt = new DThread(cnt, binSem);

        it.start();
        dt.start();

        try {
            it.join();
            dt.join();
        } catch (InterruptedException ie) {}

        System.out.println("value=" + cnt.value());
    }
}

```

Dla pięciu kolejnych uruchomień powyższego kodu otrzymaliśmy oczekiwany finalny stan licznika wynoszący 0.

Wnioski:

Zastosowanie nawet tak prostego mechanizmu jak semafora binarny sprawia, że tylko jeden wątek na raz będzie stanie wejść do sekcji krytycznej (operacja inkrementacji/dekrementacji). Dzięki temu unikniemy zjawiska race condition, a tym samym otrzymamy deterministyczny program zwracający oczekiwany wynik.

2.

Rozważmy dwie kolejki:

- Kolejkę wątków czekających na wejście do monitora, w której znajdują się wątki próbujące uzyskać dostęp do monitora (czyli zablokować obiekt za pomocą `synchronized`). Wątki te czekają na zwolnienie zamka przez inny wątek.
- Kolejka związana z instrukcją `wait`, w której znajdują się wątki, które wywołały metodę `wait()`. Te wątki czekają, aż zostaną wybudzone za pomocą `notify` lub `notifyAll()`. Dopiero po wybudzeniu będą mogły spróbować ponownie uzyskać zamek monitora.

Gdy wątek napotka na blokadę (np. semafor jest w stanie zajęty), wywołuje metodę `wait()`, zwalniając tym samym monitor (czyli zamek obiektu) i przechodząc do kolejki `wait`, gdzie czeka na sygnał `notify()`. Kiedy wątek zostanie wybudzony przez `notify()`, trafia z powrotem do kolejki wejściowej monitora, czekając na ponowne zdobycie zamka, aby kontynuować swoją pracę.

Użycie instrukcji `if` sprawia, że wątek sprawdza stan semafora tylko raz. Jeśli zostanie wybudzony przez `notify()`, założy, że stan semafora jest odpowiedni do kontynuowania. Jednak zanim wątek zdobędzie monitor, inny wątek może uzyskać dostęp do zasobu, co prowadzi do wyścigu.

Instrukcja `while` wymusza ponowne sprawdzenie warunku po każdorazowym wybudzeniu wątku. Dzięki temu wątek upewnia się, że stan semafora faktycznie pozwala mu na dalsze działanie. Jeśli nie, wątek ponownie wywoła `wait()` i wróci do kolejki czekających. Dzięki temu eliminujemy potencjalne wyścigi pomiędzy wątkami, ponieważ po wybudzeniu wątek nie zakłada, że może kontynuować, dopóki rzeczywisty stan semafora tego nie potwierdzi.

Eksperyment:

Dla implementacji semafora z wykorzystaniem instrukcji `if` oraz `while` wykonano dziesięć kolejnych uruchomień programu `Race`. Tak otrzymane wyniki zostały zaprezentowane w poniższej tabeli.

<i>Numer wywołania programu</i>	<i>Wynik programu z semaforem zaimplementowanym z użyciem instrukcji if</i>	<i>Wynik programu z semaforem zaimplementowanym z użyciem instrukcji while</i>
1	-7	0
2	-10	0
3	6	0
4	9	0
5	-1	0
6	-1	0
7	-11	0
8	-3	0
9	15	0
10	0	0

Analizując powyższe wyniki, można zauważyć, że program, wykorzystujący semafor zaimplementowany z użyciem instrukcji `if`, jest niedeterministyczny, gdyż tylko raz na dziesięć przypadków otrzymaliśmy wynik zgodny z przewidywaniami. W przypadku zastosowania instrukcji `while` przy implementacji semafora każde z dziesięciu wywołań programu dało oczekiwany wynik równy zero.

Wnioski:

Korzystanie z `while` zamiast `if` w implementacji semaforów z `wait()` i `notify()` jest kluczowe dla zapewnienia poprawnej synchronizacji wątków i zapobieganiu wystąpienia zjawiska *race condition*. Pętla `while` zapewnia, że warunek dostępu do zasobu jest zawsze sprawdzany po wybudzeniu wątku, co eliminuje potencjalne błędy wynikające z uzyskania przez więcej niż jeden wątek jednoczesnego dostępu do sekcji krytycznej.

3.

Do implementacji semafora licznikowego który wykorzystuje dwa semaforey binarne: jeden do synchronizacji dostępu do zasobu (`mutex`), a drugi do blokowania operacji w przypadku braku dostępnych zasobów.

Początkowo tworzę atrybuty:

- **resourcesCount** – reprezentujący liczbę zasobów dostępnych do współdzielenia.
- **mutex** – semafor binarny początkowo odblokowany
- **lock** – semafor binarny początkowo zablokowany
-

W metodzie `P()` odpowiedzialnej za operację zablokowania zasobu na początku, operacja `P` na `mutexie` (`mutex.P()`) zapewnia, że dostęp do zmiennej `resourcesCount` jest zablokowany dla innych wątków, aby uniknąć współbieżnych modyfikacji. Jeśli `resourcesCount` wynosi 0 (brak zasobów), wątek zwalnia `mutex` (`mutex.V()`) i czeka na zasób, blokując się na semaforze `lock.P()` (aż jakiś inny wątek zwolni zasób poprzez operację `V()`). Jeśli są dostępne zasoby zmniejszamy liczbę dostępnych zasobów (`resourcesCount--`) i zwalniamy `mutex` (`mutex.V()`), umożliwiając innym wątkom działanie.

Przechodząc do metody `V()` (operacja zwolnienia zasobu) najpierw, operacja `P` na `mutexie` (`mutex.P()`) zapewnia wyłączny dostęp do zmiennej `resourcesCount`. Następnie zwiększamy liczbę dostępnych zasobów (`resourcesCount++`). Jeśli liczba zasobów po tej operacji wynosi 1 (czyli przed zwolnieniem była równa 0), wątek budzi jeden z oczekujących wątków, wywołując operację `lock.V()`. Finalnie zwalniamy `mutex` (`mutex.V()`), umożliwiając innym wątkom dostęp do zasobów.

Poniżej została przedstawiona wyżej opisana implementacja semafora licznikowego (ogólnego) w języku Java:

```

public class CountingSemaphore {
    int resourcesCount;
    final BinarySemaphore mutex = new BinarySemaphore(true) ;
    final BinarySemaphore lock = new BinarySemaphore(false) ;

    public CountingSemaphore(int initialValue) {
        this.resourcesCount = initialValue;
    }

    public void P(){
        this.mutex.P();
        if(this.resourcesCount == 0){
            this.mutex.V();
            this.lock.P();
        }else{
            this.resourcesCount--;
            this.mutex.V();
        }
    }

    public void V(){
        this.mutex.P();
        this.resourcesCount++;
        if(this.resourcesCount == 1){
            this.lock.V();
        }
        this.mutex.V();
    }
}

```

Semafor ogólny (licznikowy) może przyjmować dowolne wartości całkowite nieujemne. Liczba ta reprezentuje liczbę dostępnych zasobów, które mogą być uzyskiwane przez wątki, zatem możemy uznać semafor binarny za specjalny przypadek semafora ogólnego, który może przyjmować tylko dwie wartości reprezentujące stan semafora 0 (zablokowany) i 1 (odblokowany).

3. Bibliografia

- https://www.dz5.pl/ti/java/java_skladnia.pdf ~Jacek Rumiński, Język Java – rozdział o wątkach
- <http://brinch-hansen.net/papers/1999b.pdf> ~ Per Brinch Hansen
- <https://www.artima.com/insidejvm/ed2/threadsynch.html> ~Bill Venners, Inside the Java Virtual Machine - Chapter 20 Thread Synchronization
- <https://docs.oracle.com/en/java/javase/19/index.html>
- [https://en.wikipedia.org/wiki/Semaphore_\(programming\)](https://en.wikipedia.org/wiki/Semaphore_(programming))