Laboratorium 2 - Teoria współbieżności

Piotr Karamon

21.10.2024r.

Treści zadań

Zadanie 1

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

Zadanie 2

Pokazać, ze 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).

Zadanie 3

Zaimplementować semafor licznikowy (ogólny) za pomocą semaforów binarnych. Czy semafor binarny jest szczególnym przypadkiem semafora ogólnego?

Zadanie 1

Implementujemy dwie metody semafora binarnego:

- `P()`: próbuje uzyskać dostęp do zasobu. Jeśli zasób jest zajęty (`isAvailable` jest `false`), wątek zwiększa licznik `waiting` i czeka (`wait()`). Gdy zasób staje się dostępny, wątek zmniejsza licznik `waiting` i ustawia `isAvailable` na `false`.

- `V()`: zwalnia zasób. Ustawia `isAvailable` na `true` i powiadamia (`notify()`) jeden z oczekujących wątków, jeśli taki istnieje (`waiting > 0`).

```
class BinarySemaphore {
    private boolean isAvailable = true;
    private int waiting = 0;

    public BinarySemaphore() {}

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

    }

    public synchronized void V() {
        isAvailable = true;
        if (waiting > 0) {
            notify();
        }
    }
}
```

Aby zsynchronizować program Wyścig, do klasy każdego wątku podajemy semafora przez konstruktor, następnie w metodzie `run()` oba wątki, używają semafora w celu synchronizacji.

Klasa `IThread`:

```
class IThread extends Thread {
    private final Counter _cnt;
    private final BinarySemaphore _sem;

    public IThread(Counter c, BinarySemaphore s) {
        _cnt = c;
        _sem = s;
    }
    public void run() {
        for (int i = 0; i < 100000000; ++i) {
            _sem.P();
```

```
            _cnt.inc();
            _sem.V();
        }
    }
}
```

Klasa `DThread`:

```
class DThread extends Thread {
    private final Counter _cnt;
    private final BinarySemaphore _sem;

    public DThread(Counter c, BinarySemaphore s) {
        _cnt = c;
        _sem = s;
    }

    public void run() {
        for (int i = 0; i < 100000000; ++i) {
            _sem.P();
            _cnt.dec();
            _sem.V();
        }
    }
}
```

W metodzie `main` podajemy semafora do wątków, uruchamiamy je, czekamy aż się zakończą i sprawdzamy finalny wynik.

```
public static void main(String[] args) {
Counter cnt = new Counter(0);

BinarySemaphore sem = new BinarySemaphore();

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

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

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

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

Wynik działania programu:

```
value=0
```

Wnioski: Stworzony przez nas semafor binarny może być użyty w celu synchronizacji działania wielu wątków, w celu wyeliminowania wyścigu.

Zadanie 2

Aby pokazać, że użycie instrukcji `if` nie wystarczy do implementacji semafora zademonstrujemy przykład pokazujący wyścig w takiej sytuacji.

Mamy trzy wątki:

nazwa	stan
a	jest uśpiony w `P()` (wait set)
b	będzie chciał wywołać `P()`
c	obecnie wywołuje metodę `V()` (posiada monitor)

Rozważmy następujący przeplot:

1. b chce wywołać metodę `P()`, ale obecnie c posiada monitor, wątek b trafia więc do kolejki monitora(entry set).

2. c w metodzie `V()` wywołuje `notify`, co sprawia, że jedyny wątek w wait set czyli wątek a jest oznaczony do wykonania. Wątek kończy metodę i zwalnia monitor.

3. b jest w entry set, a w wait set. Zająć monitor może którykolwiek z nich. b zajmuje monitor, wywołuje metodę `P()`, ustawiając `isAvailable=false` i ostatecznie zwalnia monitor.

4. Przez oznaczenie wątku a w kroku 2. wątek zajmuje monitor, ale przez brak pętli `while`, wątek ten ustawia `isAvailable=false` i zaczyna wykonywać swoją sekcję krytyczną.

Jak widać mamy tutaj do czynienia z sytuacją absolutnie niedopuszczalną, bo wątki b i a myślą, że mają ekskluzywny dostęp do danego zasobu. Gdybyśmy zastąpili `if` pętlą `while` wątek a po wybudzeniu, ponownie sprawdziłby warunek i znów trafił do wait set.

Aby pokazać, że wymagana jest w implementacji pętla `while`, w metodzie `P()` zamieniamy instrukcję `while` na `if`. Następnie w metodzie `main` tworzymy cztery wątki w celu zwiększenia szans wyścigu:

```
public static void main(String[] args) {
    Counter cnt = new Counter(0);

    BinarySemaphore sem = new BinarySemaphore();

    List<Thread> threads = List.of(
        new IThread(cnt, sem),
        new DThread(cnt, sem),
        new IThread(cnt, sem),
        new DThread(cnt, sem)
    );


    threads.forEach(Thread::start);

    try {
        for (Thread t : threads) {
            t.join();
        }
    } catch(InterruptedException ie) { }

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

Wynik programu:

```
value=-689
```

Jak widzimy wynik odbiega od poprawnego zera.

Wnioski: Przy implementacji niskopoziomowych narzędzi synchronizacji musi być bardzo uważni. Wymóg zastosowania pętli `while` na pierwszy rzut oka wcale nie musi być oczywisty. Jego konieczność wynika z zasady działania monitora w języku Java. Implementując zatem takie narzędzia musimy być niezwykle pewni tego na jakich innych narzędziach je bazujemy.

Zadanie 3

Aby zaimplementować semafor licznikowy wykorzystamy zmienną `counter` oraz dwa semafory:

- `mutex` - chroni zmienną `counter`
- `gate` - ten semafor jest podniesiony gdy `counter > 0`, czyli wtedy gdy chociaż jeden zasób jest nadal dostępny.

```
class CountingSemaphore {
    private final BinarySemaphore gate;
    private final BinarySemaphore mutex;
    private int counter;

    public CountingSemaphore(int n) {
        mutex = new BinarySemaphore();
        gate = new BinarySemaphore();
        counter = n;
        if (counter == 0) {
            gate.P();
        }
    }

    public void P() {
        gate.P();
        mutex.P();
        counter--;
        if (counter > 0) {
            gate.V();
        }
        mutex.V();
    }

    public void V() {
        mutex.P();
        counter++;
        if (counter >= 1) {
            gate.V();
        }
        mutex.V();
    }
}
```

Aby zademonstrować działanie tej klasy zasymulujemy jedno z częstych zastosowań dla takiego semafora. Ograniczymy ilość otwartych gniazd przez nasz program, z racji tego, że ta liczba w systemie ma najczęściej swoją górną granicę. Przy naiwnym programie typu web-scraper, bez kontroli ilości otwartych gniazd moglibyśmy przekroczyć ten limit co skutkowałoby wyjątkami, system odmówił by utworzenia nowego gniazda.

Kod wątku który symuluje otwarcie gniazda:

```
class OpenNetworkSocketThread extends Thread {
    private final CountingSemaphore _sem;
    private final int id;

    public OpenNetworkSocketThread(int id, CountingSemaphore s) {
```

```
        this.id = id;
        _sem = s;
    }

    public void run() {
        _sem.P();
        System.out.println("Opened network socket in " + id);
        try {
            Thread.sleep(ThreadLocalRandom.current().nextInt(100, 300));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Closing network socket in " + id);
        _sem.V();
    }
}
```

Metoda `main`.

```
public static void main(String[] args) {
    var sem = new CountingSemaphore(3);
    List<OpenNetworkSocketThread> threads =
        IntStream.range(0, 10)
            .mapToObj(i -> new OpenNetworkSocketThread(i, sem))
            .toList();

    theads.forEach(Thread::start);
    threads.forEach(t -> {
        try {
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
}
```

Wynik programu:

```
Opened network socket in 1
Opened network socket in 0
Opened network socket in 2
Closing network socket in 1
Opened network socket in 3
Closing network socket in 2
Opened network socket in 4
Closing network socket in 0
Opened network socket in 6
Closing network socket in 3
Opened network socket in 5
Closing network socket in 4
Opened network socket in 7
Closing network socket in 6
Opened network socket in 8
Closing network socket in 5
Opened network socket in 9
Closing network socket in 7
Closing network socket in 8
Closing network socket in 9
```

Jak widać stworzony przez nas semafor poprawnie umożliwia utworzenie jedynie trzech gniazd w danym momencie.

Semafor binarny jest szczególnym przypadkiem semafora licznikowego. Semafor licznikowy zachowuje się tak samo jak semafor binarny gdy liczba zasobów jest równa jeden. Nie oznacza to jednak, że zawsze powinniśmy korzystać z semaforów licznikowych skoro są ogólniejsze od semaforów binarnych. Semafor binarny ma prostszą implementację(licznikowy składa się właśnie z dwóch takich semaforów), przez to jeżeli potrzebujemy zwykłego mutex'a to powinniśmy właśnie użyć semafora binarnego.

Bibliografia

- Bill Venners: *Inside the Java Virtual Machine Chapter 20*