Lab02

Adrian Madej

9.10.2023

1. Treść zadań

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

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: czekającą na wejście do monitora obiektu oraz kolejkę związaną z instrukcja wait , rozwazyc 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. Rozwiązywanie zadań

1. Semafor binarny

Na początku implementujemy Semafor binarny używając metod wait() oraz notify()

```
class BinarySemaphore {
    private boolean _stan = true;
    private int _czeka = 0;

    public BinarySemaphore() {
    }

    public synchronized void P()  {
        while (!_stan) {
            try {
                _czeka++;
```

```
                wait();
                _czeka--;
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        _stan = false;
    }

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

Następnie wykorzystujemy go do synchronizacji programu Wyścig. W tym celu zmodyfikujemy klasę Counter

```
class Counter {
    private int _val;
    private final BinarySemaphore binarySemaphore = new
BinarySemaphore();
    public Counter(int n) {
        _val = n;
    }
    public void inc() {
        binarySemaphore.P();
        _val++;
        binarySemaphore.V();
    }
    public void dec() {
        binarySemaphore.P();
        _val--;
        binarySemaphore.V();
    }
    public int value() {
        return _val;
    }
}
```

Klasy DThread, IThread oraz Race2 pozostają bez zmian i wyglądają następująco

Klasa DThread

```
class DThread extends Thread {
    private Counter _cnt;
    public DThread(Counter c) {
```

```
        _cnt = c;
    }
    public void run() {
        for (int i = 0; i < 100000000; ++i) {
            _cnt.dec();
//      try { this.sleep(1); }
//        catch(Exception e) {}
        }
    }
}
```

Klasa IThread

```
class IThread extends Thread {
    private Counter _cnt;
    public IThread(Counter c) {
        _cnt = c;
    }
    public void run() {
        for (int i = 0; i < 100000000; ++i) {
//      try { this.sleep(50); }
//        catch(Exception e) {}
            _cnt.inc();
        }
    }
}
```

Klasa Race2

```
class Race2 {
    public static void main(String[] args) {
        Counter cnt = new Counter(0);
        IThread it = new IThread(cnt);
        DThread dt = new DThread(cnt);

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

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

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

2. Poprawność implementacji semafora

Wyjaśnijmy dlaczego do implementacji semafora za pomocą metod wait i notify nie wystarczy instrukcja if tylko potrzeba użyć while. Zauważmy, że **kolejka czekająca na wejście do monitora** to struktura, która przechowuje wątki oczekujące na dostęp do monitora, zarządzająca ich kolejnością zapewniając tym samym bezpieczny dostęp do wspólnych zasobów. Wątki które wywołały metodę wait() oczekują w **kolejce wait** na wybudzenie za pomocą notify() lub notifyAll(). Zatem dlaczego używamy while a nie if? Głównym powodem jest tzw. „fałszywe wybudzenie" (**Spurious Wakeup**) czyli wybudzenie wątku bez metody notfiy(). Może ono wystąpić na przykład na skutek optymalizacji przez JVM czy też system operacyjny. Wówczas używając jedynie warunku if, program po fałszywym wybudzeniu będzie kontynuował swoje działanie, po mimo nie spełnienia określonego warunku, co może prowadzić do wyścigu (**Race condition**). Używając pętli while, program ponownie sprawdzi czy warunek został spełniony zapobiegając wyścigowi.

Wykonajmy zatem test w praktyce. Zastąpmy instrukcję while instrukcją if i porównajmy wyniki.

Klasa BinarySemaphore po modyfikacji

```
class BinarySemaphore {
    private boolean _stan = true;
    private int _czeka = 0;

    public BinarySemaphore() {
    }

    public synchronized void P()  {
        if (!_stan) {
            try {
                _czeka++;
                wait();
                _czeka--;
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        _stan = false;
    }

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

Przed zmianą, wynikiem programu była zawsze wartość value = 0, po zmianie nie jesteśmy w stanie dokładnie przewidzieć końcowej wartości. Na trzy wywołania programu otrzymaliśmy kolejno: 145, 36, -86. Zatem widzimy że zastosowanie while zamiast if ma ogromne znaczenie.

3. Semafor licznikowy

Semafor binarny jest szczególnym przypadkiem semafora ogólnego. Oznacza to, że każdy semafor binarny można traktować jako semafor ogólny, ale z pewnymi ograniczeniami, np.: liczba pozwoleń ograniczona do 2 (0, 1)

Zaimplementujmy semafor licznikowy (ogólny) przy użyciu semafora binarnego

```
public class GeneralSemaphore {
    private final BinarySemaphore mutex;        // dostęp do
metod P i V
    private final BinarySemaphore resource;     // dostęp do
pozwoleń
    private int permits;

    public GeneralSemaphore(int permits){
        if(permits < 0){
            throw new IllegalArgumentException("Permits must >
0.");
        }else{
            mutex = new BinarySemaphore();
            resource = new BinarySemaphore();
            this.permits = permits;
        }
    }

    public void P(){
        resource.P();
        mutex.P();
        permits--;
        if (this.permits > 0) {
            resource.V();
        }
        mutex.V();
    }

    public void V(){
        mutex.P();
        permits++;
        if(permits == 1){
            resource.V();
        }
        mutex.V();
```

```
    }
}
```

W celu przetestowania działania programu utworzyłem nową klasę, która tworzy semafor z 3 pozwoleniami, oraz 5 wątków, które próbują uzyskać dostęp do semafora

```
public class Test {
    public static void main(String[] args) {
        GeneralSemaphore semaphore = new GeneralSemaphore(3);

        for (int i = 1; i <= 5; i++) {
            new Thread(() -> {
                try {
                    semaphore.P();

System.out.println(Thread.currentThread().getName() + "
uzyskał dostęp do zasobu.");
                    Thread.sleep((long) (Math.random() *
2000));

System.out.println(Thread.currentThread().getName() + "
zwalnia dostęp do zasobu.");
                    semaphore.V();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }
    }
}
```

W wyniku działania programu dostajemy na wyjściu:

Thread-2 uzyskał dostęp do zasobu.

Thread-0 uzyskał dostęp do zasobu.

Thread-1 uzyskał dostęp do zasobu.

Thread-0 zwalnia dostęp do zasobu.

Thread-3 uzyskał dostęp do zasobu.

Thread-2 zwalnia dostęp do zasobu.

Thread-4 uzyskał dostęp do zasobu.

Thread-1 zwalnia dostęp do zasobu.

Thread-4 zwalnia dostęp do zasobu.

Thread-3 zwalnia dostęp do zasobu.

Co dowodzi poprawności implementacji.

4. Wnioski

- Semafor binarny jest często wykorzystywany do rozwiązywania problemów synchronizacji między dwoma wątkami lub procesami.
- Semafor ogólny jest używany w bardziej zaawansowanych przypadkach, w których wymagana jest kontrola dostępu do zasobów, które mogą być dostępne w większej liczbie niż 2.
- Semafor ogólny może być zaimplementowany za pomocą semaforów binarnych, co jest przykładem elastycznego podejścia do synchronizacji.
- Użycie pętli while zamiast instrukcji if podczas implementacji semafora binarnego jest niezbędne do zapobiegania tzw. 'fałszywym wybudzeniom'

5. Bibliografia

- Konspekt z laboratorium TW
- Per Brinch Hansen Java Insecure Parallelism.
- Allen B. Downey The Little Book of Semaphores
- A. Udaya Shankar Implementing counting semaphores using binary semaphores
- Dokumentacja Javy