Laboratorium 2

Łukasz Wala
AGH, Wydział Informatyki, Elektroniki i Telekomunikacji
Teoria Współbieżności 2022/23

Kraków, 22 października 2022

1 Treść 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 użyc *while*. Wyjasnić teoretycznie dlaczego i potwierdzic eksperymentem w praktyce. (wskazówka: rozważyc dwie kolejki: czekajaca na wejście do monitora obiektu oraz kolejkę związaną z instrukcją *wait*, rozważyc kto i kiedy jest budzony i kiedy nastepuje wyścig).
3. Zaimplementować semafor licznikowy (ogólny) za pomocą semaforów binarnych. Czy semafor binarny jest szczególnym przypadkiem semafora ogólnego?

2 Semafor binarny

Pierwszym krokiem rozwiązania jest zaimplementowanie semafora binarnego:

```
class BinarySemaphore {
    private boolean state;

    public BinarySemaphore() {
        this.state = true;
    }

    public synchronized void P() {
        while (!state) {
            try {
                wait();
            }
```

```
            catch (InterruptedException e) {
                System.exit(0);
            }
        }

        state = false;
    }

    public synchronized void V() {
        state = true;
        notifyAll();
    }
}
```

Stworzony semafor może zostać użyty do zsynchronizowania zadania z poprzedniego laboratorium:

```
class Counter {
    private int _val;
    public Counter(int n) {
        _val = n;
    }
    public void inc() {
        _val++;
    }
    public void dec() {
        _val--;
    }
    public int value() {
        return _val;
    }
}

class IThread extends Thread {
    private Counter counter;
    private BinarySemaphore semaphore;

    public IThread(Counter counter, BinarySemaphore semaphore) {
        this.counter = counter;
        this.semaphore = semaphore;
    }

    public void run() {
        for (int i=0; i<10_000; ++i) {
            semaphore.P();
            counter.inc();
```

```
            semaphore.V();
        }
    }
}

class DThread extends Thread {
    private Counter counter;
    private BinarySemaphore semaphore;

    public DThread(Counter counter, BinarySemaphore semaphore) {
        this.counter = counter;
        this.semaphore = semaphore;
    }

    public void run() {
        for (int i=0; i<10_000; ++i) {
            semaphore.P();
            counter.dec();
            semaphore.V();
        }
    }
}

public class Race {
    public static void main(String[] args) throws InterruptedException {

        Counter cnt = new Counter(0);
        BinarySemaphore semaphore = new BinarySemaphore();
        DThread dthread = new DThread(cnt, semaphore);
        IThread ithread = new IThread(cnt, semaphore);

        dthread.start();
        ithread.start();
        dthread.join();
        ithread.join();

        System.out.println(cnt.value());
    }
}
```

Tym sposobem możemy skutecznie zaimplementować licznik.

3 Implementacja semafora - dlaczego *while*?

Mogłoby się wydawać, że w powyższej implementacji semafora możnaby zastąpić pętlę *while* wyrażeniem *if*. Niestety jednak w przypadku, gdy zostanie użyty *if*,

implementacja nie będzie poprawnie działać z kilku powodów:

1. wątek może wybudzić się nie będąc wywołanym przez funkcję *notify*, przerwanym itp. (ang. *spurious wakeup*). Co prawda jest to bardzo żadkie, jednek użycie pętli *while* zapewnia, że jeżeli warunek nie będzie spełniony, wątek nie zostanie wybudzony.

2. jeżeli do wybudzenia wątków użyta jest funkcja *notifyAll*, wówczas wszystkie wątki zablokowane przez semafor zostaną odblokowane i zaczną konkurować o to, który wznowi wykonywanie jako pierwszy. Dzięki użyciu pętli *while* wszystkie wątki, oprócz pierwszego, który się wykona, wejdą w stan oczekiwania, co jest pożądanym zachowaniem. Użycie wyrażenia *if* poskutkowałoby odblokowaniem wszystkich wątków.

```
...
public synchronized void P() {
    if (!state) {  // if instead of while
        try {
            wait();
        }
        catch (InterruptedException e) {
            System.exit(0);
        }
    }

    state = false;
}
...
```

Powyższ implementacja nie zapewnia poprawnej synchronizacji.

4 Semafor licznikowy

Za pomocą semafora binarnego można zaimplementować semafor licznikowy:

```
class CountingSemaphore {
    private int count;
    private BinarySemaphore binarySemaphore;

    public CountingSemaphore() {
        this.count = 1;
        this.binarySemaphore = new BinarySemaphore();
    }

    public void P() {
        binarySemaphore.P();
```

```
        synchronized (this) {
            --count;
            if (count > 0) {
                binarySemaphore.V();
            }
        }
    }

    public synchronized void V() {
        ++count;
        if (count == 1) {
            binarySemaphore.V();
        }
    }
}
```

Semafor licznikowy jest typem semafora, który pozwala na jednoczesny dostęp N wątków do współdzielonego stanu/sekcji krytycznej. Jak można zauważyć, semafor binarmy jest semaforem licznikowym dla $N = 1$.

5 Wnioski

Semafory są prostym, ale skutecznym sposobem na ograniczenie dostępu wielu równolegle działających wątków do współdzielonego stanu i uniknięcia problemów z dostępem do sektcji krytycznej/atomicznością operacji. Mogą zostać zaimplementowane za pomocą intrucji *wait* oraz *notify* w języku Java. Semafor licznikowy pozwala na dostęp N wątków do współdzielonej sekcji, nastomiast semafor binarny pozwala na dostęp tylko jednemu wątkowi.

6 Bibliografia

1. Dokumentacja języka Java - docs.oracle.com