Teoria Współbieżności - Sprawozdanie 2

Maciej Brzeżawski

Październik 2024

1 Treść zadania

1. Zaimplementowac semafor binarny za pomoca metod wait i notify, uzyc go do synchronizacji programu Wyscig
2. Pokazac, ze do implementacji semafora za pomoca metod wait i notify nie wystarczy instrukcja if tylko potrzeba uzyc while . Wyjasnic teoretycznie dlaczego i potwierdzic eksperymentem w praktyce. (wskazowka: rozwazyc dwie kolejki: czekajaca na wejscie do monitora obiektu oraz kolejke zwiazana z instrukcja wait , rozwazyc kto kiedy jest budzony i kiedy nastepuje wyscig).
3. Zaimplementowac semafor licznikowy (ogolny) za pomoca semaforow binarnych. Czy semafor binarny jest szczegolnym przypadkiem semafora ogolnego ?

2 Implementacja semafora binarnego za pomocą metod wait i notify, użycie go do synchroniazcji programu Wyscig

2.1 Semafor binarny

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

    public Semafor() {
    }

    // P (wait) - zajmuje semafor
    public synchronized void P() {
        while (!_stan) {  // jeśli semafor zajęty, wątek czeka
            try {
                _czeka++;
                wait();  // wątek zostaje wstrzymany
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                _czeka--;
            }
        }
        _stan = false;  // semafor zajęty po wyjściu z pętli
    }

    // V (signal) - zwalnia semafor
    public synchronized void V() {
        _stan = true;  // semafor staje się dostępny
        notify();
    }
}
```

2.2 Synchronizacja programu Wyscig

```
class Counter {
    private int _val;
    private Semafor semafor;  // dodajemy semafor

    public Counter(int n) {
        _val = n;
        semafor = new Semafor();  // inicjalizujemy semafor
    }

    public void inc() {
        semafor.P();  // zajmujemy semafor przed operacją
        _val++;
        semafor.V();  // zwalniamy semafor po operacji
    }

    public void dec() {
        semafor.P();  // zajmujemy semafor przed operacją
        _val--;
        semafor.V();  // zwalniamy semafor po operacji
    }

    public int value() {
        return _val;
    }
}

class IThread extends Thread {
    private Counter _cnt;
    public IThread(Counter c) {
        _cnt = c;
    }
    public void run() {
        for (int i = 0; i < 100000000; ++i) {
            _cnt.inc();
        }
    }
}

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();
        }
    }
}

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());
    }
}
```

Rysunek 1: Zmienna value po zastosowaniu synchronizacji

3 Różnica w implementacji semafora za pomocą instrukcji *if* i *while*

3.1 Kod semafora z *if*:

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

    public SemaforIf() {
    }

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

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

Instrukcja *if* sprawdza warunek tylko raz. Jeśli wątek zostanie wybudzony przez notify(), zakłada, że może bezpiecznie przejść do wykonywania dalszego kodu, ignorując fakt, że inne wątki mogą już zająć zasób, zanim on to zrobi. Może to prowadzić do sytuacji wyścigu, w której dwa wątki myślą, że mają dostęp do zasobu w tym samym czasie.

3.2 Kod semafora z *while*:

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

    public SemaforWhile() {
    }

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

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

Pętla while ponownie sprawdza warunek po każdym wybudzeniu. Jeśli po wybudzeniu zasób nadal jest zajęty przez inny wątek, wątek pozostaje w stanie oczekiwania. Tylko wtedy, gdy warunek jest faktycznie spełniony, wątek przechodzi dalej, co zapobiega wyścigowi wątków.

3.3 Dlaczego potrzebne jest *while*, a nie *if*?

Kolejka monitorów:

1. **Kolejka monitorowania obiektu** – kiedy wątek próbuje uzyskać dostęp do sekcji krytycznej (np. wchodzi do metody zsynchronizowanej), ale inny wątek już uzyskał dostęp do tej sekcji, wątek czekający zostaje umieszczony w kolejce monitorowania obiektu.

2. **Kolejka wait()** – jeśli wątek wykona operację wait() w metodzie zsynchronizowanej, opuszcza monitor (czyli sekcję krytyczną) i trafia do specjalnej kolejki oczekującej na powiadomienie (notify()). Wątek czekający w tej kolejce nie wykonuje kodu do momentu, aż inny wątek wywoła notify() lub notifyAll().

3.4 Wniosek

Kiedy używamy if, możliwy jest wyścig wątków. Wątek może zostać wybudzony przez notify(), ale inny wątek mógł już zająć semafor. W wyniku tego wątek ten przejdzie dalej, choć zasób jest zajęty, co prowadzi do niespodziewanych błędów. W przypadku while, po każdym wybudzeniu wątek ponownie sprawdza, czy może bezpiecznie kontynuować, co eliminuje problem wyścigu.

4 Semafor licznikowy (ogólny)

4.1 Implementacja semafora binarnego

```
class BinarySemaphore {
    private boolean state;

    public BinarySemaphore(boolean initial) {
        state = initial;
    }

    public synchronized void P() {  // Podniesienie (zajęcie)
        while (!state) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        state = false;  // Zajmujemy semafor
    }

    public synchronized void V() {  // Opuszczenie (zwolnienie)
        state = true;  // Semafor zostaje podniesiony
        notify();
    }
}
```

4.2 Implementacja semafora licznikowego za pomocą semaforów binarnych

```
class CountingSemaphore {
    private int count;
    private final BinarySemaphore mutex;
    private final BinarySemaphore waitSemaphore;

    public CountingSemaphore(int initialCount) {
        count = initialCount;
        mutex = new BinarySemaphore(true);
        waitSemaphore = new BinarySemaphore(false);
    }

    public void P() {
        mutex.P();  // Blokujemy dostęp do zmiennej count
        if (count > 0) {
            count--;  // Jeśli zasoby są dostępne, zmniejszamy licznik
            mutex.V();  // Zwalniamy semafor mutex
        } else {
            mutex.V();  // Zwalniamy semafor mutex, ale nie ma zasobów
            waitSemaphore.P();  // Wątek czeka, aż zasób zostanie zwolniony
        }
    }

    public void V() {
        mutex.P();  // Blokujemy dostęp do zmiennej count
        count++;  // Zwiększamy licznik dostępnych zasobów
        if (count == 1) {  // Jeśli zasób został zwolniony i wątek czekał
            waitSemaphore.V();  // Powiadamiamy jeden oczekujący wątek
        }
        mutex.V();  // Zwalniamy semafor mutex
    }
}
```

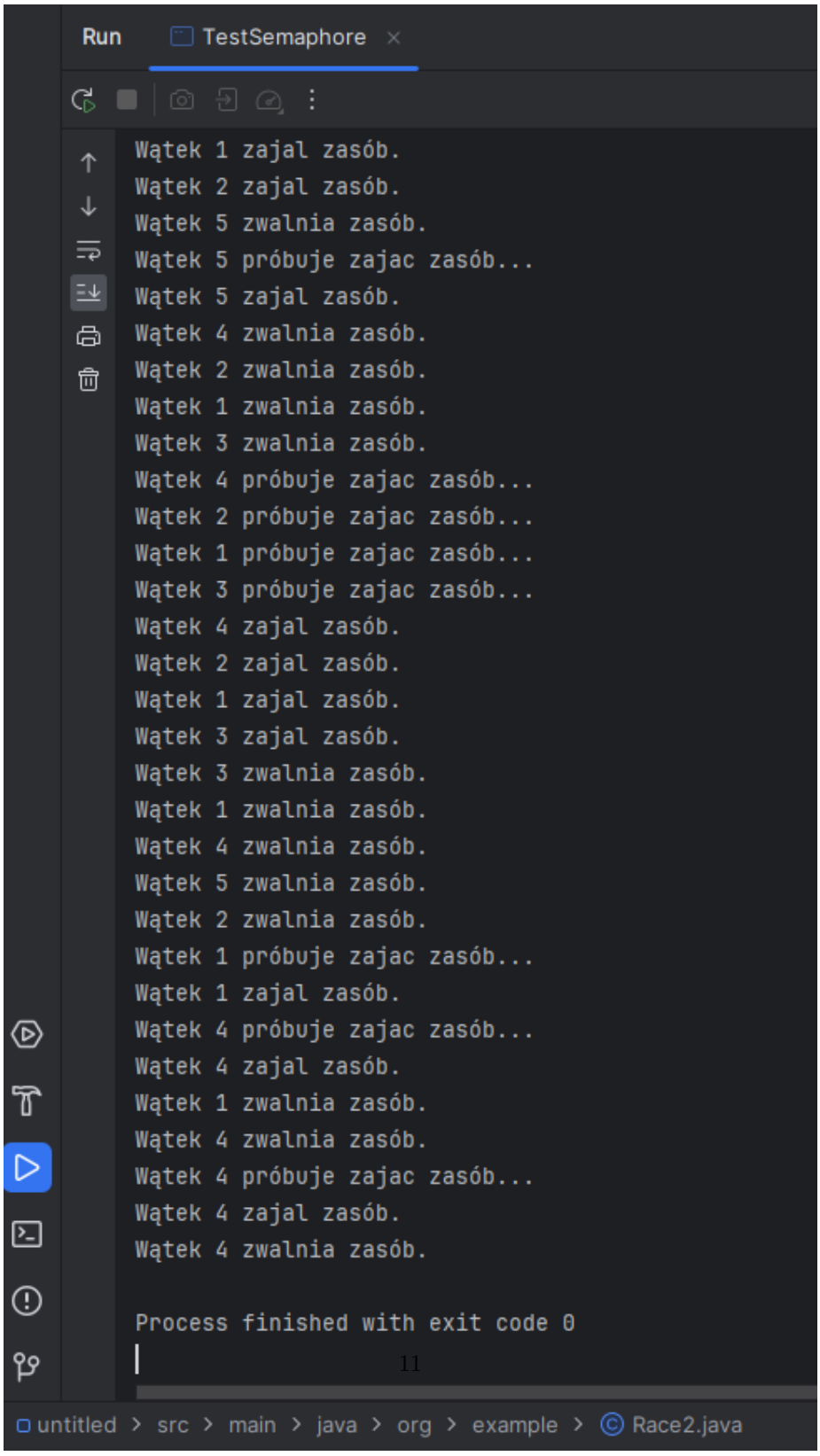
4.3 Testowanie semafora licznikowego:

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

        Runnable r = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName() + "próbuje zajac zasób")
                semaphore.P();
                System.out.println(Thread.currentThread().getName() + " zajal zasób.");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + " zwalnia zasób.");
                semaphore.V();
            }
        };

        Thread t1 = new Thread(r, "Wątek 1");
        Thread t2 = new Thread(r, "Wątek 2");
        Thread t3 = new Thread(r, "Wątek 3");
        Thread t4 = new Thread(r, "Wątek 4");
        Thread t5 = new Thread(r, "Wątek 5");

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
    }
}
```



Rysunek 2: Wynik programu TestSemaphore

4.4 Wnioski

Semafor ogólny (licznikowy) może synchronizować dostęp do więcej niż jednego zasobu, podczas gdy semafor binarny ma tylko dwa stany: dostępny lub zajęty (odpowiadające wartości licznika 1 i 0). Semafor binarny to specjalny przypadek semafora ogólnego, w którym licznik zasobów wynosi maksymalnie 1. Rozszerzając tę ideę, semafor ogólny pozwala na synchronizację dostępu do większej liczby zasobów, ale zachowuje ten sam podstawowy mechanizm blokowania i zwalniania zasobów.

5 Bibliografia

Abraham Silberschatz, Peter B. Galvin, Greg Gagne. ***Operating System Concepts***. Wiley, 2020. ISBN: 978-1119800361.
Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. ***Java Concurrency in Practice***. Addison-Wesley Professional, 2006. ISBN: 978-0321349606.
Andrew S. Tanenbaum, Herbert Bos. ***Modern Operating Systems***. Pearson, 2015. ISBN: 978-0133591620.
Oracle. ***The Java Tutorials - Concurrency***. Oracle, 2023.