



LABORATORIUM PROGRAMOWANIE WSPÓŁBIEŻNE I ROZPROSZONE

Patryk Płatek

Grupa: 2K332-PO

18.10.2025

LABORATORIUM 2: WSPÓŁBIEŻNOŚĆ W JAVIE

ZAWARTOŚĆ

Laboratorium 2: Współbieżność w Javie	1
zadania.....	3
Podejście do rozwiązania	3
Rozwiązanie	4
1 Klasa Main	4
2 Klasa Race.....	4
3 Klasa Counter	5
4 KlasY IThread i DThread	5
5 Klasa RaceThread	6
6 Klasa semafor.....	7
7 Podsumowanie	8
Wyniki	9
Wnioski.....	10
Bibliografia.....	11

ZADANIA

Rozwiązać problem wyścigu z użyciem semafora.

PODEJŚCIE DO ROZWIĄZANIA

Moje podejście do rozwiązania problemu wyścigu w Javie polegało na stworzeniu symulacji, w której dwa wątki równocześnie modyfikują wspólny licznik, oraz na zastosowaniu semafora binarnego do zapewnienia wzajemnego wykluczania dostępu do tego licznika. W programie zdefiniowałem klasę Counter, reprezentującą wspólnodzielony zasób, oraz dwie klasy wątków – IThread i DThread – które odpowiednio inkrementują i dekrementują licznik w wielu iteracjach. Oba wątki dziedziczą po klasie RaceThread, która w swojej metodzie run() używa wspólnego, statycznego semafora (Semafor) do synchronizacji sekcji krytycznej: przed modyfikacją licznika wątek wywołuje metodę P() (opuszczenie), a po zakończeniu – V() (podniesienie). Semafor został zaimplementowany przy użyciu mechanizmów monitorów Javy (synchronized, wait, notify) i działa w sposób klasyczny, dopuszczając do sekcji krytycznej tylko jeden wątek w danym momencie. Klasa Race odpowiada za uruchamianie i kontrolowanie dwóch wątków, a program główny (Main) wielokrotnie wykonuje wyścig i gromadzi wyniki w histogramie, co pozwala empirycznie potwierdzić poprawność synchronizacji – końcowa wartość licznika jest zawsze równa zero. Dzięki temu rozwiązaniu w przejrzysty sposób ilustruje działanie semaforów oraz skuteczne zapobieganie błędom współbieżności.

Main	Główny program uruchamiający wiele „wyścigów” i budujący histogram wyników.
Race	Jeden eksperyment (wyścig) – uruchamia dwa wątki (inkrementujący i dekrementujący) na wspólnym liczniku.
Counter	Licznik z metodami inc() i dec(), bez wewnętrznej synchronizacji – sekcja krytyczna.
RaceThread	Klasa bazowa dla wątków – zawiera pętlę z synchronizacją za pomocą semafora.
IThread/DThread	Wątki konkretne: inkrementują (IThread) lub dekrementują (DThread) licznik.
Semafor	Implementacja klasycznego semafora binarnego 0/1 z użyciem wait()/notify()

ROZWIAZANIE

1 KLASA MAIN

Main tworzy 100 wyścigów (`races = 100`), każdy z 100 000 000 iteracji na wspólnym liczniku (`RUNS = 100_000_000`).

Każdy wyścig:

1. Tworzy nowy obiekt `Race`,
2. Uruchamia dwa wątki (`IThread` i `DThread`),
3. Czeka aż oba skończą (`join()`),
4. Zapisuje końcową wartość licznika do histogramu.

2 KLASA RACE

Klasa `Race` pełni funkcję kontrolera.

Tworzy wspólny licznik (`Counter`) i dwa wątki:

- `IThread` (inkrementujący),
- `DThread` (dekrementujący),

a następnie uruchamia je równolegle i czeka, aż oba zakończą pracę. Na koniec zwraca końcową wartość licznika.

Oba wątki działają na tym samym obiekcie `Counter`, więc potencjalnie mogą wchodzić sobie w drogę.

3 KLASA COUNTER

To bardzo prosta klasa, ale to właśnie tutaj występuje sekcja krytyczna:

```
public class Counter {  
    private int _val;  
  
    // ...  
  
    public void inc() {  
        int n;  
        n = _val;  
        n = n + 1;  
        _val = n;  
    }  
  
    public void dec() {  
        int n;  
        n = _val;  
        n = n - 1;  
        _val = n;  
    }  
}
```

Z pozoru wygląda bezpiecznie, ale to trzy osobne instrukcje:

1. odczyt `_val`,
2. zwiększenie o 1,
3. zapis nowej wartości.

W bytecode oznacza to, że między instrukcją `getfield` i `putfield` może wejść inny wątek. Wtedy dwa wątki modyfikują ten sam `Counter` i efekt końcowy może być błędny.

4 KLASY ITHREAD I DTHREAD

Oba dziedziczą po `RaceThread` i definiują tylko metodę:

```
class IThread extends RaceThread {  
    // ...  
    public void criticalSection() {  
        _cnt.inc();  
    }  
}  
  
class DThread extends RaceThread {  
    // ...  
    public void criticalSection() {  
        _cnt.dec();  
    }  
}
```

Dzięki temu mają wspólny kod obsługi semafora i liczby iteracji w `RaceThread`, a różnią się tylko działaniem sekcji krytycznej.

5 KLASA RACETHREAD

wspólna logika współbieżna

Ważny szczegół:

Semafor jest statyczny.

Dzięki temu oba wątki współdzielą ten sam semafor, czyli tylko jeden z nich może być w sekcji krytycznej w danym momencie.

```
public abstract class RaceThread extends Thread {  
    private final int runs;  
    private static final Semafor sem = new Semafor(); // WSPÓLNY dla wszystkich  
RaceThread  
  
    // ...  
  
    @Override  
    public void run() {  
        for (int i = 0; i < this.runs; ++i) {  
            sem.P();  
            this.criticalSection();  
            sem.V();  
        }  
    }  
  
    public abstract void criticalSection();  
}
```

6 KLASA SEMAFOR

To klasyczna implementacja binarnego semafora ($s = 0/1$) z użyciem monitorów (`synchronized`, `wait`, `notify`).

- **P()** (opuszczenie):
Jeśli semafor zajęty (`stan == false`) → wątek czeka (`wait()`).
Jeśli wolny zajmuje go (`stan = false`).
- **V()** (podniesienie):
Ustawia `stan = true` i budzi jeden oczekujący wątek (`notify()`).

```
class Semafor {  
    private boolean stan = true;  
    private int _czeka = 0;  
  
    // ...  
  
    public synchronized void P() {  
        // Zgłaszamy, że wątek próbuje wejść do sekcji krytycznej  
        _czeka++;  
  
        // Jeśli semafor zajęty – czekaj aż zostanie zwolniony  
        while (!stan) {  
            try {  
                wait(); // czekaj, aż ktoś wywoła notify() w metodzie V()  
            } catch (InterruptedException e) {  
                Thread.currentThread().interrupt(); // zachowaj stan przerwania  
            }  
        }  
  
        // Wątek został dopuszczony – aktualizujemy stan  
        _czeka--; // już nie czeka  
        stan = false; // zajmuje semafor (S = 0)  
    }  
  
    public synchronized void V() {  
        // Zwalniamy semafor  
        stan = true;  
  
        // Jeśli ktoś czeka, obudź jeden wątek z kolejki  
        if (_czeka > 0) {  
            notify(); // budzimy dokładnie jeden oczekujący wątek  
        }  
    }  
}
```

Obie metody są `synchronized`, więc ich wykonanie jest atomowe względem siebie. Dokładnie jeden wątek może wejść do sekcji krytycznej w danym momencie. Czekające wątki są bezpiecznie wstrzymywane w kolejce monitora i budzone w kolejności.

7 PODSUMOWANIE

- Wątki nie mogą jednocześnie modyfikować **Counter**, bo muszą zdobyć semafor,
- Operacje **P()** i **V()** są atomowe dzięki **synchronized**,
- Użycie **wait()** i **notify()** zapewnia poprawne przechodzenie między stanami “zajęty” i “wolny”,
- **Counter** może być modyfikowany tylko przez jeden wątek w danym momencie,
- Histogram końcowych wartości pokazuje stabilny, deterministyczny wynik (zawsze 0).

WYNIKI

Bez Semafora	
razy	wartość
4	-100000000
1	-99997964
1	-99991974
1	-99990540
1	-99983505
1	-99980655
1	-99980495
1	-99966244
1	-99964780
1	-99960720
1	-99960300
1	-99945643
1	-99943968
1	-99888943
1	-98155308
1	-3446
3	0
1	99058238
1	99607674
1	99751934
1	99783920
1	99889044
1	99967440
1	99978996
1	99982206
1	99989811
1	99994691
68	100000000

Z Semaforem	
razy	wartość
100	0

WNIOSKI

Przeprowadzony eksperyment potwierdził, że zastosowanie semafora binarnego skutecznie rozwiązuje problem wyścigu w środowisku wielowątkowym w Javie. W wersji programu bez synchronizacji występowały liczne błędy wynikające z równoczesnego dostępu wątków do współdzielonego licznika końcowe wartości zmiennej były losowe i zależały od kolejności przełączeń wątków, co potwierdził zróżnicowany histogram wyników. Wprowadzenie semafora, który poprzez operacje `P()` i `V()` zapewnił wzajemne wykluczanie sekcji krytycznej, całkowicie wyeliminowało ten problem. W efekcie końcowa wartość licznika po każdej serii iteracji była zawsze równa zero, niezależnie od liczby uruchomień czy intensywności pracy wątków. Implementacja semafora przy użyciu mechanizmów monitorów (`synchronized`, `wait`, `notify`) okazała się skutecznym sposobem na kontrolowanie dostępu do zasobów współdzielonych. Otrzymane wyniki potwierdzają teoretyczne założenia dotyczące działania semaforów i ilustrują znaczenie synchronizacji w programowaniu współbieżnym. Dzięki temu doświadczeniu można jednoznacznie stwierdzić, że prawidłowe stosowanie mechanizmów synchronizacyjnych jest kluczowe dla zapewnienia poprawności i przewidywalności działania programów wielowątkowych.

BIBLIOGRAFIA

Thomas W. Christopher, George K. Thiruvathukal *High-Performance Java Platform Computing*,

Prentice Hall, Luty 2001

Per Brinch Hansen: *Java Insecure Parallelism*. ACM SIGPLAN Notices, April 1999

Repozytorium projektu:

Kod źródłowy oraz wyniki eksperymentów dostępne są w repozytorium GitHub:

https://github.com/SaberLS/concurrent_and_distributed_programming