Laboratorium Nr 1
"Współbieżność w Javie"
Radosław Kopeć
06.10.2020

I. Zadanie 1

1. Treść zadania

Napisać program, który uruchamia 2 wątki, z których jeden zwiększa wartość zmiennej całkowitej o 1, drugi wątek zmniejsza wartość o 1. Zakładając że na początku wartość zmiennej Counter była 0, chcielibyśmy wiedzieć jaka będzie wartość tej zmiennej po wykonaniu 10000 operacji zwiększania i zmniejszania przez obydwa wątki.

2. Koncepcja rozwiązania

Użycie podstawowych narzędzi Javy tj. klasy Threads. Nadpisanie metod **run()**. Uruchomienie wątków przy użyciu metody **start()**.

3. Implementacja i wyniki

Klasa Counter implementująca licznik:

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

Klasa Ithread implementująca wątek zwiększający licznik:

```java
// Watek, ktory inkrementuje licznik 100.000 razy
class IThread extends Thread {

    private Counter cnt;

    public IThread(Counter cnt) { this.cnt = cnt; }

    public void run() {
        for(int i =0;i<100000;i++) {
            cnt.inc();
        }
    }
}
```

Klasa Dthread implementująca wątek zmniejszający licznik:

```java
// Watek, ktory dekrementuje licznik 100.000 razy
class DThread extends Thread {

    private Counter cnt;

    public DThread(Counter cnt) { this.cnt = cnt; }

    public void run() {
        for(int i =0;i<100000;i++) {
            cnt.dec();
        }
    }
}
```

Klasa główna programu:

```java
public class Main extends  Thread{


    public static void main(String[] args) throws InterruptedException {
        Counter cnt = new Counter( n: 0);

        IThread iThread = new IThread(cnt);
        DThread dThread = new DThread(cnt);

        iThread.start();
        dThread.start();

        iThread.join();
        dThread.join();

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

Wynik działania programu:

```
"C:\Program Files\Java\jdk-13\bin\java.exe" "-javaa
stan=8740

Process finished with exit code 0
```

4. **Wnioski**

Na wyjściu spodziewalibyśmy się wyniku 0 jednak mamy wynik 8740. Oznacza to, że przy przetwarzaniu wielowątkowym w javie występuje zjawisko wyścigu. Wątki rywalizują o zasób. Licznik nie jest w żaden sposób chroniony zatem obliczenia wątków wchodzą w kolizję. Jeden wątek wpływa na drugi. Mamy tutaj zjawisko analogiczne do dirty-read w SQL.

5. **Bibliografia**

http://home.agh.edu.pl/~funika/tw/lab1/

II. Zadanie 2

1. Treść zadania

Na podstawie 100 wykonań programu z p.1, stworzyć histogram końcowych wartości zmiennej Counter.

2. Koncepcja rozwiązania

Oddzielimy uruchamianie wątków do innej funkcji. Powtórzymy eksperyment 1000 razy zliczając wyniki w HashMap'ie. Klucz w HashMap'ie będzie oznaczał liczbę, a wartość będzie oznaczała ile razy dany klucz wystąpił.

3. Implementacja i wyniki:

Do zadania 1 dodajemy:

funkcja testThreads() uruchamiająca zadanie 1 i zwracająca wartość licznika:

```
private static int testThreads() throws InterruptedException {
    Counter cnt = new Counter( n: 0);

    IThread iThread = new IThread(cnt);
    DThread dThread = new DThread(cnt);

    iThread.start();
    dThread.start();

    iThread.join();
    dThread.join();

    return cnt.value();
}
```

Dodajemy pole histogramu:

```
public class Zad2 extends  Thread{

    private static HashMap<Integer,Integer> histogram = new HashMap<>();
```

Zmieniamy funkcję główną programu:

```
public static void main(String[] args) throws InterruptedException {
    for(int i=0;i<1000;i++){
        int counter = testThreads();
        if(histogram.containsKey(counter)){
            int currentValue = histogram.get(counter);
            histogram.put(counter,currentValue+1);
        }
        else{
            histogram.put(counter,1);
        }
    }

    histogram.forEach((k,v) -> System.out.println(k + ": " + v));
}
```

Wynik działania(Nasz histogram dla 1000 wywołań):

```
"C:\Program Files\Java\jdk-13\bin\java.exe
0: 999
5213: 1

Process finished with exit code 0
```

4. Wnioski

Jak widać zdarzył się przypadek, że nasze wątki zaczęły rywalizować o zasób. Zastanawiające natomiast jest to, że wystąpił tylko jeden taki przypadek. Dzieje się tak z powodu prawdopodobnie bardzo szybkiego działania procesora i pamięci. Można również domniemywać, że system operacyjny lub wirtualna maszyna javy na przestrzeni lat stały się bardzo dobrze zoptymalizowane na tego typu przypadki. Niemniej jednak problem wyścigu nie zniknął i spróbujemy rozwiązać go w następnym kroku.

5. Bibliografia

http://home.agh.edu.pl/~funika/tw/lab1/

III. Zadanie 3

1. **Treść zadania**

Spróbować wprowadzić mechanizm do programu z p.1, który zagwarantowałby przewidywalną końcową wartość zmiennej Counter. Nie używać żadnych systemowych mechanizmów, tylko swój autorski.

2. **Koncepcja rozwiązania**

Po wielu nieudanych próbach postanowiłem zaimplementować najprostszy możliwy mechanizm rozwiązujący problem sekcji krytycznej dla dwóch wątków. Jeden wątek będzie blokował drugi wątek dopóki nie skończy swoich operacji nad licznikiem. Tworzymy klasę ThreadManager zawierającą licznik (Counter), dwa wątki, oraz dwie zmienne typu boolean. Zmienne te będą oznaczać czy dany wątek zakończył swoją pracę.

3. **Implementacja i wyniki:**

Klasa ThreadManager:

```
public class ThreadManager extends Thread {

    private DecThread decThread;
    private IncThread incThread;

    private boolean firstFisihedWork;
    private boolean secondFisihedWork;

    private Counter counter;

    public ThreadManager() {
        counter = new Counter(0);
        secondFisihedWork = false;
        firstFisihedWork = false;
    }
```

Klasa IncThread zwiększająca licznik. Wątek tej klasy jest blokowany przez zmienną isFirstFisihedWork mówiącą czy wątek dekrementujący zakończył swoją prace. Na końcu funkcja ustawia

secondFisihedWork na true co oznacza że drugi wątek (inkrementu jacy) zakończył pracę.

```
public class IncThread extends Thread {

    private ThreadManager threadManager;

    IncThread(ThreadManager threadManager){
        super();
        this.threadManager = threadManager;
    }

    @Override
    public void run(){
        for(int i=0;i<1000000;){
            if(threadManager.isFirstFisihedWork()){
                i++;
                threadManager.getCounter().inc();
            }
            else{
                try {
                    sleep( millis: 10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        threadManager.setSecondFisihedWork(true);
    }
}
```

Klasa DecThread. Blokuje wątek IncThread, aż zakończy pracę. Ustawiając zmienną FirstFisihhedWork na true, wątek IncThread może zacząść wykonywać swoja pracę.

```
public class DecThread extends Thread {

    private ThreadManager threadManager;

    DecThread(ThreadManager threadManager){
        super();
        this.threadManager = threadManager;
    }

    @Override
    public void run(){
        for(int i=0;i<1000000;){

            threadManager.getCounter().dec();
                i++;

        }
        threadManager.setFirstFisihedWork(true);
    }

}
```

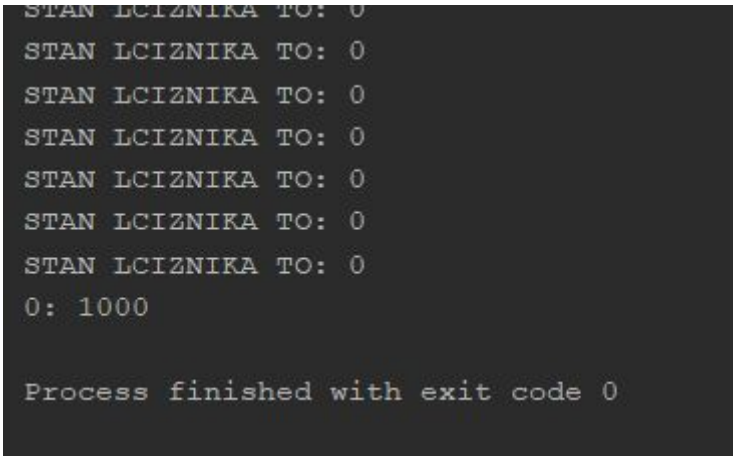
Funkcja testThread powtarzana w pętli w celu przetestowania programu dla 1000 wywołań.

```
private static int testThreads() throws InterruptedException {

    ThreadManager threadManager = new ThreadManager();
    DecThread decThread = new DecThread(threadManager);
    IncThread incThread = new IncThread(threadManager);

    threadManager.setDecThread(decThread);
    threadManager.setIncThread(incThread);

    decThread.start();
    incThread.start();
    threadManager.start();

    decThread.join();
    incThread.join();
    threadManager.join();
```

Główna funkcja bez zmian:

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

    for(int i=0;i<1000;i++){
        int counter = testThreads();
        if(histogram.containsKey(counter)){
            int currentValue = histogram.get(counter);
            histogram.put(counter,currentValue+1);
        }
        else{
            histogram.put(counter,1);
        }
    }
```

Wynik działania programu:



4. Wnioski

Jak widać mój sposób działa, jednak jest on bardzo nieoptymalny. Jeden wątek blokuje zasób na czas wszystkich swoich operacji co jest bardzo nieefektywne. Niemniej jednak problem został rozwiązany.

5. Bibliografia

http://home.agh.edu.pl/~funika/tw/lab1/

IV Zadanie dodadkowe

Treść:
W systemie dziala N wątkow, które dzielą obiekt licznika (początkowy stan licznika = 0).

Każdy wątek wykonuje w pętli 5 razy inkrementację licznika. Zakładamy, że inkrementacja składa się z sekwencji trzech instrukcji: read, inc, write (odczyt z pamięci, zwiększenie o 1, zapis do pamięci). Wątki nie są synchronizowane.

1. Jaka jest teoretycznie najmniejsza wartość licznika po zakończeniu działania wszystkich wątków i jaka kolejność instrukcji (przeplot) do niej prowadzi?

Najmniejsza wartość to 2 dla następującej sekwencji wywołań:

składnia: thread-numerWątku-operacja-numerOperacji

thread-5-read-1 // thread 5 wczytał stan licznika 0

thread 1 wykonuje 4 sekwencje

wątki 2,3,4 wykonują cala swoja prace

thread-5-inc-1
thread-5-write-1 //wątek 5 zapisuje do licznika 1

stan licznika 1

thread-1-read-5 //wątek 1 odczytuje stan licznika 1

wątek 5 wykonuje cala swoja prace

thread-1-inc-5
thread-1-write-5 // wątek 1 zapisuje do licznika wartość 2

Cała praca zostanie wykonana, a w liczniku mamy wartość 2

2. Dowód poprawności

Ilość wątków nie ma tutaj znaczenia, ponieważ już w trzeciej linijce wyeliminowaliśmy znaczenie wątków 2,3,4. Równie dobrze moglibyśmy wyeliminować dowolną liczbę wątków. Ilość sekwencji również nie jest istotna, ponieważ pierwszy wątek musi wykonać swoje n-1 sekwencji po czym odczytać wartość licznika 1. I w swojej ostatniej operacji zapisu wpisać 2 do licznika. W międzyczasie np. ostatni wątek musi dokonać wszystkich swoich operacji.

Radosław Kopeć