Programowanie współbieżne i rozproszone

mgr inż. Marcin Mrukowicz

Instrukcja laboratoryjna IV

Wątki i wzajemne wykluczanie w języku Java

Wątki w języku Java.

Ogólnie pojęcie wątku omówiono w instrukcji I. W tym miejscu, wątki zostaną opisane w kontekście ich implementacji w języku Java.

Język Java charakteryzuje się posiadaniem wsparcia dla wątków od samego powstania języka. Początkowo nie były to wątki natywne, czyli wspierane bezpośrednio przez system operacyjny, ale tzw. **zielone wątki** (ang. **green threads**), dostarczane przez maszynę wirtualną Javy (JVM). Nazwa ta wzięła się od nazwy zespołu programistycznego (the green team), który odpowiadał za ich implementację w firmie Sun Microsystems. Pojęcie to obecnie funkcjonuje jako ogólna nazwa (spotyka się też określenia ang. **user-level threads, lightweight threads, wątki poziomu użytkownika, wątki lekkie**), również poza językiem Java i oznacza wątki, zarządzane z poziomu użytkownika (ang. user level) a nie jądra systemu (ang. kernel level). Konstrukcja taka ma tę zaletę, że dostarcza funkcjonalność wątków nawet na platformie, która ich nie wspiera, jednak zwykle jest to rozwiązanie mniej efektywne (choć jest to sprawa dyskusyjna i zależna od zastosowań) i mające ograniczenia względem wątków natywnych. Ogólnie zielone wątki są dostępne w wielu językach programowania i są wciąż rozwijane, jednakże nie zastąpiły one wątków natywnych. Można powiedzieć, że zarówno wątki natywne, jak i zielone będą współistnieć w przewidywalnej przyszłości, a wybór których używać zależy od specyfiki problemu bądź nawet preferencji zespołu programistycznego.

W ciągu rozwoju języka Java, zielone wątki ustąpiły miejsca wątkom natywnym w standardowej implementacji JVM. Jednakże wątki zielone są dostępne wtórnie za pomocą bibliotek: kilim (https://github.com/kilim/kilim) oraz quasar (https://docs.paralleluniverse.co/quasar/).

Wątek w języku Java konstruujemy jako klasę dziedzicząca po klasie *Thread*, która musi przesłonić metodę *run.* Ponadto istnieje interfejs *Runnable*, również posiadający metodę *run, typu void.* Klasa implementująca ten interfejs, jest uważana za taką, która może zostać wykonana przez wątek. **Warto zaznaczyć, że utworzenie wątku, nie powoduje jego uruchomienia. Wątek uruchomi się dopiero po wywołaniu jego metody *start bądź inne skierowanie go do wykonania*.** Późniejsze wersje języka Java dodały nowe konstrukcje, takie jak *Wykonawcy* (ang. *Executors*), interfejs *Callable* i interfejs *Future. Callable* jest interfejsem zaprojektowanym na wzór Runnable, ale tutaj metoda *call* musi zwracać jakąś wartość, a nie *void* jak metoda *run* w *Runnable. Wykonawcę* można opisać jako klasę, której możemy dodawać do puli zadań zarówno klasy implementujące *Runnable*, jak i *Callable*, następnie zaś *Wykonawca* zaalokuje wątki i przekaże im do wykonania zadania z puli. Warto zaznaczyć, że wykonawca nie będzie czekał na wykonanie metody *call* klasy implementującej *Callable* i zwrócenie przez nią wyniku. Interfejs *Future* pozwala na uzyskanie wyniku jakiejś operacji asynchronicznej. Domyślnie instancja klasy, implementująca ten interfejs jest zwracana przez wykonawcę, jeżeli nakażemy mu wykonać klasę implementującą *Callable*. *Future* posiada metodę *get*, która zwraca wynik operacji asynchronicznej, ale blokuje wykonanie wątku głównego aplikacji, zmuszając go na czekanie na ukończenie pracy przez *Callable (*o ile oczywiście zadanie nie jest jeszcze ukończone*)*. Możliwe jest również wywołanie metody *get* z argumentami, określającymi ile maksymalnie czekać na wykonanie *Callable*, po czym jeżeli odpowiedź nie nadejdzie, wątek główny będzie wykonywał się nadal.

Mechanizmy wzajemnego wykluczania w języku Java

W celu synchronizacji dostępu do zmiennych współdzielonych przez wątki, stosuje się różne mechanizmy. Głównym celem jest zapobieżenie utraty spójności danych, gdy wątki niezależnie od siebie będą modyfikować jej wartości (jest to tzw. *race condition*).

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Na powyższym rysunku pokazano, wykonanie pewnego programu współbieżnego w dwóch wersjach: pierwsza zakłada, że pierwszy wątek zdążył odczytać współdzieloną zmienną, zmodyfikować ją i ponownie zapisać, po czym drugi wątek wykonał tę samą operację. Jednakże nie jest to regułą: równie dobrze może być tak jak na rysunku po lewej: wątki odczytały najpierw wartość zmiennej, zmodyfikowały ją niezależnie od siebie, po czym zapisały jej wartość. Jak widać, doprowadziło to do niepoprawnego stanu i efekt działania programu może być nieoczekiwany.

Najprostszym sposobem na zapewnienie wzajemnego wykluczania jest zastosowanie tzw. mutexu (od ang. mutual exclusion). Jest to w najprostszym przypadku zmienna binarna, współdzielona przez wątki, która blokuje dostęp do jakiegoś zasobu (np. jeżeli jest ustawiona na wartość false) lub pozwala na dostęp do niego (np. jeżeli jest ustawiona na wartość true). Jeżeli jakiś wątek dostał dostęp do zasobu, to automatycznie mutex jest negowany i inne wątki nie będą już miały dostępu do blokowanego zasobu; gdy wątek skończy pracę z zasobem, ponownie neguje on mutex, co zwalnia blokadę dla innych wątków. Blok kodu, w którym wątek uzyskuje wyłączny dostęp do zasobu nazywamy **sekcją krytyczną** (ang. **critical section**).

Uogólnieniem mutexu jest semafor (ang. semaphore). Jest to w najprostszej implementacji zmienna całkowita, która określa maksymalną liczbę wątków, które mogą korzystać z danego zasobu. Przykładowo ustawiając semafor początkowo na wartość 5, ustalamy, że maksymalnie 5 wątków może mieć dostęp do danego zasobu. Gdy wątek dostaje dostęp do zasobu, to dekrementuje semafor, a podczas gdy zwalnia ten zasób, to z kolei inkrementuje semafor. Gdy wartość semaforu osiągnie 0, to nie pozwalamy już kolejnym wątkom na dostęp do zasobu. Dopiero gdy jakiś wątek zwolni zasób, to zwiększy wartość semafora (np. na 1) i znowu jakiś wątek, oczekujący na zasób, może mieć do niego dostęp. W teorii, gdy wszystkie wątki zwolnią zasób, to semafor ponownie uzyska wartość 5. Semafor, któremu ustawimy maksymalną liczbę wątków na 1, redukuje się do mutexu. **Język Java dostarcza klasy Semaphore, implementującej semafor.**

Monitor to obiekt, którego metody są chronione przez mutexy, a więc mogą być bezpiecznie wykonywane przez wątki. W języku Java istnieje słowo kluczowe ***synchronized***, które może być używane w dwóch podstawowych przypadkach. Możemy oznaczyć nim metodę dowolnej klasy, co oznacza, że metoda ta, będzie chroniona przez mutex. Możliwe jest również wykorzystanie tzw. ***bloku synchronizowanego, synchronized block***. W tym celu podajemy podajemy w nawiasie obiekt, do którego dostęp będziemy synchronizować. Obiekt taki stanie się wtedy automatycznie opakowany przez monitor. Każdy obiekt w języku Java posiada metody: ***wait***, ***notify*** i ***notifyAll***. Metody te mogą być wywoływane wyłącznie w bloku synchronizowanym i służą do komunikacji pomiędzy wątkami, oraz dostarczają pewnych informacji monitorowi. Metoda *wait* spowoduje, że wątek będzie czekał na powiadomienie przez inny wątek, że może mieć już dostęp

do zasobu. Wywołanie metody *notify* spowoduje wybudzenie pierwszego wątku, oczekującego na dostęp do zasobu w kolejce. Wywołanie metody *notifyAll* spowoduje wybudzenie wszystkich wątków, oczekujących na dostęp do zasobu.

W języku Java występuje również słowo kluczowe **volatile.** Zostało ono wprowadzone w JDK w wersji 5, w celu rozwiązania problemu związanego z rosnącym wykorzystaniem procesorów wielordzeniowych. Architektury tego typu wykorzystują niejednokrotnie różne techniki optymalizacyjne w postaci cache'owania zmiennych. W przypadku uruchomienia wątków na różnych rdzeniach, nawet **pola statyczne klas** mogą posiadać lokalne scache'owane wartości. Ponadto kompilator posiada możliwość zmiany kolejności instrukcji, o ile nie zmieni się ich ostateczny rezultat, co jednak w warunkach programów współbieżnych może doprowadzić do race condition.

Jeżeli pole oznaczymy **volatile** wymuszamy, aby nie było ono w żaden sposób cacheowane, jak i wymuszamy na JDK, aby za każdym razem jego wartość była odczytywana z komórki pamięci; ponadto modyfikator ten zapewnia, że operacje na tym polu nie będą zmieniać kolejności wykonania (tzn. zostaną na pewno wykononane w kolejności w jakiej były wywoływane). **Należy jednak zaznaczyć, że pola volatile nie są w zupełności „bezpieczne" (ang. thread-safe). Volatile można traktować, jako pewien słaby sposób synchronizacji, który sprawdza się w sytuacji, gdy tylko jeden wątek modyfikuje dany zasób (a zwłaszcza gdy dzieje się to tylko jeden raz podczas uruchomienia programu), a pozostałe jedynie odczytują jego wartość. W każdej innej sytuacji wykorzystanie volatile nie gwarantuje poprawnego działania programu. Z drugiej strony wydajność zastosowania volatile jest zdecydowanie większa niż bloku synchronizowanego i innych metod synchronizacji. To zagadnienie powoduje, że niekiedy opłaca się w celach optymalizacyjnych stosować ten modyfikator w miejsce innych metod synchronizacji.** Na koniec warto też odnotować, że z kolei zmienna volatile jest wolniejsza niż zwykłe zmienne, zatem nie powinno się z drugiej strony również używać tego modyfikatora ponad to, co jest niezbędne.

1. Przepisz, następujące klasy w języku Java::

```
import java.util.ArrayList;

public class HelloThreadArray extends Thread {
    private ArrayList<String> buffer;

    public HelloThreadArray(ArrayList<String> buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        synchronized (buffer) {
            buffer.add("Hello, ");
            buffer.notify();
        }
    }
}
```

klasa HelloThreadArray.java

```
import java.util.ArrayList;

public class WorldThreadArray extends Thread {
    private ArrayList<String> buffer;
```

```
    public WorldThreadArray(ArrayList<String> buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        synchronized (buffer) {
            try {
                buffer.wait();
                buffer.add("World!");
                for (String s : buffer)
                    System.out.print(s);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

klasa WorldThreadArray.java

```
import java.util.ArrayList;

public class ThreadsArrayHelloWorld {
    public static void main(String [] args) {
        final ArrayList<String> buffer = new ArrayList<>();
        HelloThreadArray helloThread = new HelloThreadArray(buffer);
        WorldThreadArray worldThread = new WorldThreadArray(buffer);
        worldThread.start();
        helloThread.start();
    }
}
```

klasa ThreadArrayHelloWorld.java

Zostały zdefiniowane dwie klasy wątków, które współdzielą zasób w postaci listy. Jak widać, jeden wątek (ten, który dodaje do listy słowo: „World!”) zawsze czeka na dostępność zasobu, natomiast drugi wątek (ten który dodaje do listy słowo: „Hello, ”) po prostu modyfikuje zasób, po czym powiadamia drugi wątek, że zakończył już pracę z zasobem. Należy zauważyć, że powyższe zachowanie programu wynika z zastosowania bloku synchronizowanego.

2. Przepisz, następujące klasy w języku Java:

```
public class WorldThread extends Thread {
    private Boolean waitForHello;

    public WorldThread(Boolean waitForHello) {
        this.waitForHello = waitForHello;
    }

    @Override
    public void run() {
        synchronized (this.waitForHello) {
            try {
                this.waitForHello.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(", World!");
```

```
        }
    }
}
```

klasa WorldThread.java

```
public class HelloThread extends Thread {
    private Boolean waitForHello;

    public HelloThread(Boolean waitForHello) {
        this.waitForHello = waitForHello;
    }

    @Override
    public void run() {
        synchronized (waitForHello) {
            System.out.print("Hello");
            waitForHello.notify();
        }
    }
}
```

klasa HelloThread.java

```
public class ThreadsHelloWorld {
    public static void main(String [] args) {
        HelloThread helloThread = new HelloThread(Boolean.FALSE);
        WorldThread worldThread = new WorldThread(Boolean.FALSE);
        worldThread.start();
        helloThread.start();
    }
}
```

klasa ThreadHelloWorld.java

Ponownie wykorzystano blok synchronizowany, ale tym razem tak naprawdę nie interesowała nas wartość zmiennej synchronizowanej, jednakże osiągnięto cel w postaci synchronicznego wypisania na ekran: „Hello, World!”. Można to potraktować jako pewną ciekawostkę.

- **Przepisz powyższe przykłady, korzystając zamiast z klasy Thread, z interfejsu Runnable, zamień w nazwach klas słowo Thread na Runnable. Uruchom te klasy.**

3. Przepisz poniższe programy

```
public class Account {
    double debit;
    double actualSalary;

    public Account(double debit, double actualSalary) {
        this.debit = debit;
        this.actualSalary = actualSalary;

    }

    synchronized void subtract(double amount) {
        if (actualSalary - amount >= debit) {
            System.out.print(actualSalary + "-" + amount + "=");
            actualSalary -= amount;
            System.out.println(actualSalary);
        } else {
            System.out.print(actualSalary + "-" + amount + " < dopuszczalnego   debetu");
            System.out.println(", nie da się wykonać transakcji!!!");
        }
    }
```

```
    synchronized void add(double amount) {
        actualSalary += amount;
    }

    synchronized double getActualSalary() {
        return actualSalary;
    }

}
```

klasa Account.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AccountDemo {
    public static void main(String [] args) throws InterruptedException {
        Account account = new Account(500, 3000);

        ExecutorService executor = Executors.newFixedThreadPool(20);
        executor.execute(new Thread(() -> {account.subtract(1000);}));
        executor.execute(new Thread(() -> {account.subtract(500);}));
        executor.execute(new Thread(() -> {account.subtract(200);}));
        executor.execute(new Thread(() -> {account.subtract(1200);}));
        executor.execute(new Thread(() -> {account.subtract(1500);}));

        executor.shutdown();
    }
}
```

klasa AccountDemo.java

- **Jak działa powyższy program?**
- **Usuń z klasy *Account* słowo kluczowe synchronized i zapisz plik jako *AccountNoSync.java*. Napisz klasę *AccountNoSyncDemo.java*, która działa dokładnie jak *AccountDemo.java* ale wykorzystuje klasę *AccountNoSync.java*. Jaki jest efekt uruchomienia tej klasy? Co spowodowało usunięcie słowa *synchronized*?**

4. Przepisz poniższy program:

```
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.*;

public class CallableTemperatureSumming {
    public static void main(String [] args) throws ExecutionException, InterruptedException,
TimeoutException {
        ExecutorService executor = Executors.newFixedThreadPool(20);
        ArrayList<Integer> temperature = new ArrayList<>();
        Random r = new Random();
        for (int i = 0; i < 100000; i++) {
            temperature.add(r.nextInt(35));
        }
        Future<Integer> part1 = executor.submit(() -> {
            int partSum = 0;
            for (int i = 0; i < 25000; i++) {
                partSum += temperature.get(i);
            }
            return partSum;
        });

        Future<Integer> part2 = executor.submit(() -> {
            int partSum = 0;
            for (int i = 25000; i < 50000; i++) {
                partSum += temperature.get(i);
            }
            return partSum;
```

```
        });

        Future<Integer> part3 = executor.submit(() -> {
            int partSum = 0;
            for (int i = 50000; i < 75000; i++) {
                partSum += temperature.get(i);
            }
            return partSum;
        });

        Future<Integer> part4 = executor.submit(() -> {
            int partSum = 0;
            for (int i = 75000; i < 100000; i++) {
                partSum += temperature.get(i);
            }
            return partSum;
        });

        while (!part1.isDone() && !part2.isDone() && !part3.isDone() && !part4.isDone()) {
            Thread.sleep(0, 1);
        }
            int s = part1.get(1, TimeUnit.NANOSECONDS);
            int s2 = part2.get(1, TimeUnit.NANOSECONDS);
            int s3 = part3.get(1, TimeUnit.NANOSECONDS);
            int s4 = part4.get(1, TimeUnit.NANOSECONDS);
        System.out.println("The mean temperature is = " + (s + s2 + s3 + s4) / 100000.0);
        executor.shutdown();

    }
}
```

klasa *CallableTemperatureSumming.java*

Dla celów demonstracyjnych najpierw tworzymy listę temperatur, ale proszę mieć na uwadze, że dane te mogłyby być rzeczywiste. Chcemy obliczyć średnią temperaturę, zatem musimy najpierw zsumować wszystkie zmierzone temperatury po czym podzielić tę sumę przez liczbę pomiarów. Powyższy program rozdziela sumowanie na cztery wątki (oczywiście w celu przyspieszenia obliczeń), następnie oczekując na ich wykonanie. Jednakże nie następuje blokowanie wykonania wątku głównego w oczekiwaniu na wynik któregoś z wątków. Na koniec wyniki częściowe są sumowane (ale jest ich już niewiele) i dzielone przez liczbę pomiarów; otrzymujemy średnią temperaturę.

- **Usuń pętlę while z powyższego programu, nazwij klasę *CallableTemperatureSummingNoWhile.java,* jaki jest efekt działania programu?**
- **Zmodyfikuj czas oczekiwania na wykonanie wątku do 10 milisekund, ale również usuń pętlę while. Nazwij klasę *CallableTemperatureSummingNoWhileLongWait.java.* Jaki jest efekt działania programu?**

5. Przepisz poniższy program:

```
public class NoVisibility {
   private static boolean ready;
   private static int number;

   private static class ReaderThread extends Thread {
      public void run() {
         while (!ready)
            Thread.yield();
         System.out.println(number);
      }
   }

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

```
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Jest to klasyczny przykład programu, który nie gwarantuje, że wątek, zdefiniowany w klasie ReaderThread wypisze na ekran oczekiwaną (raczej na bazie doświadczeń z programowania sekwencyjnego) wartość 42. W bardzo wielu przypadkach rzeczywiście wypisane zostanie 42 (będzie to spowodowane tym, że JVM i system operacyjny skierują wykonanie obydwu wątków do jednego rdzenia procesora, który ma wtedy jeden cache), jednak nie jest to zachowanie **zagwarantowane w żaden sposób**.

- Jaki jest według Ciebie możliwy inny wynik działania programu?
- Dopisz do pól ready i number poprawnie modyfikator *volatile*

6. Przepisz poniższy program:

```
public class HotSpotTest {
    static long count;
    static boolean shouldContinue = true;

    public static void main(String[] args) {
        Thread t = new Thread(new Runnable() {
            public void run() {
                while(shouldContinue) {
                    count++;
                }
                System.out.println("I exited");
            }
        });
        t.start();

        do {
            try {
                Thread.sleep(1000L);
            } catch(InterruptedException ie) {}
        } while(count < 999999L);

        shouldContinue = false;
        System.out.println(
                "stopping at " + count + " iterations"
        );

        try {
            t.join();
        } catch(InterruptedException ie) {}
    }
}
```

Program ten jest napisany w ten sposób, że JDK po jakimś czasie wykona pewne optymalizacje (np. widząc że zmienna *shouldContinue* niekoniecznie zmienia się w bardzo długim horyzoncie czasowym, może podmienić jej wartość na stałą *true*, albo scache'ować jej wartość). Spowoduje to, że jeżeli nawet wątek główny w końcu zmodyfikuje *shouldContinue* to drugi wątek może nie zobaczyć tej zmiany i ugrzęźnie w pętli nieskończonej.

- Dodaj polom *count* i *shouldContinue* modyfikator *volatile*. Jaki tym razem jest efekt działania programu?
- Usuń polu *count* modyfikator *volatile.* Jaki tym razem jest efekt działania programu? Czy masz pomysł czym może być to spowodowane?