WYŻSZA SZKOŁA BANKOWA
Warszawa

Programowanie aplikacji w Java

Maciej Gowin

Zjazd 7 - dzień 2

Linki

Opis

https://maciejgowin.github.io/wsb-java/

Kod źródłowy przykładów oraz zadań

https://github.com/MaciejGowin/wsb-programowanie-aplikacji-java

Program

Kod programu opisuje instrukcje, które powinny zostać wykonane przez procesor. Instrukcje te są najczęściej niezmienne. Aby móc je wykonać konieczne są zasoby takie jak procesor czy pamięć.

Proces jest kontekstem umożliwiającym wykonanie danego programu.

Proces

Podczas uruchomienia programu (np. aplikacji Java) tworzony jest proces. Jest on identyfikowany poprzez unikatowy numer PID (ang. process identifier).

System operacyjny przypisuje procesowi zasoby umożliwiające jego działanie. System operacyjny jest również odpowiedzialny za zarządzanie procesami.

Każdy proces posiada proces nadrzędny. Może również tworzyć procesy potomne, tworząc swego rodzaju drzewo procesów. Nowe procesy tworzone są poprzez wywołania systemowe.

Składowe procesu

Do składowych procesu zaliczamy:

- kod programu,
- licznik rozkazów,
- stos programu,
- sekcję danych.

Zasoby procesu

Do zasobów przydzielanych procesowi zaliczamy:

- czas procesora,
- pamięć,
- dostęp do urządzeń wejścia-wyjścia,
- pliki.

Stany procesu

Każdy z procesów może znajdować się w jednym ze stanów, który uzależniony jest od jego działania i gotowości.

- Nowy - tworzenie procesu oraz przypisywanie zasobów, po zakończeniu oczekiwanie do przyjęcia do kolejki procesów gotowych.
- Wykonywany - wykonywanie instrukcji programu oraz działanie na zasobach.
- Oczekujący - zatrzymanie wykonywania instrukcji w związku z potrzebą dostępu do dodatkowych zasobów lub otrzymania odpowiedzi z otoczenia procesu.
- Gotowy - oczekiwanie na przydział czasu procesora.
- Zakończony - zakończenie procesu, zwolnienie zasobów oraz ewentualne przekazanie informacji o zakończeniu do innych procesów.

> Zombie to proces pozostawiony w stanie zakończony .

Planista przydziału procesora

W systemach z jednym procesorem w danym momencie w stanie wykonywany może przebywać tylko jeden proces, któremu został przydzielony czas procesora.

Planista przydziału procesora decyduje o przejściu danego procesu ze stanu gotowy do wykonywany. Procesy są szeregowane na podstawie priorytetów.

Moment przejścia procesu ze stanu wykonywany do gotowy nazywany jest wywłaszczeniem procesu i może wynikać z:

- zakończenia czasu procesora przydzielonego procesowi,
- pojawienia się procesu o wyższym priorytecie.

Istnieje szereg algorytmów, które opisują kolejkowanie oraz wywłaszczanie procesów.

Przełączanie kontekstu

Podczas zmiany procesu, który jest wykonywany przez procesor w danym momencie, dochodzi do przełączania kontekstu.

Działanie to ma na celu:

- zapamiętanie stanu procesowa dla procesu kończącego pracę oraz
- załadowanie do procesora wszystkich istotnych informacji dla nowego procesu (np. przeładowanie rejestrów procesora).

Kolejki procesów

Działanie procesu nie jest uzależnione tylko od przydzielonego czasu procesora, ale w głównej mierze od zasobów zewnętrznych.

W przypadku oczekiwania za zasów proces trafia do kolejki procesów oczekujących. Dopiero po uzyskaniu zasobów zostaje przeniesiony do kolejki procesów gotowych - nie koniecznie jest od razu wykonywany.

Kolejkowanie to mechanizm kategoryzowania procesów i umieszczania ich na odpowiedniej liście.

Zależności procesów

Procesy są od siebie izolowane. Oznacza to, że nie współdzielą między sobą zasobów.

Oczywiście jeden proces może zależeć od innego, aczkolwiek wymiana informacji pomiędzy nimi jest skomplikowana.

Procesy

Tworzenie procesów wymaga wywołań systemowych. Mówimy, że procesy są "ciężkie" ponieważ ich utworzenie wymaga wyodrębnionych zasobów. Również ich zarządzanie jest wymagające. Przełączanie procesów jest zadaniem czasochłonnym.

Wątek

Aby rozwiązać problemy z procesami, wprowadzono pojęcie procesów lekkich, czyli wątków.

W ramach danego procesu może być wykonywany jeden wątek lub kilka wątków równolegle.

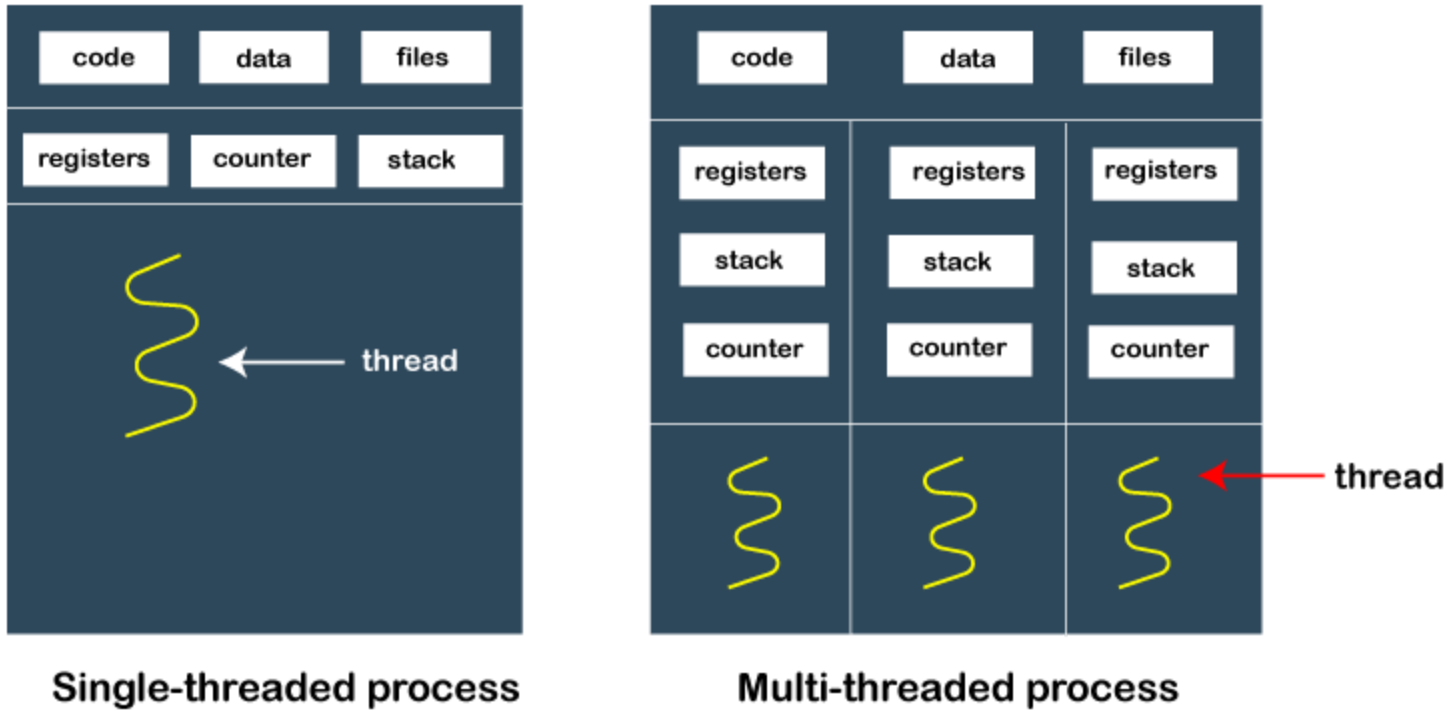
Wątki to instrukcje realizowane do pewnego stopnia niezależnie w obrębie danego procesu.

Proces a wątek

Innymi słowy, wątek jest podzbiorem procesu. Proces może posiadać więcej niż jeden wątek. Każdy z wątków jest zarządzany niezależnie przez zarządcę.

Wszystkie wątki w obrębie tego samego procesu są ze sobą powiązane. Współdzielą niektóre z informacji takie jak: wirtualną przestrzeń adresową, segmenty danych, segmenty kody czy pliki. Każdy z wątków posiada jednak swoje rejestry oraz stos.

Proces a wątek



Źródło: https://www.javatpoint.com/process-vs-thread

Proces a wątek

- Proces jest niezależny i nie zawiera się w innym procesie, podczas gdy wszystkie wątki logicznie zawierają się w procesie.
- Procesy są ciężkie, podczas gdy wątki są lekkie.
- Proces może istnieć sam w sobie, ponieważ posiada własną pamięć oraz inne zasoby, podczas gdy wątek potrzebuje procesu nadrzędnego.
- Synchronizacja pomiędzy procesami nie jest wymagana. W przeciwieństwie do wątków, gdzie synchronizacja jest kluczowa do uniknięcia nieoczekiwanych zachowań.
- Procesy mogą się komunikować jedynie przy użyciu komunikacji międzyprocesowej. W przeciwieństwie do wątków, które mogą komunikować się bezpośrednio, ponieważ współdzielą tę samą przestrzeń adresową.

Wielowątkowość

Wielowątkowość odnosi się to wykonywanie wielu wątków na poziomie systemu operacyjnego.

W skrócie dwa lub więcej wątków tego samego procesu wykonywane równolegle.

Jeżeli w obrębie procesu wykonywany jest jeden wątek mamy do czynienia z aplikacją jednowątkową (ang. single-threading).

Jeżeli w obrębie procesu wykonywany jest więcej niż jeden wątek mamy do czynienia z aplikacją wielowątkową (ang. multi-threading).

Zalety wielowątkowości

Wątki są głównie używane w celu poprawy przetwarzania aplikacji. W praktyce tylko jeden wątek jest wykonywany w danym momencie. Jednak przełączanie kontekstu pomiędzy wątkami jest dużo szybsze niż to wykonywane na poziomie procesów, co sprawia wrażenie wykonywania wielu wątków równocześnie.

Własności wielowątkowości

- Wątki współdzielą dane, pamięć, zasoby, pliki itp. razem z innymi wątkami w obrębie procesu.
- Jedno wywołanie systemowe może stworzyć więcej niż jeden wątek.
- Każdy z wątków posiada własny stos oraz rejestr.
- Wątki mogą komunikować się bezpośrednio pomiędzy sobą, ponieważ współdzielą tę samą przestrzeń adresową.
- Wątki muszą być synchronizowane w celu zapobiegania nieoczekiwanych zachowań.

Typy wątków

Wyróżniamy dwa typy wątków:

- wątki poziomu użytkownika,
- wątki poziomu jądra.

Typy wątków

Wątki poziomu użytkownika

- Wątki poziomu użytkownika są zarządzane przez użytkownika, jądro systemu nie ma o nich informacji.
- Są szybsze, łatwe w tworzeniu oraz zarządzaniu.
- Jądro traktuje wszystkie te wątki jako jeden proces i obsługuje je jako jeden proces.
- Wątki poziomu użytkownika są implementowane przez biblioteki poziomu użytkownika, nie przez wywołania systemowe.

Typy wątków

Wątki poziomu jądra

- Wątki poziomu jądra obsługiwane przez system operacyjny oraz zarządzane przez jego jądro.
- Są wolniejsze niż wątki poziomu użytkownika, ponieważ ich kontekst jest zarządzany przez jądro.
- Aby stworzyć wątki poziomu jądra, wymagane jest wywołanie systemowe.

Przykład użycia wątków

Aplikacja do przetwarzania wideo

1. Wyobraźmy sobie aplikację do przetwarzającą wideo.
2. Na czas konwersji wideo przy użyciu jednego wątku zablokowana zostałaby cała aplikacja.
3. Aby pozwolić użytkownikowi na pracę z aplikacją, na czas przetwarzania tworzony jest poboczny wątek wykonujący operację.
4. Główny wątek odpowiedzialny jest za działanie aplikacji.

Przykład użycia wątków

Serwer aplikacji dostarczający Strona internetowa

1. Użytkownik wykonuje zapytanie do serwera aplikacji w celu pobrania strony internetowej.
2. Jeżeli pobranie danych wymagałoby bardziej skomplikowanych i czasochłonnych operacji serwer zostałby zablokowany na czas tych operacji.
3. Każde zapytanie jest obsługiwane przez osobny wątek.
4. Należy zauważyć, że liczba wątków, które może stworzyć serwer aplikacji, jest ograniczona.
5. Tworzenie wątków w nieskończoności doprowadzi do błędów.

Problemy systemów współbieżnych

- synchronizacja: problem ucztujących filozofów
- zakleszczenie (ang. deadlock)
- zagłodzenie (ang. starvation)

Wątki Java

Java dostarcza bibliotekę do obsługi wyjątków. Wątki mogą wykonywać dowolny kod w obrębie danej aplikacji.

Podczas uruchomienia programu metoda `main()` jest wykonywana przez wątek główny (ang. main thread). Jest on specjalnym wątkiem tworzony przez JVM służącym do uruchomienia aplikacji.

Z poziomu aplikacji możemy tworzyć i uruchamiać kolejne wątki, które mogą wykonywać zadania równolegle do wątku głównego.

Wątki Java

Wątki w Javie reprezentowane są poprzez obiekty klasy `java.lang.Thread` lub obiekty jakiejkolwiek klasy dziedziczącej po niej.

Język Java: wątek

Utworzenie wątku odbywa się poprzez standardowe utworzenie instancji klasy:

```
Thread thread = new Thread();
```

Po utworzeniu wątek może zostań uruchomiony poprzez wywołanie metody `start()`.

```
thread.start();
```

W powyższym przykładzie wątek nie wykonuje żadnego kodu. Zostanie zakończony zaraz po rozpoczęciu.

Język Java: tworzenie wątków

Istnieją dwa podejścia pozwalające na specyfikację kodu, który powinien zostać wykonany przez wątek:

- utworzenie podklasy klasy Thread oraz nadpisanie metody `run()`,
- utworzenie instancji klasy Thread przekazując obiekt implementujący interfejs `Runnable` w konstruktorze.

Język Java: podklasa Thread

```
class FirstThread extends Thread {

    public void run() {
        System.out.println("FirstThread");
    }
}

public class Main {

    public static void main(String[] args) {
        Thread firstThread = new FirstThread();
        Thread secondThread = new Thread() {
            public void run() {
                System.out.println("SecondThread");
            }
        };

        firstThread.start();
        secondThread.start();
    }
}
```

Język Java: implementacja Runnable

Interfejs Runnable nie różni się od zwykłych interfejsów. Co więcej, jest to interfejs funkcyjny.

```
public interface Runnable() {

    public void run();
}
```

Język Java: implementacja Runnable

```
class FirstRunnable implements Runnable {

    public void run() {
        System.out.println("FirstRunnable");
    }
}

public class Main {

    public static void main(String[] args) {
        Runnable firstRunnable = new FirstRunnable();
        Runnable secondRunnable = new Runnable() {

            @Override
            public void run() {
                System.out.println("SecondRunnable");
            }
        };
        Runnable thirdRunnable = () -> System.out.println("ThirdRunnable");

        Thread firstThread = new Thread(firstRunnable);
        Thread secondThread = new Thread(secondRunnable);
        Thread thirdThread = new Thread(thirdRunnable);

        firstThread.start();
        secondThread.start();
        thirdThread.start();
    }
}
```

Język Java: Runnable czy Thread

Nie ma jednoznacznej odpowiedzi na to pytanie.

W większości przypadków może wydawać się, iż implementacja `Runnable` daje więcej możliwości podczas kolejkowania zadań oraz wykorzystywania jednego wątku do wykonywania wielu zadań przy użyciu różnych implementacji Runnable.

Język Java: nazwa wątku

Każdy z wątków otrzymuje swoją domyślną nazwę, która może zostać pobrana poprzez metodę `getName()`. Istnieje możliwość nadpisania tej wartości poprzez zadanie jej podczas wywołania konstruktora.

Język Java: nazwa wątku

```
import java.nio.file.Files;

class FirstThread extends Thread {

    public FirstThread() {
    }

    public FirstThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println("Thread name: " + getName());
    }
}

public class Main {

    public static void main(String[] args) {
        Thread thread1 = new FirstThread();
        Thread thread2 = new FirstThread("Another");
        thread1.start();
        thread2.start();
    }
}
```

Język Java: obecny wątek

W dowolnej części naszego kodu możemy pobrać informację o tym, jaki wątek wykonuje dany blok kodu.

```
Thread thread = Thread.currentThread();
```

Wywołanie to pozwala na uzyskanie referencji do wątku.

Język Java: uśpienie wątku

Do uśpienia wątku używamy metody `sleep()`.

```
Thread.sleep(1000);
```

Język Java: gotowość wątku

Do poinformowania o tym, że obecny wątek jest gotowy do wykonania tak szybko, jak tylko jest to możliwe możemy użyć metody `yield()`.

```
Thread.yield()
```

Język Java: oczekiwanie na zakończenie wątku

W momencie, w którym chcemy ,aby dany wątek został wykonany i zakończony przed kontynuacją obecnego wątku należy użyć metody `join()` na wątku, który powinien zostać zakończony.

```
Thread thread = new CounterThread();
thread.start();
thread.join()
```

Język Java: metody start() oraz run()

Metoda `run()` jest metodą, która jest wywoływana zaraz po wywołaniu metody `start()`.

Należy rozróżnić wywołanie metody `start()` od wywołania metody `run()` explicite:

- wywołanie `start()` spowoduje uruchomienie wątku oraz wykonanie kodu metody `run()`,
- wywołanie `run()` nie spowoduje uruchomienia wątku i jedynie wykona kod metody `run()`.

Język Java: metody start() oraz run()

```
class FirstThread extends Thread {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}

public class Main {

    public static void main(String[] args) {
        Thread firstThread = new FirstThread();
        firstThread.run();
        firstThread.start();
    }
}
```

Język Java: usypianie wątku

Wątek może zostać uśpiony poprzez wywołanie statycznej metody `Thread.sleep()`. Przyjmuje ona jako parametr wartość czasu zadaną w milisekundach. Jest to czas, na jaki wątek ma zostać uśpiony. Po jego upływie wykonywanie wątku może zostać wznowione.

```
Thread.sleep(5000);
```

> Wartość ta nie jest w pełni precyzyjna.

Programowanie: przykład 63

Tworzenie, uruchamianie, usypianie oraz pobieranie nazwy wątku.

```
package pl.wsb.programowaniejava.maciejgowin.przyklad63;

public class Main {

    public static void main(String[] args) {
        Runnable r1 = () -> {
            safeSleep(3000);
            System.out.println("R1: " + Thread.currentThread().getName());
        };
        Runnable r2 =  () -> {
            safeSleep(1000);
            System.out.println("R2: " + Thread.currentThread().getName());
        };

        System.out.println("Start: " + Thread.currentThread().getName());
        new Thread(r1).start();
        new Thread(r2).start();
        System.out.println("End: " + Thread.currentThread().getName());
    }

    private static void safeSleep(int time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Język Java: wykonywanie wątków

Należy zwrócić uwagę na fakt, że wątki nie są wykonywane sekwencyjnie na podstawie kolejności wywołania.

Wynika to z ich natury - w założeniu powinny one być równoległe, a nie sekwencyjne. To JVM wraz z systemem operacyjnym decyduje o ich kolejności wykonywania. Kolejność ta nie musi odpowiadać kolejności wywołania. Co więcej, kolejność ta może różnić się nawet podczas kolejnych uruchomień programu.

Wyścig (ang. race condition)

Wyścig (ang. race condition) występuje, w momencie, w którym dwa lub więcej wątków uzyskuje dostęp do współdzielonych danych i stara się je zmienić w tym samym momencie.

Blok kodu, w którego obrębie wiele wątków może spowodować wystąpienie wyścigu, nazywamy sekcją krytyczną.

W sekcji krytycznej na wyniki może wpłynąć nie tylko liczba wątków, ale również kolejność wykonania danych instrukcji.

Programowanie: przykład 64

Przykład wyścigu oraz sekcji krytycznej dla dwóch wątków dzielących dane.

```
package pl.wsb.programowaniejava.maciejgowin.przyklad64;

public class Counter {

    private int i;

    public void increment() {
        i = i + 1;
    }

    public int get() {
        return i;
    }
}
```

Typy wyścigów

Wyróżniamy dwa typy wyścigów:

- read-modify-write
- check-then-act

Read-modify-write

Dwa lub więcej wątków rozpoczyna odczyt danej zmiennej, następnie modyfikuje jej wartość, a na końcu zapisuje wartość do zmiennej.

Problem pojawia się w momencie, w którym dwa watki odczytują tę samą wartość przed jej zmodyfikowanie.

Typy wyścigów

Wyróżniamy dwa typy wyścigów:

- read-modify-write
- check-then-act

Check-then-act

Dwa lub więcej wątków sprawdza dany warunek, np. czy mapa zawiera daną wartość, następnie wykonuje inną operację na podstawie tego warunki.

Problem może powstać, jeżeli dwa wątki sprawdzają te samą wartość w mapie, a następnie starają się usunąć tę i pobrać tę samą wartość.

Zapobieganie wyścigom

Aby zapobiec wyścigom, musimy zapewnić, aby sekcja krytyczna została zawsze wykonana jako instrukcja atomowa.

Innymi słowy - jeżeli jeden wątek wykonuje sekcję krytyczną, żaden inny nie może jej wykonać, dopóki pierwszy jej nie opuścił (nie zakończył jej przetwarzania).

Zapobieganie wyścigom

Rozwiązaniem problemu jest synchronizacja wątków w sekcjach krytycznych. Synchronizacja ta może zostać osiągnięta poprzez:

- blok synchronizacyjny
- konstrukcje synchronizacyjne, takie jak blokady (ang. locks)
- zmienne atomowe, takie jak `java.util.concurrent.atomic.AtomicInteger`

Język Java: synchronizacja na poziomie metody

W kontekście synchronizacji Java pozwala na oznaczenie całej metody jako synchronizowanej. Dzięki temu w przypadku wywołania danej metody na danym obiekcie tylko jeden wątek będzie w stanie ją wykonać w danym momencie

```
public synchronized void increment() {
    i = i + 1;
}
```

Programowanie: przykład 65

Użycie `synchronized` na poziomie metody dla sekcji krytycznej dla współdzielonych danych.

```
package pl.wsb.programowaniejava.maciejgowin.przyklad65;

public class Counter {

    private int i;

    public synchronized void increment() {
        i = i + 1;
    }

    public int get() {
        return i;
    }
}
```

Język Java: synchronizacja na poziomie bloku

Jeżeli chcielibyśmy ograniczyć zasięg synchronizacji, do osiągnięcia podobnego efektu możemy użyć bloku synchronizacyjnego.

```
public  void increment() {
    synchronized (this) {
        i = i + 1;
    }
}
```

Należy zauważyć, że blok synchronizacyjny sparametryzowany jest referencją do obiektu. Obiekt ten nazywany jest monitorem. Mówimy, że kod jest synchronizowany na danym obiekcie monitora.

W danym momencie tylko jeden wątek może wykonać kod w bloku synchronizowanym na danym obiekcie monitora.

Język Java: wbudowane klasy atomowe

Java dostarcza też szereg klas, które definiują operacje atomowe i są bezpieczne w kontekście wielowątkowym.

Przykładam takiej klasy jest `AtomicInteger`, która mogłaby zostać użyta w zastępstwie naszego licznika.

Programowanie: przykład 66

Użycie `AtomicInteger` dla licznika w środowisku wielowątkowym.

```
package pl.wsb.programowaniejava.maciejgowin.przyklad66;

import java.util.concurrent.atomic.AtomicInteger;

public class Counter {

    private final AtomicInteger i = new AtomicInteger();

    public void increment() {
        i.incrementAndGet();
    }

    public int get() {
        return i.get();
    }
}
```

Optymalizacja oraz zmienne volatile

W przypadku braku synchronizacji kompilator, środowisko uruchomieniowe oraz kompilator mogą dokonywać optymalizacji na kodzie. Optymalizacje te mogą wprowadzać nieoczekiwane błędy.

Wyobraźmy sobie sytuację, w której dwa wątki zostają uruchomione na dwóch różnych procesorach. Dodatkowo pierwszy wątek pisze do zmiennej, przy czym drugi czyta wartości z tej samej zmiennej.

Każdy z wątków kopiuje wartość zmiennej z pamięci głównej do pamięci cache procesora dla ulepszenia wydajności. JVM nie definiuje momentu, w którym wartości te zostaną ponownie zapisane do pamięci głównej z pamięci cache.

Programowanie: przykład 67

Problemy widoczności zmiennej.

```
package pl.wsb.programowaniejava.maciejgowin.przyklad67;

public class Main {

    private static int value;
    private static boolean finished;

    private static class Reader extends Thread {

        @Override
        public void run() {
            while (!finished) {
                Thread.yield();
            }

            System.out.println(value);
        }
    }

    public static void main(String[] args) {
        new Reader().start();
        finished = true;
        value = 86;
    }
}
```

Język Java: zmienne volatile

Zmienne oznaczone jako `volatile` są pisane oraz odczytywane bezpośrednio z pamięci głównej. Nie są one przechowywane w pamięci cache procesora.

```
private volatile boolean check;
```

Volatile oraz Visibility Guarantee

Gwarancja widoczności (ang. visibility guarantee) zapewnia, że wartości zmienny volatile są widoczne od razu podczas zapisu i odczytu. Oznacza to, że zawsze operujemy na pamięci głównej.

Volatile oraz Happens-Before Guarantee

Wirtualna maszyna oraz procesor często optymalizują kolejność wykonywania niezależnych instrukcji, oraz wykonują je równolegle w celu poprawy wydajności.

Gwarancje wydarzy się - przed (ang. happens-before guarantee) zapobiega zmienianie kolejności instrukcji dla zmiennych volatile.

Gwarancja ta zapewnia, że:

- wszystkie instrukcje występujące przed instrukcją pisania do zmiennej volatile nie zostaną przeniesione za instrukcję przypisania,
- wszystkie instrukcje występujące po instrukcji czytania ze zmiennej volatile nie zostaną przeniesione przed instrukcję odczytu.

Volatile a synchronized

Dla zapewnienia poprawności działania aplikacji wielowątkowych i obsługi sekcji krytycznych muszą zostać spełnione warunki:

- wzajemne wykluczanie (ang. mutual exclusion) - w danym momencie tylko jeden wątek może wykonywać sekcję krytyczną
- widoczność (ang. visibility) - zmiany dokonane przez jeden wątek na współdzielonych danych są widoczne dla wszystkich innych wątków w celu zapewnienia spójności danych

Założenia te są realizowane poprzez poprawną synchronizację kosztem wydajności aplikacji.

Zmienne volatile pozwala na dostarczanie poprawnej widoczności, nie wspierając wzajemnego wykluczania.

Bezpieczeństwo wątków

Bezpieczeństwo wątków (ang. thread safety) to zapewnienie poprawnego wykonania kodu bez nieprzewidzianych zachowań. Istnieje szereg rozwiązań pozwalających na zapewnienie tego bezpieczeństwa.

1. Bezstanowość.
2. Niezmienność obiektów (ang. immutability).
3. Wartości lokalne w obrębie wątku (ang. thread-local).
4. Wbudowane klasy obsługujące wielowątkowość (np. `ConcurentHashMap`) lub opakowania obiektów (np. `Collections.synchronizedCollection`).
5. Wbudowane klasy z operacjami atomowymi (np. `AtomicInteger`).
6. Metody i bloki synchronizacyjne.
7. Zmienne `volatile`.
8. Wbudowane mechanizmy blokad (ang. lock).

Wątki i ExecutorService

W przypadku wątków i zadań do wykonania nie zawsze chcemy tworzyć nowy wątek. Java wprowadza ideę puli wątków, które mogą zostać użyte do wykonywania danego zadania. Pule wątków obsługiwane są poprzez interfejs `java.util.concurrent.ExecutorService`.

Aby stworzyć pulę wątków, możemy skorzystać ze statycznych metod należących do klasy `java.util.concurrent.Executors`.

```
ExecutorService executorService = Executors.newFixedThreadPool(3);
```

Wątki i ExecutorService

Po utworzeniu puli dodajemy zadania do wykonania za pomocą metody `ExecutorService.submit()`.

```
executorService.submit(new Runnable() {
    @Override
    public void run() {
        System.out.println("Executing task");
    }
});

executorService.submit(new Callable<String>() {
    @Override
    public String call() throws Exception {
        return null;
    }
});
```

Wątki i ExecutorService

Java rozszerza `Runnable` o kolejny interfejs `java.util.concurrent.Callable`, który ma podobne zadanie. Główna różnica polega na tym, że interfejs ten zwraca wartość oraz metoda dla interfejsu funkcyjnego może zgłosić wyjątek.

Wątki i ExecutorService

Po dodaniu zadań do kolejki będą one wykonywane, aż do momentu zakończenia. Operacja kolejkowania elementów zwraca obiekty typy `java.util.concurrent.Future` opisujące zadanie oraz przyszłe rezultaty wywołania.

Możemy zablokować działanie wątku głównego w oczekiwaniu na rezultaty poprzez wywołanie `Future.get()`.

Wątki i ExecutorService

Należy pamiętać, że pula wątków powinna zostać wyłączona po zakończeniu pracy.

```
executorService.shutdown();
```

W przypadku braku wyłączenia pula oraz wątki do niej przypisane będą oczekiwały na kolejne zadania.

Programowanie: przykład 68

Pula wątków i `ExecutorService` w praktyce.

```
package pl.wsb.programowaniejava.maciejgowin.przyklad68;

import java.time.OffsetDateTime;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {

    public static void main(String[] args) {
        System.out.println("Started processing");
        ExecutorService executorService = Executors.newFixedThreadPool(3);

        List<Future<String>> futures = Stream.of(100, 200, 100, 300, 400, 600)
                .map(Main::getCallable)
                .map(executorService::submit)
                .collect(Collectors.toList());

        futures.stream()
                .map(future -> {
                    try {
                        return future.get();
                    } catch (Exception ex) {
                        return ex.getMessage();
                    }
                })
                .forEach(System.out::println);

        executorService.shutdown();

        System.out.println("Finished processing");
    }

    public static Callable<String> getCallable(final int sleep) {
        return () -> {
            Thread.sleep(sleep);
            return String.format("Thread: %s: finished task: %s",
                    Thread.currentThread().getName(), OffsetDateTime.now());
        };
    }
}
```