Programowanie współbieżne i rozproszone

WYKŁAD 4

Krzysztof Pancerz

Modyfikator volatile

- **volatile** - modyfikator informujący kompilator, że oznaczona nim zmienna może zostać nieoczekiwanie zmieniona przez inną część programu (wątek).
- Ze względu na wydajność każdy wątek może przechowywać prywatną kopię współdzielonej zmiennej. Rzeczywista wersja zmiennej uaktualniana jest tylko co jakiś czas.
- **volatile** wymusza na kompilatorze stosowanie rzeczywistej wersji zmiennej lub ciągłe uaktualnianie każdej jej kopii.

Pakiet java.util.concurrent.atomic

- Pakiet **`java.util.concurrent.atomic`** pozwala na wykorzystywanie zmiennych w środowisku wielowątkowym.
- Pakiet ten umożliwia aktualizację wartości zmiennej bez konieczności wykorzystywania blokad.

Pakiet java.util.concurrent.atomic

- Przykładowe klasy pakietu **`java.util.concurrent.atomic`**:
  - **`AtomicBoolean`**
  - **`AtomicInteger`**
  - **`AtomicLong`**
  - **`AtomicIntegerArray`**
  - **`AtomicReference`**
  - itp.

Pakiet java.util.concurrent.atomic

- Przykładowe metody klasy **`AtomicInteger`**:
  - **`int decrementAndGet()`**
  - **`int getAndDecrement()`**
  - **`int addAndGet(int delta)`**
  - **`int getAndAdd(int delta)`**
  - itp.

Kolekcje współbieżne

- Pakiet **`java.util.concurrent`** zawiera definicje kilkunastu klas kolekcji, opracowanych specjalnie dla potrzeb programów współbieżnych.
- Przykładowe kolekcje współbieżne:
  - **`ConcurrentHashMap`**
  - **`ConcurrentLinkedQueue`**
  - **`ConcurrentSkipListMap`**
  - itp.

Pakiet java.util.concurrent

- Klasa **`CountDownLatch`**:
  - Klasa pozwala na określenie pewnej liczby zdarzeń, które muszą się pojawić aby zatrzask, na którego zwolnienie czeka wątek został zwolniony.
  - Po każdym zdarzeniu zmniejszana jest wartość licznika, gdy osiągnie ona zero, zatrzask jest zwolniony.
  - Oczekiwanie wątku na zwolnienie zatrzasku realizowane jest przez metodę **await**.
  - Sygnalizowanie wystąpienia zdarzenia odbywa się przy pomocy metody **countDown**. Każde jej wywołanie zmniejsza wartość licznika o 1.

Pakiet java.util.concurrent

- Klasa **`CountDownLatch`**:
  - Klasa pozwala na określenie pewnej liczby zdarzeń, które muszą się pojawić aby zatrzask, na którego zwolnienie czeka wątek został zwolniony.
  - Po każdym zdarzeniu zmniejszana jest wartość licznika, gdy osiągnie ona zero, zatrzask jest zwolniony.
  - Oczekiwanie wątku na zwolnienie zatrzasku realizowane jest przez metodę **await**.
  - Sygnalizowanie wystąpienia zdarzenia odbywa się przy pomocy metody **countDown**. Każde jej wywołanie zmniejsza wartość licznika o 1.

Pakiet java.util.concurrent

- Klasa `CyclicBarrier`:
  - Klasa pozwalająca dwóm lub więcej liczbie wątków oczekiwać w określonym punkcie programu do chwili, gdy wszystkie wątki go osiągną.

Pakiet java.util.concurrent

- Procedura wykorzystania klasy **`CyclicBarrier`**:
  - Utworzenie obiektu klasy `CyclicBarrier` i określenie liczby wątków, na które w punkcie bariery będzie realizowane czekanie.
  - W oczekiwanych wątkach, muszą zostać wywołane metody **await**. Gdy wszystkie oczekiwane wątki wywołają tą metodę, zostanie ona zakończona, a program wznowiony.

Pakiet java.util.concurrent

- Klasa **`Exchanger`**:
  - Klasa upraszczająca wymianę danych pomiędzy dwoma wątkami.
- Procedura wykorzystania klasy **`Exchanger`**:
  - Utworzenie obiektu klasy `Exchanger`.
  - W wątkach wymieniających dane, muszą zostać wywołane metody **exchange**. Gdy oba wątki wywołają tą metodę, realizowana jest wymiana danych.

Pakiet java.util.concurrent

- Klasa **`Phaser`**:
  - Klasa upraszczająca synchronizację wątków wykonujących wiele faz operacji.
  - Przejście do kolejnej fazy operacji możliwe jest tylko wówczas, gdy wszystkie wątki zakończą wykonywanie fazy poprzedniej.

Pakiet java.util.concurrent

- Procedura wykorzystania klasy **`Phaser`**:
  - Utworzenie obiektu klasy `Phaser`.
  - Metoda **register** służy do zarejestrowania strony biorącej udział w synchronizacji (tj. wątku).
  - Metoda **arrive** służy do sygnalizacji zakończenia wykonywania danej fazy przez stronę.
  - Metoda **arriveAndAwaitAdvance** służy do sygnalizacji zakończenia wykonywania danej fazy przez stronę i przejście w stan oczekiwania do momentu zakończenia tej samej fazy przez wszystkie pozostałe zarejestrowane strony.

Pakiet java.util.concurrent

- Procedura wykorzystania klasy **`Phaser`**:
  - Metoda **arriveAndDeregister** służy do sygnalizacji zakończenia wykonywania danej fazy przez stronę i wyrejestrowania się z dalszego udziału w synchronizacji.

Pakiet java.util.concurrent

- Klasa **`Executors`** i interfejs **`ExecutorService`**:
  - Zestaw tworzący narzędzie zwane egzekutorem, który grupuje wątki w pule wątków i kontroluje ich uruchamianie.

Pakiet java.util.concurrent

- Schemat wykorzystania egzekutora:

```
ExecutorService es=Executors.newFixedThreadPool(2);

// ...

es.execute(new Watek());
es.execute(new Watek());
es.execute(new Watek());
es.execute(new Watek());

// ...

es.shutdown();
```

Pula wątków zawiera dwa wątki. Wszystkie wątki zostaną wykonane, ale tylko dwa w tym samym czasie

Pominięcie wywołania shutdown spowoduje, że program się nie zakończy (wątki w puli będą nadal aktywne)

Programowanie równoległe przy użyciu Fork/Join

- Pakiet **`java.util.concurrent`** zawiera definicje frameworku **Fork/Join** wspierającego programowanie równoległe.
- Framework Fork/Join upraszcza tworzenie i stosowanie wielu wątków.
- Framework Fork/Join pozwala na automatyczne korzystanie z wielu procesorów.

Programowanie równoległe przy użyciu Fork/Join

- Tradycyjne narzędzia programowania wielowątkowego w Javie nie są optymalizowane dla środowisk wieloprocesorowych (wielordzeniowych).
- Te narzędzia w środowiskach jednoprocesorowych umożliwiają przede wszystkim efektywne współdzielenie czasu procesora przez wiele zadań (np. jeden wątek może wykonywać swoje operacje w czasie, gdy inny wątek oczekuje na jakieś zdarzenie).

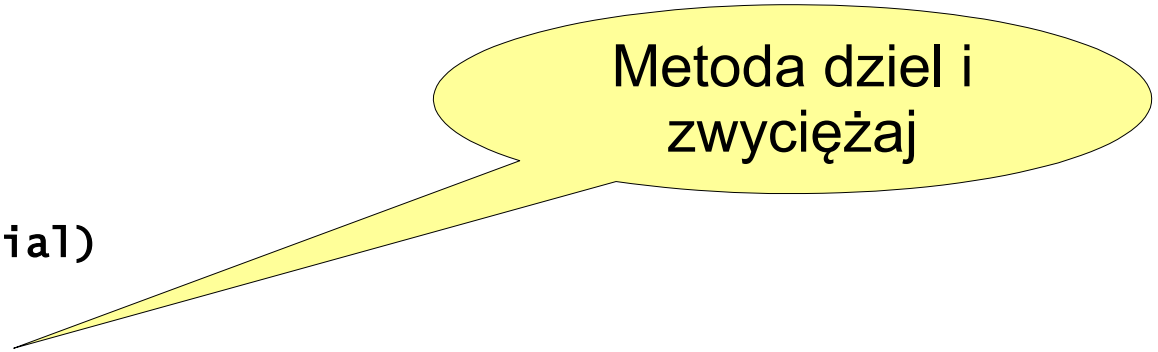
Programowanie równoległe przy użyciu Fork/Join

- Podstawowe klasy frameworku Fork/Join:
  - **`ForkJoinTask<V>`** - klasa abstrakcyjna definiująca zadanie
  - **`ForkJoinPool`** - klasa umożliwiająca zarządzanie zadaniami ForkJoinTaks
  - **`RecursiveAction`** - podklasa klasy ForkJoinTaks, dla zadań które nie zwracają żadnej wartości
  - **`RecursiveTask<V>`** - podklasa klasy ForkJoinTaks, dla zadań zwracających wartość

Pakiet java.util.concurrent

- Schemat wykorzystania Fork/Join:

```
class Operacja extends RecursiveAction
{
  // ...
  void compute()
  {
     if(dalej_podzial)
     {
        invokeAll(new Operacja(dane, poczatek, srodek),
                  new Operacja(dane, srodek+1, koniec));
     }
     else
     {
         // ... obliczenia ...
     }
  }
}
```



Pakiet java.util.concurrent

- Schemat wykorzystania Fork/Join (cd.):

```
class Glowna
{
  // ...
  public static void main(String[] args)
  {
     ForkJoinPool fjp=new ForkJoinPool();
     // ...
     fjp.invoke(new Operacja(dane, 0, dane.length));
  }
}
```