Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium: Wielowątkowość w Java

Przedmiot: Przetwarzanie Równoległe i Rozproszone

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący ćwiczenia: dr inż. Mateusz Sitko

Data: 6 Grudnia 2024

Numery lekcji: 6, 7, 8

Grupa laboratoryjna: 4

Cel Zajęć

Celem zajęć było zapoznanie nas z wielowątkowością w języku Java. Wykorzystanie w praktycznych przykładach wybranych mechanizmów zarządzania wątkami.

Wielowątkowość w Java

Wątki i Podstawy Wielowątkowości

Klasa **Thread** jest podstawowym narzędziem w Javie do tworzenia i zarządzania wątkami. Nowy wątek tworzony jest poprzez nadpisanie metody **run()** w klasie dziedziczącej po Thread.

```
public class ThreadVariant extends Thread
{

    // Atrybuty
    // ......

    // Konstruktor
    public ThreadVariant1(){}

    // Metoda wywołana przez wątek
    public void run()
    {
        // Kod wykonywany przez wątek
        // ......
    }

    // Przykładowy getter
    public int get_result()
    {
        // Zwrot wyniku
        // ......
    }
}
```

Metoda **start()** uruchamia wątek, który następnie wykonuje swoją logikę zawartą w metodzie **run()**. Bezpośrednie wywołanie **run()** wywoła metodę na głównym wątku.

```
int thread_count = 5; // Liczba wątków
ThreadVariant[] threads = new ThreadVariant[thread_count];

// Tworzenie wątków i uruchamianie ich
for (int i = 0; i < thread_count; i++)
{
    threads[i] = new ThreadVariant();
    threads[i].start();
}
```

Podobnie jak w wątkach POSIX, musimy zaczekać na wątki aby zakończyły swoją pracę. Możemy oczywiście używać innych metod naszej klasy reprezentującej wątek, na przykład gettera zwracającego wynik.

```
for (int i = 0; i < thread_count; i++)
{
    try
    {
        threads[i].join();
        int result = threads[i].get_result();
        System.out.printf("Wynik wątku: %d = %d\n", i, result);
    }
    catch (InterruptedException e)
    {
        System.err.printf("Błąd podczas dołączania wątku: %d\n", i);
    }
}
```

Ponieważ Java nie obsługuje wielokrotnego dziedziczenia klas, klasy dziedziczące po **Thread** tracą możliwość dziedziczenia po innych klasach.

Rozwiązaniem tego problemu jest interfejs **Runnable**, który umożliwia definiowanie logiki wątku w metodzie **run()** bez konieczności dziedziczenia po **Thread**. Dzięki temu klasa może dziedziczyć po innej klasie, a jednocześnie implementować interfejs **Runnable**, by była kompatybilna z mechanizmami wielowątkowości w Javie.

Tworzenie klasy w tym wypadku wygląda niemal tak samo z różnicą, że interfejs zastępuje dziedziczenie.

```
public class RunnableThread implements Runnable
{
    // Obecność metody wymuszona przez interfejs
    public void run()
    {
        // Kod wykonywany przez wątek
    }
}
```

Wykorzystanie różni się wyłącznie tworzeniem.

```
// Tworzenie instancji klasy implementującej Runnable
Runnable runnable_task = new RunnableThread();

// Tworzenie wątku i przekazywanie obiektu Runnable
Thread thread = new Thread(runnable_task);

// Uruchomienie wątku
thread.start();
```

ThreadPool – Mechanizm zarządzający pracą

ThreadPool jest mechanizmem zarządzającym kontrolą nad wątkami i zadaniami. Mechanizm ten przydziela zadania do wątków wewnątrz dynamicznej lub stałej puli ustalonej przez użytkownika. Duża zaleta mechanizmu jest to, że potrafi on rozdysponować zadania w taki sposób aby zmaksymalizować wydajność i ograniczyć sytuacje w których wątki są bezczynne.

Przykładowe użycie Executora i puli wątków. Executorowi możemy przekazać zadanie poprzez metody:

- **execute(Runnable command)** – przyjmuje zadanie interfejsu **Runnable**.
- **submit(Callable<T> task)** – przyjmuje zadanie interfejsu **Callable** i zwraca obiekt **Future** jako wynik.

```
// Utworzenie puli wątków o rozmiarze 3
ExecutorService executor = Executors.newFixedThreadPool(3);

// Tworzenie i uruchamianie wątków w pętli
for (int i = 1; i <= 5; i++)
    executor.execute(new RunnableThread(i));
```

Executor działa dopóki nie zostanie on zatrzymany przez metodę **shutdown()** lub wątek główny się skończy. Koniec zadań nie zakończy jego pracy. Wywołanie metody spowoduje, że executor przestanie przyjmować nowe zadania i po zakończeniu już przyjętych zadań zakończy pracę.

```
// Tell the executor to wrap up
executor.shutdown();
```

Wywołanie tej metody nie zatrzyma dalszej egzekucji kodu w wątku głównym. Aby upewnić się, że Executor dostanie tyle czasu ile potrzebuje musimy wstrzymywać wątek główny.

```
try
{
    boolean ignore =  executor.awaitTermination(1, TimeUnit.MINUTES);
}
catch (InterruptedException exception)
{
    System.err.println("Exception: " + exception);
}
```

Obiekty **Future** pozwalają nam na odbieranie wyników bez używania getterów. Obiekty tę są zwracane przez klasy / zadania implementujące interfejs **Callable**. Interfejs wymusza implementacje metody **call()**, która będzie wykonywana przez wątek oraz finalnie zwróci wynik.

```
public class CallableThread implements Callable<int>
{
      public int call()
      {
          // Zwrot wyniku
      }
}
```

Przykład dodania zadań **Callable** do **Executora.**

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

// Lista przechowująca Future wyników
List<Future<Integer>> results = new ArrayList<>();

// Tworzenie i przekazywanie zadań do executora
for (int i = 0; i < 5; i++)
    results.add(executor.submit(new CallableThread(i)));
```

Wyników oczywiście nie będziemy mogli odczytać od razu. Pojawią się one dopiero później w obiektach **Future**. Możemy je odczytać poprzez metodę **get()**, która zatrzyma wątek główny dopóki dany wynik nie będzie dostępny.

```
// Pobieranie wyników z Future
try
{
    for (int i = 0; i < results.size(); i++)
        System.out.println("Wynik zadania " + (i + 1) + ": " + results.get(i).get());
}
catch (InterruptedException | ExecutionException e)
{
    System.err.println("Błąd podczas wykonywania zadania: " + e.getMessage());
}
finally
{
    executor.shutdown();
}
```

ForkJoin – Rekurencyjne Przetwarzanie Wielowątkowe

ForkJoin jest frameworkiem służącym do przetwarzania równoległego zadań, które można podzielić na mniejsze podzadania. Zadania dla ForkJoinPool dziedziczą po RecursiveAction (bez zwrotu wyniku) lub RecursiveTask<V>.

Praca odbywa się w dziedziczonej metodzie Compute. Poniżej jest przykład. Metoda **fork()** tworzy nowy wątek z zadaniem, podczas gdy metoda **join()** czeka na zakończenie pracy wątku i zwrot wyniku. Wywołanie metody **compute()** działa jak zwykła rekurencja i dzieje się na bezczynnym wątku.

```
class SumTask extends RecursiveTask<Integer>
{

    protected Integer compute()
    {
        // Warunek stopu
        // if ( ... )
        // {

        // }
        // else
        // {
            // Podziel zadanie na dwie części
            // ...

            // Wykonaj równolegle
            left_task.fork();
            int right_result = right_task.compute();
            int left_result = left_task.join();

            return left_result + right_result;
        // }
    }
}
```

Uruchamiamy zadanie z ForkJoinPool zgodnie z poniższym przykładem.

```
// Array = przykładowy argument
ForkJoinPool pool = new ForkJoinPool();
SumTask task = new SumTask(array);

int result = pool.invoke(task);
System.out.println("Rozwiązanie zadania: " + result);
```

Mechanizmy Synchronizacji

Deklaracja **synchronized** – blokuje metodę tak aby wykonywał ją tylko jeden wątek w czasie. Deklaracja pozwala na używanie metod takich jak **wait()**, która pozwala na zatrzymanie działania kodu do otrzymania sygnału **notify()** lub **notifyAll()**.

```
public synchronized void wait_for_signal() throws InterruptedException
{
    while (!ready)
    {
        System.out.println("Czekanie na sygnał...");
        wait(); // Wstrzymanie wątku
    }
    System.out.println("Sygnał otrzymany!");
}

public synchronized void send_signal()
{
    ready = true;
    System.out.println("Wysyłanie sygnału...");
    notify(); // Obudzenie jednego wątku
}
```

ReentrantLock – Działa jak mutex w wątkach POSIX, jednak oferuje większą kontrolę nad mechanizmami synchronizacji. Pozwala na tworzenie wielu niezależnych kolejek warunkowych za pomocą obiektów **Condition**. Dzięki temu umożliwia bardziej precyzyjne zarządzanie sygnalizacją między wątkami, np. wstrzymując wątki w różnych kolejkach (**await()**) i budząc je selektywnie (**signal()** lub **signalAll()**).

```
private final ReentrantLock lock = new ReentrantLock();
private final Condition condition = lock.newCondition();

public void wait_for_signal() throws InterruptedException
{
    lock.lock();
    try
    {
        condition.await();
    }
    finally{lock.unlock();}
}
public void send_signal()
{
    lock.lock();
    try
    {
        condition.signal();
    }
    finally {lock.unlock();}
}
```