Mechanizmy synchronizacji wątków w Javie w mniej niż 15 minut

Spis treści:

Mechanizmy synchronizacji wątków w Javie – jakie problemy może rozwiązać?

Dzięki zastosowaniu wielowątkowości możemy w pełni wykorzystać zasoby naszego komputera. W momencie, gdy program czeka na odczyt czy zapis danych na dysk, akcje użytkownika lub odbiera dane z sieci, procesor czeka bezczynnie. W tym momencie za pomocą dodatkowych wątków możemy zrównoleglić wiele działań i utrzymać procesor w ciągłej pracy. Dzięki temu w możemy wielokrotnie przyspieszyć czas uzyskania końcowego rezultatu.

Dużym problemem staje się jednak synchronizacja pracy wątków. W momencie gdy o nią nie zadbamy, możemy bardzo łatwo doprowadzić do zdeformowania stanu naszego systemu i skończyć z błędnymi danymi.

Jeśli na przykład nie zsynchronizujemy w odpowiedni sposób operacji wypłaty pieniędzy z bankomatu, to możemy albo skończyć z debetem na koncie (pobrane pieniądze z konta, ale niewypłacone), albo ze sztucznym wykreowaniem pieniądza (pieniądze wypłacone, ale niepobrane z konta). **Dlatego koniecznym jest, byśmy rozumieli mechanizmy synchronizacji kodu i odpowiednio z nich korzystali.**

Zapraszam do krótkiego przeglądu dostępnych możliwości.

1. synchronized

Pierwszym mechanizmem służącym do synchronizacji kodu jest słowo kluczowe `synchronized`. Można je założyć na całej metodzie lub na fragmencie kodu. Spójrzmy na poniższy, problematyczny fragment.

```
// DON'T DO THAT! :(
void transferMoney(BigDecimal amount, Account fromAccount, Account toAccount) throw
  if(fromAccount.balance() >= amount) {
    fromAccount.reduce(amount);
    toAccount.add(amount);
  } else {
    throw new InsufficientFundsException("Unable to transfer money. Insufficient fu
  }
}
```

Metoda `transferMoney` nie jest w żadnej sposób synchronizowana przed dostępem wielowątkowoym. Co to oznacza? A no to, że jeśli dwa wątki będą próbowały ją równocześnie wykonać, to może dojść do... tragedii! Zarówno wątek pierwszy, jak i drugi mogą sprawdzić warunek `if` w tym samym momencie i wejść do bloku kodu związanego z przesyłaniem pieniędzy. Jeśli okaże się, że pieniędzy na koncie było tylko tyle, by wystarczyło na jeden przelew, to skończymy z błędnym stanem konta. Ilość pieniędzy na tym koncie będzie ujemna. Dlatego musimy się przed tym jakoś zabezpieczyć.

W tym momencie cały na biało wchodzi blok `synchronized`, za pomocą którego **pozwolimy tylko jednemu wątkowi** wykonać kod w jednej chwili. W ten sposób upewnimy się, że cała operacja wykona się poprawnie i w przypadku braku funduszy przelew nie powiedzie się.

Tym samym **zagwarantujemy poprawność działania naszego systemu i spójność danych**.

W jaki sposób zastosować mechanizm `synchronized`? Jedną z możliwości jest dodanie słowa kluczowego w sygnaturze metody.

```
synchronized void transferMoney(BigDecimal amount, Account fromAccount, Account toA
  if(fromAccount.balance() >= amount) {
    fromAccount.reduce(amount);
    toAccount.add(amount);
  } else {
    throw new InsufficientFundsException("Unable to transfer money. Insufficient fu
  }
}
```

Lub jako blok kodu.

```
void transferMoney(BigDecimal amount, Account fromAccount, Account toAccount) throw
  synchronized(this) {
    if(fromAccount.balance() >= amount) {
      fromAccount.reduce(amount);
      toAccount.add(amount);
    } else {
      throw new InsufficientFundsException("Unable to transfer money. Insufficient
    }
  }
}
```

Od tej pory dwa wątki, które na tym samym obiekcie będą próbowały wołać metodę `transferMoney`, będą musiały zaczekać i wykonają się nie jednocześnie, a jeden po drugim.

Słowo kluczowe `synchronized` można również nakładać na dedykowanych obiektach. W powyższych przykładach zostało ono założone na obecnej instancji danej klasy (`this`), ale nic nie stoi na przeszkodzie, by przekazać tam inny obiekt.

```
class AccountManager {
  private final Object lock = new Object();

  void transferMoney(BigDecimal amount, Account fromAccount, Account toAccount) thr
    synchronized(lock) {
      if(fromAccount.balance() >= amount) {
        fromAccount.reduce(amount);
        toAccount.add(amount);
      } else {
        throw new InsufficientFundsException("Unable to transfer money. Insufficien
      }
    }
  }
}
```

Dzięki temu zyskujemy większą kontrolę, na których obiektach zakładane są bloki `synchronized` i możemy lepiej decydować, które części kodu mogą wykonać się równolegle, a które nie.

Co więcej, nieuważny programista mógłby w innym miejscu wykorzystać instancję klasy `AccountManager` do synchronizacji i w ten sposób doprowadzić do problemów z `deadlockami`,

których chcemy uniknąć.

2. Locki

Drugim sposobem blokowania sekcji krytycznej, o którym warto wiedzieć są `Locki`, a szczególnie ich sub-interfejs – `ReadWriteLock`. Działają one podobnie jak słowo kluczowe `synchronized`, ale dają programistom dużo większą swobodę działania.

Jak sama nazwa wskazuje, w tym przypadku **mamy do czynienia z dwoma rodzajami locków: *read* – do odczytu oraz *write* – do zapisu.**

W sekcjach, w których korzystamy z *read locków* wiele wątków naraz może uzyskać dostęp do bloku kodu. W miejscu, w którym pojawia się *write lock*, tylko jeden wątek na raz może wykonywać kod.

Poniżej prezentuję przykład zastosowania tego mechanizmu. W klasie `Book` mamy dwie metody: `editLine` oraz `readLine`.

Pierwsza synchronizowana jest z pomocą `writeLocka`, co oznacza, że **dozwolony jest dostęp tylko dla jednego wątku**. W momencie uzyskania tego locka, żaden inny wątek nie korzysta z metody `editLine`, ani z metody `readLine`.

Druga metoda chroniona jest `readLockiem`. **W tej sytuacji wiele wątków jednocześnie może wołać tę metodę**, ale w sytuacji, gdy jakiś wątek woła metodę `editLine` wówczas `readLock` nie może być uzyskany.

Oznacza to, że w momencie modyfikacji zawartości książki tylko jeden wątek może korzystać z tej klasy. W momencie czytania, dostęp ma wiele wątków.

```
class Book {
  private String[] lines;
  private final ReadWriteLock lock = new ReentrantReadWriteLock();
  private final Lock readLock = lock.readLock();
  private final Lock writeLock = lock.writeLock();

  public void editLine(int lineNumber, String content) {
    writeLock.lock();
    try {
      lines[lineNumber] = content;
    } finally {
      writeLock.unlock();
    }
  }

  public void readLine(int lineNumber) {
    readLock.lock();
    try {
      String content = lines[lineNumber];
      return content;
    } finally {
      readLock.unlock();
    }
  }
}
```

`ReadWriteLocki` świetnie sprawdzają się w klasach, w których liczba odczytów przewyższa liczbę zapisów. Dzięki zastosowaniu dwóch rodzajów locków **uzyskujemy lepszą wydajność** danej klasy,

jednocześnie zachowując spójność zawartych w niej danych.

3. Semaphory

Wiemy już, w jaki sposób synchronizować dostęp do sekcji krytycznych wielu wątków. W tej części pokażę Ci, **jak limitować** dostęp do pewnego zasobu.

Służą do tego `Semaphory`. Dzięki nim możemy określić, ile maksymalnie wątków może korzystać z danego zasobu na raz. Jeśli dany limit nie został osiągnięty, kolejny wątek uzyskuje dostęp. W przeciwnym wypadku czeka w kolejce.

Wyobraź sobie na przykład wizytę w gabinecie fryzjerskim, w którym pracuje troje fryzjerów. Dopóki są wolni, kolejni klienci mogą korzystać z ich usług. Gdy wszystkie miejsca są zajęte, następni chętni ustawiani są w kolejce. Do strzyżenia mogą podejść, gdy jakiejś miejsce się zwolni.

```
class HairSalon {
  private Semaphore chairs = new Semaphore(3);

  public void takeAHaircut(Customer customer) {
    chairs.acquire();
    cutHair(customer);
    chairs.release();
  }
}
```

W ten łatwy sposób udało nam się zarządzić kosztownym zasobem. Semafory świetnie się w takich sytuacjach sprawdzają i pomagają w utrzymywaniu określonych limitów. Warto je używać na przykład do **ograniczania połączeń do bazy danych, zewnętrznych serwisów HTTP czy kontroli ciężkich operacji typu** `IO` (wejścia-wyjścia).

4. Latche

Czasami potrzebujemy zsynchronizować moment startu pracy wielu wątków. W tym celu możemy korzystać z `Latchy`, a konkretnie służy do tego `CountDownLatch`.

Wyobraź sobie bieg na 100m. Nie chcesz, żeby zawodnicy wystartowali w momencie zajęcia miejsca na blokach startowych, ale dopiero na konkretny znak. I właśnie tym znakiem może być sygnał od klasy `CountDownLatch`.

W poniższym przykładzie tworzymy taki przykładowy wyścig, w którym na dany sygnał `startSignal.countDown()` pozwalamy wcześniej stworzonym uczestnikom wyścigu rozpocząć bieg.

Na koniec natomiast czekamy aż wszyscy z nich dadzą znać, że bieg ukończyli i dopiero wtedy możemy udekorować zwycięzców. Dzięki temu, że `CountDownLatch` pozwala nam ustalić liczbę sygnałów, których oczekujemy, możemy w łatwy sposób zsynchronizować te dwa zdarzenia.

Do wystartowania biegu wystarczy nam jeden sygnał, który zwalniamy w momencie utworzenia wszystkich biegaczy. Do zakończenia biegu będziemy potrzebować tyle sygnałów, ile jest biegaczy. Wtedy mamy pewność, że kolejne instrukcje kodu wykonają się po skończeniu biegu przez wszystkich uczestników.

```
class Race {
  Executor executor = Executors.newCachedThreadPoolExeutor();

  public void start(int runners) {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(runners);
```

```
    for(int i = 0; i < runners; ++i) {
      Runner runner = new Runner(startSignal, doneSignal);
      executor.submit(runner);
    }
    doSomethingElse();
    startSignal.countDown();
    doSomethingElse();
    doneSignal.await();
    log.info("Race finished! All runners run.");
  }
}

class Runner implements Runnable {
  private final CountDownLatch startSignal;
  private final CountDownLatch doneSignal;

  public Runner(CountDownLatch startSignal, CountDownLatch doneSignal) {
     this.startSignal = startSignal;
     this.doneSignal = doneSignal;
  }

  public void run() {
    try {
      startSignal.await(); // wait for start signal
      runVeryHard();
      doneSignal.countDown(); // mark race finish
    } catch (InterruptedException E) {
      //
```

```
    }
  }
}
```

`CountDownLatch` świetnie przydaje się również do testów współbieżnych. Możemy najpierw stworzyć wiele wątków wykonujących operacje na metodach naszej klasy, ale rozpocząć ich działanie dopiero na konkretny sygnał. Dzięki temu zwiększamy szanse na to, że będą one uruchomione jednocześnie i lepiej sprawdzą zachowanie klas w środowisku wielowątkowym.

5. Barriery

Co w sytuacji, **gdy chcesz grupować wiele wątków tak, by wykonywały pracę w określonej liczbie?** Na przykład tworząc kosztowny zasób, chcesz by był stworzony dopiero w momencie, gdy pojawiła się na niego określona liczba chętnych. Jeśli do tego doszło, tworzysz ten zasób i uruchamiasz operacje.

Kolejny taki zasób tworzysz dopiero w momencie, gdy pojawi się kolejna określona grupa chętnych. Właśnie w tym celu została stworzona struktura `CyclicBarrier`.

Przykład z życia? Wyobraź sobie przepływ tratwami na Dunajcu. Chcemy w pełni wykorzystać nasze zasoby, dlatego tratwy ruszają dopiero, gdy znajdzie się pełna grupa 12 turystów. Dopóki chętnych jest mniej, dopóty czekamy na kolejne osoby.

W ten sposób, dzięki cyklicznej barierze, możemy łączyć wątki w grupy i kontrolować sposób ich pracy. Przykład poniżej.

```
class Rafting {
  private Barrier capacity = new CyclicBarrier(12);
```

```
  void join(Tourist tourist) {
    capacity.await(); // wait for 12 tourists to join
    Raft raft = newRaft();
    raft.takeASeat(tourist);
    raft.sail();
  }
}
```

Za pomocą barier możemy też kontrolować kiedy zostaną wykonane obliczenia, gdy spodziewamy się wyników z wielu źródeł by zoptymalizować kalkukacje, grupować uczestników wycieczek i tym podobne.

W przeciwieństwie do `Latchy`, `Barriery` automatycznie się resetują i w momencie osiągnięcia określonego progu za pierwszym razem, bariera zostaje opuszczona kolejny raz, gdy próg zostanie osiągnięty ponownie. W `Latchach` raz osiągnięty próg powoduje uruchomienie operacji już na zawsze.

Podsumowanie. Mechanizmy synchronizacji wątków w Javie

Jak widzisz, w Javie dostępnych mamy wiele mechanizmów synchronizacji pracy wątków. W zależności od potrzeb, możemy korzystać ze słowa kluczowego `synchronized`, *z locków, semaphorów, latchy* czy *barrier*. Zapoznaj się z nimi, aby lepiej decydować, z których kiedy korzystać.