Laboratorium Przetwarzanie Rozproszone

Instrukcje do laboratoriów z Javy

Jan Majkutewicz

04.2024

1 Wielowątkowość w Javie

1.1 Wstęp

Java, podobnie jak inne języki programowania, umożliwia implementację przetwarzania wielowątkowego. W chwili startu programu w Javie wykonywany jest jeden wątek (główny), a w czasie działania programu możliwe jest tworzenie kolejnych wątków, które będą wykonywane równolegle. Wątki mogą posiadać dostęp do wspólnych zmiennych w ramach jednej aplikacji. Współczesne aplikacje napisane w Javie bardzo często składają się z kilkuset wątków.

Szczegółowy opis mechanizmów wielowątkowości, koncepcji i metody potrzebnych do pisania wątkowo-bezpiecznych (ang. thread-safe) i skalowalnych programów w Javie można znaleźć w książce "Java Concurrency in Practice".

1.2 Implementacja przetwarzania wielowątkowego

1.2.1 Klasa Thread

Jednym ze sposobów wykonania wielu wątków jest rozszerzanie standardowej klasy `Thread` dostępnej w pakiecie java.lang. Należy zdefiniować własną klasę (np. `HelloThread`), dziedziczącą po klasie `Thread`, i nadpisać w niej metodę `run`, która będzie wykonywana w oddzielnym wątku.

Dalej należy utworzyć obiekt klasy `HelloThread`. Uruchomienie nowego wątku następuje w wyniku wywołania metody `start` (nie `run`!) na obiekcie klasy "wątkowej". Metoda `start` jest metodą klasy `Thread`, która faktycznie tworzy nowy wątek i przekazuje sterowanie do kodu zdefiniowanego w metodzie `run`.

Przykładowy kod jest widoczny poniżej:

```
public class Main {
    public static class HelloThread extends Thread {
        @Override
        public void run() {
            try {
                Thread.sleep(1000);
                System.out.println("Napis z watku utworzonego");
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        HelloThread helloThread = new HelloThread();
        helloThread.start();
        System.out.println("Napis z watku glownego");
        helloThread.join();
    }
}
```

Metoda `join` klasy `Thread` powoduje, że wątek oczekuje na zakończenie przetwarzania w innym wątku. Metodę `join` wywołuje na obiekcie wątku, na którego zakończenie oczekuje.Brak wywołania `helloThread.join()`, mógłby spowodować, że wątek główny zakończyłby działanie zanim wątek `helloThread` zdążyłby wykonać instrukcje w swojej metodzie `run()`.

Metoda `sleep` powoduje, że bieżący wątek zawiesza swoje działanie na określony czas.

1.2.2 Pule wątków i klasa ExecutorService

Tworzenie wątków ręcznie nie jest jednak dobrą praktyką z kilku powodów:

- Tworzenie nowych wątków nie jest darmowe. Utworzenie nowego wątku zajmuje czas, i może wprowadzać opóźnienia. Znacznie lepiej jest utworzyć wątek jeden raz, a następnie używać go do wykonywania wielu zadań.
- Ograniczona kontrola nad liczbą uruchomionych wątków. Aktywne wątki zużywają zasoby systemowe, a w skrajnych przypadkach wiele wątków konkurujących o wykonanie na procesorze może również powodować inne problemy z wydajnością.

Z tych powodów, do zarządzania wątkami stosuje się pule wątków (ang. thread pools). Pozwalają one na efektywne wykonywanie wielu zadań w ograniczonej liczbie wątków. Zamiast tworzyć nowy wątek dla każdego zadania, zadania te są przypisywane do wątku z puli dostępnych wątków. Gdy wątek z puli zakończy wykonywanie jednego zadania, może zostać wykorzystany do wykonania innego zadania. Gdy wszystkie wątki są zajęte, nowe zadania są umieszczane w kolejce i czekają na zwolnienie wątku.

Pakiet java.util.concurrent zawiera kilka klas do zarządzania pulami wątków, m.in. `ExecutorService`:

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

public class ExecutorServiceExample {
    public static void main(String[] args) throws InterruptedException {
        final ExecutorService executor = Executors.newFixedThreadPool(2);

        executor.submit(() -> {
            System.out.println("Task 1 executed by thread: " + Thread.currentThread().
                getName());
        });
        executor.submit(() -> {
            System.out.println("Task 2 executed by thread: " + Thread.currentThread().
                getName());
        });
        executor.submit(() -> {
            System.out.println("Task 3 executed by thread: " + Thread.currentThread().
                getName());
        });

        executor.shutdown();
        executor.awaitTermination(1, TimeUnit.MINUTES);
    }
}
/**
 * Output:
 * Task 1 executed by thread: pool-1-thread-1
 * Task 2 executed by thread: pool-1-thread-2
 * Task 3 executed by thread: pool-1-thread-2
 */
```

W powyższym przykładzie utworzono pulę wątków z dwoma wątkami, wykorzystując metodę `Executors.newFixedThreadPool(2)`. Następnie, za pomocą obiektu `executor`, przesłano trzy zadania do wykonania. Każde zadanie jest zaimplementowane za pomocą wyrażenia lambda, wewnątrz którego zdefiniowano konkretne działania do wykonania (tutaj: wypisanie wiadomości na wyjście).

Metoda `shutdown` inicjuje proces łagodnego zamykania serwisu egzekutora, przestając przyjmować nowe zadania, ale dokańczając już przekazane. Metoda `awaitTermination` blokuje wątek, w którym jest wywoływana, do czasu zakończenia wszystkich zadań lub do upłynięcia określonego limitu czasu, co jest podobne do działania metody `Thread.join()` w kontekście wątków.

Alternatywnym rozwiązaniem utworzenia nowego zadania jest zdefiniowanie klasy, która implementuje interfejs `Runnable`. Następnie należy przekazać obiekt tej klasy do obiektu `executor`:

```
public class TaskClassExample {

    private static class Task implements Runnable {
        @Override
        public void run() {
            System.out.println("Task executed by thread: " + Thread.currentThread().
                getName());
        }
    }

    public static void main(String[] args) throws InterruptedException {
        final ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.submit(new Task());
    }
}
```

1.3 Dostęp do zmiennych współdzielonych

Wykonywane wątki wykorzystują tę samą przestrzeń adresową pamięci, zatem posiadają dostęp do współdzielonych zmiennych. Jednym ze sposobów współdzielenia zmiennych jest przekazanie obiektu zmiennej współdzielonej w konstruktorze zadania lub wątku. Innym sposobem jest wykorzystanie mechanizmu domknięcia (ang. closure) w wyrażeniu lambda. Oba mechanizmy zilustrowano poniżej:

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

public class SharedVariableExample {
    private static class SharedVariable {
        private int counter = 0;

        public void increment() { counter++; }
        public int get() { return counter; }
    }

    private static class MyTask implements Runnable {
        private final SharedVariable sharedVariable;

        private MyTask(SharedVariable sharedVariable) {
            this.sharedVariable = sharedVariable;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(1000);
                sharedVariable.increment();
                System.out.println("Variable incremented");
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
```

```
    }

    public static void main(String[] args) throws InterruptedException {
        final ExecutorService executor = Executors.newFixedThreadPool(3);
        final SharedVariable sharedVariable = new SharedVariable();

        executor.submit(new MyTask(sharedVariable));
        executor.submit(new MyTask(sharedVariable));
        executor.submit(() -> {
            while (sharedVariable.get() == 0) {
            }
            System.out.println("Variable has changed");
        });

        executor.shutdown();
        if (!executor.awaitTermination(10, TimeUnit.SECONDS)) {
            System.out.println("Timeout occurred while waiting for tasks to finish.
                Kill the program manually");
        }
    }
}
```

W powyższym programie uruchomiono 2 zadania, które czekają 1 sekundę, po czym zwiększają wartość współdzielonej zmiennej. Trzecie zadanie oczekuje, aż wartość współdzielonej zmiennej zostanie zmieniona. Kod ten zawiera jednak dwa krytyczne błędy, które sprawiają, że nie będzie działa poprawnie, a w skrajnym wypadku nigdy się nie zakończy:

- zmiana wartości współdzielonej zmiennej, nie jest w żaden sposób zsynchronizowana. Występuje zjawisko tzw. "wyścigu" w wyniku którego współdzielona zmienna może mieć wartość 1 lub 2 po zakończeniu działania wątków
- wątek oczekujący na zmianę współdzielonej zmiennej może nigdy nie zobaczyć zmian dokonanych przez pozostałe dwa wątki

Oba problemy oraz sposoby ich rozwiązania zostaną omówione poniżej.

1.4 Mechanizmy synchronizacji

Problem synchronizacji może dokładniej ilustrować następujący przykład. Załóżmy, że dwa wątki A i B zyskują dostęp do zmiennej współdzielonej `SharedVariable.counter` i zwiększają jej wartość. Oczekiwany końcowy wynik to 2, jednak w wyniku niewłaściwego wzajemnego wykluczania wątków wynik może być inny. Faktyczne wykonanie operacji `counter++` przez jeden wątek przebiega w następujących krokach:

1. pobierz `counter`
2. dodaj 1 do `counter`
3. zapisz `counter`

W przypadku przeplatania się akcji wątku A i B wykonanie może wyglądać następująco:

1. Wątek A: pobierz `counter` → `counter` ma wartość 0
2. Wątek B: pobierz `counter` → `counter` ma wartość 0

3. Wątek A: zwiększ `counter` o 1
4. Wątek B: zwiększ `counter` o 1
5. Wątek A: zapisz `counter` → `counter` ma wartość 1
6. Wątek B: zapisz `counter` → `counter` ma wartość 1

Wynik działania wątku A jest utracony i nadpisany przez wątek B. Istnieje wiele możliwych kolejności wykonania i wyników, co powoduje, że wykrycie błędów synchronizacji może być trudne.

W kontekście tego ćwiczenia używane jest słowo synchronizacja do opisania sekcji krytycznych, które zapewnią wzajemne wykluczanie. Trzeba jednak pamiętać, że ogólne pojęcie synchronizacja odnosi się również do zapewnienia kolejności wykonania, a nie jedynie wzajemnego wykluczania.

1.4.1 Słowo kluczowe synchronized

W Javie zdefiniowano słowo kluczowe `synchronized` dla określenia sekcji krytycznych. Ten mechanizm synchronizacji jest dostępny na dwóch głównych poziomach: synchronizowane metody lub wyrażenia. Zdefiniowanie metody jako synchronizowanej odbywa się przez dodanie słowa kluczowego synchronized do deklaracji, np.:

```
public synchronized void increment() {
    counter++;
}
```

Taka deklaracja powoduje, że wykonanie metody w ramach jednego obiektu może odbywać się przez jeden wątek. Jeżeli jakiś wątek wykonuje metodę, to inne wątki, które chcą wykonać tę metodę, zostają zablokowane do czasu zakończenia wykonania metody. **Jeżeli w jednej klasie istnieje kilka metod określonych jako synchronized, to wszystkie one tworzą jedną sekcję krytyczną i w danym momencie może wykonywać się tylko jedna z nich.** Należy pamiętać, że synchronizacja taka jak pokazano, odnosi się do pojedynczego obiektu. Jeżeli utworzono wiele obiektów tej samej klasy, to każdy obiekt będzie posiadał własną sekcję krytyczną.

Uwaga! Powszechnym błędem jest używanie metod typu synchronized w klasie wątku. Metody synchronized powinny być definiowane w klasie współdzielonego zasobu. W typowych przypadkach wykorzystywanie metod synchronized w klasie wątku jest niepoprawne i nie gwarantuje wzajemnego wykluczania między wątkami.

Drugim sposobem synchronizacji jest synchronizacja na poziomie bloków kodu. W tym przypadku synchronizacja (sekcja krytyczna) wymaga podania obiektu, na którym jest uzyskiwana:

```
public void increment() {
    synchronized (this) {
        counter++;
    }
}
```

Synchronizacja z użyciem `this` wykorzystuje domyślny obiekt synchronizacji, który jest związany z każdym obiektem istniejącym w aplikacji. Innym przykładem sekcji krytycznej jest synchronizacja z użyciem innego obiektu niż bieżący.

```
private Object lock1 = new Object ();
private Object lock2 = new Object ();

public void inc1() {
    synchronized (lock1) {
        c1 ++;
    }
}
```

```
public void inc2() {
    synchronized (lock2) {
        c2 ++;
    }
}
```

Bardziej zaawansowane mechanizmy synchronizacji zostaną omówione na kolejnym laboratorium.

1.5 Spójność pamięci pomiędzy wątkami

Synchronizacja dotyczy jednak nie tylko kolejności wykonania i wzajemnego wykluczania. Ma również inny subtelny aspekt: zagwarantowanie spójności pamięci pomiędzy wątkami. Chcemy nie tylko zapobiec modyfikowaniu stanu obiektu przez jeden wątek, gdy inny go używa, ale także upewnić się, że gdy wątek zmodyfikuje już stan obiektu, inne wątki będą mogły faktycznie zobaczyć wprowadzone zmiany. Bez odpowiedniej synchronizacji może się to nie wydarzyć.

Brak takiej synchronizacji występuje w pokazanym wcześniej programie. Wątek oczekujący na zmianę współdzielonej zmiennej, może nigdy nie zobaczyć zmian dokonanych przez pozostałe dwa wątki.

Najprostszym rozwiązaniem tego problemu jest dodanie `synchronized` do metody `get()`, wymuszając tym samym synchronizację podczas odczytu współdzielonej zmiennej:

```
private static class SharedVariable {
    private int counter = 0;

    public synchronized void increment() { counter++; }
    public synchronized int get() { return counter; }
}
```

Synchronizacja zapewnia nie tylko wzajemne wykluczanie, ale także poprawną widoczność zmian w pamięci. Aby upewnić się, że wszystkie wątki widzą najbardziej aktualne wartości współdzielonych zmiennych, wątki modyfikujące zmienną i wątki odczytujące tą zmienną muszą synchronizować się na wspólnym obiekcie synchronizacji.

1.5.1 Zmienne volatile

Java udostępnia również alternatywną, słabszą formę synchronizacji: zmienne `volatile`. Zapis do zmiennej `volatile` jest natychmiast widoczny dla wszystkich innych wątków. Zmienna taka gwarantuje relacje happens-before, co oznacza, że zmiany w zmiennej volatile są natychmiast widoczne dla innych wątków, a kolejne odczyty i zapisy do tej zmiennej zobaczą tę aktualizację. Zmienne `volatile` nie są zapisywane w rejestrach ani w pamięci podręcznej procesora, gdzie są niewidoczne dla innych rdzeni, więc odczyt takiej zmiennej zawsze zwraca ostatni zapis dokonany przez dowolny wątek.

Przykład użycia zmiennej `volatile` jest widoczny poniżej. Metoda `get` nie jest tym razem synchronizowana:

```
private static class SharedVariable {
    private volatile int counter = 0;

    public synchronized void increment() { counter++; }
    public int get() { return counter; }
}
```

Należy pamiętać, że zmienne volatile nie służą do synchronizacji sekcji krytycznych i nie gwarantują niepodzielności złożonych operacji. Nawet jeśli `counter` jest zadeklarowany jako `volatile`, nie ma gwarancji, że sekwencja odczyt-modyfikacja-zapis zostanie wykonana jako pojedyncza, niepodzielna operacja:

```
@Override
public void run() {
```

```
    if (sharedVariable.get() == 1) {
        sharedVariable.increment();
    }
}
```

Powyższy kod jest błędny, ponieważ inny wątek ma możliwość zmodyfikowania współdzielonej zmiennej pomiędzy odczytem a zapisem wykonywanym w powyższym kodzie.

1.6 Zmienne o zasięgu wątku

Zmienne o zasięgu wątku (ang. thread-local variables) w Javie pozwalają na przechowywanie danych, które są izolowane dla poszczególnych wątków. Oznacza to, że każdy wątek ma własną, niezależną instancję zmiennej, co pozwala na bezpieczne używanie zmiennych w środowiskach wielowątkowych bez konieczności stosowania synchronizacji.

W Javie do tworzenia zmiennych o zasięgu wątku wykorzystywana jest klasa `ThreadLocal<T>`. Przykład jej użycia został pokazany poniżej:

```
public class ThreadLocalExample {
    private static final ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(()
        -> 0);

    public static void main(String[] args) {
        new Thread(() -> {
            threadLocal.set(1);
            System.out.println("Wartosc w watku 1 to zawsze 1: " + threadLocal.get());
        }).start();

        new Thread(() -> {
            try {
                Thread.sleep(1000);
                System.out.println("Wartosc w watku 2 to zawsze 0: " + threadLocal.get
                    ());
            } catch (InterruptedException ignored) {}
        }).start();
    }
}
```

Mimo że zmienna `threadLocal` jest zdefiniowana jako statyczna, to każdy wątek będzie miał własną niezależną instancję tej zmiennej.

1.6.1 ThreadLocalRandom

Innym przykładem zmiennej o zasięgu wątku jest `ThreadLocalRandom`.

Java udostępnia klasę `Random`, która pozwala na generowanie losowych wartości. Obiekty tej klasy nie działają jednak dobrze w środowisku wielowątkowym. Jeżeli wiele wątków współdzieli tę samą instancję `Random`, to dochodzi do zjawiska rywalizacji (ang. contention), które prowadzi do niskiej wydajności.

`ThreadLocalRandom` jest połączeniem klas `ThreadLocal` i `Random`. Obiekty tej klasy są izolowane do bieżącego wątku. W ten sposób osiągana jest lepsza wydajność w środowisku wielowątkowym, po prostu unikając współbieżnego dostępu do instancji `Random`. Na liczbę losową uzyskaną przez jeden wątek nie ma wpływu inny wątek, podczas gdy `Random` zapewnia liczby losowe globalnie dla wszystkich wątków w programie.

Przykład użycia `ThreadLocalRandom` został pokazany poniżej. Należy zwrócić uwagę, żeby instancję `ThreadLocalRandom` pozyskać w wątku, który wykonuje metodę `run`, a nie wątku, który tworzy instancję klasy `TaskUsingThreadLocalRandom`.

```
class TaskUsingThreadLocalRandom implements Runnable {
```

```
    @Override
    public void run() {
        final ThreadLocalRandom random = ThreadLocalRandom.current();
        ...
    }
}
```

1.7 Zadanie

Zaimplementować wielowątkowy program, który implementuje model producent-konsument. W programie dostępny jest magazyn, który posiada ograniczoną ilość miejsca i przechowuje "produkowane" towary - typ i ilość. Działa wiele wątków producentów i konsumentów (należy do tego wykorzystać 2 pule wątków).
Producenci zajmują się produkowaniem towarów (jeżeli w magazynie jest miejsce), losują ich typ i ilość. Konsumenci losują typ towaru i jego ilość, następnie odszukują w magazynie, czy mogą taki zakup wykonać. Jeżeli mogą, to dokonują zakupu w całości lub częściowo zmniejszając ilość towaru w magazynie.
Wątki powinny wykonywać transakcje cyklicznie, a losowane opóźnienia i ilości powinny zapewnić, że program nie będzie zawieszał się albo chodził w pustych pętlach. Należy zwrócić uwagę na synchronizację metod dostępu do magazynu w celu uniknięcia "wyścigów".
Program powinien działać aż do momentu naciśnięcia przez użytkownika klawisza enter na klawiaturze. Po tym wątki konsumentów i producentów powinny zostać poprawnie zakończone (należy do tego użyć zmiennej `volatile`), a cały program się zakończyć.

2 Wielowątkowość w Javie - zagadnienia zaawansowane

2.1 Wstęp

Zwykły blok `synchronized` umożliwia tworzenie jedynie prostych sekcji krytycznych, które są ograniczone do jednej metody. O ile pozwala to na tworzenie wątkowo-bezpiecznych aplikacji, to niesie to za sobą poważne ograniczenia, które mogą negatywnie wpływać na wydajność aplikacji.

Jeżeli w jednej klasie kilka metod jest określona jako `synchronized`, to wszystkie one tworzą jedną sekcję krytyczną i w danym momencie może wykonywać się tylko jedna z nich. Może to prowadzić do problemu konkurencja o blokadę (ang. lock contention). Jeżeli wiele wątków próbuje jednocześnie uzyskać dostęp do tego samego zasobu chronionego przez blokadę, to tylko jeden z nich może posiadać blokadę w danym momencie i wykonywać swoje zadanie. Pozostałe wątki muszą czekać, aż blokada zostanie zwolniona, co prowadzi do opóźnień i zmniejszenia przepustowości aplikacji.

Bloki `synchronized` nie zapewniają równego ani sprawiedliwego (ang. fairness) dostępu dla wszystkich wątków ubiegających się o tę blokadę. Każdy wątek może przejąć blokadę po jej zwolnieniu i nie można określić, czy któryś wątek (np. najdłużej oczekujący) ma pierwszeństwo w dostępie.

Dodatkowo bloki `synchronized` nie pozwalają na sprawdzenie, czy blokada jest wolna, czy na próbę uzyskania blokady bez blokowania jeżeli natychmiastowe uzyskanie blokady się nie powiedzie.

2.2 Zaawansowane mechanizmy synchronizacji

Java udostępnia wiele mechanizmów synchronizacji, które pozwalają na dużo bardziej elastyczną i rozbudowaną synchronizację wątków i efektywniejsze zarządzanie dostępem do sekcji krytycznej. Poniżej omówiono trzy wybrane mechanizmy.

2.2.1 Klasa ReentrantLock

Klasa `ReentrantLock` oferuje koncepcję synchronizacji dostępu do sekcji krytycznych podobną do bloków `synchronized`. Rozszerza je jednak o dodatkowe możliwości.

Proste wykorzystanie `ReentrantLock` pokazano poniżej. Metoda `lock()` pozwala na uzyskanie blokady. Jeśli blokada nie jest dostępna, wątek zostaje zablokowany do momentu zwolnienia blokady. Metoda `unlock()` zwalnia blokadę. W celu zapewnienia, że blokada zawsze zostanie zwolniona zaleca się użycie bloku try-finally.

```
private final ReentrantLock lock = new ReentrantLock();
public void doStuff() {
    lock.lock();
    try {
        // critical section here
    } finally {
        lock.unlock();
    }
}
```

Klasa `ReentrantLock` oferuje ponadto dodatkowe mechanizmy:

- utworzenie instancji z ustawioną flagą fair: `new ReentrantLock(true)` pozwala na wymuszenie stosowania sprawiedliwego dostępu do blokady
- metoda `lock.tryLock()` próbuje zdobyć blokadę bez blokowania wątku wywołującego. Zwraca wartość `true` jeśli blokada została pomyślnie zdobyta lub `false`, jeśli blokada jest już zajęta
- metoda `lock.isLocked()` pozwala sprawdzić, czy blokada jest aktualnie zajęta

2.2.2 Klasa ReentrantReadWriteLock

Klasa `ReentrantLock` umożliwia tylko 1 wątkowi na pozyskanie blokady. Taka restrykcyjna strategia blokowania zapobiega nakładaniu się:

- kilku wątków, które próbują jednocześnie zmodyfikować współdzieloną zmienną
- kilku wątków, które próbują jednocześnie zmodyfikować lub odczytać współdzieloną zmienną
- kilku wątków, które próbują jednocześnie odczytać współdzieloną zmienną

Takie wzajemne wykluczenie jest jednak często bardziej restrykcyjne, niż jest to konieczne i tym samym niepotrzebnie ogranicza współbieżność aplikacji.

W wielu przypadkach dane przechowywane we współdzielonych zmiennych są odczytywane znacznie częściej, niż są modyfikowane. W takich przypadkach dobrze jest złagodzić wymagania dotyczące blokowania, i umożliwić wielu wątkom jednoczesny odczyt współdzielonych zmiennych. Należy przy tym zagwarantować, że tylko jeden wątek ma dostęp do zmiennej w momencie modyfikacji.

Klasa `ReentrantReadWriteLock` oferuje właśnie taki mechanizm synchronizacji: dostęp do zasobu może mieć wiele wątków odczytujących zmienną albo jeden wątek modyfikujący zmienną. Wątek modyfikujący zmienną nigdy nie ma dostępu do zmiennej jednocześnie z wątkiem odczytującym tę zmienną.

Klasa `ReentrantReadWriteLock` udostępnia dwie blokady: jedną do odczytu i jedną do zapisu. Aby odczytać dane strzeżone przez ReadWriteLock, należy uzyskać blokadę odczytu. Aby zmodyfikować dane, należy uzyskać blokadę zapisu. Przykład użycia `ReentrantReadWriteLock` pokazano poniżej:

```
public class SyncHashMap {
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Map<String,String> map = new HashMap<>();

    public String get(final String key) {
        lock.readLock().lock();
        try {
            return map.get(key);
        } finally {
            lock.readLock().unlock();
        }
    }

    public void put(final String key, final String value) {
        lock.writeLock().lock();
        try {
            map.put(key, value);
        } finally {
            lock.writeLock().unlock();
        }
    }
}
```

Chociaż może się wydawać, że istnieją dwie oddzielne blokady: blokada odczytu i blokada zapisu, to są po prostu różne widoki zintegrowanego obiektu blokady odczytu i zapisu.

2.2.3 Klasa CyclicBarrier

`CyclicBarrier` jest mechanizmem synchronizacji w Javie, który umożliwia grupie wątków czekanie na siebie nawzajem w określonym punkcie wykonania programu. Umożliwia to koordynację działań w scenariuszach, gdzie

wykonanie pewnych części zadania przez różne wątki musi być zakończone, zanim wszystkie mogą przejść do kolejnego etapu.

Poniżej przedstawiono przykład użycia CyclicBarrier, w którym dwa wątki koordynują swój punkt synchronizacji. Mimo że tylko drugi wątek zawiera opóźnienie (`sleep`) trwające 10 sekund, oba wątki zakończą się mniej więcej w tym samym czasie. Pierwszy wątek będzie czekać na drugi wątek, aż ten zakończy swoje opóźnienie i dotrze do bariery.

```
public class CyclicBarrierExample {
    public static void main(String[] args) throws InterruptedException {
        final CyclicBarrier cyclicBarrier = new CyclicBarrier(2);
        final ExecutorService executorService = Executors.newFixedThreadPool(2);
        executorService.submit(() -> {
            try {
                cyclicBarrier.await();
                System.out.printf("Watek %s konczy prace%n", Thread.currentThread().
                    getName());
            } catch (InterruptedException | BrokenBarrierException e) {
                throw new RuntimeException(e);
            }
        });
        executorService.submit(() -> {
            try {
                Thread.sleep(TimeUnit.SECONDS.toMillis(10));

                cyclicBarrier.await();
                System.out.printf("Watek %s konczy prace%n", Thread.currentThread().
                    getName());
            } catch (InterruptedException | BrokenBarrierException e) {
                throw new RuntimeException(e);
            }
        });

        executorService.shutdown();
        executorService.awaitTermination(1, TimeUnit.MINUTES);
    }
}
```

2.3 Wątkowo-bezpieczne kolekcje

Podstawowe kolekcje w Javie, takie jak `HashMap` czy `ArrayList` nie są wątkowo-bezpieczne i nie mogą być współdzielone przez kilka wątków bez dodatkowej synchronizacji.

Z tego względu, Java udostępnia wiele struktur danych zaprojektowanych specjalnie do użycia w środowisku wielowątkowym, takie jak `ConcurrentHashMap` czy `CopyOnWriteArrayList`. Te kolekcje oferują bardzo dobrą skalowalność i wydajność poprzez zastosowanie zaawansowanych technik zarządzania współbieżnością, takich jak segmentacja blokad czy mechanizmy "write-on-copy".

Należy pamiętać, że kolekcje te gwarantują bezpieczeństwo jedynie na poziomie pojedynczych operacji oraz iteracji po kolekcji. Złożone operacje wymagają dodatkowej synchronizacji. Np. poniższy kod jest błędny, ponieważ pomiędzy sprawdzeniem, czy w mapie istnieje już taki klucz, a dodaniem nowego obiektu do mapy, inny wątek może zmodyfikować mapę i wstawić własną wartość pod ten sam klucz:

```
private final Map<String, String> concurrentMap = new ConcurrentHashMap<>();

public void putUnsafe(final String key, final String value) {
```

```
    if (!concurrentMap.containsKey(key)) {
        concurrentMap.put(key, value);
    }
}
```

W celu naprawienie tego błędu można zastosować dodatkową synchronizację, ale jeżeli jest to możliwe, dużo lepiej jest wykorzystać bardziej zaawansowane metody kolekcji, np.:

```
public void putSafe(final String key, final String value) {
    concurrentMap.putIfAbsent(key, value);
}
```

2.3.1 Kolekcje synchronizowane

W Javie występują również kolekcje synchronizowane takie jak `Vector` i `Hashtable`. Istnieje również możliwość ręcznego stworzenie synchronizowanej kolekcji z już istniejącej, np.: `Collections.synchronizedMap(new HashMap<>())`.

Kolekcje takie jednak opierają się na prostym blokowaniu dostępu na poziomie całego obiektu, co oznacza, że każdy wątek próbujący uzyskać dostęp do dowolnej metody kolekcji musi poczekać na zwolnienie blokady przez inne wątki. Dodatkowo iteracja po takich obiektach nie jest wątkowo-bezpieczna i wymaga dodatkowej synchronizacji. W związku tym stosowanie takich kolekcji jest **niezalecane**.

2.4 Zakleszczenia

Zakleszczenie (ang. deadlock) określa sytuację, w której co najmniej dwa wątki są na zawsze zablokowane, oczekując na siebie nawzajem. Rozważmy przykładowy scenariusz zakleszczenia. Wątek A oraz wątek B chcą uzyskać dostęp do sekcji krytycznych chronionych przez semafory (obiekty synchronizacji) X oraz Y. Scenariusz wygląda następująco:

1. Wątek A wchodzi do synchronized (X)
2. Wątek B wchodzi do synchronized (Y)
3. Wątek A chce wejść do synchronized (Y) //zablokowany, czeka na B
4. Wątek B chce wejść do synchronized (X) //zablokowany, czeka na A

W rezultacie ani wątek A nie może przejść dalej (czeka na B), ani wątek B nie może przejść dalej (czeka na A).

Poniżej przedstawiono kod innego przykładu, w którym dwa wątki wykonują wzajemnie metody synchronizowane `ping` oraz `pingBack`. Jeżeli oba wątki wykonają `ping` przed wykonaniem `pingBack`, dojdzie do zakleszczenia.

```
public class DeadlockPing {
    public static class PingPong {
        private final String name;

        public PingPong(String name) {
            this.name = name;
        }

        public synchronized void ping(PingPong other) {
            System.out.format("%s:%s wyslal pinga!%n", this.name, other.name);
            other.pingBack(this);
        }
```

```
        private synchronized void pingBack(PingPong other) {
            System.out.format("%s:%s odpowiedzial na pinga!%n", this.name, other.name);
        }
    }

    public static void main(String[] args) {
        final PingPong pingPierwszy = new PingPong("Pierwszy");
        final PingPong pingDrugi = new PingPong("Drugi");

        new Thread(() -> pingPierwszy.ping(pingDrugi)).start();
        new Thread(() -> pingDrugi.ping(pingPierwszy)).start();
    }
}
```

Innym typowym błędem prowadzącym do zakleszczeń jest niepoprawne zwalnianie blokad. Zwalnianie blokady zawsze powinno odbywać się w bloku try-finally, tak żeby zagwarantować, że zawsze zostanie zwolniona.

```
private final Lock lock = new ReentrantLock();

public void incorrectUnlock() {
    lock.lock();
    methodThatMightThrowException();

    // Lock won't be unlocked if previous method throws exception
    lock.unlock();
}

public void correctUnlock() {
    lock.lock();
    try {
        methodThatMightThrowException();
    } finally {
        // Lock will be unlocked even if previous method throws exception
        lock.unlock();
    }
}
```

2.5 Zrzut wątków

Jednym ze sposobów na analizę zakleszczeń, problemów z wydajnością oraz ogólnego stanu działania programu, jest wykonanie i analiza zrzutu wątków (ang. thread dump). Zrzut wątków zapewnia obraz bieżącego stanu uruchomionego procesu Javy.

W ramach ćwiczenia wykonamy i przeanalizujemy thread-dump dla poniższego programu:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ThreadDumpExample {
    private static class AwesomeClass {
        private final Lock lock = new ReentrantLock();
```

```
        public void method1() {
            lock.lock();
            try {
                method2();
            } finally {
                lock.unlock();
            }
        }

        private void method2() {
            method3();
        }

        private void method3() {
            try {
                Thread.sleep(TimeUnit.SECONDS.toMillis(60));
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(3);
        final AwesomeClass ac = new AwesomeClass();

        executorService.submit(ac::method1);
        executorService.submit(ac::method1);

        executorService.shutdown();
        executorService.awaitTermination(5, TimeUnit.MINUTES);
    }
}
```

2.5.1 Generowanie zrzutu wątków

Istnieje wiele sposobów na wygenerowanie zrzutu wątku Javy. W pierwszej kolejności należy uzyskać PID procesu Javy. Użyć do tego można komendy `jps` lub zwykłej komendy `ps` z odpowiednimi flagami (np. `ps aux | grep java`):

```
$ jps
14068
4328 ThreadDumpExample
6552 Jps
6808 Launcher
```

Następnie należy użyć narzędzia `jstack` (dostarczanym razem z JDK) w celu wykonania zrzutu wątków. Zrzuty wątków są sporządzane w postaci zwykłego tekstu, więc można zapisać ich zawartość do pliku i przejrzeć je później w edytorze tekstowym.

```
$ jstack -l 4328 > threaddump.txt
```

Alternatywnie, w systemie Linux można wykorzystać metodę `kill -3` do wykonania zrzutu:

```
$ kill -3 4328 > threaddump.txt
```

2.5.2 Analiza zrzutu wątków

Fragment zrzutu wątków jest widoczny na listingu poniżej. Nawet najprostsza aplikacja Javy startuje kilkanaście wątków, więc poniżej pokazano tylko wybrane wątki.

```
2024-03-05 21:57:30
Full thread dump OpenJDK 64-Bit Server VM (17.0.10+7 mixed mode, sharing):

Threads class SMR info:
_java_thread_list=0x00000209ec5e43d0, length=15, elements={
0x00000209bfb06220, 0x00000209e9a8e470, 0x00000209e9a8f2f0, 0x00000209e9aa5600,
0x00000209e9aa71d0, 0x00000209e9aa9ba0, 0x00000209e9aab570, 0x00000209e9aac260,
0x00000209e9aba230, 0x00000209ec41a7c0, 0x00000209ec3f0a30, 0x00000209ec61c920,
0x00000209ec61d210, 0x00000209ec5e64f0, 0x00000209ec5e86a0
}

"main" #1 prio=5 os_prio=0 cpu=46.88ms elapsed=56.08s tid=0x00000209bfb06220 nid=0x22b4 waiting on condition  [0
      x000000b09c0fe000]
   java.lang.Thread.State: TIMED_WAITING (parking)
    at jdk.internal.misc.Unsafe.park(java.base@17.0.10/Native Method)
    - parking to wait for  <0x0000000621235500> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
    at java.util.concurrent.locks.LockSupport.parkNanos(java.base@17.0.10/LockSupport.java:252)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(java.base@17.0.10/
        AbstractQueuedSynchronizer.java:1672)
    at java.util.concurrent.ThreadPoolExecutor.awaitTermination(java.base@17.0.10/ThreadPoolExecutor.java:1464)
    at ThreadDumpExample.main(ThreadDumpExample.java:41)

   Locked ownable synchronizers:
    - None

"pool-1-thread-1" #16 prio=5 os_prio=0 cpu=0.00ms elapsed=55.99s tid=0x00000209ec5e64f0 nid=0x3750 waiting on condition
      [0x000000b09d4fe000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(java.base@17.0.10/Native Method)
    at ThreadDumpExample$AwesomeClass.method3(ThreadDumpExample.java:26)
    at ThreadDumpExample$AwesomeClass.method2(ThreadDumpExample.java:21)
    at ThreadDumpExample$AwesomeClass.method1(ThreadDumpExample.java:14)
    at ThreadDumpExample$$Lambda$15/0x00000209ee001410.run(Unknown Source)
    at java.util.concurrent.Executors$RunnableAdapter.call(java.base@17.0.10/Executors.java:539)
    at java.util.concurrent.FutureTask.run(java.base@17.0.10/FutureTask.java:264)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(java.base@17.0.10/ThreadPoolExecutor.java:1136)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(java.base@17.0.10/ThreadPoolExecutor.java:635)
    at java.lang.Thread.run(java.base@17.0.10/Thread.java:840)

   Locked ownable synchronizers:
    - <0x0000000621236c08> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)
    - <0x000000062123dcf8> (a java.util.concurrent.ThreadPoolExecutor$Worker)

"pool-1-thread-2" #17 prio=5 os_prio=0 cpu=0.00ms elapsed=55.99s tid=0x00000209ec5e86a0 nid=0x26d0 waiting on condition
      [0x000000b09d5fe000]
   java.lang.Thread.State: WAITING (parking)
    at jdk.internal.misc.Unsafe.park(java.base@17.0.10/Native Method)
    - parking to wait for  <0x0000000621236c08> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)
    at java.util.concurrent.locks.LockSupport.park(java.base@17.0.10/LockSupport.java:211)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(java.base@17.0.10/AbstractQueuedSynchronizer.java
        :715)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(java.base@17.0.10/AbstractQueuedSynchronizer.java
        :938)
    at java.util.concurrent.locks.ReentrantLock$Sync.lock(java.base@17.0.10/ReentrantLock.java:153)
    at java.util.concurrent.locks.ReentrantLock.lock(java.base@17.0.10/ReentrantLock.java:322)
    at ThreadDumpExample$AwesomeClass.method1(ThreadDumpExample.java:12)
    at ThreadDumpExample$$Lambda$16/0x00000209ee002000.run(Unknown Source)
    at java.util.concurrent.Executors$RunnableAdapter.call(java.base@17.0.10/Executors.java:539)
    at java.util.concurrent.FutureTask.run(java.base@17.0.10/FutureTask.java:264)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(java.base@17.0.10/ThreadPoolExecutor.java:1136)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(java.base@17.0.10/ThreadPoolExecutor.java:635)
    at java.lang.Thread.run(java.base@17.0.10/Thread.java:840)

   Locked ownable synchronizers:
    - <0x00000006212401a8> (a java.util.concurrent.ThreadPoolExecutor$Worker)

"VM Thread" os_prio=2 cpu=0.00ms elapsed=56.07s tid=0x00000209e9a89430 nid=0x3108 runnable

"GC Thread#0" os_prio=2 cpu=0.00ms elapsed=56.08s tid=0x00000209bfb71bb0 nid=0x3980 runnable
```

Pierwsze dwa wiersze zawierają informację o czasie wykonania rzutu oraz wersji JVM. Następna sekcja "Threads class SMR info" jest użyteczna głównie dla osób rozwijających JVM lub debugujących bardzo skomplikowane problemy związane z wielowątkowością.

Następnie zrzut wyświetla listę wątków. Każdy wątek zawiera następujące informacje:

- Nazwa i numer identyfikacyjny
- Priorytet
- Czas CPU i czas działania
- ID wątku

- Stan wątku i adres pamięci, który wskazuje miejsce, w którym wątek jest zatrzymany
- Ślad stosu (ang. stack trace)
- Listę blokad pozyskanych przed dany wątek (`Locked ownable synchronizers`)

Ślad stosu pokazuje sekwencję wywołań metod prowadzących do bieżącego stanu wątku. Jest to najważniejsze źródło informacji do zrozumienia, tego co dzieje się z naszą aplikacją.

Najważniejsze z punktu widzenia programisty są wątki aplikacji. W zrzucie wątków są 3 takie wątki: "main", "pool-1-thread-1" i "pool-1-thread-2". W ich śladach stosów można zobaczyć nazwy klas i metod aplikacji (np. `ThreadDumpExample` , `AwesomeClass`). Dokładnie analizując ich ślady stosów, można poznać, które metody są wywoływane przez wątki i poznać aktualny stan aplikacji.

W zrzucie wątków można znaleźć wiele wątków maszyny wirtualnej Javy, np.:

- wątki garbage collectora (np. "GC Thread")
- wątki kompilatora JIT (np. "C1 CompilerThread")
- i inne (np. "Monitor Ctrl-Break", "VM Thread", "Finalizer")

2.5.3 Analiza wątku main

Dla przykładu przeanalizujmy wątek `main` . Jest to pierwszy wątek na widocznym wyżej zrzucie wątków. Na początku odczytujemy informacje o wątku:

- `main` - nazwa wątku
- `#1` - numer identyfikacyjny wątku
- `prio=5` - priorytet wątku w JVM
- `os_prio=0` - priorytet wątku na poziomie systemu operacyjnego
- `cpu=46.88ms` - czas procesora zużyty przez wątek
- `elapsed=56.08s` - czas, który upłynął od startu wątku
- `tid=0x00000209bfb06220` - identyfikator wątku w JVM
- `nid=0x22b4` - identyfikator wątku na poziomie systemu operacyjnego
- `waiting on condition` - wątek czeka na spełnienie pewnego warunku.

Następnie odczytujemy stan wątku: `java.lang.Thread.State: TIMED_WAITING (parking)` . Oznacza to, że wątek jest w stanie oczekiwania z limitem czasowym (`TIMED_WAITING`), czyli czeka na pewien warunek, ale nie przez nieskończony czas. `parking` wskazuje na to, że wątek jest zawieszony i czeka na odblokowanie.

Na końcu analizujemy ślad stosu wątku. Czytając go od dołu, widzimy, że:

- `at ThreadDumpExample.main(ThreadDumpExample.java:41)` - wątek zaczął się od metody `main`
- `at java.util.concurrent.ThreadPoolExecutor.awaitTermination` - w chwili wykonania zrzutu czekał na zakończenie zadań przez `Executor` poprzez wywołanie metody `awaitTermination`

Dalej widzimy wywołanie wewnętrznych metod Javy:

- `at java.util.concurrent.locks.AbstractQueuedSynchronizer`
- `at java.util.concurrent.locks.LockSupport.parkNanos`

Następna linia informuje nas o tym, że wątek czeka na obiekt `ConditionObject`, co jest typowe dla wątków czekających na zwolnienie blokady lub spełnienie innego warunku:

- `- parking to wait for <0x0000000621235500>`

Na końcu widzimy, że wątek jest zatrzymany w metodzie `Unsafe.park`, która jest metodą natywną i służy do zawieszania i wznawiania wątków.

- `at jdk.internal.misc.Unsafe.park`

2.5.4 Analiza stanu blokad

Ze zrzutu wątków możemy też odczytać ważne informacje o stanie blokad, jakie blokady są posiadane przez dany wątek czy na jakie blokady dany wątek czeka. Sekcja `Locked ownable synchronizers` wymienia obiekty blokad, które są posiadane przez wątek w momencie zrzutu stosu. Informacje o blokadach, na które wątek czeka, pojawiają się zazwyczaj w śladzie stosu (np. `parking to wait for <addres>`).

Na przykład, odczytując sekcje `Locked ownable synchronizers` wątku "pool-1-thread-1", widzimy, że jest on właścicielem 2 blokad: obiektu `ReentrantLock` o adresie 0x0000000621236c08, oraz obiektu ThreadPoolExecutor$Worker (blokada związaną z pulą wątków):

```
Locked ownable synchronizers:
    - <0x0000000621236c08> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)
    - <0x000000062123dcf8> (a java.util.concurrent.ThreadPoolExecutor$Worker)
```

Odczytując ślad stosu wątku "pool-1-thread-2", widzimy że czeka on na pozyskanie blokady o adresie 0x0000000621236c08:

```
- parking to wait for  <0x0000000621236c08>
```

Jest to ta sama blokada `ReentrantLock`, której właścicielem w momencie zrzutu wątku był "pool-1-thread-1".

Zrzuty wątków są zazwyczaj bardzo długie, a ich analiza może być czasochłonna i skomplikowana. Zidentyfikowanie przyczyn problemów z wydajnością lub zakleszczeń często wymaga dogłębnej analizy i powiązania ze sobą odpowiednich informacji.

2.6 Zadanie

Zaimplementuj wielowątkowy program, który będzie sprawdzał, czy podane przez użytkownika liczby są liczbami pierwszymi. Program powinien cache'ować już raz obliczone wyniki i wykorzystywać je, jeżeli użytkownik ponownie zapyta o tę samą liczbę. Program powinien działać w następujący sposób:

- W aplikacji utworzono pulę wątków z 4 wątkami
- Główna pętla programu pobiera od użytkownika 4 liczby. Następnie dla każdej z tych liczb przekazuje do puli wątków nowe zadanie, na sprawdzenie czy dana liczba jest liczbą pierwszą
- Pętla główna programu oczekuje na zakończenie wszystkich 4 zadań przed pobraniem od użytkownika kolejnych 4 liczb
- Wątki oczekują na wystartowanie wszystkich 4 zadań przed przystąpieniem do wykonywania obliczeń
- Po sprawdzeniu czy liczba jet liczbą pierwszą, wątek wypisuje tę informację na wyjście program

- Wątki zapisują już raz sprawdzone liczby w cach'u (czyli mapie). Jeżeli użytkownik zapyta o już raz sprawdzoną liczbę należy pobrać wynik z cach'u i nie obliczać go ponownie
- Należy zwrócić uwagę na scenariusz gdy użytkownik poda na wejściu 4 takie same liczby. W takim przypadku, tylko 1 wątek powinien wykonać obliczenia. Pozostałe 3 wątki powinny zaczekać i również pobrać wynik z cach'u.

Dodatkowo, w czasie działania programu należy wykonać zrzut wątków i przeanalizować stan aplikacji.

Jak punkt wyjściowy implementacji należy użyć widoczny poniżej szkielet programu:

```
public class JavaExercise2 {
    private static class CachingPrimeChecker {
        // TODO: dokończ implementację cachu
        //  (ustal typ przechowywanej wartości oraz rodzaj wykorzystywanej mapy)
        private final Map<Long, > cache = ;

        public boolean isPrime(final long x) {
            // TODO: dokoncz implementację sprawdzania czy liczba x jest liczba pierwszą
            // Należy zagwarantować, że dla każdej unikalnej liczby obliczenia zostaną
                wykonane tylko 1 raz
            // Ponowne (w tym równległe) sprawdzenie czy dana liczba jest liczbą pierwszą
                 powinny wykorzystać cache
        }

        // Funkcja sprawdzająca czy dana liczba jest liczbą pierwszą, należy jej użyć do
            wykonywania obliczeń
        // Nie należy jej w żaden sposób modyfikować!
        private boolean computeIfIsPrime(long x) {
            final String currentThreadName = Thread.currentThread().getName();
            System.out.printf("\t[%s] Running computation for: %d%n", currentThreadName,
                x);
            try {
                // simulating long computations
                Thread.sleep(TimeUnit.SECONDS.toMillis(10));
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }

            if (x < 2) {
                return false;
            }
            for (long i = 2; i * i <= x; i++) {
                if (x % i == 0) {
                    return false;
                }
            }
            return true;
        }
    }

    public static void main(String[] args) {
        // TODO: zaimplementuj pętlę główną programu
    }
}
```