Programowanie współbieżne i rozproszone

mgr inż. Marcin Mrukowicz

Instrukcja laboratoryjna V

Wątki i wzajemne wykluczanie w języku Java

Blokada

Blokada (ang. Lock) to w języku Java mechanizm podobny w działaniu do mutexu. Zasadniczo efekt uzyskany w wyniku uzyskania blokady jest podobny do zastosowania słowa kluczowego *synchronized*, ale pozostawia więcej kontroli po stronie programisty. Oznacza to jednak jednocześnie mniejsze zautomatyzowanie procesu udzielania dostępu do zasobu. Każda blokada posiada charakterystyczne metody. Metoda *lock* blokuje zasób **jeśli jest wolny i wykonuje sekcję krytyczną wątku**, jednakże gdy zasób jest **aktualnie zajęty, wykonanie wątku zostanie zablokowane** do czasu zwolnienia tego zasobu. Nierozważne stosowanie tej metody spowoduje, że wątki będą długo na siebie czekać i całość będzie działać nieefektywnie! Metoda *tryLock* również będzie blokować dany zasób, ale tutaj następuje **próba zablokowania zasobu**, która jeżeli się powiedzie, spowoduje wykonanie sekcji krytycznej, w przeciwnym razie **natomiast wykonanie wątku jest kontynuowane.** Jest to tak zwana **nieblokująca (ang. non-blocking)** wersja metody *lock*. Istnieje również wersja tej metody, która oczekuje przez pewien z góry ustalony okres czasu na dostęp do zasobu i albo go otrzymuje i wejdzie do sekcji krytycznej, albo po upływie tego czasu kontynuuje swoje wykonanie. Metoda *unlock* zwalnia dostęp do zasobu, umożliwiając innym wątkom dostęp do tego zasobu. Blokada umożliwia również sprecyzowanie czy dostęp do zasobu ma być sprawiedliwy (ang. fairness), dla wszystkich chętnych wątków. Sprawiedliwość oznacza tutaj, że każdy wątek ma to samo prawdopodobieństwo uzyskania dostępu do zasobu. Mimo że wydaje się to naturalne podejście, to istnieją problemy, w których lepiej preferować pewne wątki nad innymi. W każdym razie blokada, w przeciwieństwie do bloku *synchronized,* pozostawia podjęcie tej decyzji programiście.

Blokady pozwalają również na korzystanie z klasy *Condition*. Klasa ta pozwala utworzyć zdefiniowane przez programistę, własne zdarzenia, które wątki mogą tworzyć i na które wątki mogą czekać. Jest to właściwie tożsame z użyciem metod *wait*, *notify* i *notifyAll* w bloku synchronizowanym. Jednakże w tamtej sytuacji występuje właściwie tylko jedno zdarzenie (nie możemy w prosty sposób jakoś wyróżnić, że jedno wywołanie *notify* ma inne znaczenie niż drugie). Natomiast obiekty klasy *Condition* pozwalają nam, w obrębie jednej blokady posiadać wiele różnych zdarzeń. Może być nawet tak, że na pewne zdarzenie reaguje tylko jeden wątek, natomiast na inne mogą reagować wszystkie wątki. Metody w klasie signal nazywają się: *await* (odpowiednik *wait*), *signal* (odpowiednik *notify*) i *signalAll* (odpowiednik *notifyAll*).

Bariera

Bariera (ang. Barrier) to struktura, synchronizująca wykonanie grupy wątków w ten sposób, że wykonuje pewną akcję, dopiero, gdy wszystkie wątki zakończą wykonywanie swoich zadań (mówi się, że wszystkie wątki osiągną pewien punkt wykonania, ang. execution point). Zadania wykonywane przez wątki są tu zazwyczaj ściśle powiązane i przejście do dalszego wykonywania programu wymaga najpierw poczekania na wszystkie wątki. Konstruktor tej klasy zwykle przyjmuje liczbę wątków, które mają zakończyć wykonanie (zatem ta liczba musi być znana). Opcjonalnie można podać również obiekt, implementujący interfejs Runnable, który zostanie wykonany, gdy wszystkie wątki zakończą pracę. Wywołanie metody *await* tej klasy informuje, że dany wątek zakończył już pracę (jednak bariera wstrzyma wykonanie do czasu, aż wszystkie wątki zgłoszą ten fakt).

Zmienne atomowe

Zmienne atomowe (ang. atomic variables) to pewne nowe rozwiązanie, które umożliwia wykonanie operacji odczytu i zapisu zmiennej, **jako operacji atomowej. Oznacza to, że nie ma możliwości pojawienia się tzw. race condition**. Niepodzielność zmiennej **może być zapewniana sprzętowo, poprzez zastosowanie odpowiedniej instrukcji w języku maszynowym procesora. Zwykle stosowanie zmiennych atomowych jest znacznie szybsze** niż tradycyjne podejście oparte o stosowanie blokad, **a zwłaszcza blokad w wersji blokującej**. W języku Java istnieje pakiet *java.util.concurrent.atomic*, który przechowuje implementacje zmiennych atomowych. Ważne założenie tego pakietu jest takie, że klasy te **zadziałają** nawet jeżeli środowisko uruchomieniowe **nie wspiera sprzętowych operacji atomowych.** Jednakże spowoduje to konieczność zastąpienia tych instrukcji za pomocą blokad (w implementacji klas atomowych), a zatem zanikają wszelkie zyski wydajnościowe.

1. Przepisz, następującą klasę w języku Java:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.concurrent.*;
import java.util.concurrent.locks.ReentrantLock;

public class WordCount {

    public static void main(String[] args) {
        String text = "This is some text. It is intended to be counted by a threads.\n" +
                "Each thread will be taking a line and then will count words.\n" +
                "It is supposed to be faster than a standard, sequential way";
        ArrayList<String> lines = new ArrayList<>(Arrays.asList(text.split("\n")));
        HashMap<String, Integer> wordsCounted = new HashMap<>();
        ReentrantLock lock = new ReentrantLock(true);
        int numberOfThreads = 3;

        CyclicBarrier cb = new CyclicBarrier(numberOfThreads, () -> {
            for (String s : wordsCounted.keySet()) {
                if (wordsCounted.get(s) > 1) {
                    System.out.println(s + " has " + wordsCounted.get(s) + " occurrences.");
                } else {
                    System.out.println(s + " has " + wordsCounted.get(s) + " occurrence.");
                }
            }
        });

        class Counter extends Thread {
            @Override
            public void run() {
                while (!lines.isEmpty()) {
                    lock.lock();
                    try {
                        if (!lines.isEmpty()) {

                            System.out.println("I am thread" + this.getId());
                            System.out.println("I got line: " + lines.get(0));
                            for (String s : lines.get(0).split(" ")) {
                                if (wordsCounted.containsKey(s)) {
                                    wordsCounted.put(s, wordsCounted.get(s) + 1);
                                } else {
                                    wordsCounted.putIfAbsent(s, 1);
                                }
                            }
                            lines.remove(0);
                        }
                    } finally {
                        lock.unlock();
                        try {
                            cb.await();
                        } catch (InterruptedException | BrokenBarrierException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
```

```
        }

        ExecutorService executor = Executors.newFixedThreadPool(numberOfThreads);

        executor.submit(new Counter());
        executor.submit(new Counter());
        executor.submit(new Counter());

        executor.shutdown();

    }

}
```

Klasa *WordCount.java*

Celem tej klasy jest zliczenie słów w tekście. Do celów demonstracyjnych oczywiście zaprezentowano niewielki tekst, jednakże należy pamiętać, ze tekst ten mógłby np. być wczytany z pliku i być bardzo długi. Arbitralnie podzielono to zadanie na 3 wątki. Wątki wykorzystują klasę *ReentrantLock* **w wersji blokującej, co ma wpływ na działanie programu.** W przykładzie zaprezentowano też barierę, która oczekuje na zakończenie pracy przez wszystkie wątki, po czym wyświetla na koniec zliczone słowa.

- **Uruchom klasę, jaki jest rezultat jej działania?**
- **Czy zauważasz pewną prawidłowość w działaniu programu?**

2. Przepisz, następującą klasę w języku Java:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.ReentrantLock;

public class WordCountNonBlocking {
    public static void main(String[] args) throws Exception {
        String text = "This is some text. It is intended to be counted by a threads.\n" +
                "Each thread will be taking a line and then will count words.\n" +
                "It is supposed to be faster than a standard, sequential way.\n" +
                "This example is non blocking and should be different from\n" +
                "the previous one.\n";
        ArrayList<String> lines = new ArrayList<>(Arrays.asList(text.split("\n")));
        HashMap<String, Integer> wordsCounted = new HashMap<>();
        ReentrantLock lock = new ReentrantLock(true);
        int numberOfThreads = 3;

        CyclicBarrier cb = new CyclicBarrier(numberOfThreads, () -> {
            for (String s : wordsCounted.keySet()) {
                if (wordsCounted.get(s) > 1) {
                    System.out.println(s + " has " + wordsCounted.get(s) + " occurrences.");
                } else {
                    System.out.println(s + " has " + wordsCounted.get(s) + " occurrence.");
                }
            }
        });


         class Counter extends Thread {
            int begin, end;

            public Counter(int begin, int end) {
                this.begin = begin;
                this.end = end;
            }

            @Override
            public void run() {
                for (int i = begin; i < end; ) {
                    String line = lines.get(i);
                    System.out.println("I am thread" + this.getId() + " and i got line " + line);
                    String[] splittedWords = line.split(" ");
```

```
                    boolean locked = lock.tryLock();
                    if (locked) {
                        for (String s : splittedWords) {
                            if (wordsCounted.containsKey(s)) {
                                wordsCounted.put(s, wordsCounted.get(s) + 1);
                            } else {
                                wordsCounted.putIfAbsent(s, 1);
                            }
                        }
                        i++;
                        lock.unlock();
                    }

                }

                try {
                    System.out.println(this.getId() + " called await");
                    cb.await();
                } catch (InterruptedException | BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }
        }


        ExecutorService executor = Executors.newFixedThreadPool(numberOfThreads);

        executor.submit(new Counter(0, 2));
        executor.submit(new Counter(2, 3));
        executor.submit(new Counter(3, lines.size()));

        executor.shutdown();

        if (executor.isShutdown()) {
            for (String s : wordsCounted.keySet()) {
                System.out.println(s + " has " + wordsCounted.get(s) + " occurrences.");
            }
        }

    }
}
```

Klasa *WordCountNonBlocking.java*

Ten program działa teoretycznie tak samo, jak poprzedni, ale tutaj użycie metody *tryLock* **jest nieblokujące.** Ponownie wykorzystano barierę (jej użycie jest bez zmian).

- **Uruchom klasę, jaki jest rezultat jej działania?**
- **Czym różni się od rezultatu działania klasy *WordCount*?**
- **Napisz klasę *WordCountNonBlockingImproved.java*, która jeszcze ulepszy powyższy proces. W tym celu, niech zliczanie słów w linii będzie wykonywane bez użycia blokady, natomiast niech jedynie nadpisanie wartości hash mapy będzie synchronizowane. Możesz to osiągnąć stosując lokalną hash mapę, którą resetujesz w momencie nadpisywania globalnej hash mapy.**

3. Przepisz poniższe klasy:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

public class Invoice {
    private long id;
    private List<String> items;
    private List<Integer> counts;
    private List<Double> prices;
    private boolean invoicePdf;

    private Invoice(long id, List<String> items, List<Integer> counts, List<Double> prices) {
        this.id = id;
```

```
        this.items = items;
        this.counts = counts;
        this.prices = prices;
    }

    public static Invoice createRandomInvoice() throws InterruptedException {
        Thread.sleep(1000);
        Random r = new Random();
        List<String> availableItems = Arrays.asList("laptop", "smartphone", "HDMI cable", "USB cable",
                                    "switch", "router", "tablet", "mouse", "keyboard", "USB C to USB adapter",
                                    "32 GB microSD card", "500 GB external SSD drive");
        List<Double> availablePrices = Arrays.asList(2500.0, 1400.0, 20.0, 10.0, 75.0, 100.0,
                                     1700.0, 100.0, 80.0, 40.0, 50.0, 650.0);

        int noItems = r.nextInt(5);
        while (noItems == 0) {
            noItems = r.nextInt(5);
        }
        ArrayList<String> selectedItems = new ArrayList<>();
        ArrayList<Integer> selectedCounts = new ArrayList<>();
        ArrayList<Double> selectedPrices = new ArrayList<>();
        for (int i = 0; i < noItems; i++) {
            int randomItemIndex = r.nextInt(availableItems.size());
            selectedItems.add(availableItems.get(randomItemIndex));
            selectedCounts.add(r.nextInt(2));
            selectedPrices.add(availablePrices.get(randomItemIndex));
        }

        return new Invoice(r.nextLong(), selectedItems, selectedCounts, selectedPrices);
    }

    public void printJustCreated() {
        System.out.println("\033[1;32m");
        System.out.println("The new invoice with id " + id + " has been received");
        printInfo();
        System.out.println("\u001B[0m");
    }

    public void makeInvoicePdf() {
        invoicePdf = true;
        System.out.println("\u001B[32m");
        System.out.println("I \"made following invoice pdf\" with id: " + id);
        printInfo();
        System.out.println("\u001B[0m");
    }

    public void sendEmail() {
        System.out.println("\u001B[34m");
        System.out.println("I send email to customer, having following invoice id " + id);
        printInfo();
        System.out.println("\u001B[0m");
    }

    public void printInfo() {
        for (int i = 0; i < items.size(); i++) {
            System.out.println(items.get(i) + "\t" + counts.get(i) + "\t" + prices.get(i));
        }
        System.out.println("pdf created: " + invoicePdf);
    }

    public long getId() {
        return id;
    }

}
```

Klasa Invoice.java

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class InvoiceGeneratorAndSender {

    public static void main(String[] args) {

        ArrayList<Invoice> invoices = new ArrayList<>();
        LinkedList<Invoice> invoicesWithPdf = new LinkedList<>();

        final ReentrantLock lock = new ReentrantLock(true);
        final Condition createdInvoicePdf = lock.newCondition();
        AtomicInteger noInvoicesCreated = new AtomicInteger();
        AtomicInteger noPdfCreated = new AtomicInteger();
        AtomicInteger noEmailsSend = new AtomicInteger();

        new Thread(() -> {
            while (noInvoicesCreated.get() < 10) {
                boolean aquired = false;
                try {
                    aquired = lock.tryLock(100, TimeUnit.MILLISECONDS);
                    if (aquired) {
                        Thread.sleep(500);
                        Invoice i = Invoice.createRandomInvoice();
                        i.printJustCreated();
                        invoices.add(i);
                        System.out.println(invoices.size());
                        noInvoicesCreated.getAndIncrement();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    if (aquired)
                        lock.unlock();
                }
            }
        }).start();

        class Pdf extends Thread {
            @Override
            public void run() {
                while (noPdfCreated.get() < 10) {
                    boolean acquired = false;
                    try {
                        acquired = lock.tryLock();
                        if (acquired) {
                            if (!invoices.isEmpty()) {
                                try {
                                    Thread.sleep(5000);
                                } catch (InterruptedException e) {
                                    e.printStackTrace();
                                }
                                System.out.println("Current thread id:" + this.getId());
                                Invoice first = invoices.get(0);
                                invoices.remove(0);
                                first.makeInvoicePdf();
                                invoicesWithPdf.addLast(first);
                                createdInvoicePdf.signal();
                                noPdfCreated.getAndIncrement();
                            }
                        }
                    } finally {
                        if (acquired)
                            lock.unlock();
                    }
                }
            }
        }

        new Pdf().start();
        new Pdf().start();
```

```
        new Thread(() -> {
            boolean acquired = false;
            while (noEmailsSend.get() < 10) {
                try {
                    acquired = lock.tryLock();
                    if (acquired) {
                        while (invoicesWithPdf.size() == 0) {
                            createdInvoicePdf.await();
                        }
                        invoicesWithPdf.getFirst().sendEmail();
                        invoicesWithPdf.removeFirst();
                        noEmailsSend.getAndIncrement();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    if (acquired)
                        lock.unlock();
                }
            }
        }).start();
    }

}
```

Klasa InvoiceGeneratorAndSender.java

Powyższy program próbuje odwzorować sytuację, w której posiadamy kolejkę zamówień w sklepie internetowym i dzielimy program na dwa zadania: generowanie faktury (zakładamy że zadanie to zajmuje więcej czasu) i wysyłanie maila z wygenerowaną fakturę (jest to zadanie, zajmujące mniej czasu). W celu optymalizacji 2 wątki wykonują zadanie generowania faktury, natomiast jeden wątek zajmuje się wysyłaniem maili. Zakładamy, że zamówienia napływają na bieżąco (w programie dla celów emulacji, oczywiście zamówienia są generowane losowo), natomiast wątki generujące faktury, po „wykonaniu" tej czynności (tutaj też oczywiście emulowanej), wysyłają sygnał do wątków wysyłających maile, aby się wybudziły (chociaż w praktyce wątki te, jeżeli kolejka zamówień z wygenerowaną fakturą nie jest pusta, będą wysyłać maile, jednak gdy ta kolejka jest pusta, wątki usypiają się, oczekując na sygnał wybudzenia). Aby odwzorować sytuację, w której generowanie faktury trwa dłużej niż wysłanie maila, wątek generujący fakturę jest usypiany na 500 ms. W celu ułatwienia analizy, domyślnie wygenerowane zostanie 10 zamówień, ponadto wypisywane są na ekran pokolorowane rezultaty działania wątków, wraz z informacją, który wątek w danym momencie pracuje.

- **Uruchom powyższą klasę 5 razy, co możesz powiedzieć o wynikach tych uruchomień?**
- **Zmień implementacje pętli while, tak aby były to pętle nieskończone; jeżeli wątek wysyłający email czeka, to również wypisuj tę informację na ekran („I have to wait!"). Usuń również Thread.sleep(500) z implementacji wątku tworzącego faktury. Nazwij klasę *InvoiceGeneratorAndSenderInfinity.java.* Co obserwujesz?**
- **Do zmienionej poprzednio implementacji dodaj jeszcze jedną zmianę: uruchom 4 instancję wątków tworzących faktury (a nie dwie, jak było). Nazwij klasę *InvoiceGeneratorAndSenderInfinity4Threads.java.* Czy zmienia to działanie programu?**