Programowanie współbieżne i rozproszone

mgr inż. Marcin Mrukowicz

Instrukcja laboratoryjna VI

Klasyczne problemy synchronizacji i ich rozwiązania w języku Java

1. Problem producenta i konsumenta (ang. producer–consumer problem, bounded-buffer problem)

W problemie tym występuje podział na dwie klasy procesów (wątków): jeden proces (wątek), nazywany producentem, dodaje elementy do współdzielonej kolekcji (tablicy, listy, ogólnie bufora), natomiast drugi proces (wątek) pobiera elementy z tej współdzielonej kolekcji, przetwarza je i potem usuwa. Teoretycznie, gdyby bufor był nieskończony, problem sprowadzałby się jedynie do tego, aby zachować spójność danych: producent nie powinien dodać jakiegoś elementu więcej niż raz, podobnie jak konsument powinien tylko raz przetwarzać dany element. Jednakże ponieważ nieskończony bufor wymagałby nieskończonej pamięci, należy dodatkowo założyć, że posiada on maksymalny rozmiar. Powoduje to, że producent nie może dodać elementu do pełnego bufora i musi czekać aż konsument pobierze dane z bufora i zwolni miejsce w buforze. Jednocześnie konsument nie może pobrać danych z pustego bufora, zatem w takiej sytuacji musi poczekać aż producent wyprodukuje jakiś element i doda go do bufora. Oczywiście dopuszczalne jest sytuacja, w której istnieje wielu producentów i wielu konsumentów.

1. Przepisz, następującą klasę w języku Java:

```
import java.util.LinkedList;
import java.util.Random;

public class ProducerConsumerSleep {
    static LinkedList<String> buffer = new LinkedList<>();
    static final int BUFFER_SIZE = 10000;

    public static void main(String[] args) {
        class Producer extends Thread {

            private String name;

            public Producer(String name) {
                this.name = name;
                this.setPriority(Thread.MAX_PRIORITY);
            }

            @Override
            public void run() {
                while (true) {
                    synchronized (buffer) {
                        if (buffer.size() == BUFFER_SIZE) {
                            try {
                                Thread.sleep(10);
                                System.out.println("Producer " + name + " is sleeping, full buffer");
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        } else {
                            Random r = new Random();
                            String[] words = {"aaa", "bbb", "ccc", "ddd", "eee", "fff"};
                            String s = words[r.nextInt(words.length)];
                            buffer.addLast(s);
```

```
                            System.out.println("Producer " + name + " produces: " + s);
                        }
                    }
                }
            }
        }

        class Consumer extends Thread {
            private String name;

            public Consumer(String name) {
                this.name = name;
                this.setPriority(Thread.MIN_PRIORITY);
            }

            @Override
            public void run() {
                while (true) {
                    synchronized (buffer) {
                        if (buffer.size() == 0) {
                            try {
                                Thread.sleep(10);
                                System.out.println("Consumer " + name + " is sleeping, empty buffer");
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        } else {
                            String elem = buffer.getFirst();
                            buffer.removeFirst();
                           System.out.println("Consumer" + name + " consumes: " + elem + " and process it
                                               to: " + elem.toUpperCase());

                        }
                    }
                }
            }
        }

        new Consumer("I").start();
        new Producer("I").start();
        new Consumer("II").start();
    }

}
```

ProducerConsumerSleep.java

Powyższa klasa to teoretycznie poprawne, ale bardzo nieefektywne rozwiązanie problemu producenta i konsumenta (z uwagi na usypianie wątków). Ponieważ program jest poglądowy producent produkuje losowy ciąg znaków, a konsument pobiera ten ciąg znaków i wypisuje go, zamieniając litery na wielkie. Ważne jest spełnienie założeń dotyczących bufora: nie można odczytać z pustego bufora, ani zapisać do pełnego bufora. Ponadto program spełnia założenie, że konsumenci nie przetwarzają tego samego elementu wiele razy.

- **ustaw rozmiar bufora na 10, czy zmieni to istotnie działanie programu?**
- **dodaj jeszcze 3 kolejnych konsumentów, co się stało?**
- **zauważ, że wątkom ustawiono priorytety wykonania, usuń te priorytety, czy coś to zmieniło?**

2. Przeanalizuj działanie poniższej klasy

```
public class ProducerConsumerWait {

    static LinkedList<String> buffer = new LinkedList<>();
    static final int BUFFER_SIZE = 10;

    public static void main(String[] args) {
        class Producer extends Thread {

            private String name;
```

```
            public Producer(String name) {
                this.name = name;
            }
            @Override
            public void run() {
                while (true) {
                    synchronized (buffer) {
                        if (buffer.size() == BUFFER_SIZE) {
                            try {
                                buffer.wait();
                                System.out.println("Producer " + name + " is waiting, full buffer");
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        } else {
                            Random r = new Random();
                            String[] words = {"aaa", "bbb", "ccc", "ddd", "eee", "fff"};
                            String s = words[r.nextInt(words.length)];
                            System.out.println("Producer " + name + " produces: " + s);
                            buffer.addLast(s);
                            buffer.notify();
                        }
                        try {
                            Thread.sleep(10);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        }

        class Consumer extends Thread {
            private String name;

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

            @Override
            public void run() {
                while (true) {
                    synchronized (buffer) {
                        if (buffer.size() == 0) {
                            try {
                                buffer.wait();
                                System.out.println("Consumer " + name + " is waiting, empty buffer");
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        } else {
                            String elem = buffer.getFirst();
                            System.out.println(this.getPriority());
                            System.out.println("Consumer" + name + " consumes: " + elem + " and process it
                            to: " + elem.toUpperCase());

                            try {
                                Thread.sleep(10);
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }

                            buffer.removeFirst();
                            buffer.notify();
                        }
                    }

                }
            }
        }

        new Consumer("I").start();
        new Producer("I").start();
        new Producer("II").start();
        new Producer("III").start();
        new Producer("IV").start();
    }
```

```
}
```

ProducerConsumerWait.java

Powyższa klasa to rozwiązanie problemu producenta i konsumenta za pomocą bloku synchronizowanego i wywołań *wait* i *notify* na współdzielonym buforze. Rozwiązanie to jest na pewno lepsze niż poprzednie, ale jest to rozwiązanie blokujące: gdy któryś wątek aktualnie korzysta z bufora, to pozostałe wątki nie mogą już tego zrobić i są w tym czasie bezczynne.

3. Przepisz poniższą klasę:

```
public class ProducerConsumerLock {
    static LinkedList<String> buffer = new LinkedList<>();
    static final int BUFFER_SIZE = 20;
    static ReentrantLock lock = new ReentrantLock(true);
    static Condition bufferNotFull = lock.newCondition();
    static Condition bufferNotEmpty = lock.newCondition();


    public static void main(String[] args) {
        class Producer extends Thread {

            private String name;

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

            @Override
            public void run() {
                while (true) {
                    lock.lock();
                    if (buffer.size() == BUFFER_SIZE) {
                        try {
                            bufferNotFull.await();
                            System.out.println("Producer " + name + " is waiting, full buffer");
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    } else {
                        Random r = new Random();
                        String[] words = {"aaa", "bbb", "ccc", "ddd", "eee", "fff"};
                        String s = words[r.nextInt(words.length)];
                        System.out.println("Producer " + name + " produces: " + s);
                        buffer.addLast(s);
                        bufferNotEmpty.signal();
                    }
                    lock.unlock();
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }

        class Consumer extends Thread {
            private String name;

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

            @Override
            public void run() {
                while (true) {
                    lock.lock();
                    if (buffer.size() == 0) {
                        System.out.println("Consumer " + name + " is waiting, empty buffer");
                        try {
                            bufferNotEmpty.await();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    } else {
```

```
                        String elem = buffer.getFirst();
                        System.out.println("Consumer" + name + " consumes: " + elem + " and process it to: "
+ elem.toUpperCase());

                        try {
                            Thread.sleep(10);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                        buffer.removeFirst();
                        bufferNotFull.signal();
                    }
                    lock.unlock();
                }

            }
        }

        new Producer("I").start();
        new Producer("II").start();
        new Producer("III").start();
        new Consumer("I").start();
    }


}
```

ProducerConsumerLock.java

W istocie rzeczy blok synchronizowany jest równoważny powyższemu programowi (oczywiście poza aktywną liczbą producentów i konsumentów); widać tutaj wyraźnie, że kiedy wątek zablokuje dostęp do bufora, pozostałe wątki będą zmuszone czekać na dostęp do bufora i nie będą w tym czasie wykonywać żadnej pracy. Uśpienie wątku jest tutaj wykorzystane tylko w celu ułatwienia analizy wyników wypisywanych w konsoli.

- **zmodyfikuj przykład z blokiem synchronizowanym, tak aby było dwóch producentów i dwóch konsumentów. Niech jeden producent wyprodukuje dokładnie jeden element, a drugi dokładnie 3, po czym zakończą oni działanie (zmodyfikuj warunek zatrzymania pętli while). Z kolei niech konsumenci skonsumują dokładnie po 2 elementy i zakończą swoje działanie (rówież zmodyfikuj działanie ich pętli). Nazwij klasę *ProducerConsumerExactlyRuns.java***

4. Przepisz nastepującą klasę:

```
public class ProducerConsumerNonBlocking {
    static LinkedList<String> buffer = new LinkedList<>();
    static final int BUFFER_SIZE = 20;
    static ReentrantLock lock = new ReentrantLock(true);
    static Condition bufferNotFull = lock.newCondition();
    static Condition bufferNotEmpty = lock.newCondition();


    public static void main(String[] args) {
        class Producer extends Thread {

            private String name;
            private ArrayList<String> localItems;

            public Producer(String name) {
                this.name = name;
                this.localItems = new ArrayList<>();
            }

            void produceElem() {
                Random r = new Random();
                String[] words = {"aaa", "bbb", "ccc", "ddd", "eee", "fff"};
                localItems.add(words[r.nextInt(words.length)]);
            }

            void produce() {
                if (localItems.size() < 10) {
                    produceElem();
                }
```

```
        }

        @Override
        public void run() {
            while (true) {
                if (localItems.size() == 0) {
                    System.out.println("Producer has 0 items");
                }

                boolean gained = lock.tryLock();
                if (gained) {
                    if (buffer.size() == BUFFER_SIZE) {
                        try {
                            bufferNotFull.await();
                            System.out.println("Producer " + name + " is waiting, full buffer");
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    } else {
                        buffer.addAll(localItems);
                        localItems.clear();
                        bufferNotEmpty.signal();
                    }
                    lock.unlock();
                }

                produce();

            }
        }
    }

    class Consumer extends Thread {
        private String name;

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

        @Override
        public void run() {
            while (true) {
                lock.lock();
                if (buffer.size() == 0) {
                    System.out.println("Consumer " + name + " is waiting, empty buffer");
                    try {
                        bufferNotEmpty.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                } else {
                    String elem = buffer.getFirst();
                    System.out.println("Consumer" + name + " consumes: " + elem + " and process it to: "
                    + elem.toUpperCase());

                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    buffer.removeFirst();
                    bufferNotFull.signal();
                }
                lock.unlock();
            }

        }
    }

    new Producer("I").start();
    new Producer("II").start();
    new Producer("III").start();

    new Consumer("I").start();
    new Consumer("II").start();
    new Consumer("III").start();
    new Consumer("IV").start();
```

```
    }
}
```

ProducerConsumerNonBlocking.java

Powyższy przykład, posiada producenta zaimplementowanego w wersji nieblokującej, natomiast konsumenta w wersji blokującej. Oznacza to, że producent tworzy elementy lokalnie (co może w ogólności trwać długo), po czym, po otrzymaniu dostępu do bufora, jedynie szybko dodaje do niego te elementy. Rozwiązanie to powoduje, że kiedy producent zajmuje się tworzeniem nowych elementów nie blokuje innym wątkom dostępu do współdzielonego zasobu.

- **Wzorując się na tym, jak działa producent, zaimplementuj konsumenta również w wersji nieblokującej**

5*. Pobierz Apache Kafka i rozpakuj archiwum. Przejdź do folderu. Następnie uruchom:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

W kolejnym oknie terminala wpisz:

```
bin/kafka-server-start.sh config/server.properties
```

Efektem wykonania poleceń jest uruchomienie Apache Kafka w domyślnej konfiguracji.

Za pomocą polecenia utwórz nowy topic uppercaseing:

```
bin/kafka-topics.sh --create --topic uppercaseing --bootstrap-server localhost:9092
```

- **Napisz w 1-2 zdaniach co to jest topic w Apache Kafka (https://kafka.apache.org/documentation/#intro_concepts_and_terms)**

Następnie uruchom implementację konsolowego producenta:

```
bin/kafka-console-producer.sh –topic uppercaseing  --bootstrap-server localhost:9092
```

Wpisz następujące linie:

```
aaa
bbb
ccc
ddd
eee
ctrl+c
```

W innym oknie terminala wpisz:

```
bin/kafka-topics.sh --describe --topic uppercaseing --bootstrap-server localhost:9092
```

Następnie uruchom implementację konsolowego konsumenta:

```
bin/kafka-console-consumer.sh --topic uppercaseing --from-beginning --bootstrap-server
localhost:9092
```

- **Jak działa domyślny producent i konsument?**

Poniżej zaprezentowano programowy przykład producenta i konsumenta w Apache Kafka:

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;
```

```
public class KafkaProducerExample {


   public static void main(String [] args) {

      Properties props = new Properties();
      props.put("bootstrap.servers", "localhost:9092");
      props.put("linger.ms", 1);
      props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
      props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

      Producer<String, String> producer = new KafkaProducer<>(props);
      int noWords = 0;
      while(noWords < 10) {
         producer.send(new ProducerRecord<>("uppercaseing", "words", "bbb"));
         noWords++;
      }
      producer.close();

   }
}
```

KafkaProducerExample.java

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

public class KafkaConsumerExample {
   public static void main(String [] args) {
      Properties props = new Properties();
      props.setProperty("bootstrap.servers", "localhost:9092");
      props.setProperty("group.id", "test");
      props.setProperty("enable.auto.commit", "true");
      props.setProperty("auto.commit.interval.ms", "1000");
      props.setProperty("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
      props.setProperty("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
      KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
      consumer.subscribe(Arrays.asList("uppercaseing"));
      while (true) {
         ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
         for (ConsumerRecord<String, String> record : records) {
            if (record.key().equals("words")) {
               System.out.printf("offset = %d, value = %s%n", record.offset(),
record.value().toUpperCase());
```

```
            }
        }
    }
  }
}
```

KafkaConsumerExample.java

Skopiuj klasy do katalogu głównego Kafka, po czym wywołaj najpierw klasę producenta:

```
bin/kafka-run-class.sh ./KafkaProducerExample.java
```

Następnie wywołaj klasę konsumenta:

```
bin/kafka-run-class.sh ./KafkaConsumerExample.java
```

Konsument w tej wersji rzeczywiście wykonuje operacje na danych – zamienia otrzymany łańcuch znaków z małych liter na wielkie.

- **Zauważ, że konsumowany ten sam topic, jak we wcześniejszym przykładzie. Czy ma to jakiś związek, z tym, co wykonał programowy konsument?**
- **Na podstawie https://kafka.apache.org/documentation/#intro_concepts_and_terms opisz co w terminologii Apache Kafka znaczy key (event key)**
- **Na podstawie: https://kafka.apache.org/documentation/#semantics opisz jakie trudności występują w zapewnieniu teoretycznej własności, że dany event jest przetwarzany dokładnie jeden raz (ang. exactly once)?**

*** - zadanie dla osób chętnych**