Problém producenta a konzumenta

(paralelné programovanie v Jave)

Róbert Novotný

robert.novotny@upjs.sk

Producent-konzument

- Majme sklad (s obmedzenou kapacitou)
  - dodávateľ doň vkladá palety s tovarom
  - odberatelia ich vyberajú
- Dodávateľ je producent – vkladá do kontajnera položky, ak je voľné miesto
- Odberateľ je konzument – vyberá z kontajnera položky, ak tam nejaké sú.
- Aby sme predišli bitkám v sklade, vyhlásime, že v sklade môže byť nanajvýš 1 producent alebo 1 konzument.

Producent-konzument

- Každé z vlákien producenta a konzumenta bude pri práci uzamykať sklad
- Vlákna budú synchronizované čakania na sklad
- Konzument sa musí aktívne pozerať, či v sklade niečo nie je.
- Producent sa musí aktívne pozerať, či sklad nie je náhodou plný.
- Toto budú robiť paralelne.

Producent-konzument

```
public class Channel {
    private int[] data;

    private int elementCount;

    public Channel(int capacity) {
        this.data = new int[capacity];
    }

    public void put(int element) { ... }

    public int get() { .... }
}
```

interne budeme dáta ukladať v poli

Producent-konzument

```
public void put(int element) {
  while(elementCount == data.length) { }
  data[elementCount] = element;
  elementCount++;
}
```

aktívne čakáme kým sa sklad neuvoľní

```
public int get() {
  while(elementCount == 0) { }
  int element = data[0];
  int[] newData = new int[data.length];

  System.arraycopy(data, 1, newData, 0, data.length - 1);
  data = newData;
  elementCount--;
  return element;
}
```

aktívne čakáme na príchod prvku

Implementácia producenta

```
public class Producer extends Thread {
    private Channel channel;

    public Producer(Channel channel) {
        this.channel = channel;
    }

    public void run() {
        while(true) {
            channel.put(...);
        }
    }
}
```

producent ukladá prvky do skladu

Implementácia konzumenta

```
public class Consumer extends Thread {
    private Channel channel;

    public Consumer(Channel channel) {
        this.channel = channel;
    }

    public void run() {
        while(true) {
            int item = channel.get();
            ...
        }
    }
}
```

konzument vyberá prvky zo skladu a spracováva ich

Producent-konzument

- Výhody:
  - producent netuší o konzumentovi
  - konzument netuší o producentovi
  - kanál (sklad) ich robí nezávislými
  - takto je možné zadefinovať aj viacero producentov / konzumentov
- Nevýhody našej implementácie
  - implementácia nie je thread-safe!
    - pole je zdieľané viacerými vláknami!
  - pri vkladaní / vyberaní prvkov sa čaká aktívne: to žerie procesorový čas:
    - v cykle márnime čas

Porušenie thread-safe princípu

```
public int get() {
  while(elementCount == 0) { }
  int element = data[0];
  int[] newData = new int[data.length];
  System.arraycopy(data, 1, newData, 0, data.length - 1);
  data = newData;
  elementCount--; return element;
}
```

get(), po arraycopy():

premenná **data:**

3	7	-	-

premenná **newData:**

7	-	-	-

konzument je odstavený, producent vloží do **data**

premenná **data:**

3	7	4	-

zobudí sa producent, vykoná **data = newData**

premenná **data** zrazu obsahuje **newData:**

7	-	-	-

prvok vložený konzumentom sa stratil!

Porušenie thread-safe princípu pri viacerých producentoch

```
public void put(int element) {
  while(elementCount == data.length) { }
  data[elementCount] = element;
  elementCount++;
}
```

- majme pole s kapacitou 4 (indexy 0—3)
- v poli tri prvky (elementCount = 3)
- producent 1 vkladá nový prvok, strčíme ho na index 3 (koniec poľa)
- odstavíme producent a1, zobudíme producent a2
- producent 2 vkladá nový prvok, strčíme ho na index 3 (koniec poľa)
- zobudí sa producent 1, zvýši elementCount na 4
- zobudí sa producent 2, zvýši elementCount na 5
- zobudí sa konzument, chce brať prvok z piatej pozície => prekročenie rozsahu!

Nápad na zaistenie thread-safe

```
public synchronized void put(int element) {
   ...
}
```

```
public synchronized int get() {
   ...
}
```

- Máme dve zdieľané premenné:
  - pole prvkov
  - počet prvkov v poli
- Inštancia si mení svoje premenné, teda definujeme kritickú sekciu, kde uzamykáme samotnú inštanciu

Nápad je zlý, pretože raz nastane situácia, keď...

1. Producent zavolá put(), vojde do skladu
2. Konzument zavolá get(). Chce vojsť do skladu.
   - sklad je zavretý, konzument musí počkať na odomknutie
3. Producent opustí sklad, a zhodou okolností hneď dostane príležitosť plniť sklad ďalej. Toto sa opakuje, až kým...
4. ...producent zaplní sklad a začne aktívne čakať na konzumenta, aby vyprázdnil sklad
   - aktívne čakanie je vo vnútri kritickej sekcie = producent je vo vnútri skladu!
   - lenže konzument stále stojí vo fronte pred skladom a nemôže sa dostať dnu, tam je totiž producent!

Deadlock!

Producent čaká na konzumenta a naopak!

Podmienky pre výskyt deadlocku

„Ak sa na križovatke stretnú dve súpravy, obe sú povinné zastaviť. Súprava môže pokračovať v jazde až po tom, čo druhá súprava opustí križovatku.“

-- nelogický železničný zákon v Kansase

1. **vzájomné vylúčenie**: prostriedok (dáta) môže využívať najviac jedno vlákno
2. **drží a čaká**: vlákno drží jeden prostriedok a čaká na iný
3. **vylúčenie preemptívnosti**: vlákno sa musí vzdať prostriedku dobrovoľne. Nie je k dispozícii spôsob, ktorým je možné odobrať vláknu prostriedok.
4. **cyklické čakanie**: dve či viac vlákien tvoria cyklickú postupnosť, kde jedno vlákno čaká na prostriedok nasledujúcim vláknom v postupnosti.

Princíp *wait-notify*

- producent/konzument sa vo vnútri skladu vie oprieť o stenu a spať, kým nie je sklad voľný/plný.
- Idea wait-notify:
  - vlákno sa môže dočasne vzdať zámku a čakať na splnenie podmienky
  - čaká dovtedy, kým ho niekto neupozorní, že podmienka bola splnená
  - čaká sa v kritickej sekcii

Wait-notify umožňuje vláknu čakať na splnenie podmienky

Princíp *wait-notify*

- metóda wait():
  - vlákno sa vzdá zamknutého objektu a čaká
  - zavolaná na inštancii objektu, ktorý je uzamknutý
  - musí byť volaná v kritickej sekcii
- notify() / notifyAll():
  - upozorní vlákno, ktoré drží zamknutý objekt a zobudí ho
  - volaná v kritickej sekcii, ktorá uzamkla príslušný objekt
    - zobudené vlákna počkajú na jej dobehnutie
  - zavolaná na inštancii objektu, ktorý je uzamknutý
  - *notifyAll*: upozorní všetky vlákna, *notify* upozorní jedno (náhodne)
    - lepšie používať notifyAll

Vkladanie prvkov

```
public synchronized void put(int element) {
  while(elementCount == data.length) {
    try {
      // v kritickej sekcii uzamykáme inštanciu
      // pasívne čakáme
      this.wait();
    } catch (InterruptedException e) {
      // nerobíme nič
    }
  }
  data[elementCount] = element;
  elementCount++;
  // oznámime všetkým držiacim inštanciu, že sa majú
  // zobudiť
  this.notifyAll();
}
```

Vyberanie prvkov

```
public synchronized int get() {
  while(elementCount == 0) {
    try {
      // v kritickej sekcii uzamykáme inštanciu
      // pasívne čakáme
      this.wait();
    } catch (InterruptedException e) {
      // nerobíme nič
    }
  }
  int element = data[0]; int[] newData = new int[data.length];
  System.arraycopy(data, 1, newData, 0, data.length - 1);
  data = newData; elementCount--;

  this.notifyAll();

  return element;
}
```

Podpora producenta-konzumenta v Jave

- interfejs java.util.concurrent.BlockingQueue
- kolekcia podporujúca front presne spĺňajúci požiadavky nášho skladu
- implementácia java.util.concurrent.LinkedBlockingQueue
  - umožňuje špecifikovať kapacitu
  - ak je front plný, ďalšie vkladanie čaká dovtedy, kým sa neuvoľní miesto
  - ak je front prázdny, vyberanie čaká dovtedy, kým sa neuvoľní miesto
  - navyše je to kolekcia, môžeme používať klasické metódy (add, remove, iterácie...)
- nemusíme používať žiadnu synchronizáciu
- nemusíme si komplikovať život s *wait-notify*

Producent a konzument ešte raz

```
class Producent implements Runnable {

   private final BlockingQueue<String> front;

   Producer(BlockingQueue<String> front) {
     this.front = front;
   }

   public void run() {
     try {
       while(true) {
         front.put( ... );
       }
     } catch (InterruptedException ex) {
        // niekto nás prerušil
     }
   }
}
```

Producent a konzument ešte raz

```
class Konzument implements Runnable {

   private final BlockingQueue<String> front;

   Konzument(BlockingQueue<String> front) {
     this.front = front;
   }

   public void run() {
     try {
       while(true) {
         String s = front.take( ... );
       }
     } catch (InterruptedException ex) {
        // niekto nás prerušil
     }
   }
}
```

Príklad s dvoma konzumentami

```
public class Tester {
  public static void main(String[] args) {
     BlockingQueue front = new LinkedBlockingQueue ();

     Producent p = new Producent(front);

     Konzument c1 = new Konzument(q);
     Konzument c2 = new Konzument(q);

     new Thread(p).start();
     new Thread(c1).start();
     new Thread(c2).start();
   }
 }
```