KONKURENTNÉ PROGRAMOVANIE

4. cvičenie: Konkurentné kolekcie

Stav objektu/triedy

- Stav objektu je uložený v jeho inštančných premenných
- Stav triedy je uložený v jej statických premenných
- Stav objektu môže mať definované obmedzenia
  - početKolies musí byť nezáporný a konečný
  - ak pohlavie==„žena“, rodnéČíslo musí mať 4. cifru 5 alebo 6
- Stav = stavové premenné + obmedzenia

Konzistentnosť stavu

- Konzistentný stav je podstatou vláknovej bezpečnosti
- Spôsoby na zachovanie konzistentnosti stavu
  - cez vlastníctvo stavu: nezdieľať stav medzi vláknami (čítanie ani zmena)
  - sprístupňovať konečný/nemenný stav
  - celý stav realizovať prostredníctvom thread-safe stavovej premennej
  - premenlivý stav, ktorý nie je sám o sebe thread-safe, strážiť tým istým zámkom **vždy**, keď sa k nemu pristupuje (čítanie aj zmena)

Viachodnotový stav

- Ak nemám žiadne obmedzenia a nezávislé premenné, môžem delegovať vláknovú bezpečnosť na vláknovo bezpečné stavové objekty

```
public class VisualComponent {
  private final List<KeyListener> keyListeners =
                      new CopyOnWriteArrayList<KeyListener>();
  private final List<MouseListener> mouseListeners =
                     new CopyOnWriteArrayList<MouseListener>();

  public void addKeyListener(KeyListener listener) {
     keyListeners.add(listener);
  }
  public void addMouseListener(MouseListener listener) {
     mouseListeners.add(listener);
  }
...
```

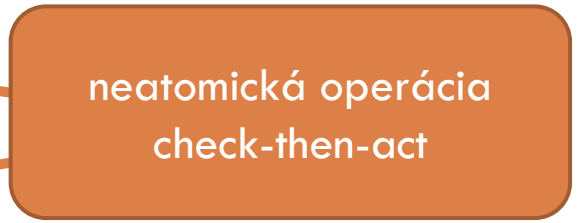
Viachodnotový stav

- Ak máme obmedzenia alebo závislé stavové premenné, delegácia nefunguje

```
@NotThreadSafe
public class Interval {
  private final AtomicInteger lower = new AtomicInteger(0);
  private final AtomicInteger upper = new AtomicInteger(0);

  public void setLower(int i) {
    if (i <= upper.get()) lower.set(i);
  }

  public void setUpper(int i) {
    if (i >= lower.get()) upper.set(i);
  }
...
```



Synchronizované kolekcie

- Všetko sú to thread-safe triedy
- Riešenie vláknovej bezpečnosti cez synchronizované metódy
  - zamknutie cez synchronized(this)
- napr. Vector, HashTable
- ľubovoľná kolekcia obalená cez niektorý zo synchronizovaných pohľadov:
  - Collections.synchronizedXxx(Xxx xxx)
    - Napr. synchronizedSet, synchronizedList, …

Rozsah vláknovej bezpečnosti

- Z definície vláknovej bezpečnosti: Konkurentné volanie metód na vláknovo bezpečnom objekte neporuší konzistentnosť stavu tohto objektu
- Ak je objekt thread-safe, nemusí to znamenať bezstarostné používanie pre volajúci kód bez ďalšej synchronizácie
  - Vždy sa treba pozrieť do dokumentácie (ak existuje)
  - Typické problémy
    - volanie viacerých metód po sebe
    - iterovanie cez kolekciu

Client-side zamykanie

- Ak thread-safe trieda využíva zamykanie cez synchronizované metódy
- Zamkneme viacnásobné volania metód alebo iterácie cez rovnaký kľúč – teda cez referenciu na objekt
  - Že je to OK, by sme mali zistiť z dokumentácie

```
public static Object getLast(Vector list) {
  synchronized (list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
  }
}
```

Synchronizovaná kolekcia

atomizujem check-then-act

Iterovanie cez kolekcie

- Často je ťažké na prvý pohľad odhaliť, že nejaká metóda iteruje cez kolekciu

```
public class SynchronizedSet {
  private final Set<Double> set = new HashSet<Double>();

  public synchronized void add(Double i) { set.add(i); }

  public synchronized void remove(Integer i) {set.remove(i); }

  ...
}
```

```
public void addTenThings() {
  for (int i = 0; i < 10; i++)
    add(Math.random());
  System.out.println("added ten elements to " + set);
}
```

Iterovanie cez kolekcie

- Často je ťažké na prvý pohľad odhaliť, že nejaká metóda iteruje cez kolekciu

```
public class SynchronizedSet {
  private final Set<Double> set = new HashSet<Double>();

  public synchronized void add(Double i) { set.add(i); }

  public synchronized void remove(Integer i) {set.remove(i); }

  ...
}
```

```
public void addTenThings() {
  for (int i = 0; i < 10; i++)
    add(Math.random());
  System.out.println("added ten elements to " + set);
}
```

.toString()

Iterovanie cez kolekcie

- Typické metódy, ktoré iterujú cez kolekciu
  - toString()
  - hashCode()
  - equals()
  - containsAll()
  - removeAll()
  - retainAll()
  - konštruktory s kolekciou na vstupe
- Ak používame synchronizované kolekcie, niektoré z týchto metód sa musia extra zamykať

Iterácie cez synchronizované kolekcie

- Môžeme aplikovať client-side zamykanie
  - Ak je kolekcia dlhá, môže to značne spomaliť
  - Počas iterovania nemôže žiadne vlákno pristupovať ku kolekcii – ani čítať, ani meniť
- Ak je akcia s prvkom dlhá, môžeme synchronizovane riešiť skopírovanie kolekcie a pracovať s prvkami až potom
  - Ak je málo zmien a veľa iterovaní, je to neefektívne
  - Robustnejším riešením sú konkurentné kolekcie

Konkurentné kolekcie

- Balíček **java.util.concurrent**
- Nezamyká sa celá štruktúra, iba nutná časť
- Dodané nové atomické metódy na typické akcie
  - putIfAbsent, addIfAbsent
  - replace
  - podmienené mazanie
- Iterátory sú slabo konzistentné
  - Nezablokujú kolekciu počas iterovania
- Typicky neaktuálne operácie
  - iterácia, size(), isEmpty()

Konkurentné kolekcie

- HashMap → ConcurrentHashMap
- TreeMap → ConcurrentSkipListMap
- TreeSet → ConcurrentSkipListSet
- ArrayList → CopyOnWriteArrayList
  - Vytvára novú kópiu štruktúry pri zmene
  - Iterátory majú „fotku“ zoznamu v čase ich vytvorenia
  - Vhodné, ak je viac čítaní ako zápisov
    - Ak je to naopak, vhodnejší je synchronizovaný zoznam s vytváraním kópie pred iterovaním

Blokovaný rad

- Implementácie interfejsu BlockingQueue
  - Najčastejšie ArrayBlockingQueue, LinkedBlockingQueue
  - Blokované operácie
    - put(prvok) – zaspím, kým nebude v rade voľné miesto
    - take() – zaspím, kým nebude v rade nejaký prvok
  - Dočasne blokované operácie
    - offer(prvok, čas, jednotka)
    - poll(čas, jednotka)
    - Ak sa nepodarí akcia za uvedený čas, vrátim false/null
- Vhodné pre návrhový vzor producenti-konzumenti

Ukončenie práce konzumentov

- Ak producenti nemajú čo produkovať, skončia
  - Konzumenti by mali byť informovaní, nech už nečakajú na ďalšie prvky a tiež skončia
- Ak máme jediného konzumenta
  - Producenti dekrementujú nejakú atomickú premennú „početAktívnychProducentov“
  - Konzument chce konzumovať:
    - v prípade, že je rad prázdny pozrie, či je nejaký producent aktívny, ak áno čaká na prvok, ak nie tak končí
    - v prípade, že rad nie je prázdny, skonzumuje prvok

Ukončenie práce konzumentov

- Ak máme ľubovoľný počet konzumentov
  - „Chutnejšie“ riešenie je dať do radu otrávené pilulky (poison pills)
    - Špeciálny prvok, znamenajúci: skonči!
    - Aspoň toľko piluliek, koľko je konzumentov

```
public class Konzument {
  …
  public void run() {
    while(true) {
      Element element = queue.take();
      if (element.equals(POISON_PILL)) break;
      consume(element);
    }
  }
}
```

Zadanie

- Stiahnite si z Gitu poslednú verziu:
  - https://gitlab.science.upjs.sk/peter.gursky/kopr2019
- Nasledovný balíček:
  - cviko04.zadanie

1. **Zabezpečte súbežnú prácu úlohy Searcher a úlohy FileAnalyzer cez blokovaný rad**
2. Zvýšte počet vlákien súbežne vykonávajúcich úlohu FileAnalyzer
3. Zvýšte počet vlákien súbežne vykonávajúcich úlohu Searcher

Riešenie 1

- Spustíme úlohu Searcher v samostatnom vlákne
- Na rad súborov na analýzu použijeme LinkedBlockingQueue
- Využijeme poison pill

Zadanie

- Stiahnite si z GitHubu poslednú verziu:
  - https://gitlab.science.upjs.sk/peter.gursky/kopr2019
- Nasledovný balíček:
  - cviko04.zadanie

1. Zabezpečte súbežnú prácu úlohy Searcher a úlohy FileAnalyzer cez blokovaný rad
2. **Zvýšte počet vlákien súbežne vykonávajúcich úlohu FileAnalyzer**
3. Zvýšte počet vlákien súbežne vykonávajúcich úlohu Searcher

Riešenie 2

- Spustíme N vlákien s úlohou FileAnalyzer
- Úloha Searcher vygeneruje N otrávených piluliek
- Ak chceme počkať s výpisom riešenia potrebujeme mechanizmus na informovanie o skončení N vlákien s úlohou FileAnalyzer
  - CountDownLatch

CountDownLatch

- Funguje ako brána, ktorá sa otvorí, keď sa dosiahne konečný stav
- Blokované čakanie na otvorenie:
  - countDownLatch.**await**();
  - Spím pokiaľ sa neotvorí
- Štartovací stav : počet vlákien
- Keď vlákno končí úlohu, nakoniec dekrementuje stav
  - countDownLatch.**countDown**();
- Konečný stav – hodnota 0

Riešenie 2

- Spustíme N vlákien s úlohou FileAnalyzer
- Úloha Searcher vygeneruje N otrávených piluliek
- Využijeme CountDownLatch
  - main → await();
  - FileAnalyzer → countDown();
- Zabezpečíme atomickosť zmeny počtu výskytov slov v úlohe FileAnalyzer
  - cez kritickú sekciu
  - alebo cez konkurentnú mapu slov na AtomicInteger
  - alebo cez metódu konkuretnej mapy `merge`

Zadanie

- Stiahnite si z GitHubu poslednú verziu:
  - https://gitlab.science.upjs.sk/peter.gursky/kopr2019
- Nasledovný balíček:
  - cviko04.zadanie

1. Zabezpečte súbežnú prácu úlohy Searcher a úlohy FileAnalyzer cez blokovaný rad
2. Zvýšte počet vlákien súbežne vykonávajúcich úlohu FileAnalyzer
3. **Zvýšte počet vlákien súbežne vykonávajúcich úlohu Searcher**

Riešenie 3

- Potrebujeme rad adresárov na prehľadávanie
- Spustíme N vlákien s úlohou Searcher
- Úloha Searcher končí, keď už žiaden adresár nebude prehľadávaný
  - Pozor na dočasne prázdny rad adresárov na prehľadávanie
  - AtomicInteger unprocessedDirs
- Aspoň jedna úloha Searcher vygeneruje N otrávených piluliek
- Na ukončenie potenciálne donekonečna čakajúcich úloh Searcher môžeme opäť využiť otrávené pilulky