Sinhronizacija niti

Sadržaj

Uvod

- Kod konkurentnih programa možemo imati veliki broj niti koje se istovremeno izvršavaju
- Izvršavaju se na jednom računaru sa više procesora, ali sa jednim memorijskim prostorom
- Veliki problemi mogu nastati kada više niti koriste iste promenljive

Uvod

```
public class Brojanje extends Thread {
    private static int brojac = 0;

    public void run() {
        for (int i=0; i<5000; i++) {
            brojac = brojac + 1;
        }
    }

    public static void main(String[] args) {
        Brojanje nit1 = new Brojanje();
        Brojanje nit2 = new Brojanje();
        nit1.start();
        nit2.start();
        try {
            nit1.join();
            nit2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(brojac);
    }
}
```

Uvod

- Nit 1 i nit 2 istovremeno pristupaju brojaču i čitaju broj 0
- Nit 1 dodaje 1 na brojač i to iznosi 1
- Nit 2 dodaje 1 na brojač i to je takođe 1
- Nit 1 dodeljuje brojaču vrednost 1
- Nit 2 takođe dodeljuje brojaču vrednost 1

Dakle, dva istovremena inkrementa su uvećala brojač samo jedanput

Uvod

- Prethodno opisani problem se na engleskom zove race condition
- Postoje dve mogućnosti za rešavanje ovog problema:
  - Program koji ne koristi deljene promenljive i ne čuva stanje: izvodljivo, ali obično veoma teško u objektnom orijentisanom programiranju
  - Sinhronizacija niti: kontrolisan pristup promenljivima od strane niti

Sinhronizacija niti u Javi

- Kontrolisan pristup promenljivima se može postići korišćenjem atomičkih regiona
- Atomički regioni su delovi koda koji se mogu izvršavati odjednom i SEKVENCIJALNO, slično transakcijama
- Ovo nam omogućava da niti jedna po jedna koriste neku promenljivu, čime se izbegava trka za resursima

synchronized blok

- Atomički regioni u Javi se najlakše implementiraju upotrebom synchronized bloka
- Sinhronizacioni blokovi zaključavaju pristup nekom objektu
- Potrebno je proslediti referencu na taj objekat prilikom pisanja synchronized bloka
- Primer:

```
synchronized(referenca_na_objekat) {
    // Sekvencijalni pristup uz zakljucavanje
    // referenciranog objekta
}
```

Sinhronizacija uz pomoć synchronized bloka

```
public class Brojac {
    public int vrednost = 0;
}
public class Brojanje extends Thread {
    private static Brojac brojac = new Brojac();

    public void run() {
        for (int i=0; i<5000; i++) {
            synchronized(brojac) {
                brojac.vrednost = brojac.vrednost + 1;
            }
        }
    }

    public static void main(String[] args) {
        Brojanje nit1 = new Brojanje();
        Brojanje nit2 = new Brojanje();
        nit1.start();
        nit2.start();
        try {
            nit1.join();
            nit2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(brojac.vrednost);
    }
}
```

synchronized metoda

- Drugi, sličan način je definisanjem synchronized metoda
- Potrebno je pored deklaracije metode dodati synchronized
- Razlika je u tome što se zaključava ceo objekat klase umesto jedne promenljive (referenca this)
- Primer:

```
public synchronized void metod() {
  // sekvencijalni pristup uz zakljucavanje
  // celog objekta (this)
}
```

Uvod

```
public class Brojac {
    public int vrednost = 0;
    public synchronized void uvecaj() {
      vrednost++;
    }
}
public class Brojanje extends Thread {
    private static Brojac brojac=new Brojac();

    public void run() {
        for (int i=0; i<5000; i++) {
           synchronized(brojac) {
                brojac.uvecaj();
           }
        }
    }

    public static void main(String[] args) {
        Brojanje nit1 = new Brojanje();
        Brojanje nit2 = new Brojanje();
        nit1.start();
        nit2.start();
        try {
            nit1.join();
            nit2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(brojac.vrednost);
    }
}
```

Dodatne napomene za sinhronizaciju

- Synchronized blok, odnosno metoda bi trebalo da bude što je manja moguća
- Ako se cela nit stavi u synchronized blok ili metodu, program postaje sekvencijalan
- Ako se kod koji se dugo izvršava (npr. pisanje u datoteku, komplikovana matematička izračunavanja) stavi u synchronized blok ili metodu, ostale niti će se blokirati dok se kod ne izvrši (što dugo traje)
- Pametno odabrati koji deo koda se sinhronizuje
- Trebalo bi da se u synchronized blok ili metodu stave samo delovi koda za pristupanje i izmenu nekog polja (recimo samo čitanje i pisanje)

```
// dugotrajna operacija
double proizvod = komplikovaniUpitNadBazom();
synchronized (obracun) {
  // Jedini bitan deo za sinhronizaciju
  // Pristupanje i pisanje u promenljivu
  obracun.suma += proizvod;
}
```

volatile polje

- Pri korišćenju varijabli od strane različitih niti je moguće da jedna nit 'ne zna' da je druga nit promenila vrednost varijable
- Ovo se dešava zato što Java optimizuje izvršavanje koda i kešira jednom očitanu vrednost
- ```
  public class BeskonacnaPetlja extends Thread {
      static boolean zaustaviSe = false;
      public void run() {
          while (!zaustaviSe) {

          }
          System.out.println("Niz zaustavljena");
      }

      public static void main(String[] args) {
          BeskonacnaPetlja t = new BeskonacnaPetlja();
          t.start();
          try {
              Thread.sleep(1000);
          } catch (InterruptedException e) {
              e.printStackTrace();
          }
          zaustaviSe = true;
          System.out.println("zaustaviSe = true");
      }
  }
  ```
  ```

Primer sa volatile poljem

```
public class VolatilePrimer extends Thread {
    static volatile boolean zaustaviSe = false;
    public void run() {
        while (!zaustaviSe) {

        }
        System.out.println("Niz zaustavljena");
    }

    public static void main(String[] args) {
        VolatilePrimer t = new VolatilePrimer();
        t.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        zaustaviSe = true;
        System.out.println("zaustaviSe = true");
    }
}
```

Vector

- Vector je sinhronziovana kolekcija u Javi
- Operacije nad njom (dodavanje, izmena, brisanje...) su automatski sinhronizovane
- Preporuka je da se koristi samo ako je potrebno thread-safe rešenje
- Upotrebljava se na isti način kao ArrayList

Primer: kontrola leta

- Primer predstavlja aplikaciju za kontrolu leta aerodroma
- Svaki avion leti 3 sekunde
- Da bi se izbegao sudar, samo jedan avion može da sleti u jednom trenutku; dok jedan sleće, ostali čekaju
- Kada avion sleti, dozvoljava se sletanje sledećem avionu
- Avioni, koji čekaju, svake sekunde proveravaju da li je sletanje dozvoljeno