Sinhronizacija niti

Sadržaj

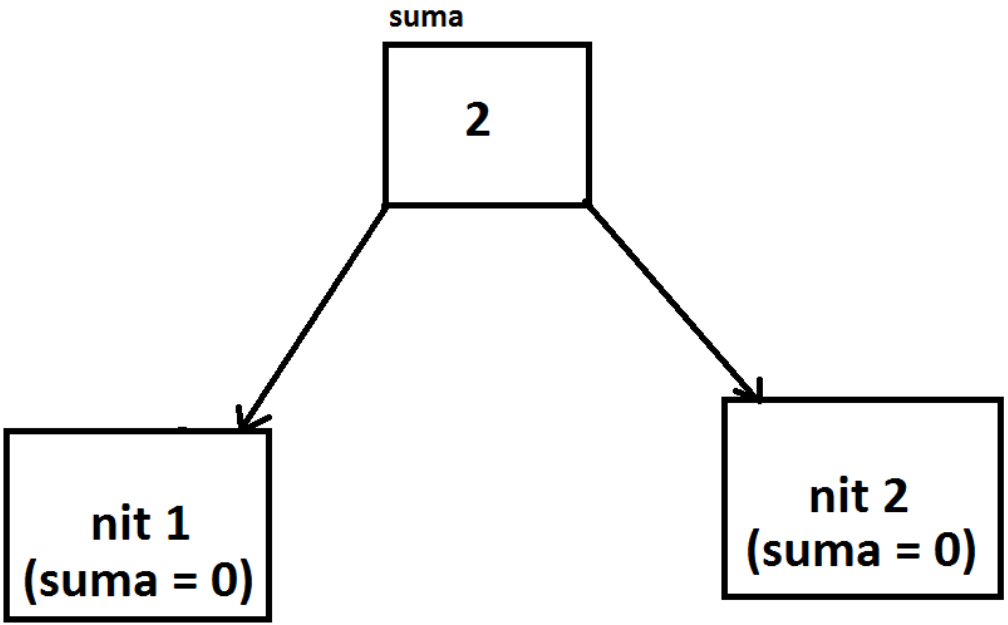
Uvod

- 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

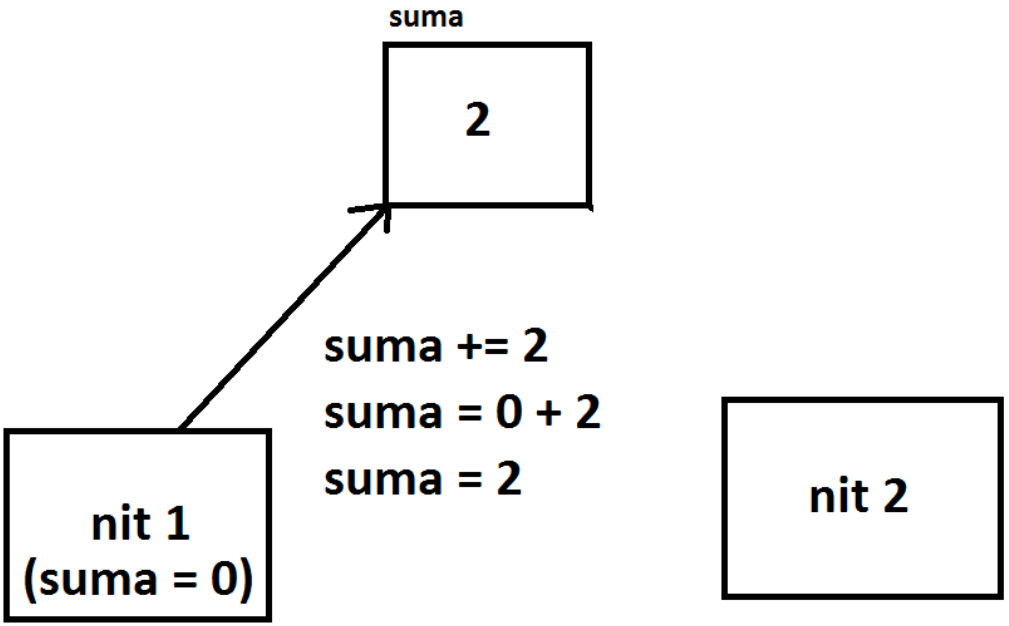
Uvod

```
public class KonkurentniProgram {
    // Deljena promenljiva
    public static Integer suma = 0;

    public static void main(String [] args){
        int niz[] = {2, 4, 7, 5, 9};
        for (int i=0; i< niz.length; i++){
            SumaNit nit = new SumaNit(niz);
            nit.start();
        }
    }
}

public class SumaNit extends Thread {

    public int broj;

    // Ako bismo sabrali ovde, dobili bismo sekvencijalni program
    // Sacuvacemo u polju klase da bismo mogli da pristupimo u run
    public KonkurentniProgram(int broj){
        this.broj = broj;
    }

    public void run(){
        KonkurentniProgram.suma += broj;
    }
}
```

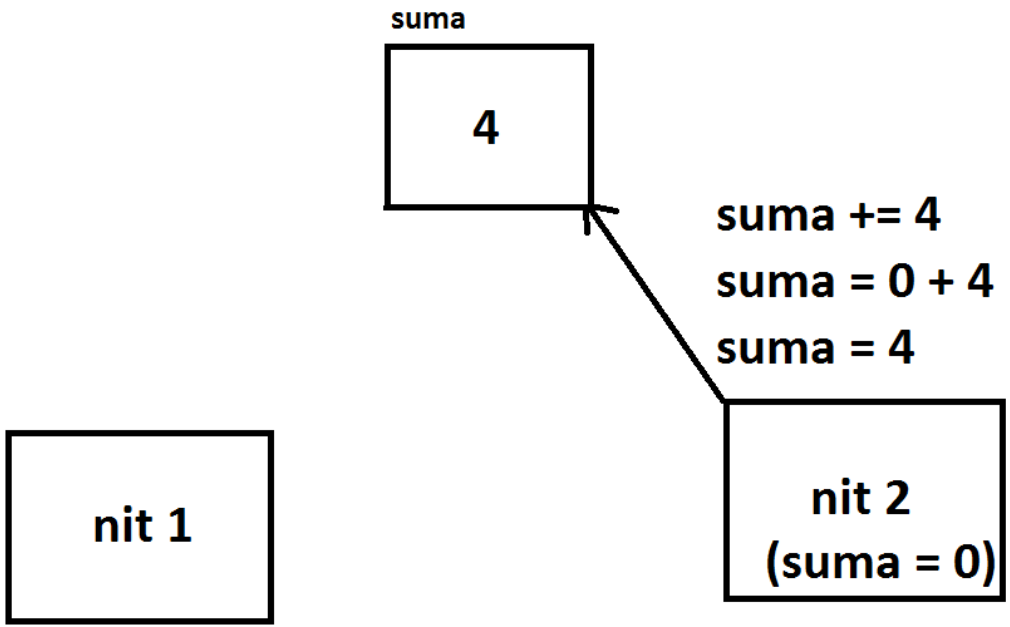
Obe niti zatiču sumu 0



Nit 1 dodaje na zatečenu sumu 2



Nit 2 dodaje na zatečenu sumu 4



Uvod

- Nit 1 i nit 2 istovremeno pristupaju sumi i čitaju broj 0
- Nit 1 dodaje 2 na sumu; po njoj je suma 2 to se čuva
- Nit 2 dodaje 4 na sumu; po njoj je suma 4 i to se čuva
- Nit 2 je pregazila ono što je nit 1 uradila: umesto 6 imamo 4

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 nekoj promenljivoj
- Potrebno je proslediti referencu na tu promenljivu prilikom pisanja synchronized bloka
- Mora biti objekat, tako da ce se umesto int koristiti Integer, umesto double koristiti Double i sl.
- Primer:

```
synchronized(suma) {
    // Sekvencijalni pristup promenljivoj suma
    this.suma += broj;
}
```

Sinhronizacija uz pomoć synchronized bloka

```
public class KonkurentniProgram {
    // Deljena promenljiva
    public static Integer suma = 0;

    public static void main(String [] args){
        int niz[] = {2, 4, 7, 5, 9};
        for (int i=0; i< niz.length; i++){
            SumaNit nit = new SumaNit(niz[i]);
            nit.start();
        }
    }
}

public class SumaNit extends Thread {

    private int broj;

    public KonkurentniProgram(int broj){
        this.broj = broj;
    }

    public void run(){
        // Sinhronizovan pristup
        synchronized(suma){
            KonkurentniProgram.suma += broj;
     }

    }
}
```

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 void method() {
public synchronized void method() {
    // Sekvencijalni pristup objektu ove
       klase (this)
    this.suma += broj;
}
```

Uvod

```
public class KonkurentniProgram {

    public static void main(String [] args){
        int niz[] = {2, 4, 7, 5, 9};
        // Deljena promenljiva sada u main da bi je prosledili niti
        int suma = 0;
        for (int i=0; i< niz.length; i++){
            SumaNit nit = new SumaNit(niz[i], suma);
            nit.start();
        }
    }
}

public class SumaNit extends Thread {

    private int broj;
    // Suma mora biti ovde da bi je zastitila synchronized metoda
    private int suma;

    public KonkurentniProgram(int broj, int suma){
        this.broj = broj;
        this.suma = suma;
    }

    public void synchronized run(){
        this.suma += this.broj;
    }
}
```

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(suma){
    // Jedini bitan deo za sinhronizaciju
    // Pristupanje i pisanje u promenljivu
    suma+=proizvod;
}
```

volatile polje

- Još jedan način sinhronizacije niti je upotrebom volatile polja
- Sve niti koje pokušavaju da čitaju volatile polje će uvek dobiti njenu poslednju vrednost, čak iako se ona promeni u međuvremenu
- `public static volatile int suma = 0;`

Primer sa volatile poljem

```
public class KonkurentniProgram {
// suma je sada volatile
    public static volatile int suma = 0;
    public static void main(String [] args){
        int niz[] = {2, 4, 7, 5, 9};
        for (int i=0; i< niz.length; i++){
            SumaNit nit = new SumaNit(niz[i], suma);
            nit.start();
        }
    }
}

public class SumaNit extends Thread {

    private int broj;

    public KonkurentniProgram(int broj, int suma){
        this.broj = broj;
        this.suma = suma;
    }

    public void run(){
        this.suma += this.broj;
    }
}
```

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