Laboratorijska vježba 10.

Korištenje principa višenitnosti

10.1. Višenitno izvršavanje i spajanje tokova

Višenitno izvršavanje neophodno je kako bi se paralelno moglo izvršavati više operacija i na taj način skratiti ukupno vrijeme izvršavanja programskih rješenja, ali i u druge svrhe (npr. simultano pisanje u bazu podataka dok se korisniku omogućava vršenje drugih operacija). U okviru ove laboratorijske vježbe za kreiranje višenitnih programskih rješenja neće se koristiti desktop JavaFX aplikacije, već isključivo Java konzolne aplikacije, s obzirom da je prikaz u konzoli dovoljan za prikaz svih principa višenitnosti koji će biti korišteni.

Za demonstriranje osnovnih mogućnosti višenitnosti, biti će iskorišten jednostavni primjer. Neka je definisana lista cjelobrojnih vrijednosti koja sadrži veoma veliki broj elemenata (reda 10^7, što je odabrano uzimajući u obzir količinu RAM memorije koja je dostupna na većini računara, a u slučaju veće memorije mogu se koristiti i veće kolekcije). U Listingu 1 prikazan je način na koji se može klasično izvršiti sabiranje svih brojeva unutar kolekcije kako bi se dobila konačna suma. Sama inicijalizacija liste traje nekoliko sekundi, a isto toliko traje i prolazak kroz petlju kako bi se izvršilo sabiranje svih elemenata liste. Na kraju se vrši ispis sume svih elemenata u konzoli, na način prikazan na Slici 1.

```
public class Main {
   public static void main(String[] args) {
        List<Integer> brojevi = new ArrayList<Integer>();
        for (int i = 0; i < 10000000; i++)
            brojevi.add(5);
        Integer suma = 0;
        for (int i = 0; i < brojevi.size(); i++)
            suma += brojevi.get(i);
        System.out.println("Suma svih brojeva u kolekciji je " +
suma);
    }
}
```

Listing 1. Sumiranje veoma velike kolekcije brojeva

```
Suma svih brojeva u kolekciji je 50000000

Process finished with exit code 0
```

Slika 1. Prikaz sume svih elemenata u kolekciji brojeva

Neka se sada želi ubrzati ovaj proces sabiranja, tako što će se vršiti paralelno sabiranje različitih dijelova liste od strane više niti. U tu svrhu može se kreirati nova klasa koja će naslijediti klasu **Thread** i implementirati njenu metodu **run()**. U okviru ove metode definiše se ono što se želi uraditi u okviru date niti. Listing 2 prikazuje klasu *SabiranjeNit* koja sadrži informacije o početnom i krajnjem indeksu kolekcije koji želi koristiti, a zatim u metodi *run()* vrši iterativni prolazak kroz elemente kolekcije u svrhu formiranja parcijalne sume, koja se na kraju ispisuje na ekranu.

```
public class SabiranjeNit extends Thread
{
    Integer pocetak, kraj, suma;
    public SabiranjeNit (Integer pocetak, Integer kraj)
    {
        this.pocetak = pocetak;
        this.kraj = kraj;
        suma = 0;
    }
    public void run ()
    {
        for (int i = pocetak; i < kraj; i++)
            suma += Main.brojevi.get(i);

        System.out.println("Parcijalna suma za nit [" + pocetak +
"," + kraj + ") iznosi " + suma);
    }
}
```

Listing 2. Klasa SabiranjeNit koja definiše operaciju koju će izvršavati više niti

U Listingu 2, u okviru metode *run()* pristupa se kolekciji brojeva koristeći varijablu **Main.brojevi**, što indicira da je u okviru klase *Main* definisana javna statička varijabla (umjesto u funkciji *main()*, kao što je urađeno u Listingu 1). Ova izmjena bila je neophodna kako bi niti, koje su definisane u zasebnoj klasi, mogle pristupati kolekciji koja se nalazi izvan te klase (naravno, umjesto javne varijable mogla se koristiti i *get()* metoda, međutim isto nije urađeno radi pojednostavljenja programskog koda). Listing 3 prikazuje izmijenjenu *Main* klasu, u okviru koje se sada nalazi inicijalizacija osam niti klase *SabiranjeNit* nakon određivanja odgovarajućih indeksa u kolekciji za pretragu, kao i poziv metode **run()** svih niti.

```
public class Main {
    public static List<Integer> brojevi;
    public static void main(String[] args) {
```

```
        brojevi = new ArrayList<Integer>();
        for (int i = 0; i < 10000000; i++)
            brojevi.add(5);

        for (int i = 0; i < 8; i++)
        {
            Integer pocetak = i * brojevi.size() / 8,
                    kraj = (i + 1) * brojevi.size() / 8;
            SabiranjeNit nit = new SabiranjeNit(pocetak, kraj);
            nit.start();
        }
    }
}
```

Listing 3. Računanje sume kolekcije brojeva koristeći više niti

Na Slici 2 prikazani su rezultati nakon pokretanja ovako definisanog programskog koda, odakle se vidi da parcijalne sume odgovaraju očekivanim vrijednostima, kao i indeksi kolekcije za koje se vrši sabiranje u pojedinačnim nitima. Osim toga, može se vidjeti da izvršavanje niti nije završeno onim redoslijedom kojim su niti pokrenute, s obzirom da su se niti borile za procesorsko vrijeme i simultano su bile pokrenute.

```
Parcijalna suma za nit [2500000,3750000) iznosi 6250000
Parcijalna suma za nit [5000000,6250000) iznosi 6250000
Parcijalna suma za nit [0,1250000) iznosi 6250000
Parcijalna suma za nit [6250000,7500000) iznosi 6250000
Parcijalna suma za nit [7500000,8750000) iznosi 6250000
Parcijalna suma za nit [1250000,2500000) iznosi 6250000
Parcijalna suma za nit [8750000,10000000) iznosi 6250000
Parcijalna suma za nit [3750000,5000000) iznosi 6250000

Process finished with exit code 0
```

Slika 2. Ispisivanje parcijalnih suma koristeći više niti

S obzirom da su parcijalne sume definisane kao atributi klase *SabiranjeNit*, iste je moguće iskoristiti kako bi se izvršilo ispisivanje samo konačne sume nakon završetka izvršavanja niti. U Listingu 4 prikazan je način na koji je to urađeno, koristeći kolekciju niti i metodu *getSuma()* za svaku pojedinačnu nit. Međutim, rezultati prikazani na Slici 3 pokazuju da ova implementacija nije ispravna, jer se konačna suma ispisuje na samom početku, prije prikaza rezultujućih parcijalnih suma za svaku pojedinačnu nit. Osim toga, dobivena vrijednost (suma = 33165) ne odgovara očekivanoj, i njena mala vrijednost indicira da je *getSuma()* funkcija pozvana nad nitima koje su tek započele s izvršavanjem.

```
public static void main(String[] args) {
    brojevi = new ArrayList<Integer>();
    for (int i = 0; i < 10000000; i++)
        brojevi.add(5);
    List<SabiranjeNit> niti = new ArrayList<SabiranjeNit>();
    for (int i = 0; i < 8; i++) {
        Integer pocetak = i * brojevi.size() / 8,
                kraj = (i + 1) * brojevi.size() / 8;
        SabiranjeNit nit = new SabiranjeNit(pocetak, kraj);
        niti.add(nit);
        nit.start();
    }
    Integer suma = 0;
    for (SabiranjeNit nit : niti)
        suma += nit.getSuma();
    System.out.println("Konačna suma je " + suma);
}
```

Listing 4. Ispisivanje konačne sume sabiranjem parcijalnih suma pojedinačnih niti

```
Konačna suma je 33165
Parcijalna suma za nit [8750000,10000000) iznosi 6250000
Parcijalna suma za nit [5000000,6250000) iznosi 6250000
Parcijalna suma za nit [7500000,8750000) iznosi 6250000
Parcijalna suma za nit [1250000,2500000) iznosi 6250000
Parcijalna suma za nit [2500000,3750000) iznosi 6250000
Parcijalna suma za nit [3750000,5000000) iznosi 6250000
Parcijalna suma za nit [0,1250000) iznosi 6250000
Parcijalna suma za nit [6250000,7500000) iznosi 6250000

Process finished with exit code 0
```

Slika 3. Prikaz pogrešne konačne sume na početku rada programa

Kako bi se riješio ovaj problem, potrebno je sačekati da sve niti završe s izvršavanjem, a što se postiže korištenjem funkcije **join()**, koja indicira da je potrebno spojiti tok *main()* funkcije i niti koja se samostalno izvršava i nastaviti sa izvršavanjem daljih dijelova *main()* funkcije tek nakon završetka izvršavanja niti koja je pokrenuta. Listing 5 prikazuje mjesto na kome se poziva **join()** funkcija - u okviru petlje za svaku pojedinačnu nit, nakon njenog pokretanja, a prije formiranja konačne sume. S obzirom da se tokom rada niti može pojaviti izuzetak, okruženje zahtijeva da se taj izuzetak obradi, što je u programskom kodu i urađeno.

```
public static void main(String[] args) {
    brojevi = new ArrayList<Integer>();
    for (int i = 0; i < 10000000; i++)
        brojevi.add(5);
```

```
    List<SabiranjeNit> niti = new ArrayList<SabiranjeNit>();
    try {
        for (int i = 0; i < 8; i++) {
            Integer pocetak = i * brojevi.size() / 8,
                    kraj = (i + 1) * brojevi.size() / 8;
            SabiranjeNit nit = new SabiranjeNit(pocetak, kraj);
            niti.add(nit);
            nit.start();
        }
        Integer suma = 0;
        for (SabiranjeNit nit : niti) {
            nit.join();
            suma += nit.getSuma();
        }
        System.out.println("Konačna suma je " + suma);
    }
    catch (InterruptedException e) {
        System.out.println(e.getMessage());
    }
}
```

Listing 5. Spajanje tokova prije ispisivanja konačne sume

Osim navedenog, može biti korisno i ispisati stanje niti odmah nakon njihovog pokretanja i na samom kraju nakon formiranja parcijalne sume. Isto je urađeno na jednostavan način prikazan u Listingu 6. Za ovako definisan programski kod dobivaju se rezultati prikazani na Slici 4, odakle je vidljivo da je dobiven željeni rezultat - prvo su pokrenute sve niti, zatim su sve niti završile s radom, i na kraju se ispisala ukupna suma, čije vrijednost sada odgovara očekivanoj. Važno je napomenuti da se ovakvi rezultati ne bi dobili da se funkcija *join()* pozvala na dnu prve petlje, jer bi se u tom slučaju zaustavilo izvršavanje *main()* funkcije i pokretanje svih sljedećih niti dok prethodna nit ne bi završila s izvršavanjem, čime bi se ostvarilo sekvencijalno, a ne paralelno izvršavanje niti.

```
public void run ()
{
    System.out.println("Nit [" + pocetak + "," + kraj + ") započinje
s radom");
    for (int i = pocetak; i < kraj; i++)
        suma += Main.brojevi.get(i);
    System.out.println("Nit [" + pocetak + "," + kraj + ") završava
s radom");
}
```

Listing 6. Ispis informacija o izvršavanju svake pojedinačne niti

```
Nit [0,1250000) započinje s radom
Nit [5000000,6250000) započinje s radom
Nit [1250000,2500000) započinje s radom
Nit [2500000,3750000) započinje s radom
Nit [3750000,5000000) započinje s radom
Nit [6250000,7500000) započinje s radom
Nit [8750000,10000000) započinje s radom
Nit [5000000,6250000) završava s radom
Nit [8750000,10000000) završava s radom
Nit [1250000,2500000) završava s radom
Nit [7500000,8750000) započinje s radom
Nit [3750000,5000000) završava s radom
Nit [6250000,7500000) završava s radom
Nit [7500000,8750000) završava s radom
Nit [0,1250000) završava s radom
Nit [2500000,3750000) završava s radom
Konačna suma je 50000000

Process finished with exit code 0
```

Slika 4. Ispis tačne konačne sume nakon završetka rada svih niti

10.2. Rješavanje konflikta korištenjem mehanizama zaključavanja

U prethodno prikazanom primjeru izbjegnuti su konflikti koji mogu nastati kada više niti koristi iste varijable putem specifikacije indeksa pretraživanja, koji su različiti za sve niti. Međutim, ponekad to nije moguće izbjeći i u tom slučaju potrebno je koristiti mehanizme za sprječavanje konflikta među nitima. U tu svrhu biti će upotrijebljen primjer u kojem se koristi brojačka varijabla koja se mijenja u nitima. U prethodnom dijelu laboratorijske vježbe korišten je princip nasljeđivanja klase *Thread* za definisanja operacije koja se želi koristiti, međutim moguće je koristiti i lambda funkciju kako bi se taj proces pojednostavio. U Listingu 7 prikazana je osnovna struktura *Main* klase koja omogućava korištenje i izmjenu vrijednosti statičke varijable *brojac* u okviru 1000 iteracija, koristeći lambda funkciju i osnovnu klasu *Thread*. S obzirom da se nakon pokretanja niti ne vrše nikakve akcije u *main()* funkciji, nema potrebe za čekanjem na završetak njihovog izvršavanja putem *join()* funkcije.

```
public class Main {
    public static Integer brojac = 0;
    public static void main(String[] args) {
        for (int i = 0; i < 8; i++) {
            Thread nit = new Thread(() ->
            {
                while (brojac < 1000) {
                    brojac++;
                    System.out.print(brojac + " ");
                }
            });
            nit.start();
        }
    }
}
```

Listing 7. Korištenje lambda funkcije za uvećavanje vrijednosti brojača

Rezultati nakon pokretanja nakon ovako definisanog programskog koda prikazani su na Slici 5. Ne dolazi do pojave izuzetka zbog pristupa dijeljenoj memoriji, međutim rezultati se ispisuju na veoma čudan i neočekivan način. Bez obzira na činjenicu da je pokrenuto 8 niti, očekuje se da koja god nit da izvrši liniju koda kojom se u konzoli prikazuje trenutna vrijednost varijable *brojac*, ona će ispisati vrijednost varijable sekvencijalno (1, 2, 3, 4,...). Druga očekivana mogućnost je da određeni broj niti prvo izvrši inkrementaciju, pa zatim jedna nit izvrši ispis, čime bi došlo do preskakanja određenog broja vrijednosti prije narednog ispisivanja (npr. 1, 5, 6, 9, …). Međutim, rezultati pokazuju da su sve vrijednosti brojača ispisane u konzoli (dakle, i 1, i 2, i 3, i 4, i sve ostale), ali ne sekvencijalnim redoslijedom (npr. broj 1 se ispisao 7. po redu). Nakon opsežne analize, može se otkriti da je krivac za pojavu ovakvog ponašanja korištenje konzole za ispis, koji implicitno koristi zaključavanje varijabli i preuzima trenutnu vrijednost brojača, a zatim je sporo ispisuje. Zbog toga se ranije vrijednosti brojača kasnije ispisuju - njihovo ispisivanje je započelo kasnije, ali koristeći vrijednost koja je dobivena odmah nakon inkrementiranja brojača u okviru procesa koji zaključava datu vrijednost.

```
3 5 6 7 8 9 1 11 13 14 15 16 18 19 20 21 4 23 6 10 27 28 12 17 2 32 33 22 35 36 37 38 39 40 41 42 43 44 45 46 47
Process finished with exit code 0
```

Slika 5. Ispis vrijednosti brojača bez ikakve hronologije

Kako bi se uklonio utjecaj odgođenog ispisivanja u konzoli na tumačenje rezultata, može se koristiti kolekcija za zapisivanje rezultata bez potrebe za čekanjima niti implicitnog korištenja zaključavanja, kao što je prikazano u Listingu 8. S obzirom da se sadržaj kolekcije vrijednosti brojača sada ispisuje nakon što sve niti završe s operacijom povećanja vrijednosti brojača, potrebno je iskoristiti *join()* funkciju. Osim toga, za povećavanje utjecaja više niti na rezultate, ukupan broj niti je povećan na 32, a broj iteracija za svaku nit na 1000. Na ovaj način očekuje se da krajnja vrijednost brojača bude *brojac = 32000*.

```
public static void main(String[] args) {
    List<Integer> brojevi = new ArrayList<Integer>();
    List<Thread> niti = new ArrayList<Thread>();
    for (int i = 0; i < 32; i++) {
        Thread nit = new Thread(() ->
        {
            for (int j = 0; j < 1000; j++) {
                brojac++;
                brojevi.add(brojac);
            }
        });
        nit.start();
        niti.add(nit);
    }
    try {
        for (Thread nit : niti)
            nit.join();
        System.out.print(brojevi);
```

```
    }
    catch (Exception e)
    {
        System.out.println(e.getMessage());
    }
}
```

Listing 8. Korištenje kolekcije za zapis vrijednosti brojača

Rezultati izvršavanja sadrže veoma veliki broj elemenata, a posljednji elementi u ispisanoj kolekciji prikazani su na Slici 6, odakle je vidljivo da nije dostignuta vrijednost od 32000, već 20953. Pri ponovljenom izvršavanju ovaj broj se mijenja, što ukazuje da postoji nestabilnost pri izvršavanju. Međutim, kolekcija zaista sadrži 32000 elemenata, ali neki od njih sadrže ponavljanja. Na Slici 7 prikazani su neki od primjera anomalija koje se mogu pronaći u nizu, poput višestrukog ponavljanja istih elemenata, dekrementacije elemenata, kao i pojave *null* vrijednosti. Sve ovo je posljedica simultanog pristupanja istoj memorijskoj lokaciji, pokušaja pristupa lokaciji od strane jedne niti dok druga nit piše u memoriju i drugih problema koji nastaju zbog nedostatka mehanizama za osiguravanje da neće doći do ovakvih pojava. Ukoliko se izvršavanje ponovi dovoljan broj puta, može doći i do pojave izuzetaka zbog pristupanja elementima niza koji su zaključani i sl.

```
20803, 20804, 20805, 20806, 20807, 20808, 20809, 20810, 20811, 20812, 20813, 20814, 20815, 20816, 20817, 20818, 20819, 20820, 20821, 20822, 20823,
20824, 20825, 20826, 20827, 20828, 20829, 20830, 20831, 20832, 20833, 20834, 20835, 20836, 20837, 20838, 20839, 20840, 20841, 20842, 20843, 20844,
20845, 20846, 20847, 20848, 20849, 20850, 20851, 20852, 20853, 20854, 20855, 20856, 20857, 20858, 20859, 20860, 20861, 20862, 20863, 20864, 20865,
20866, 20867, 20868, 20869, 20870, 20871, 20872, 20873, 20874, 20875, 20876, 20877, 20878, 20879, 20880, 20881, 20882, 20883, 20884, 20885, 20886,
20887, 20888, 20889, 20890, 20891, 20892, 20893, 20894, 20895, 20896, 20897, 20898, 20899, 20900, 20901, 20902, 20903, 20904, 20905, 20906, 20907,
20908, 20909, 20910, 20911, 20912, 20913, 20914, 20915, 20916, 20917, 20918, 20919, 20920, 20921, 20922, 20923, 20924, 20925, 20926, 20927, 20928,
20929, 20930, 20931, 20932, 20933, 20934, 20935, 20936, 20937, 20938, 20939, 20940, 20941, 20942, 20943, 20944, 20945, 20946, 20947, 20948, 20949,
20950, 20951, 20952, 20953]
```

Slika 6. Krajnji elementi kolekcije vrijednosti brojača

```
5175, 5176, 5177, 5177, 5178, 5178, 5179, 5179, 5180, 5180, 5181, 5181, 5182, 5183, 5184, 5184, 5185, 5186, 5186, 5187, 5187, 5188, 5188, 5189, 5189,
5190, 5191, 5192, 5191, 5192, 5193, 5194, 5194, 5195, 5195, 5196, 5196, 5198, 5199, 5200, 5201, 5201, 5204, 5205, 5206, 5206, 5207, 5208, 5209, 5209,
5210, 5210, 5211, 5211, 5212, 5213, 5214, 5214, 5215, 5215, 5216, 5216, 5217, 5217, 5217, 5219, 5219, 5220, 5221, 5222, 5222, 5223, 5223, 5224, 5224,
5225, 5226, 5226, 5227, 5228, 5228, 5229, 5229, 5230, 5231, 5231, 5232, 5233, 5233, 5234, 5234, 5235, 5235, 5236, 5236, 5237, 5238, 5239, 5240, 5241,
5242, 5242, 5243, 5243, 5244, 5244, 5245, 5245, 5246, 5247, 5247, 5248, 5248, 5249, 5250, 5250, 5251, 5251, 5252, 5252, 5253, 5253, 5254, 5254, 5255,
5255, 5256, 5257, 5258, 5258, 5259, 5260, 5260, 5261, 5261, 5262, 5263, 5263, 5264, 5264, 5265, 5266, 5266, 5267, 5267, 5268, 5268, 5269, 5269, 5270,
5270, 5271, 5271, 5272, 5272, 5273, 5273, 5274, 5275, 5275, 5276, 5276, 5277, null, null, null, null, null, null, null, null, null, null, null, null,
5290, 5291, 5292, 5291, 5293, 5294, 5294, 5295, 5296, 5296, 5297, 5298, 5298, 5299, 5299, 5300, 5300, 5301, 5301, 5302, 5302, 5303, 5303, 5304, 5304,
5305, 5305, 5306, 5306, 5307, 5307, 5308, 5308, 5309, 5309, 5310, 5310, 5311, 5312, 5313, 5314, 5314, 5315, 5315, 5316, 5316, 5317, 5317, 5318, 5319,
5320, 5321, 5322, 5323, 5323, 5324, 5324, 5325, 5325, 5326, 5327, 5328, 5328, 5329, 5330, 5330, 5331, 5331, 5332, 5332, 5333, 5333, 5334, 5334, 5335,
5335, 5336, 5336, 5337, 5338, 5338, 5339, 5340, 5341, 5342, 5342, 5343, 5343, 5344, 5344, 5345, 5345, 5346, 5346, 5347, 5347, 5348, 5348, 5349, 5349,
5350, 5350, 5351, 5351, 5352, 5352, 5353, 5353, 5354, 5354, 5355, 5355, 5356, 5356, 5357, 5357, 5358, 5358, 5359, 5359, 5360, 5360, 5361, 5361, 5362,
```

Slika 7. Anomalije u elementima kolekcije vrijednosti brojača

Kako bi se spriječila pojava prethodno opisanih anomalija, neophodno je koristiti zaključavanje (*lock*) varijabli koje koriste date niti. Ovu operaciju vrše same niti, onemogućavajući drugim nitima da pristupaju varijablama koje se koriste u datoj iteraciji, što dovodi do pojave čekanja niti koja nije ostvarila pristup. Time se smanjuje ubrzanje koje se postiže korištenjem višenitnosti, ali se povećava sigurnost memorije i konzistentnost dobivenih rezultata. Da bi se to postiglo, potrebno je definisati novu klasu za niti *BrojacNit*, koja je prikazana u Listingu 9. Ova klasa ne sadrži atribute niti konstruktor, već samo *run()* metodu koja ima isti programski kod kao prethodno definisana lambda funkcija za niti iz *main()* funkcije.

Jedina razlika je u stereotipu metode **synchronized** (vidjeti sljedeći link), koji onemogućava vršenje izmjena nad objektima koji se koriste u okviru metode od strane drugih niti, dok se oni koriste. *Main* klasa sada poprima oblik prikazan u Listingu 10, pri čemu je bilo neophodno kolekciju vrijednosti brojača *brojevi* deklarisati kao statičku varijablu zbog pristupa u nitima, a kod *main()* metode je značajno pojednostavljen bez lambda funkcije.

```
public class BrojacNit extends Thread
{
   public synchronized void run () {
        for (int j = 0; j < 1000; j++) {
            Main.brojac++;
            Main.brojevi.add(Main.brojac);
        }
   }
}
```

Listing 9. Kreiranje nove vrste niti za rad s brojačem koristeći mehanizam zaključavanja

```
public class Main {
   public static Integer brojac = 0;
   public static List<Integer> brojevi = new ArrayList<Integer>();
   public static void main(String[] args) {
        List<Thread> niti = new ArrayList<Thread>();
        for (int i = 0; i < 32; i++) {
            niti.add(new BrojacNit());
            niti.get(i).start();
        }
        try {
            for (Thread nit : niti)
                nit.join();
            System.out.print(brojevi);
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
   }
}
```

Listing 10. Korištenje nove vrste niti za rad s brojačem

Dobiveni rezultati prikazani su na Slici 8. Vidljivo je da ponovo nije dobiven niz sa jedinstvenim elementima, odnosno da postoje ponavljanja. Međutim, više nema *null* vrijednosti jer je korištenje sinhronizacije tj. mehanizma zaključavanja onemogućilo pokušaj pisanja u memoriju dok je zauzeta od strane druge niti. S druge strane, nije onemogućen pokušaj čitanja vrijednosti brojača od strane više niti, što je dovelo do višestrukog upisivanja istih vrijednosti u kolekciju. Na ovaj način je samo djelimično riješen problem pristupa dijeljenim varijablama u okviru višenitnih programskih rješenja.

```
23205, 23206, 23207, 23208, 23209, 23210, 23211, 23212, 23213, 23214, 23215, 23216, 23217, 23218, 23219, 23220, 23221, 23222, 23223, 23224, 23225,
23226, 23227, 23228, 23229, 23230, 23231, 23232, 23233, 23234, 23235, 23236, 23237, 23238, 23239, 23240, 23241, 23242, 23243, 23244, 23245, 23246,
23247, 23248, 23249, 23250, 23251, 23252, 23253, 23254, 23255, 23256, 23257, 23258, 23259, 23260, 23261, 23262, 23263, 23264, 23265, 23266, 23267,
23268, 23269, 23270, 23271, 23272, 23273, 23274, 23275, 23276, 23277, 23278, 23279, 23280, 23281, 23282, 23283, 23284, 23285, 23286, 23287, 23288,
23289, 23290, 23291, 23292, 23293, 23294, 23295, 23296, 23297, 23298, 23299, 23300, 23301, 23302, 23303, 23304, 23305, 23306, 23307, 23308, 23309,
23310, 23311, 23312, 23313, 23314, 23315, 23316, 23317, 23318, 23319, 23320, 23321, 23322, 23323, 23324, 23325, 23326, 23327, 23328, 23329, 23330,
23331, 23332, 23333, 23334, 23335, 23336, 23337, 23338, 23339, 23340, 23341, 23342, 23343, 23344, 23345, 23346, 23347, 23348, 23349, 23350, 23351,
23352, 23353, 23354, 23355, 23356, 23357, 23358, 23359, 23360, 23361, 23362, 23363, 23364, 23365, 23366, 23367, 23368, 23369, 23370, 23371, 23372,
23373, 23374, 23375, 23376, 23377, 23378, 23379, 23380, 23381, 23382, 23383, 23384, 23385, 23386, 23387, 23388, 23389, 23390, 23391, 23392, 23393,
23394, 23395, 23396, 23397, 23398, 23399, 23400, 23401, 23402, 23403, 23404, 23405, 23406, 23407, 23408, 23409, 23410, 23411, 23412, 23413, 23414,
23415, 23416, 23417, 23418, 23419, 23420, 23421, 23422, 23423, 23424, 23425, 23426, 23427, 23428, 23429, 23430, 23431, 23432, 23433, 23434, 23435,
23436, 23437, 23438, 23439, 23440, 23441, 23442, 23443, 23444, 23445, 23446, 23447, 23448, 23449, 23450, 23451, 23452, 23453, 23454, 23455, 23456,
23457, 23458, 23459, 23460]
```

Slika 8. Novi prikaz krajnjih elemenata kolekcije vrijednosti brojača

10.3. Korištenje atomskih varijabli

Prethodno opisani načini za sinhronizaciju niti i sprječavanje paralelnog pisanja u memoriju samo djelimično mogu riješiti probleme koji nastaju pri neophodnosti pristupanja dijeljenim resursima. Najbolji način za sprječavanje pojave ovakvih problema je korištenjem **atomskih varijabli** (za više informacija pogledati sljedeći link). Atomskim varijablama moguće je proglasiti cjelobrojne, *boolean* vrijednosti i reference na objekte. U prethodnom primjeru, dijeljeni resurs je brojač koji je cjelobrojnog tipa, pa je potrebno definisati ga kao atomsku varijablu, čime će se onemogućiti da više niti istovremeno pristupa varijabli, bez da je nit koja je prva započela operaciju istu i završila (zbog čega se ove varijable i nazivaju atomskim - čitanje memorije, izmjena vrijednosti varijable i pisanje u memoriju se posmatraju kao jedinstvena operacija ili *atom*). U Listingu 11 nalazi se izmijenjena definicija *Main* klase, gdje je vidljivo da je umjesto *Integer* tipa, varijabla *brojac* sada **AtomicInteger** tipa, a za njenu inicijalizaciju u *main()* funkciji koristi se metoda **set(inicijalnaVrijednost)**. Sav ostali kod *Main* klase nije promijenjen.

```
public class Main {
    public static AtomicInteger brojac = new AtomicInteger();
    public static List<Integer> brojevi = new ArrayList<Integer>();
    public static void main(String[] args) {
        brojac.set(0);
        List<Thread> niti = new ArrayList<Thread>();
        for (int i = 0; i < 32; i++) {
            niti.add(new BrojacNit());
            niti.get(i).start();
        }
        try {
            for (Thread nit : niti)
                nit.join();
            System.out.print(brojevi);
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Listing 11. Korištenje atomske cjelobrojne varijable kao dijeljenog resursa

U Listingu 12 nalazi se izmijenjena definicija klase *BrojacNit* koja koristi varijablu *brojac*. Ovu varijablu više nije moguće inkrementirati koristeći operaciju **++**, već koristeći metodu **incrementAndGet()**, koja kao rezultat vraća povećanu vrijednost brojača i koja se, kao takva, može odmah upotrijebiti za smještanje dobivene vrijednosti u kolekciju vrijednosti brojača. Nakon pokretanja ovako definisanog programskog koda, dobivaju se rezultati prikazani na Slici 9, koji napokon odgovaraju onome što je očekivano - tačan broj vrijednosti brojača (32 niti sa po 1000 iteracija), bez ponavljajućih, dekrementiranih ili *null* vrijednosti. Na ovaj način riješen je problem sinhronizacije, korištenja dijeljenih resursa i mogućnosti krahiranja programskog rješenja zbog svih prethodno opisanih problema.

```
public class BrojacNit extends Thread
{
    public synchronized void run () {
        for (int j = 0; j < 1000; j++) {
            Main.brojevi.add(Main.brojac.incrementAndGet());
        }
    }
}
```

Listing 12. Korištenje atomske varijable u okviru niti

```
31832, 31833, 31834, 31835, 31836, 31837, 31838, 31839, 31840, 31841, 31842, 31843, 31844, 31845, 31846, 31847, 31848, 31849, 31850, 31851, 31852,
31853, 31854, 31855, 31856, 31857, 31858, 31859, 31860, 31861, 31862, 31863, 31864, 31865, 31866, 31867, 31868, 31869, 31870, 31871, 31872, 31873,
31874, 31875, 31876, 31877, 31878, 31879, 31880, 31881, 31882, 31883, 31884, 31885, 31886, 31887, 31888, 31889, 31890, 31891, 31892, 31893, 31894,
31895, 31896, 31897, 31898, 31899, 31900, 31901, 31902, 31903, 31904, 31905, 31906, 31907, 31908, 31909, 31910, 31911, 31912, 31913, 31914, 31915,
31916, 31917, 31918, 31919, 31920, 31921, 31922, 31923, 31924, 31925, 31926, 31927, 31928, 31929, 31930, 31931, 31932, 31933, 31934, 31935, 31936,
31937, 31938, 31939, 31940, 31941, 31942, 31943, 31944, 31945, 31946, 31947, 31948, 31949, 31950, 31951, 31952, 31953, 31954, 31955, 31956, 31957,
31958, 31959, 31960, 31961, 31962, 31963, 31964, 31965, 31966, 31967, 31968, 31969, 31970, 31971, 31972, 31973, 31974, 31975, 31976, 31977, 31978,
31979, 31980, 31981, 31982, 31983, 31984, 31985, 31986, 31987, 31988, 31989, 31990, 31991, 31992, 31993, 31994, 31995, 31996, 31997, 31998, 31999,
32000]
Process finished with exit code 0
```

Slika 9. Novi prikaz krajnjih elemenata kolekcije vrijednosti brojača sa ispravnim brojem elemenata

10.4. Zadaci za samostalni rad

Za sticanje bodova na prisustvo laboratorijskoj vježbi, potrebno je uraditi sljedeće zadatke tokom laboratorijske vježbe i postaviti ih u repozitorij studenta na odgovarajući način:

Zadatak 1.

Definisati programsko rješenje koje omogućava paralelno pretraživanje kolekcije brojeva. Pri definisanju kolekcije koristiti nasumično generisane elemente, a za pretragu odrediti nasumični broj u kolekciji, pri čemu kolekcija treba imati 10^8 elemenata. Za pretraživanje koristiti 16 niti i prikazati rezultat čim prva nit pronađe dati broj. Ne koristiti principe sinhronizacije, zaključavanja niti atomske varijable, već dodijeliti nitima različite dijelove kolekcije za pretragu.

Uputstvo:

- Za dobivanje nasumične vrijednosti, može se koristiti klasa *Random*, kao što je opisano na sljedećem linku. Listing 13 prikazuje primjer generisanja nasumičnog cijelog broja u rasponu od 0 do 1000.

```
Random random = new Random();
int broj = random.nextInt(0, 1001);
```

Listing 13. Korištenje nasumičnih vrijednosti

Zadatak 2.

Definisati programsko rješenje koje koristi 50 niti za sortiranje kolekcije cijelih brojeva, od najmanjeg ka najvećem broju. Koristiti algoritam sortiranja po želji, međutim algoritam mora biti ručno implementiran (npr. *bubble sort* - poređenje svaka dva elementa i njihova zamjena kako bi element manje vrijednosti bio ispred elementa veće vrijednosti). Obavezno je koristiti atomske varijable, a sve niti moraju imati neograničen opseg tj. koristiti cijelu kolekciju brojeva pri sortiranju.