Mehanizmi sinkronizacije, izvršitelji, kolekcije za rad u višedretvenom okruženju

Matej Mihelčić

Prirodoslovno-matematički fakultet, Sveučilište u Zagrebu

matmih@math.hr

21. prosinca, 2022.

Mehanizmi sinkronizacije višeg reda - Lokoti

Sinkronizirani kod (`synchronized`) koristi **jednostavni lokot** koji **dopušta višestruko zaključavanje**. Ta vrsta lokota je jednostavna za korištenje ali **ima ograničenja**. Zbog toga su napravljeni **sofisticiraniji** objekti za zaključavanje i nalaze se u paketu `java.util.concurrent.locks`. Objekti lokoti funkcioniraju na sličan način kao lokoti pridruženi objektu, **samo jedna dretva može posjedovati objekt lokota u danom trenutku**. Lokoti **podržavaju mehanizam** `wait/notify` **preko povezanog objekta** `Condition`. Najveća prednost objekata lokota naspram lokota pridruženih monitorima objekata je njihova **mogućnost da dretva odustane od pokušaja dohvaćanja lokota**. Metoda `tryLock` **odustaje od pokušaja dohvaćanja lokota ukoliko on nije dostupan odmah ili prije isteka zadanog vremena čekanja**. Metoda `lockInterruptibly` **odustaje od dohvaćanja lokota ukoliko neka druga dretva pošalje signal za prekid prije nego što je lokot dohvaćen**. Objekti lokoti **omogućavaju stvaranje poštenih lokota** (daju prioritet dohvaćanja lokota dretvama **koje dugo čekaju** na dohvaćanje lokota).

Mehanizmi sinkronizacije višeg reda - Lokoti

```
1  public class Objekt {
2      ReentrantLock lokot = new ReentrantLock();
3      int brojac = 0;
4     //ista funkcionalnost kao synchronized metoda
5      public void inkrement() {
6          lock.lock();
7          try { brojac++;} finally {
8              lock.unlock();
9          } }
10
11       public void pokusajInkrement(){
12         boolean lokotDohvacen = lock.tryLock(1, TimeUnit.
       SECONDS);//pokusaj dohvatiti lokot, cekaj 1s
13      if(isLockAcquired) {//ako je lokot dohvacen radi
14         try {  brojac++;} finally {
15             lock.unlock();
16         } } } }
17 //lokot s vremenski postenim pristupom zasticenim resursima
18 ReentrantLock posteniLokot = new ReentrantLock(true);
```

Korištenje lokota.

Mehanizmi sinkronizacije višeg reda - Lokoti čitanja i pisanja

Lokot čitanja i pisanja **sadrži dva lokota, jedan za čitanje i jedan za pisanje. Lokot za čitanje se može istovremeno dohvaćati od strane više dretvi ukoliko niti jedna dretva nije dohvatila lokot za pisanje. Lokot za pisanje smije dohvatiti samo jedna dretva** i to onda kada **niti jedna druga dretva ne koristi lokot za pisanje ili čitanje**.

```
1  public class SinkroniziranaMapa {
2      Map<String,String> sinkHashMap = new HashMap<>();
3      ReadWriteLock lokot = new ReentrantReadWriteLock();
4      Lock lokotPisanja = lokot.writeLock();
5
6      public void put(String key, String value) {
7          try {
8              lokotPisanja.lock();
9              sinkHashMap.put(key, value);
10         } finally {
11             lokotPisanja.unlock();
12         } } }
```

Korištenje lokota pisanja i čitanja.

Mehanizmi sinkronizacije višeg reda - Lokoti čitanja i pisanja

```
1  Lock lokotCitanja = lokot.readLock();
2
3  public String get(String key){
4      try {
5          lokotCitanja.lock();
6          return sinkHashMap.get(key);
7      } finally {
8          lokotCitanja.unlock(); } }
9
10 public boolean sadrziKljuc(String key) {
11     try {
12         lokotCitanja.lock();
13         return sinkHashMap.containsKey(key);
14     } finally {
15         lokotCitanja.unlock();
16     }
17 }
```

Korištenje lokota pisanja i čitanja.

Mehanizmi sinkronizacije višeg reda - Lokoti sa žigom

Lokoti sa žigom (`StampedLock`) **vraćaju žig nakon zaključavanja** koji se **mora koristiti za otključavanje lokota**. Ova vrsta lokota **ne dopušta višestruko zaključavanje**. Dopušta tri načina dohvaćanja: a) **ekskluzivno pisanje** (nakon dohvaćanja lokota on je nedostupan svim drugim dretvama), b) **neekskluzivno čitanje** (lokot je dostupan svim dretvama za čitanje, ukoliko neka dretva proba dohvatiti lokot za pisanje dolazi do zaključavanja), c) **optimistično čitanje** (ne dolazi do zaključavanja lokota čak niti kada neka dretva počne pisati - potrebno validirati čitanje korištenjem funkcije `validate(long stamp)`).

```
1 public String optimisticnoCitanje(String kljuc) {
2     long zig = lokot.tryOptimisticRead();
3     String vrijednost = mapa.get(kljuc);
4     if(!lokot.validate(zig)) {
5         zig = lokot.readLock();
6         try { return mapa.get(kljuc); } finally {
7             lokot.unlock(zig);  } }
8     return vrijednost; }
```

Korištenje lokota sa žigom.

Izvršitelji

Ukoliko želimo **razdvojiti** zadatak koji izvršava dretva i ostale dijelove programa od **stvaranja**, **održavanja** i **upravljanja dretvom** koristimo **izvršitelje**. Postoje tri glavna načina rada s izvršiteljima:
- Sučelja izvršitelja koja definiraju tri tipa objekata izvršitelja (izvršitelj `Executor`, servis izvršitelja `ExecutorService` i dodijeljen servis izvršitelja `ScheduledExecutorService`).
- Grupe dretvi - najčešća vrsta implementacije izvršitelja.
- `Fork/Join` - skup alata za stvaranje višedretvenih aplikacija.

Sučelje izvršitelja, `Executor` je **jednostavno sučelje koje podržava pokretanje novih zadataka**, sučelje servisa izvršitelja `ExecutorService` je podsučelje sučelja izvršitelj koje **sadrži dodatke koji omogućavaju upravljanje životnim ciklusom individualnih zadataka i izvršitelja**, sučelje dodijeljenog servisa izvršitelja `ScheduledExecutorService` je podsučelje sučelja servisa izvršitelja koje **omogućava dodijeljivanje i izvršavanje budućih ili periodičkih zadataka**.

Sučelje izvršitelja

Sučelje izvršitelja **sadrži jednu metodu** `execute` koja **zamijenjuje** `start` kod standardnog načina pokretanja dretve. Dakle umjesto `(new Thread(r)).start();` koristimo `e.execute(r);`. Dok kod korištenja standardnog načina s kreiranjem nove dretve i pokretanjem `start` metode, **nova dretva odmah počinje izvršavati kod metode** `run` objekta koji implementira sučelje `Runnable`, `execute` nekog izvršitelja **može napraviti istu stvar ali i dodijeliti zadatak nekoj postojećoj dretvi ili staviti zadatak na red čekanja** dok dretva ne bude slobodna za njegovo izvršavanje.

```
1 public class IzvrsiteljZadatka implements Executor{
2   @Override public void execute(Runnable r){
3    new Thread(r).start();  } }// r.run (dretva pozivatelj)
4 //ovdje po potrebi kreiramo novu dretvu
5 public class Zadatak implements Runnable{
6   public void run(){ System.out.println("Dretva izvodi
      zadatak!"); } }
```

Implementacija i korištenje izvršitelja.

Sučelje izvršitelja

```
1  public class Zadatak1 implements Runnable {
2      public void run(){ System.out.println("Neki drugi
       zadatak!"); } }
3
4  public static void main(String args[]){
5    Executor e = new IzvrsiteljZadatka();// moze se izvrsiti
6    e.execute(new Zadatak());// u dretvi pozivatelju, novoj
7    e.execute(new Zadatak1());// dretvi ili u dretvi iz grupe
       dretvi ovisno o implementaciji
8  //obican red smijemo koristiti kada sekvencijalno stavimo
       zadatke na njega i pokrecemo ih na jednoj ili vise
       dretvi (postujuci korektnost visedretvenih programa)
9          Queue<Runnable> redZadataka = new LinkedList<>();
10         Zadatak1 z1 = new Zadatak1();
11       for(int i=0;i<100;i++)
12           if(i%3==0) redZadataka.add(new ZadatakGen(i+1));
13           else redZadataka.add(z1);
14      while(redZadataka.size()>0){
15          e.execute(redZadataka.poll()); } }
```

Implementacija i korištenje izvršitelja.

Sučelje izvršitelja

```
1  public class ZadatakGen implements Runnable {
2      int broj;
3
4      public ZadatakGen(int i){
5          broj = i;
6      }
7
8      public void run(){
9          System.out.println("Zadatak: "+broj);
10     }
11
12 }
```

Implementacija i korištenje izvršitelja.

Sučelje servisa izvršitelja

Sučelje servisa izvršitelja pruža **nadograđenu** funkcionalnost metode `execute` preko **svestranije** metode `submit`. Kao i `execute`, `submit` radi nad objektima koji implementiraju sučelje `Runnable` ali **dodatno** i nad objektima koji implementiraju sučelje `Callable`. Prednost objekata koji implementiraju sučelje `Callable` je u tome što **dozvoljavaju da zadatak vraća vrijednost**. Povratna vrijednost je **objekt tipa** `Future` koji se **koristi za dobivanje povratne vrijednosti** i za **upravljanje statusom** i `Callable` i `Runnable` zadataka. Servisi izvršitelja sadrže metode za **slanje velikih kolekcija zadataka** u obliku `Callable` objekata i metode za **gašenje izvršitelja** (u tom slučaju treba ispravno obraditi potencijalne prekide).

Sučelje servisa izvršitelja obično **koristi grupe dretvi** (*thread pool*) za izvršavanje zadataka.

Sučelje servisa izvršitelja

```
1  public static void main(String args[])throws
       InterruptedException, ExecutionException{
2          ExecutorService izvrsitelj = Executors.
       newFixedThreadPool(4);
3
4          Runnable runnableZadatak = () -> {
5       try {
6          TimeUnit.MILLISECONDS.sleep(300);
7          System.out.println("Rad bez vracanja vrijednosti");
8       } catch (InterruptedException e) {
9          e.printStackTrace(); } };
10
11 Callable<String> callableZadatak = () -> {
12      TimeUnit.MILLISECONDS.sleep(300);
13      return "Izvodenje zadatka!"; };
14
15 List<Callable<String>> callableZadaci = new ArrayList<>();
16 callableZadaci.add(callableZadatak); callableZadaci.add(
       callableZadatak); callableZadaci.add(callableZadatak);
```

Implementacija i korištenje servisa izvršitelja.

Sučelje servisa izvršitelja

```
1  izvrsitelj.execute(runnableTask);
2  List<Future<String>> povratne = izvrsitelj.invokeAll(
       callableZadaci);
3  izvrsitelj.shutdown();
4  try {
5      if (!izvrsitelj.awaitTermination(800, TimeUnit.
       MILLISECONDS)) {
6          izvrsitelj.shutdownNow();
7      }
8  } catch (InterruptedException e) {
9      izvrsitelj.shutdownNow();
10 }
11         for(Future<String> t:povratne){
12             System.out.println(t.get());
13          }
14     }
```

Implementacija i korištenje servisa izvršitelja.

Sučelje dodijeljenog servisa izvršitelja

Sučelje dodijeljenog servisa izvršitelja (`ScheduledExecutorService`) **proširuje metode** sučelja `ExecutorService` **s rasporedom**, koji izvršava `Runnable` ili `Callable` zadatak **nakon definiranog vremenskog perioda**. Također, sučelje definira metode `scheduleAtFixedRate` i `scheduleWithFixedDelay` koje **kontinuirano izvode neki definirani zadatak u predefiniranim vremenskim intervalima**.

```
1 public class ServisDodijeljenogIzvrsitelja {
2     public static void main(String args[]) throws
      InterruptedException, ExecutionException{
3         ScheduledExecutorService izvrsitelj = Executors
4   .newSingleThreadScheduledExecutor();
```

Implementacija i korištenje dodijeljenog servisa izvršitelja.

Sučelje dodijeljenog servisa izvršitelja

```
1  Runnable runnableZadatak = () -> {
2      try {
3          TimeUnit.MILLISECONDS.sleep(300);
4          System.out.println("Rad bez vracanja vrijednosti");
5      } catch (InterruptedException e) {
6          e.printStackTrace(); } };
7
8  Callable<String> callableZadatak = () -> {
9      TimeUnit.MILLISECONDS.sleep(300);
10     return "Izvodenje zadatka"; };
11
12   Future<String> rezultat =
13   izvrsitelj.schedule(callableZadatak, 1, TimeUnit.SECONDS);
14   System.out.println("Povratni rezultat: "+rezultat.get());
15
16   izvrsitelj.scheduleAtFixedRate(runnableZadatak, 100, 450,
       TimeUnit.MILLISECONDS);  } }
```

Implementacija i korištenje dodijeljenog servisa izvršitelja.

Grupe dretvi

Većina implementacija izvršitelja u paketu `java.util.concurrent` **koristi grupe dretvi** koje sadrže **više dretvi radnika**. Ta vrsta dretvi **postoji neovisno o zadacima** tipa `Runnable` i `Callable` koje izvodi i često se **koristi za izvođenje više zadataka**. Takvo korištenje **minimizira trošenje računalnih resursa na stvaranje dretvi**. Objekti dretvi **koriste značajnu količinu memorije** stoga **konstantno alociranje/dealociranje** u aplikacijama koje koriste puno dretvi može izazvati **znatno trošenje resursa za upravljanje memorijom**.

Često korišteni tip grupa dretvi je **grupa dretvi fiksne duljine**. Kod tog tipa se **uvijek izvodi konstantan broj dretvi** i ukoliko je neka dretva ugašena pri korištenju, **automatski se zamijenjuje novom dretvom**. Zadaci se šalju grupi preko **internog reda** koji **sadrži dodatne zadatke kada postoji više zadataka od dretvi**. Korištenjem grupa dretvi, aplikacija se **može na kontrolirani način nositi s iznenadnim povećanjem broja zadataka**. Zadaci će duže čekati u redu čekanja, ali neće doći do povećanja broja dretvi, trošenja resursa na stvaranje i uništavanje dretvi ili potencijalnog kraha servera.

Grupe dretvi

Jednostavan način za stvaranje izvršitelja koji koristi grupu dretvi **fiksne duljine** je pozivom metode `newFixedThreadPool` iz paketa `java.util.concurrent.Executors`. Ta klasa pruža i druge metode stvaranja grupa dretvi. Metoda `newCachedThreadPool` stvara izvršitelja s **proširivom duljinom** grupe dretvi. Taj izvršitelj je **pogodan za korištenje kod aplikacija koje obrađuju puno kratkih zadataka**. Metoda `newSingleThreadExecutor` stvara izvršitelja koji izvršava **po jedan zadatak u danom trenutku**. Postoje i verzije dodijeljenog servisa izvršitelja sa sličnom funkcionalnosti. Također, stvaranje instanci `java.util.concurrent.ThreadPoolExecutor` ili `java.util.concurrent.ScheduledThreadPoolExecutor` nudi dodatne opcije (ove klase se mogu i naslijediti). Ove klase bilježe neke osnovne statistike kao što je broj završenih zadataka, također nudi brojne parametre (`corePoolSize`, `maximumPoolSize`, `prestartCoreThread`, `prestartAllCoreThreads`) itd. Dretvama u *Javi* (uključujući one sadržane u grupi dretvi) se može postaviti tzv. `deamon` status.

Grupe dretvi

Taj status određuje **smije li** *Java* program **završiti izvođenje dok se dana dretva još izvodi**. Ukoliko je deamon status dretve postavljen na `true`, **program može završiti izvođenje prije završetka izvođenja dretve, inače mora čekati gašenje dretve**.

```
1  public class ZadatakGrupe implements Runnable {
2        private String ime;
3
4      public ZadatakGrupe(String ime) { this.ime = ime; }
5      public String getIme() { return ime; }
6
7     @Override public void run() {
8        try {
9              Long duration = 1l;
10             System.out.println("Izvrsavam : " + ime);
11             TimeUnit.SECONDS.sleep(duration);
12         } catch (InterruptedException e) {
13             e.printStackTrace(); } } }
14
```

Korištenje izvršitelja grupe dretvi.

Grupe dretvi

```
1  public class IzvrsiteljGrupeDretvi {
2   public static void main(String args[]){
3     ThreadPoolExecutor tpe = new ThreadPoolExecutor
      (16,64,100, TimeUnit.SECONDS, new ArrayBlockingQueue<
      Runnable>(100));//moramo koristiti kontejnere koji
      podrzavaju visedretvene operacije (imaju blocking u
      imenu)
4
5   for (int i = 1; i <= 100; i++){
6     ZadatakGrupe zad = new ZadatakGrupe("zadatak " + i);
7     System.out.println("Stvoren : " + zad.getIme());
8     tpe.execute(zad);}
9           tpe.shutdown();
10     }// vrijeme izvrsavanja na Intel I3-5005U, 2.00 Ghz
11  }// format: (stalniBrojDretvi, maksimalniBrojDretvi) -
      vrijeme. (1, 2) - 1m 41s, (2, 4) - 51s, (4, 8) - 26s,
      (8, 16) - 14s, (16, 32) - 8s, (32, 64) - 5s, (64, 128) -
       3s, (128, 256) - 2s
12
```

Korištenje izvršitelja grupe dretvi.

Grupe dretvi

Parametri konstruktora izvršitelja grupi dretvi su: **stalni broj dretvi, maksimalni broj dretvi, vrijeme toleriranja neaktivnosti dretvi** (za sve dretve koje su višak - one koje ne spadaju u stalne), **jedinica vremena za čekanje, kontejner** (mora biti `blocking`) u koji se spremaju zadaci.

Standardno se **dretve na parnim jezgrama kreiraju i pokreću samo kada dolaze novi zadaci**, no ta **funkcionalnost se može promijeniti** korištenjem metoda `prestartCoreThread` ili `prestartAllCoreThreads` (uglavnom se radi ukoliko se pokreće izvršitelj s **nepraznim redom izvršavanja**). Standardno dretve stvorene korištenjem izvršitelja grupi dretve imaju `deamon` status postavljen na `false`, **sve dretve pripadaju istoj grupi dretvi s istim prioritetom izvršavanja**. Te postavke se mogu promijeniti **korištenjem drugačije definirane klase** izvedene iz `ThreadFactory`. Ukoliko je aktivno više dretvi od parametra `corePoolSize` (prvi parametar konstruktora), one se **gase ukoliko su neaktivne nakon vremena specificiranog** kao 3. parametar (`keepAliveTime`).

Grupe dretvi

Postavljanjem zastavice `allowCoreThreadTimeOut` se mogu gasiti i **neaktivne stalne dretve** (no `keepAliveTime` u tom slučaju mora biti veći od nula). Zadaci koji se šalju izvršitelju se spremaju u proizvoljni `BlockingQueue`.

Novi zadaci koji se daju na izvršavanje izvršitelju grupi dretvi **će biti odbijeni ako**: a) **izvršitelj je ugašen**, b) **izvršitelj ima ogradu na maksimalan broj dretvi i kapacitet reda zadataka i kapacitet je popunjen**. U slučaju odbijanja zadatka se poziva metoda `RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor)` koji nudi **četiri moguća načina obrade odbijanja zadatka**: a) standardno se poziva `ThreadPoolExecutor.AbortPolicy` koji vraća `RejectedExecutionException` nakon odbijanja, b) `ThreadPoolExecutor.CallerRunsPolicy` definira da **dretva koja je pozvala** `execute` **sama izvrši zadatak**, time se usporava dodavanje novih zadataka i omogućava oslobađanje dretvi radnika,

Grupe dretvi

c) `ThreadPoolExecutor.DiscardPolicy` definira da se zadatak koji se ne može izvršiti **ispusti**, e) `ThreadPoolExecutor.DiscardOldestPolicy` definira da se **odbacuje zadatak na početku reda čekanja** (najstariji zadatak) te se pokuša izvršiti neki zadatak. Moguće je definirati i druge vrste klasa `RejectedExecutionHandler`.

Klasa izvršitelja grupa dretvi sadrži i niz metoda (koje se mogu nadjačati) tipa `beforeExecute(Thread, Runnable)` i `afterExecute(Runnable, Throwable)` koje se **izvršavaju prije i nakon izvršavanja nekog zadatka zadanog izvršitelju**. Te metode se mogu koristiti za **reinicijalizaciju lokalnih dretvi, skupljanje statistika, dodavanje log zapisa** itd.

Grupa dretvi koja **više nije referencirana i ne sadrži niti jednu dretvu** može biti očišćena od strane sakupljača smeća.

Grupe dretvi

```
1  class PausableThreadPoolExecutor extends ThreadPoolExecutor
       {
2    private boolean isPaused;
3    private ReentrantLock pauseLock = new ReentrantLock();
4    private Condition unpaused = pauseLock.newCondition();
5
6    public PausableThreadPoolExecutor(...) { super(...); }
7
8    protected void beforeExecute(Thread t, Runnable r) {
9      super.beforeExecute(t, r);
10     pauseLock.lock();
11     try {
12       while (isPaused) unpaused.await();
13     } catch (InterruptedException ie) { t.interrupt();
14     } finally { pauseLock.unlock(); } }
15
```

Nasljeđivanje klase ThreadPoolExecutor.

Grupe dretvi

```
1  public void pause() {
2      pauseLock.lock();
3      try { isPaused = true;
4      } finally { pauseLock.unlock(); } }
5
6    public void resume() {
7      pauseLock.lock();
8      try {
9        isPaused = false;
10       unpaused.signalAll();
11     } finally {
12       pauseLock.unlock();
13     }
14   }
15 }
16
```

Nasljeđivanje klase ThreadPoolExecutor.

Fork/Join

`Fork/join` sustav je **implementacija sučelja** `ExecutorService` koja **omogućuje korištenje više procesora**. Sustav je **dizajniran za rješavanje zadataka koji se mogu rekurzivno rastaviti na manje zadatke**. Fork/join sustav **dijeli zadatke na dretve radnike u grupi dretvi** po posebnom principu, tzv. **principu krađe posla** (eng. *work-stealing*). Dretve radnici koji obave svoje zadatke **mogu ukrasti** (preuzeti) zadatke drugih dretvi koje još izvršavaju zadatke. Sustav `fork/join` je baziran na klasi `ForkJoinPool` koja nasljeđuje klasu `AbstractExecutorService`. `ForkJoinPool` implementira osnovni algoritam krađe zadataka i može izvršavati procese zadataka `fork/join` (klasa `ForkJoinTask`).

Sustav `fork/join` se koristi tako da se **prvo konstruira kod koji rješava dio problema**. Zadaci koji su **preveliki se dijele i paralelno rješavaju prije definiranim postupkom**.

Fork/Join

```
1
2 if(moj dio posla je dovoljno malen)
3   obavi posao definiranim algoritmom
4 else
5   podijeli posao na dva dijela
6   pokreni rjesavanje podproblema i cekaj rezultat
7
```

Pristup rješavanju problema koristeći sustav fork/join.

Nakon kreiranja podklase klase `ForkJoinTask`, treba **kreirati objekt koji reprezentira sav posao koji treba napraviti** i taj objekt treba **proslijediti metodi** `invoke()` **instance klase** `ForkJoinPool`. Klasa koja definira način rješavanja problema proširuje klasu `RecursiveAction` a algoritam za rješavanje problema se definira ili poziva unutar metode `compute()` koja se počinje izvoditi nakon dodjeljivanja dretve zadatku.

Fork/Join

```
1  public class ForkZamuti extends RecursiveAction {
2      private int[] mIzvor;
3      private int mStart;
4      private int mDuljina;
5      private int[] mOdrediste;
6      private int mSirinaZamuti = 15;
7
8    public ForkZamuti(int[] pocetak, int start, int duljina,
       int[] kraj) {
9        mIzvor = pocetak; mStart = start; mDuljina = duljina;
       mOdrediste = kraj; }
10   //racuna prosjek susjednih piksela (zamucivanje)
11     protected void racunajDirektno() {
12         int susjedniPikseli = (mSirinaZamuti - 1) / 2;
13         for (int i = mStart; i < mStart + mDuljina; i++) {
14             float rt = 0, gt = 0, bt = 0;
15
```

Upotreba sustava fork/join za rješavanja problema zamučivanja slike.

Fork/Join

```
1        for (int mi = -susjedniPikseli; mi <= susjedniPikseli; mi++) {
2            int mindeks = Math.min(Math.max(mi + index, 0), mIzvor.length - 1);
3            int piksel = mIzvor[m];
4            rt += (float)((piksel & 0x00ff0000) >> 16)
5                    / mSirinaZamuti;
6            gt += (float)((piksel & 0x0000ff00) >>  8)
7                    / mSirinaZamuti;
8            bt += (float)((piksel & 0x000000ff) >>  0)
9                    / mSirinaZamuti;
10         }
11
12           int odrpiksel = (0xff000000 ) | (((int)rt) << 16) | (((int)gt) <<  8)   |  (((int)bt) <<  0);
13           mOdrediste[indeks] = odrpiksel;
14       } }//racuna se za male zadatke sekvencijalno
15
```

Upotreba sustava fork/join za rješavanja problema zamučivanja slike.

Fork/Join

```
1  protected static int sPrag = 100000;
2
3  protected void compute() { //racunaj sekvencijalno
4      if (mDuljina < sPrag) { //ako duljina komada <100000
5          racunajDirektno();
6          return;
7      }
8
9      int split = mDuljina / 2; //inace podijeli zadatak
10     //na pola i pokreni polovice u novim dretvama
11     invokeAll(new ForkZamuti(mIzvor, mStart, split,
       mOdrediste), new ForkZamuti(mIzvor, mStart + split,
       mDuljina - split, mOdrediste));
12   } }
13 //poziv u glavnom programu
14 ForkZamuti fz = new ForkZamuti(izvor, 0, izvor.length,
       odrediste);
15 ForkJoinPool grupa = new ForkJoinPool();
16 grupa.invoke(fz);
```

Upotreba sustava fork/join za rješavanja problema zamučivanja slike.

Kolekcije za rad u višedretvenom okruženju

Paket `java.util.concurrent` **sadrži brojne kolecije za rad u višedretvenom okruženju**. Mogu se podijeliti prema sučeljima koje implementiraju na:

- `BlockingQueue` (implementiraju ga `ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`). Postoji i `BlockingDequeue` (implementira ga `LinkedBlockingDequeue`). Strukture iz ove skupine **blokiraju kada dretva pokuša dodati element u puni red ili dohvatiti element iz praznog reda**. Kolekcije su **sigurne za korištenje u višedretvenom okruženju**, **operacije nad elementima se mogu smatrati atomarnim (pristup je zaštičen lokotima i drugim sinkronizacijskim mehanizmima)**. Operacije koje rade nad kolekcijama `addAll`, `containsAll`, `retainAll`, `removeAll` **nisu nužno atomarne** (ovisi o implementaciji) stoga može doći do iznimke prije dodavanja svih elemenata.
- Druga skupina spremnika su **spremnici koji osiguravaju korektno izvođenje u višedretvenom okruženju ali ne blokiraju** (ne postoji mehanizam kojim bi mogli dobiti ekskluzivni pristup cijelom spremniku).

Kolekcije za rad u višedretvenom okruženju

U ovu skupinu spadaju klase koje implementiraju sučelje `ConcurrentMap` kao što su `ConcurrentHashMap` (analogon klase `HashMap` za rad u višedretvenom okruženju), `ConcurrentNavigableMap` (podržava približna poklapanja) i `ConcurrentSkipListMap` (višedretveni analogon klase `TreeMap`). `ConcurrentMap` **definira atomarne operacije kojima se uklanjaju parovi ključ-vrijednost ukoliko je ključ prisutan ili dodaje par ključ-vrijednost ukoliko je ključ odsutan** (time se otklanja potreba za sinkronizacijom). Postoje i klase `ConcurrentLinkedQueue` i `ConcurrentLinkedDequeue` koje su također sigurne za izvršavanje u višedretvenom okruženju. Sve navedene kolekcije **izbjegavaju probleme s konzistentnosti memorije na način da osiguravaju da su operacije dodavanja u kolekciju i daljnje operacije pristupa ili uklanjanja u relaciji** `dogodilo-se-prije`.

Kolekcije za rad u višedretvenom okruženju

```
1  ArrayBlockingQueue<Integer> red = new ArrayBlockingQueue
       (100);
2
3  red.add(1); red.add(2); red.add(3); red.add(4); red.add(5);
       red.remove(2); red.offer(5);
4        //offer preferirana metoda za dodavanje u red jer ne
       izbacuje iznimku ako je red pun
5            System.out.println(red);
6            try{
7            red.offer(125, 20, TimeUnit.SECONDS);//dretva ceka
8            }//dok ne istekne vrijeme ili se red ne oslobodi
9            catch(InterruptedException e){
10               e.printStackTrace();
11           }
12           System.out.println(red);
13
14           red.removeIf((Integer x)->{return x<5;});
15           System.out.println(red);
```

Upotreba spremnika iz paketa java.util.concurrent.

Kolekcije za rad u višedretvenom okruženju

`DelayQueue` omogućava postavljanje vremena za svaki element reda, element se može maknuti iz reda tek nakon što zadano vrijeme prođe dok `size` vraća sumu elemenata kojima je vrijeme čekanja isteklo i onih kojima nije. `LinkedBlockingQueue` implementira blokirajući red korištenjem povezane liste, `PriorityBlockingQueue` nudi funkcionalnost blokirajućeg prioritetnog reda.

```
1  public class ElementiSKasnjenjem implements Delayed {
2      int broj;
3     long vrijemeURedu;
4
5    public  ElementiSKasnjenjem(long t, int broj){
6          vrijemeURedu = t;
7          vrijemeURedu =  System.currentTimeMillis() + t;
8          this.broj = broj; }
9
10   public int getBroj(){ return broj; }
```

Upotreba spremnika iz paketa java.util.concurrent.

Kolekcije za rad u višedretvenom okruženju

```
1   @Override
2    public long getDelay(TimeUnit t){
3     long diff = vrijemeURedu - System.currentTimeMillis();
4     return t.convert(diff, TimeUnit.MILLISECONDS);
5    }
6
7   @Override
8    public int compareTo(Delayed drugi)
9    {
10        if (vrijemeURedu < ((ElementiSKasnjenjem)drugi).
    vrijemeURedu) {
11            return -1;
12        }
13        if (vrijemeURedu > ((ElementiSKasnjenjem)drugi).
    vrijemeURedu) {
14            return 1;
15        }
16        return 0; } }
```

Upotreba spremnika iz paketa java.util.concurrent.

Kolekcije za rad u višedretvenom okruženju

```
1 java.util.concurrent.DelayQueue redskasnjenjem = new
    DelayQueue();
2       for(int i=0;i<100;i++){
3           ElementiSKasnjenjem elem = new
    ElementiSKasnjenjem((long)(Math.random()*1000),i);//
    dodaj slucajni broj milisekundi kasnjenja elementa
4           redskasnjenjem.add(elem);
5       }
6
7      //Thread.sleep(1000);
8       ElementiSKasnjenjem prvi = (ElementiSKasnjenjem)
    redskasnjenjem.remove();//dobijemo iznimku java.util.
    NoSuchElementException ukoliko je zakomentirana linija
    7, inace element s najmanjim vremenom kasnjenja
9        System.out.println("Element: "+prvi.getBroj());
```

Upotreba spremnika iz paketa java.util.concurrent.

Kolekcije za rad u višedretvenom okruženju

ConcurrentHashMap dozvoljava istovremeno čitanje i pisanje grupe dretvi. Iteratori ove mape imaju svojstvo slabe konzistentnosti (ne garantiraju da je nužno uočena zadnja upisana vrijednost).

```
1  public class Citac extends Thread {
2       private ConcurrentHashMap<Integer, String> mapa;
3       private String ime;
4
5      public Citac(ConcurrentHashMap<Integer, String> map,
       String dretvaIme) {
6           this.mapa = map; this.ime = dretvaIme; }
7
8      public void run() {
9           while (true) {
10             ConcurrentHashMap.KeySetView<Integer, String>
       kljucevi = mapa.keySet();
11             Iterator<Integer> iterator = kljucevi.iterator();
```

Upotreba spremnika iz paketa java.util.concurrent.

Kolekcije za rad u višedretvenom okruženju

```
1            long time = System.currentTimeMillis();
2            String izlaz = time + ": " + ime + ": ";
3
4             while (iterator.hasNext()) {
5                 Integer kljuc = iterator.next();
6                 String vrijednost = mapa.get(kljuc);
7                 izlaz += kljuc + "=>" + vrijednost + "; ";
8             }
9
10            System.out.println(izlaz);
11
12            try {
13                Thread.sleep(300);
14            } catch (InterruptedException ex) {
15                System.out.println("Citac: "+ime+" zavrsava
    citanje...");
16                return;
17            } } } }
```

Upotreba spremnika iz paketa java.util.concurrent.

Kolekcije za rad u višedretvenom okruženju

```
1  public class Pisac extends Thread  {
2      private ConcurrentMap<Integer, String> mapa;
3      private Random random;
4      private String ime;
5
6      public Pisac(ConcurrentMap<Integer, String> map,
7                           String imeDretve, long randomSeed) {
8          this.mapa = map;
9          this.random = new Random(randomSeed);
10         this.ime = imeDretve;
11     }
12
13     public void run() {
14         while (true) {
15             Integer kljuc = random.nextInt(10);
16             String vrijednost = ime;
17             if(mapa.putIfAbsent(kljuc, vrijednost) == null){
18                 long time = System.currentTimeMillis();
```

Upotreba spremnika iz paketa java.util.concurrent.

Kolekcije za rad u višedretvenom okruženju

```
1                   String ispis = String.format("%d: %s je
    zapisao [%d => %s]", time, ime, kljuc, vrijednost);
2                   System.out.println(ispis); }
3
4               Integer kljucZaIzbaciti = random.nextInt(20);
5
6               if (mapa.remove(kljucZaIzbaciti, vrijednost)) {
7                   long time = System.currentTimeMillis();
8                   String izlaz = String.format("%d: %s je
    izbacio [%d => %s]", time, ime, kljucZaIzbaciti,
    vrijednost);
9                   System.out.println(izlaz); }
10
11              try {
12                  Thread.sleep(500);
13              } catch (InterruptedException ex) {
14                  System.out.println("Pisac: "+ime+" zavrsava
    pisanje...");
15                  return; } } } }
```

Upotreba spremnika iz paketa java.util.concurrent.

Kolekcije za rad u višedretvenom okruženju

```
1  public class PrimjerConcurrentHashMap{
2      public static void main(String[] args) throws
       InterruptedException {
3        ConcurrentHashMap<Integer, String> mapa = new
       ConcurrentHashMap<>();
4        Thread p1 = new Thread(new Pisac(mapa, "Pisac-1", 1));
5        Thread p2 = new Thread(new Pisac(mapa, "Pisac-2", 2));
6        Thread p3 = new Thread(new Pisac(mapa, "Pisac-3", 3));
7        Thread citaci[] = new Thread[10];
8
9        p1.start(); p2.start(); p3.start();
10       for (int i = 0; i < 10; i++) {
11   citaci[i] = new Thread(new Citac(mapa, "Citac-" + (i+1)));
12             citaci[i].start(); }
13
14       Thread.sleep(10000);
15       for(int i=0;i<10;i++) citaci[i].interrupt();
16       p1.interrupt(); p2.interrupt(); p3.interrupt();
17       System.out.println("Sve dretve su prekinute!"); } }
```

Upotreba spremnika iz paketa java.util.concurrent.

Sinkronizacija barijerama

Barijere su posebni sinkronizacijski mehanizmi koji za grupu dretvi **pauziraju izvođenje dretve iz grupe** (na poziciji u programu na kojoj se javlja barijera) **dok sve dretve iz grupe u svom izvršavanju ne izvrše naredbu ulaska u barijeru**. **U tom trenutku se sve dretve grupe odblokiraju i nastavljaju izvršavanje**. *Java* sadrži nekoliko različitih vrsta barijera: `CyclicBarrier` (može se ponovo koristiti nakon što su dretve otpuštene), `Phaser` (nudi dodatne mogućnosti kao što je varijabilni broj dretvi koje trebaju stići do barijere da bi se otpustile, taj broj može varirati kroz vrijeme). Postoji i specijalni sinkronizacijski mehanizam `CountDownLatch` koji ima brojač koji broji koliko dretvi treba pozvati funkciju dekrementa brojača na toj klasi. Za razliku od barijere, dretve koje pozovu dekrement brojača ne trebaju čekati dok druge dretve ne naprave dekrement (odnosno dok brojač ne dođe do 0). Osigurava se jedino da dretve ne mogu zaobići potencijalni `await` dok se ne osiguraju uvjeti za prolazak svih dretvi.

Sinkronizacija barijerama

Problem: Izračunati sumu svih elemenat matrice u višedretvenoj aplikaciji. Svaka dretva će računati sumu dijela redaka matrice. Sumu suma redaka će računati glavna dretva i prijaviti rezultat.

```
1  public class SumatorRedakaMatrice implements Runnable{
2       int id, pocetniRedak, konacniRedak;
3       double matrica[][];
4       double sume[];
5       CyclicBarrier barijera;
6
7       SumatorRedakaMatrice(int i, int p, int z, double m[][],
       double s[], CyclicBarrier b){
8           id = i; pocetniRedak = p; konacniRedak = z;
9           matrica = m; sume = s; barijera = b;
10       }
```

Sinkronizacija barijerama.

Sinkronizacija barijerama

```
1      @Override public void run(){
2          sume[id] = 0.0;
3          for(int i=pocetniRedak;i<=konacniRedak;i++)
4              for(int j=0;j<matrica[pocetniRedak].length;j++)
5                  sume[id]+=matrica[i][j];
6           System.out.println("Broj dretvi koji ceka: " +
    barijera.getNumberWaiting());
7           try{
8           barijera.await();
9           }
10          catch(InterruptedException e){
11              e.printStackTrace();
12          }
13          catch(BrokenBarrierException e){
14              e.printStackTrace();
15          }
16     }
17 }
```

Sinkronizacija barijerama.

Sinkronizacija barijerama

```
1  public class SumirajElementeMatrice {
2      public static void main(String args[]){
3          double matrica[][] = new double[10000][10000];
4          double sume[] = new double[12];
5
6        for(int i=0;i<matrica.length;i++)
7            for(int j=0;j<matrica[0].length;j++)
8              matrica[i][j] = Math.random()*2000+Math.random();
9
10           Thread dretve[] = new Thread[12];
11           int brojRedaka = matrica.length/12;
12           int poc = 0, kraj =-1;
13
14 CyclicBarrier barijera = new CyclicBarrier(dretve.length+1);
15
16           SumatorRedakaMatrice s;
17           for(int i=0;i<dretve.length;i++){
18               kraj+=brojRedaka;
```

Sinkronizacija barijerama.

Sinkronizacija barijerama

```
1            if((i+1)<dretve.length)
2                s = new SumatorRedakaMatrice(i,poc,kraj,
    matrica,sume, barijera);
3            else  s = new SumatorRedakaMatrice(i,poc,matrica
    [0].length-1,matrica,sume, barijera);
4            poc+=brojRedaka;
5            dretve[i] = new Thread(s);
6        }
7
8        for(int i=0;i<dretve.length;i++){
9            dretve[i].start(); }
10
11       try {
12            barijera.await();
13     System.out.println("Broj dretvi koji ceka: " +
    barijera.getNumberWaiting());
14    } catch (InterruptedException | BrokenBarrierException e
    ) { e.printStackTrace(); }
```

Sinkronizacija barijerama.

Sinkronizacija barijerama

```
1          double ukupnaSuma = 0.0;
2          for(int i=0;i<sume.length;i++)
3              ukupnaSuma+=sume[i];
4
5
6          double sekvencijalnaSuma = 0.0;
7
8          for(int i=0;i<matrica.length;i++)
9              for(int j=0;j<matrica[i].length;j++)
10                 sekvencijalnaSuma+=matrica[i][j];
11
12       //uz barijeru dobijemo identicnu vrijednost, bez nje
     se vrijednost razlikuje
13       System.out.println(ukupnaSuma+" "+sekvencijalnaSuma);
14     }
15 }
```

Sinkronizacija barijerama.

Atomarne varijable

Paket `java.util.concurrent.atomic` **definira klase koje podržavaju atomarne operacije nad jednom varijablom**. Sve klase imaju `get` i `set` metode koje rade kao čitanje i pisanje nad `volatile` varijablama. Metoda `set` je u relaciji `dogodilo-se-prije` **sa svim pozivima** `get` **koji se dogode nakon tog poziva** `set`. Metoda `compareAndSet` također **ima svojstvo nedjeljivosti** kao i jednostavne atomarne operacije nad atomarnim cijelim brojevima.

```
1 class AtomarniBrojac {
2     private AtomicInteger c = new AtomicInteger(0);
3
4     public void inkrement() { c.incrementAndGet(); }
5     public void dekrement() { c.decrementAndGet(); }
6     public int vrijednost() { return c.get(); }
7 }
```

Primjer korištenja atomarnih varijabli.

Generator slučajnih brojeva u višedretvenim programima

Paket `java.util.concurrent` sadrži klasu `ThreadLocalRandom` za aplikacije **koje će koristiti slučajne brojeve u višedretvenim aplikacijama** ili u `fork/join` **zadacima**. Glavna dobrobit korištenja ove klase u odnosu na `Math.random()` je u tome što je potrebno manje **sinkronizacije** za dobivanje slučajnih brojeva. Pozivom `ThreadLocalRandom.current()` i odgovarajuće metode za dohvaćanje brojeva (npr. `nextDouble()`, `nextInt()`) itd. dobijemo pseudoslučajni broj odgovarajućeg tipa i raspona. Integer u rasponu $[4, 77>$ generiramo pozivom `int r = ThreadLocalRandom.current() .nextInt(4, 77);`

Zadatak

U primjeru s pisačem i čitačem unutar klase pisač kreiramo novi objekt za generiranje slučajnih brojeva za svaki objekt tipa pisač. Zamijenite navedeni način generiranja slučajnih brojeva pozivima `ThreadLocalRandom`. Izmjerite koliko puta je došlo do konflikata (pokušaja dodavanja para s ključem koji već postoji u mapi) s oba načina generiranja slučajnih brojeva.