Višenitnost u Javi

Sadržaj

Uvod u višenitnost

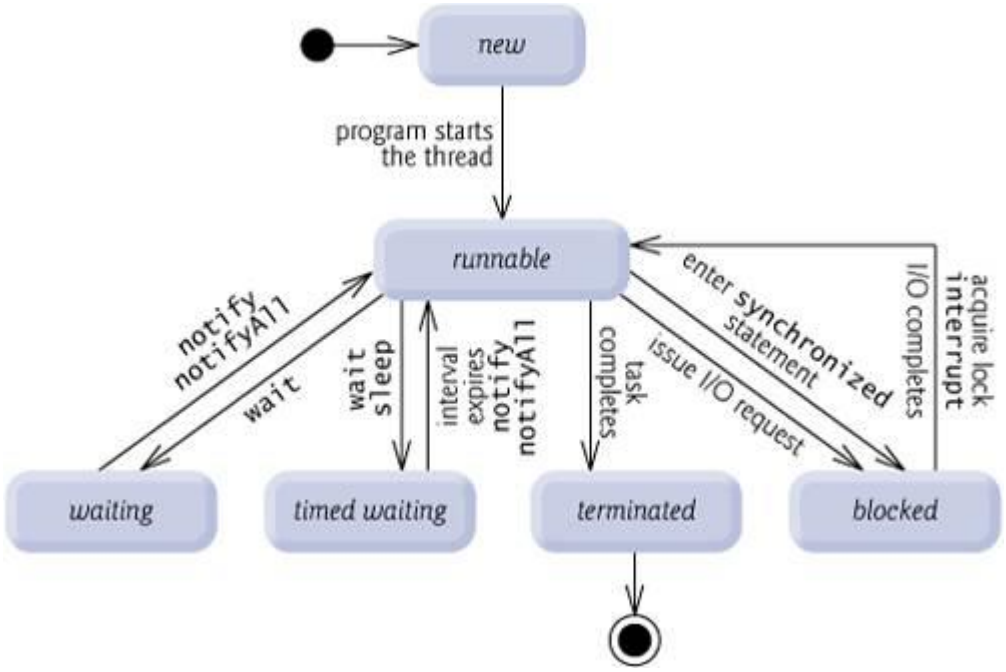
- Višenitnost se kod programiranja koristi u slučaju kad je potrebno istovremeno paralelno izvođenje više zadataka
- Na jednoprocesorskim (jednojezgrenim) sustavima prividna paralelnost postizala se od strane operacijskog sustava koji je svaki zadatak izvodio neko kratko vrijeme, ali je brzo izmjenjivao zadatke koji su se izvodili
- Tako je bila postignuta paralelnost, ali se zadaci u stvari nisu izvodili istovremeno
- Paralelno izvođenje zadataka podrazumijeva simultano izvođenje zadataka koje je moguće u slučaju višejezgrenih procesora

Višenitnost u Javi

- U Javi je moguće koristiti niti uz pomoć API-a koji je ugrađen u jezik
- Java programi mogu izvoditi više niti pri čemu svaka nit ima vlastiti stog za pozivanje metoda (engl. *method-call stack*) i vlastito programsko brojilo (engl. *program counter*)
- Time je omogućeno da niti međusobno mogu dijeliti resurse poput memorije ili datoteke
- To svojstvo naziva se **višenitnost** (engl. *multithreading*)
- Kod korištenja niti prilikom programiranja potrebno je pripaziti na sinkronizaciju među nitima kako ne bi došlo do neželjenih posljedica poput potpunog zastoja (engl. *deadlock*) i slično

Stanja i životni ciklus niti

- Nit može biti u jednom od sljedećih stanja:



Stanja i životni ciklus niti

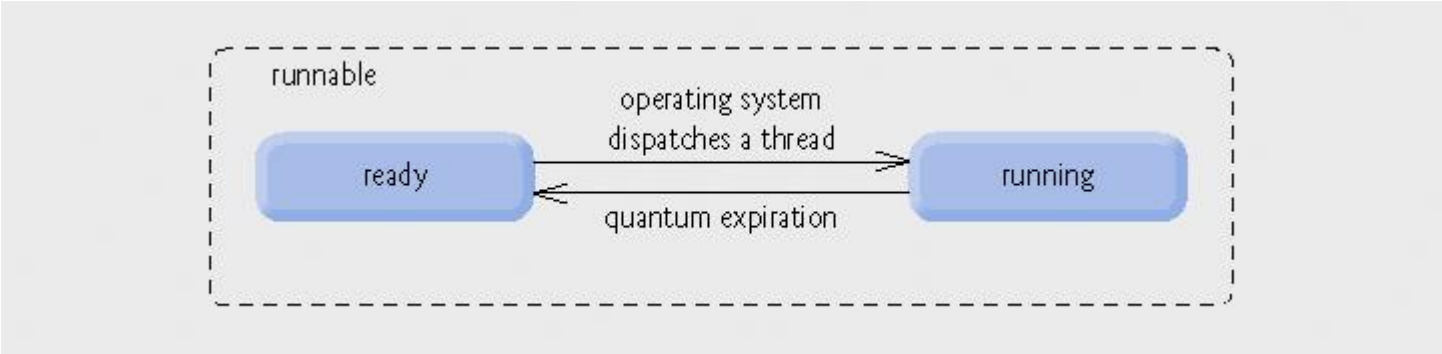
- Stanja **New** i **Runnable**: nit započinje svoj životni ciklus u New stanju i ostaje u njemu dok program ne pokrene nit, a nakon toga ulazi u Runnable stanje u kojem izvodi zadatak
- Stanje **Waiting**: nit može iz stanja Runnable prijeći u stanje Waiting u kojem čeka na to da neka druga nit obavi zadatak, kako bi nakon toga ona mogla nastaviti s radom
- Stanje **Timed Waiting**: nit može ući u stanje Timed Waiting u slučaju kad je potrebno čekati zadani vremenski interval (*sleep* interval)
- Niti u Waiting i Timed Waiting stanjima ne mogu koristiti procesor

Stanja i životni ciklus niti

- Stanje **Blocked**: nit ulazi u stanje Blocked u slučaju kad treba obaviti neki zadatak koji ne može odmah završiti, jer ovisi o nekim resursima koje su zauzele druge niti i mora čekati na njih
- Nit u stanju Blocked ne može koristiti procesor
- Stanje **Terminated**: nit ulazi u stanje Terminated kad završi svoj zadatak ili ako se tijekom izvođenja zadatka dogodi neka pogreška
- Iz stanja Terminated nit ne može prijeći u neko drugo stanje, ali se iz objekta koji označava nit mogu dohvaćati podaci koji su u njemu spremljeni

Operacijski sustav i niti

- Točan trenutak u kojem će se nit početi izvoditi ne određuje programer, već operacijski sustav, točnije njegov organizator izvođenja niti (engl. *thread scheduler*)
- Zato pokretanje niti u programskom kodu zapravo šalje nit u stanje Ready (spremno za izvođenje), a operacijski sustav u konačnici određuje kad će se ta nit izvoditi:



Prioriteti i raspoređivanje niti

- Svaka nit u Javi ima svoj prioritet koji određuje kad će se ta nit izvoditi s obzirom na ostale niti koje čekaju na izvođenje
- Niti nasljeđuju prioritete od nadređenih niti
- Niti s većim prioritetom uglavnom ranije dođu „na red za izvođenje”, ali to ovisi o internoj implementaciji operacijskog sustava
- Ako u Ready stanje uđe nit s najvišim prioritetom, operacijski sustav zaustavlja izvođenje trenutne niti, nju vraća u Ready stanje, a nit s najvišim prioritetom započinje svoje izvođenje

Sučelje Runnable

- Da bi se implementirala klasa koja predstavlja nit, ta klasa treba implementirati sučelje koje sadrži jednu metodu – **run**
- Metoda **run** sadrži programsku logiku koja se izvodi kad je nit aktivna (u stanju Runnable)
- Objekti koji predstavljalju niti moraju se kreirati i pokrenuti uz pomoć Execution frameworka
- Korištenjem statičkih metoda iz klase **Thread** moguće je prebaciti nit u stanje čekanja (Thread.sleep()) ili prekinuti izvođenje niti (Thread.interrupt())

Sučelje Runnable

```
public class PrintTask implements Runnable
{
   private final static SecureRandom generator = new SecureRandom();
   private final int sleepTime; // random sleep time for thread
   private final String taskName; // name of task

   // constructor
   public PrintTask(String taskName)
   {
      this.taskName = taskName;

      // pick random sleep time between 0 and 5 seconds
      sleepTime = generator.nextInt(5000); // milliseconds
   }
```

Sučelje Runnable

```
   // method run contains the code that a thread will execute
   public void run()
   {
      try // put thread to sleep for sleepTime amount of time
      {
         System.out.printf("%s going to sleep for %d milliseconds.%n",
            taskName, sleepTime);
         Thread.sleep(sleepTime); // put thread to sleep
      }
      catch (InterruptedException exception)
      {
         exception.printStackTrace();
         Thread.currentThread().interrupt(); // re-interrupt the thread
      }

      // print task name
      System.out.printf("%s done sleeping%n", taskName);
   }
} // end class PrintTask
```

Kreiranje i izvršavanje niti korištenjem Execution frameworka

- Za izvođenje niti u Javi koristi se Execution Framework koji uz pomoć **Executor** objekta pokreće izvršavanje **run** metoda u objektima klasa koje implementiraju sučelje Runnable
- Execution framework brine se za kreiranje i upravljanje grupom niti koja se često naziva i ***thread pool***
- Sučelje Executor sadrži jednu metodu pod nazivom **execute** koja prima Runnable tipove objekata
- Executor dodjeljuje svakom Runnable objektu jednu od slobodnih niti iz *thread poola* (ako nema slobodnih niti, kreira se nova ili se čeka dok neka nit ne postane slobodna) i tako omogućava optimiziranje broja korištenih niti

Kreiranje i izvršavanje niti korištenjem Execution frameworka

```
public class TaskExecutor
{
   public static void main(String[] args)
   {
      // create and name each runnable
      PrintTask task1 = new PrintTask("task1");
      PrintTask task2 = new PrintTask("task2");
      PrintTask task3 = new PrintTask("task3");

      System.out.println("Starting Executor");

      // create ExecutorService to manage threads
      ExecutorService executorService = Executors.newCachedThreadPool();

      // start the three PrintTasks
      executorService.execute(task1); // start task1
      executorService.execute(task2); // start task2
      executorService.execute(task3); // start task3

      // shut down ExecutorService--it decides when to shut down threads
      executorService.shutdown();

      System.out.printf("Tasks started, main ends.%n%n");
   }
} // end class TaskExecutor
```

```
Starting Executor
task1 going to sleep for 1826 milliseconds.
Tasks started, main ends.

task2 going to sleep for 3374 milliseconds.
task3 going to sleep for 2844 milliseconds.
task1 done sleeping
task3 done sleeping
task2 done sleeping
```

Sinkronizacija niti

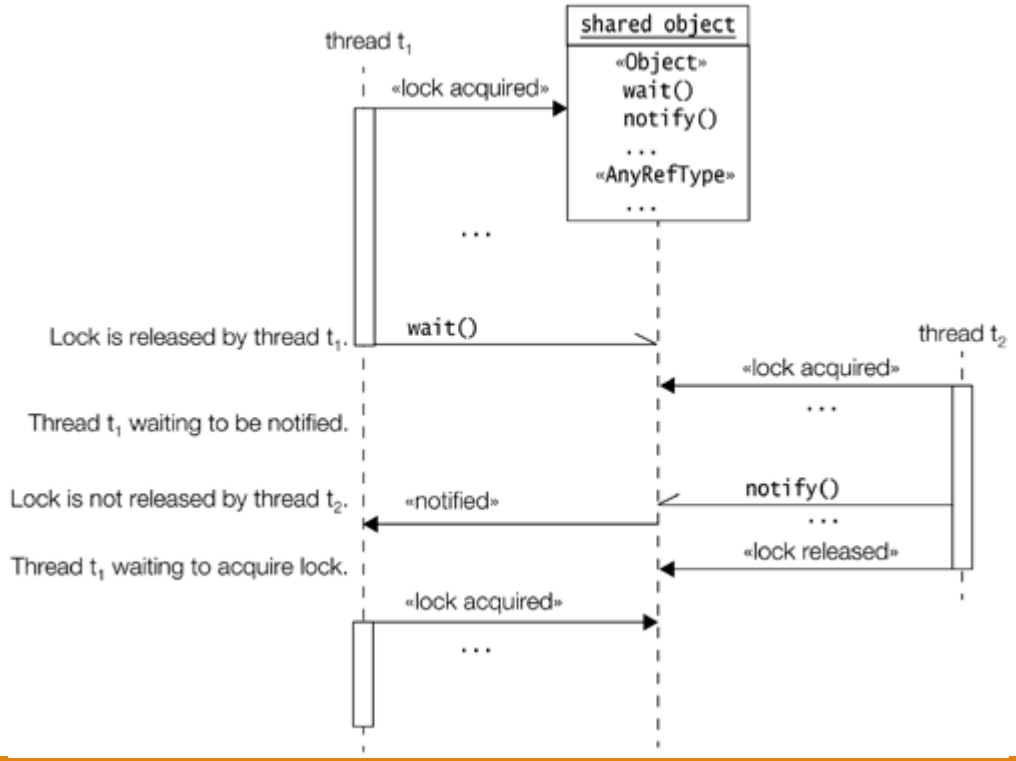
- U slučaju da više niti dijeli isti objekt i istovremeno ga mijenja, može doći do neočekivanih vrijednosti
- U takvim situacijama potrebno je koristiti mehanizme sinkronizacije koji omogućavaju kreiranje kritičnih isječaka programskog koda u kojima se u nekom trenutku može nalaziti samo jedna nit koja se izvodi
- Često se koristi i termin „monitor lock”
- Da bi nit mogla ući u kritični isječak programskog koda, mora zauzeti „monitor lock”
- U Javi se kritični isječci označavaju ključnom riječju **synchronized**
- Za čekanje u glavnom programu da završe sve niti, koristi se metoda **awaitTermination** iz objekta ExecutorService

Metode za sinkronizaciju niti

- U slučaju kad nit izlazi iz kritičnog odsječka, a sve druge niti čekaju „signal” da se mogu početi „natjecati” za ulazak u kritični odsječak, potrebno je koristiti metode za upravljanje sinkronizacijom
- U klasi Object postoji nekoliko metoda koje služe za tu namjenu:
  - **wait**: koristi se u slučaju kad nit uđe u kritični isječak (zauzme „monitor lock”) i nakon toga ustanovi da joj za nastavak rada na zadatku treba biti ispunjen neki od uvjeta (npr. slobodan resurs)
  - Tada se otpušta „monitor lock” i nit ulazi u Waiting stanje da druge niti mogu ulaziti u kritični isječak i osloboditi traženi resurs (što tvori preduvjet za nastavak rada prve niti)
  - **notify**: služi za obavještavanje da se nit iz Waiting stanja vrati u Runnable stanje
  - Može se dogoditi da ni nakon toga resurs nije slobodan pa nit ulazi ponovno u Waiting stanje

Metode za sinkronizaciju niti

- **notifyAll**: sve niti u Waiting stanju prelaze u Running stanje i pokušavaju ući u kritični isječak te provjeriti je li resurs koji im je potreban za izvršavanje zadatka slobodan



Pitanja?