Inženjerstvo serverskog sloja – Vežbe 1
Konkurentno programiranje

Konkurentno programiranje u programskom jeziku Java

Moderni procesori se sastoje od više jezgara koje mogu paralelno da izvršavaju više instrukcija iz jednog ili više programa. Jedan program može paralelno izvršavati svoje instrukcije na više jezgara korišćenjem apstrakcije niti (eng. thread). Ako nije specificirano drugačije, svaki Java program koristi samo jednu nit, što dovodi do neefikasnog korišćenja raspoloživih resursa modernog procesa. U Javi je moguće koristiti niti na dva glavna načina:

- **Nasleđivanjem klase Thread:** moguće je napisati novu klasu koja nasleđuje klasu Thread. U tom slučaju treba redefinisati metodu **run()** tako da sadrži programski kod niti. Nit se pokreće tako što se ova klasa instancira, pa se pozove metoda **start()**. Videti **thread.IncrementThread.**

- **Implementacijom interfejsa Runnable**: moguće je napisati novu klasu koja implementira interfejs Runnable. U tom slučaju treba redefinisati metodu run() tako da sadrži programski kod niti. Nit se pokreće tako što se instancira klasa Thread, koristeći konstruktor koji kao parametar prima objekat koji implementira interfejs Runnable. Proslediti implementiranu klasu ovom konstruktoru, pa onda nad novim Thread objektom pozvati metodu **start()**. Videti **thread.IncrementRunnable.**

U oba slučaja parametri i ostale vrednosti koje niti koriste moraju biti prosleđene kroz konstruktor, zato što metoda run ne može da primi parametre.

Java program neće čekati sve niti da se završe: ako program dođe do kraja izvršavanja, sve niti se zaustavljaju bez rezultata. Ukoliko je neophodno čekati da se niti izvrše do kraja, moguće je koristiti više pristupa. U nastavku su objašnjena tri ovakva pristupa:

- **Poziv metode Thread.join():** nad objektom klase Thread je moguće pozvati metodu join. Program se na ovoj liniji zaustavlja sa izvršavanjem dok se ne završi data nit. Videti **primer.Main.primerJoin().**

- **Korišćenje klase java.util.concurrent.CountDownLatch:** Moguće je instancirati objekat klase **CountDownLatch** i u konstruktoru specificirati očekivani broj niti što je ujedno i početna vrednost brojača. Pozivom metode **CountDownLatch.await()** program čeka da se brojač smanji na 0 pre nastavka izvršavanja. Objekti klase niti moraju imatu ovu referencu na ovaj objekat, kako bi mogli da pozovu **CountDownLatch.countDown()** metodu kada završe sa izvršavanjem, čime se dekrementuje vrednost brojača. Videti **primer.Main.primerLatch().**

- **Koriščenjem implementacije intefejsa java.util.concurrent.ExecutorService:** U ovom slučaju je moguće specificirati maksimalan broj niti koji se izvršava u svakom trenutku. Ovo je dobro iz dva glavna razloga: 1) niti zauzimaju memoriju i 2) veliki broj niti dovodi do velikog udela

vremena procesora u „context switching“. Runnable objekat (što uključuje i naslednice klase Thread) se prosleđuje kao parametar metode **execute**.

Glavna razlika između konkurentnog i paralelnog izvršavanja je to što se niti kod konkurentnog izvršavanja takmiče za resurse. Kod nepažljivog rukovanja deljenim resursima može doći do nepravilog ishoda, tj. *anomalija*. Da bi se anomalije izbegle, niti se moraju *sinhronizovati* strogom kontrolom pristupa deljenim resursima, uvođenjem *kritičnih regiona*. Kritični regioni su delovi koda gde je zagarantovano serijsko izvršavanje. Ovi regioni moraju biti minimalne dužine, zato što dugački regioni svode izvršavanje na serijsko. Implementacija atomičkih regiona je u Javi moguće na više načina. U nastavku su objašnjena tri ovakva pristupa:

- **Međusobno isključivi regioni / brave (eng. mutex / lock):** instanciranjem klase koja implementra Lock je moguće kontrolisati pristup. **Sve niti moraju da dele istu instancu Lock objekta da bi ovo funkcionisalo**. Pozivom **lock()** metode nit pokušava da preuzme bravu. Ako je ona već zaključana, nit mora da čeka dok se brava ne otključa. Otključavanje se vrši pozivom metode **unlock()**. Postoji više vrsta brava. Najprostija je **java.util.concurrent.locks.ReentrantLock.** Sofisticiranija varijanta je **ReentrantReadWriteLock** koja implementira **ReadWriteLock** interfejs i omogućava da se razdvoji čitanje (**readLock**) od upisa (**writeLock**).

- **Semafori:** semafor je komponenta koja kontroliše pristup ograničenom broju niti istovremeno. Sadrži celobrojnu vrednost i dve metode: **wait** i **signal**. Nit koja želi da pristupi resursu semafora poziva metodu **wait** i pokušava da dekrementuje vrednost semafora, što uspeva samo ako je vrednost > 0. U suprotnom čeka dok vrednost ne postane veća od 0, što se dešava kada druga nit pozove metodu **signal** koja inkrementira vrednost semafora. Mutex se može posmatrati kao poseban slučaj semafora gde je vrednost semafora == 1. Videti klasu **java.util.concurrent.Semaphore.**

- **Sinhronizovane metode:** dodavanjem ključne reči **synchronized** na Java metodu dovodi do toga da samo jedna nit može da pozove tu metodu u datom trenutku. Ovaj pristup nudi manje kontrole od mutex / lock pristupa, ali je lakše za implementaciju.

- **Atomičke vrednosti:** procesori nude atomičke operacije nad određenim tipovima podataka. Videti paket **java.util.concurrent.atomic**. Primer je **java.util.concurrent.atomic.AtomicInteger**, sa glavnim metodama **get, set I compareAndSet.** Metoda **compareAndSet** omogućava da se u jednom koraku (jednoj procesorskoj instrukciji) proveri vrednost i postavi nova ako je uslov zadovoljen.

Korišćenje međusobno isključivih regiona može dovesti do večnog blokiranja niti koje čekaju na resurs. Ovo se najčešće događa tako što 1) nit nikada ne otključa Lock objekat zbog greške u kodu ili 2) kod se izvršava korektno, ali postoji ciklična zavisnost dve niti. Ovaj fenomen se zove *mrtva petlja (eng. deadlock)***.** Jedan od najpoznatijih problema je problem filozofa za ručkom, definisan u nastavku:

Pet filizova ručaju za stolom. Svaki filizof ima ispred sebe tanjir, a između svakog tanjira se nalazi viljuška. Da bi filozof mogao da ruča, mora da uzme viljušku sa svoje desne, pa viljušku sa svoje leve strane u sledećim koracima:

1. Misli nasumično vreme pa uzmi levu viljušku, ako je dostupna.
2. Misli nasumično vreme pa uzmi desnu viljušku, ako je dostupna.
3. Ako su obe viljuške u rukama, ručaj.
4. Vrati levu viljušku.
5. Vrati desnu viljušku.
6. Počni od koraka 1.

Zadaci

1. Implementirati klasu ValueHolderSync koja sinhronizuje niti pomoću sinhronizovane metode.
2. Implementirati klasu ValueHolderLock koja sinhronizuje niti pomoću brave:
    a. Koristiti ReentrantLock klasu.
    b. Koristiti ReentrantReadWriteLock klasu.
3. Implementirati klasu ValueHolderAtomic koja sinhronizuje niti pomoću atomičkih tipova podataka.
4. Implementirati klasu Semaphore koja kontroliše pristup klasi ValueHolder. Postaviti početnu vrednost semafora na 1.
5. Implementirati rešenje za problem filozofa za ručkom.