Mrezno racunarstvo

Java, niti

Niti (eng. threads)

- delovi koji su relativno nezavisni jedni od drugih
- njihovo izvršavanje može biti efikasnije ako se vremenski preklope
- Niti to obezbeđuju
- Mašina sa više procesora – koliko procesora toliko konkurentnih izračunavanja
- Jedan procesor – moguće je preklopiti ulazno/izlazne operacije sa procesiranjem

Niti

- Još jedna primena: npr. za preklapanje izvršavanja animacije i drugih aktivnosti u istom programu
- Apleti se izvršavaju unutar jednog programa – našeg browser-a, a niti omogućuju konkurentno izvršavanje većeg broja apleta.

Primer

- Čitanje određenog broja blokova podataka iz fajla
- Neka izračunavanja nad svakim blokom
- Upisivanje rezultata u drugi fajl

Figure 16-1 (724. str) Horton

Ulazno-izlazne operacije zahtevaju relativno malo procesorskog vremena dok se izvršavaju

Niti

- Svaki Java program ima bar jednu nit
- Preveliki broj niti može uvećati ukupno vreme izvršavanja programa zbog implicitnog vremena potrebnog za kontrolu i prelazak sa jedne niti na drugu.
- Neophodna su i sredstva pomoću kojih niti mogu međusobno da komuniciraju (pre početka računanja, blok mora biti učitan; pre početka pisanja, mora biti završeno izračunavanje)

Kreiranje niti

- Program ima uvek bar jednu nit: onu koja se kreira kada počne izvršavanje (ta nit za aplikaciju počinje na početku main(), dok je za aplet browser glavna nit)
- Niti su objekti klase java.lang.Thread ili neke njene potklase
- Startovanje niti vrši se pozivom metoda start() nad objektom koji predstavlja nit
- Time se započinje izvršavanje odgovarajućeg run() metoda niti

Niti

- Nitima upravlja operativni sistem i on je jedini koji može kreirati i pokrenuti novu nit
- Kada bismo sami pozivali run() metod niti, a ne preko start(), on bi se ponašao kao svaki običan metod i izvršavao u istoj niti u kojoj je i pozvan

Kreiranje niti

- Klasu koja predstavlja niti možemo kreirati na dva načina:
  1. kao potklasu od Thread, uz predefinisanje metoda run()
  2. kao klasu koja implementira Runnable interfejs (koji deklariše run() metod) (potom se kreira Thread objekat u programu tamo gde je potreban)

Primer: potklasa od Thread

- primer1
  - Daemon i User niti
- setDaemon(true);
- Demonska nit je nit koja se izvršava u pozadini. Kada se završi nit koja ju je kreirala, završava se i demonska nit
- Niti koje nisu demonske nazivaju se korisničkim. One su nezavisne od niti koje su ih kreirale. Nastavljaju da se izvršavaju i nakon što se završe niti koje su ih kreirale
- Niti koje se neograničeno izvršavaju obično se definišu kao demonske

- setDaemon() se može zvati samo pre startovanja niti. Inače dovodi do IllegalThreadStateException
- Nit koju je kreirala demonska nit i sama će biti demonska
- Sve 3 niti kreirane u primeru su demonske

Implementiranje run() metoda

- sleep() metod (klase Thread)
- može izbaciti InterruptedException
- Suspenduje izvršavanje niti za onoliko milisekundi koliko se zada argumentom
- Time se daje šansa drugim nitima da se izvršavaju
- Izvršavanje read() metoda u main() blokira dok ne pritisnemo Enter, ali za to vreme se izvršavaju druge niti
- Nakon što pritisnemo Enter, omogućuje se nastavljanje main() niti koja se onda završava
- Pošto su ostale niti demonske, i one se takođe završavaju

Zaustavljanje niti

- Nit može signalizirati drugoj niti da treba da se zaustavi pozivajući njen interrupt() metod
- To, samo po sebi, ne zaustavlja nit, već samo postavlja fleg koji kaže da postoji zahtev za prekidom
- sleep() metod proverava da li je taj fleg postavljen i u slučaju da jeste izbacuje InterruptedException

Modifikacija primera

- zakomentarisati setDaemon() za sve 3 niti
- u main() nakon ”Enter pressed...”

  dodati .interrupt() za sve 3 niti
- sleep() metod sve 3 niti izbacuje odgovarajući izuzetak koji se hvata u run() metodu. Pošto je catch izvan while petlje run() metod svake niti se završava, čime se završava i sama nit.

- isInterrupted() metod proverava da li je za neku nit pozvan interrupt() metod
- time se samo proverava da li je postavljen odgovarajući fleg, ne i da li se nit još uvek izvršava
- moguće je da je fleg postavljen, ali je izvršavanje nastavljeno – nit nije u obavezi da se završi jer je pozvan interrupt()
- isAlive() metod vraća true ako se nit nije završila

- isInterrupted() metod nema uticaja na fleg (ako je bio postavljen, ostaće)
- statički metod interrupted() klase Thread testira da li je tekuća nit koja se izvršava prekinuta, i ako jeste, čisti fleg i vraća true
- nakon izbacivanja izuzetka InterruptedException, fleg se čisti pa naredni pozivi isInterrupted() i interrupted() vraćaju false

Povezivanje niti

- Ako je u jednoj niti potrebno da se čeka da neka druga nit umre, poziva se join() metod za tu nit koja treba da se okonča
- thread1.join();

  čeka se neograničeno dugo, sve dok se nit thread1 ne okonča (suspenduje se tekuća nit sve dok nit thread1 ne umre)
- thread1.join(1000);

  argument tipa long: broj milisekundi
- može izbaciti InterruptedException ako je tekuća nit prekinuta nekom drugom (try-blok)

Thread Scheduling

- Svaka nit će sigurno dobiti šansu da se izvršava dok ostale ”spavaju”
- preemptive multitasking (o.s.) – program će raditi i bez sleep() metoda u run()
- inače, prva nit će zauzeti procesor i izvršavaće se neograničeno (bez sleep())

- yield() metod klase Thread
- daje drugim nitima šansu da se izvršavaju
- izvršavaće se ako ih ima koje čekaju, ali ne želimo da suspendujemo tekuću nit na određeno vreme
- Kada zovemo sleep() nit se neće izvršavati bar zadato vreme, čak i ako nema niti koje čekaju na izvršavanje
- yield() uzrokuje da nit nastavi odmah sa izvršavanjem kada nema drugih niti koje čekaju

Implementiranje Runnable interfejsa

- alternativa definisanju potklase od Thread
- moguće je izvesti klasu iz neke druge (ne iz Thread) a da i dalje predstavlja niti
- Runnable interfejs deklariše samo metod run() koji se izvršava kada se pokrene nit

Primer2:

- Iste članice
- Konstruktor je skoro isti, nije moguće u njemu zvati setDaemon() jer klasa više nije izvedena iz Thread. To se sada čini u main() nakon kreiranja objekata koji predstavljaju niti
- Implementacija run() metoda je slična. Klasa nema sleep() metod kao član, ali se poziva statički metod klase Thread.
- U main se objekti koji predstavljaju niti kreiraju korišćenjem konstruktora klase Thread koji prihvata objekat tipa Runnable kao argument.

Imena niti

- konstruktor sa 2 argumenta, drugi je String koji predstavlja ime koje dajemo niti (moguće je da veći broj niti ima isto ime, a to ime se koristi samo za prikaz informacija o niti)
- getName()
- setName()

Upravljanje nitima

- U primerima do sada, niti su pokretane u puštane da se nadmeću za računarske resurse.
- Obično je potrebno kontrolisati kako se izvršavaju kako se ne bi mešale
- Kada dve ili više niti dele isti resurs poput fajla ili bloka memorije neophodno je preduzeti mere da jedna nit ne menja resurs dok ga koristi neka druga nit.
- Jedan način za razrešavanje ovog problema je sinhronizacija

Sinhronizacija

- osnovne mogućnosti
- specijalizovano:
  - java.util.concurrent
  - java.util.concurrent.atomic
  - java.util.concurrent.locks
- Cilj sinhronizacije je obezbediti da kada nekoliko niti želi da pristupi istom resursu, u jednom trenutku samo jedna ima pristup

Sinhronizacija

- sinhronizacija metoda
- sinhronizacija bloka
    - Sinhronizacija metoda
- ključna reč synchronized
- samo jedan (sinhronizovani) metod se može izvršavati u datom trenutku nad zadatim objektom

Sinhronizacija metoda

- Proces sinhronizacije koristi interni ”lock” koji je pridružen uz svaki objekat. To je neka vrsta flega koji postavlja proces (kada sinhronizovani metod započinje izvršavanje)
- Svaki sinhronizovani metod za objekat proverava da li je neki drugi metod postavio lock, pa ako jeste ne započinje izvršavanje dok se lock ne ukloni

Sinhronizacija metoda

- Ne postoji uslov da se ne mogu simultano izvršavati sinhronizovani metodi nad različitim objektima klase
- Kontroliše se samo konkurentni pristup jednom objektu
- Metodi koji nisu deklarisani kao sinhronizovani mogu se izvršavati uvek, bez obzira da li je u toku izvršavanje sinhronizovanog metoda nad objektom ili ne u nekoj drugoj niti

Sinhronizacija statičkih metoda

- Ako je sinhronizacija primenjena na statičke metode, u svakom trenutku može se izvršavati samo jedan od njih

Korišćenje sinhronizovanih metoda

- Primer jednostavnog modela banke
- Inicijalno: postoji samo jedan račun
- 2 službenika
- 1 radi isplate, drugi radi uplate
- banka u modelu je zapravo računar koji izvršava operacije na računu
- svaki službenik direktno komunicira sa bankom

Primer 3

- 4 klase:
  1. Banka – predstavlja računar banke
  2. Račun
  3. Transakcija – uplata ili isplata
  4. Službenik
  5. Test-klasa

klasa Banka

- ne čuva lokalno nikakve podatke
- ima samo jedan metod koji obavlja transakciju
- objekat koji predstavlja transakciju ima sve neophodne podatke: koja transakcija je u pitanju i na koji račun se primenjuje
- podržane su samo uplate i isplate (jednostavno se može proširiti i na druge vrste transakcija)
- obe podržane vrste transakcija uključuju neko čekanje koje se simulira metodom sleep() klase Thread (za to vreme mogu se izvršavati druge stvari u drugim nitima)
- nema instancnih promenljivih – nema konstruktora

klasa Transakcija

- tip transakcije (uplata – isplata)
- suma
- račun
- metodi su prilično pravolinijski (samo get*() koji se koriste u klasi Banka i toString())

klasa Racun

- takođe jednostavna
- stanje i broj računa
- metodi za očitavanje i postavljanje stanja
- operacije na računu se obavljaju eksterno, od strane banke

klasa Sluzbenik

- informacije o banci (u kojoj je zaposlen)
- detalji o tekućoj transakciji
- svaki službenik radi nezavisno od ostalih, pa će oni predstavljati odvojene niti
- Sluzbenik je nit (klasa implementira Runnable interfejs)

klasa Sluzbenik

- svaki službenik ima polje za čuvanje jedne transakcije. Kada to polje nije null, službenik je zauzet
- transakcija se smešta u odgovarajuće polje pozivom odgovarajućeg metoda
- pozivom metoda zauzet() može se proveriti da li je službenik zauzet
- metod run(): sve dok je polje transakcije prazno, nema posla, pa nakon malo spavanja, petlja ponovo proverava da li je došlo do promena u polju transakcije, pa ako jeste, poziva se metod banke da izvrši transakciju, a polje transakcije postavlja na null.

Test-klasa (BankarskeOperacije)

- main() inicijalizuje sve i generiše transakcije i prosleđuje ih službenicima
- (1 račun, 2 službenika)

1. kreiranje računa, banke i službenika...
2. kreiranje niti službenika kao demona i njihovo pokretanje
3. generisanje transakcija i njihovo prosleđivanje službenicima
4. čekanje da oba službenika završe
5. štampanje rezultata

Test-klasa (BankarskeOperacije)

- promenljive u main() metodu prate ukupne isplate i isplate i inicijalno stanje na računu
- broj izvršavanja uplata i isplata se takođe čuva u jednoj promenljivoj (ukupan broj transakcija je *2)
- uplate: slučajne svote od 50 do 75
- pre prosleđivanja transakcije službeniku, moramo biti sigurni da on nije zauzet, inače bismo prepisali transakciju preko njegove tekuće. Zato imamo while-petlju: sve dok zauzet() vraća true, poziva se metod sleep() sa čekanjem od 25 milisekundi pre ponovne provere. Kada zauzet() vrati false, zove se metod službenika za izvršavanje transakcije
- Treća while-petlja radi isto, samo dok je bilo koji od službenika zauzet

primer

- nakon pokretanja, konačno stanje biće pogrešno
- problem je što oba službenika istovremeno rade na istom računu
- oba zovu metod banke za izvršavanje transakcije
- odvojeni pozivi istog metoda se preklapaju

popravka

- jedan način da se program popravi jeste da se metod banke učini sinhronizovanim (samo se ispred njega doda ključna reč synchronized)
- time se sprečava da jedan njegov poziv bude izvršen dok je drugi još u toku
- Iako ovo rešava problem, banka je jako neefikasna. Dok jedan službenik radi, drugi ne može. U svakom trenutku najviše jedan službenik radi

Sinhronizacija blokova naredbi

- moguće je postaviti ”lock” na proizvoljan objekat za dati blok naredbi
- kada se izvršava blok koji je sinhronizovan za dati objekat, ne može se izvršavati nijedan drugi blok koji je sinhronizovan za taj objekat
- synchronized(theObject)

  statement;

Primer 4: više računa

- izmene u main()
- u petlji se kreira niz računa
- broj računa određen je brojem inicijalnih stanja u odgovarajućem nizu
- brojevi računa kreću od 1
- sada postoje nizovi: inicijalnih stanja, ukupnih isplata i uplata za svaki račun

Primer 4: više računa

- deklarisanjem metoda banke kao synchronized, program je značajno ograničen: nijedna operacija se ne može izvršavati dok je neka druga u toku
- to je nepotrebno restriktivno jer nema razloga da se sprečava transakcija na jednom računu dok je u toku transakcija na drugom

Primer 4: više računa

- ono što mi zaista želimo jeste program koji sprečava preklapanje operacija na istom računu
- ovde je od pomoći deklarisanje sinhronizovanih blokova naredbi na određenom objektu
- metod klase Banka: želimo da sprečimo simultano procesiranje istog računa, a ne da zabranimo procesiranje drugih računa
- sinhronizujemo kod za procesiranje transakcije na objektu klase Račun nad kojim se vrši transakcija

- blokovi sinhronizovani u odnosu na određeni objekat ne moraju biti u istoj klasi. Mogu biti bilo gde u programu
- neophodno je pomeriti kod za pristup i restauriranje stanja na računu tako da budu unutar sinhronizovanih blokova
- radi provere da zaista dolazi do preklapanja operacija može se dodati kod koji ispisuje tip transakcije i informacije o računu na početku i kraju svakog sinhronizovanog bloka

- U opštem slučaju, jako je teško biti siguran da je program koji koristi niti adekvatno istestiran
- Najbitniji je pažljivi dizajn
- Nikada ne možemo biti sigurni da je program 100% korektan, već samo da radi korektno veći deo vremena!

Deadlock-ovi

- uzajamna međuzavisnost dveju niti
- jedna izvršava neki kod sinhronizovan na datom objektu, theObject, recimo, a zatim treba da izvrši metod koji sadrži kod sinhronizovan na drugom objektu, theOtherObject, npr. Pre nego što se ovo desi, druga nit izvršava kod sinhronizovan da theOtherObject i treba da izvrši metod koji sadrži kod sinhronizovan na objektu theObject.
- Nijedna nit nema mogućnost da nastavi
- Otkrivanje i popravljanje ovakve vrste problema može biti jako teško

Komunikacija između niti

- Sinhronizacija daje određeni stepen kontrole, ali i dalje unosi neefikasnosti
- petlje u prethodnom primeru (za čekanje dok službenik ne završi transakciju: npr. nismo mogli proslediti službeniku novu transakciju dok je on zauzet prethodnom)
- Postoji mnogo bolji način od korišćenja while-petlje za testiranje zauzetosti službenika s vremena na vreme i zvanja sleep() metoda u međuvremenu

wait(), notify(), notifyAll()

- Klasa Object definiše metode wait(), notify() i notifyAll()
- Sve klase su izvedene iz Object pa mogu koristiti ove metode
- Ovi metodi se mogu pozivati samo iz sinhronizovanog metoda ili bloka
- inače izbacuju IllegalMonitorStateException

wait()

- suspenduje tekuću nit dok se ne pozove notify() ili notifyAll() za objekat za koji je pozvan wait()
- Kada se pozove bilo koja (od 3) verzija wait() metoda, nit oslobađa ”lock” koji ima na objekat, tako da drugi metod ili blok sinhronizovan na isti objekat može da se izvrši (ovim se omogućuje da druga nit zove notify(), notifyAll() ali i wait() za isti objekat)
- try-blok (InterruptedException) – sve verzije wait() metoda

wait(long timeout)

- suspenduje tekuću nit dok ne istekne timeout milisekundi ili dok se ne pozove notify() ili notifyAll() za objekat kome pripada wait() ako se to desi ranije
- wait(long timeout, int nanos)

notify()

- restartuje nit koja je pozvala wait() metod za objekat kome pripada notify() metod
- ako je nekoliko niti pozvalo wait(), nemamo kontrolu nad tim koja je obaveštena, pa je bolje zvati notifyAll()
- ako nema niti koje čekaju, ništa se ne dešava

notifyAll()

- restartuje sve niti koje su pozvale wait() za metod kome pripada metod notifyAll()

- Glavna ideja wait() i notify() metoda je da obezbede način da metodi ili blokovi sinhronizovani na određenom objektu komuniciraju
- Blok može pozvati wait() da suspenduje svoju operaciju dok neki metod ili drugi blok sinhronizovan na istom objektu ne izvrši neku promenu na njemu i pozove notify() da izvesti da je promena gotova.
- Nit tipično zove wait() jer neko svojstvo sinhronizovanog objekta nije postavljeno ili neki uslov nije zadovoljen, što zavisi od akcije neke druge niti
- Prosto: resurs je zauzet jer ga menja druga nit

wait()

- Glavna razlika između sleep() i wait() je u tome što wait() oslobađa sve objekte na koje tekuća nit ima ”lock”, a sleep() ne
- tipična upotreba wait():

```
synchronized(anObject){
  while(condition_not_met)
      anObject.wait();
  // Condition is met, so continue…
}
```

wait()

- Nit će suspendovati operaciju kada se pozove metod wait() dok neka druga nit sinhronizovana na istom objektu ne pozove notify() (notifyAll())
- while-petlja se nastavlja i proverava uslov ponovo
- on ponovo može da ne bude ispunjen, pa se ponovo zove wait(), tako da druga nit može raditi na anObject.
- wait() dakle ne služi samo za pristup objektu, već omogućava pristup drugim nitima dok se ne ispuni neki uslov

notifyAll()

- bolje je koristiti notifyAll() nego notify() kada ima više od 2 niti sinhronizovane na istom objektu
- sa notify() biće startovana jedna, ali nemamo kontrolu koja
- a može se desiti da ta ponovo pozove wait() jer nije ispunjen uslov koji ona zahteva
- onda sve niti jedna drugu čekaju, bez mogućnosti da nastave

Primer 5, banka, wait(), notifyAll()

- wait() i notifyAll() za oslobađanje while-petlji iz prethodnog primera
- main() – 2 while-petlje koje proveravaju da li je službenik zauzet
- izmena klase Sluzbenik tako da može da koristi wait() i notifyAll()

klasa Sluzbenik

- želimo da učinimo metod za izvršavanje transakcije klase Sluzbenik svesnim stanja polja transakcije
- Ako ono nije null, želimo da metod čeka dok polje ne postane null
- Da bismo koristili wait(), moramo sinhronizovati blok ili metod na objekat, u ovom slučaju objekat klase Sluzbenik (jer smo zainteresovani za njegovo polje)
- možemo sinhronizovati metod da bismo to uradili
- kada je polje null, čuva se transakcija i poziva notifyAll() kako bi se obavestile ostale niti koje čekaju na promenu ovog Sluzbenik objekta
- ako polje transakcije nije null, metod čeka dok neka druga nit ne pozove notifyAll() kao signal promene Sluzbenik objekta

- Sada treba razmotriti gde polje transakcije može biti promenjeno
- Odgovor je u run() metodu klase Sluzbenik, pa se menja i on
- sinhronizuje se metod run() i čeka ako je polje transakcije null.
- Može delovati čudno da se dva metoda iste klase sinhronizuju nad istim objektom koji ih poseduje, ali ova 2 metoda se izvršavaju u različitim nitima

Primer 5, banka, wait(), notifyAll()

- main() izmene
- brišu se 2 while-petlje koje čekaju da se službenici oslobode

- sa malom izmenom metoda za ispitivanje da li je službenik zauzet, moguće je izbeći potrebu za while-petljom pre ispisa rezultata
- metod se završava samo kada službenik nema transakciju na čekanju, pa više nije neophodna povratna vrednost metoda. While-petlja se zamenjuje sa dva poziva ovog metoda – po 1 za svakog službenika

- metod službenika za izvršavanje transakcije poziva wait() metod ako polje transakcije sadrži referencu na objekat, što znači da službenik obrađuje transakciju. Tekuća nit (a to je main-nit) je suspendovana dok se notifyAll() ne pozove iz run() metoda službenika da ukaže da je došlo do promene
- pošto je run() metod takođe sinhronizovan na službenika, on takođe može pozvati wait() kada je u polju transakcije null, pošto to ukazuje da nema posla koji čeka. Poziv metoda za obradu transakcije rezultovaće smeštanjem transakcije u odgovarajuće polje, a notifyAll() će probuditi run() metod da nastavi sa izvršavanjem

- kako je metod za proveru zauzetosti deklarisan kao sinhronizovan, moguće je zvati wait() za suspendovanje tekuće niti ako su transakcije još u toku

Prioriteti niti

- svaka nit ima prioritet koji određuje koja će se nit izvršavati ako ima više niti koje čekaju
- ako jedna nit u programu zahteva sve resurse, a druge zahtevaju relativno malo resursa
- damo zahtevnoj niti niži prioritet i time obezbedimo da se i ostale niti izvršavaju

Prioritet niti

- moguće vrednosti prioriteta
- statički članovi klase Thread, final, int
- MAX_PRIORITY 10
- MIN_PRIORITY 1
- NORM_PRIORITY 5
- Kada kreiramo novu nit, ona podrazumevano ima isti prioritet kao nit koja ju je kreirala
- setPriority() (IllegalArgumentException)
- getPriority()

Korišćenje prioriteta niti

- clerk1Thread.setPriority(Thread.MIN_PRIORITY);
- clerk2Thread.setPriority(Thread.MAX_PRIORITY);
- ovo neće imati puno uticaja jer svaki službenik može da obrađuje samo po jednu transakciju, a one se alternirajuće dodeljuju službenicima

- omogućimo Službeniku da ima red transakcija
- LinkedList<Transakcija>
- (Samo klasa Vector<> je thread-safe od svih kolekcijskih klasa) (bezbedna za upotrebu od strane više niti)
- Moguće je ostale koristiti sa thread-safe wrapper-om.
- Operativni sistem takođe treba da podržava prioritet niti da bi on imao efekta

Primer 5:

- Klasa Sluzbenik se može proširiti tako da rukuje većim brojem transakcija tako što se polju transakcije pridruži kapacitet za smeštanje nekoliko transakcija u listu
- Klasa java.util.Collections obezbeđuje metode za kreiranje sinhronizovanih skupova, listi i mapa od nesinhronizovanih objekata.
- Statički metod synchronizedList() prihvata argument tipa List<T> i vraća referencu tipa List<T> na sinhronizovani objekat kolekcijske klase
- Moguće je definisati objekat kao povezanu listu, a onda iskoristiti synchronizedList() da ga učinimo sinhronizovanim za čuvanje transakcija

- menja se i metod za obradu transakcije u klasi Sluzbenik tako da smešta transakcije u listu sve dok se ne napuni (imamo max broj transakcija)
- transakcija se može dodati u listu samo ako je tamo manje od max broja transakcija

- size() metod za listu vraća broj objekata trenutno smeštenih u njoj
- add() dodaje objekat na kraj liste

- run metod dohvata objekte iz liste
- remove() uklanja objekat sa zadate pozicije
- provera zauzetosti službenika se takođe menja – službenik nije zauzet ako nema transakcija u listi
- postaviti i vremena čekanja na iste vrednosti za uplate i isplate (u klasi Banka)