Johdatus Java 1.5:n semaforeihin

Anne Pääkkö, Petri Vuorio
Helsingin yliopisto
Tietojenkäsittelytieteen laitos
2006

Sisällys

Mitä ovat semaforit? (Lyhyt oppimäärä)

Samanaikaisesti suoritettavien prosessien yhteistoimintaan liittyy monenlaisia ongelmia ja haasteita, jotka on tavalla tai toisella ratkaistava. Esimerkiksi mikä on oikea suoritusjärjestys, milloin prosessin vaihto tehdään, saako sitä edes tehdä, miten ja milloin yhteisiä muuttujia saadaan muuttaa ja lukea, onko mahdollista, että jokin saa aikaan prosessien lukkiutumisen tai jonkun prosessin nälkiintymisen jne. Yksi tärkeimmistä rinnakkaisuuden ongelmista on kriittisen vaiheen ongelma. Kriittiseksi vaiheeksi kutsutaan prosessin sitä osaa, joka täytyy suorittaa alusta loppuun ilman keskeytyksiä, ilman että toinen prosessi tekee samaan aikaan mitään muuta. Semaforit ovat yksi tämän poissulkemis- (*mutual exclusion*) ja synkronointiongelman ratkaisuvaihtoehto.

Hollantilainen Edsger Dijkstra esitteli semaforit jo 1960-luvulla. Semaforit ovat siis melko vanha käsite. Ne ovat kuitenkin edelleen yleisesti käytettyjä, koska ne ovat yksinkertaisia, mutta toimivia. Semaforit tarjoavat työkalun rinnakkaisuuden hallintaan konekieltä ylemmällä tasolla.

Yksinkertaistettuna semaforien idea voidaan ymmärtää eräänlaisena lupien antajana. Semafori pitää sisällään tiedon lupien määrästä. Dijkstran määrittelemässä semaforissa sen arvo on aina positiivinen. Kun prosessi haluaa suoritusvuoron, se pyytää semaforilta lupaa. Jos semaforilla on lupia jäljellä, prosessille myönnetään se ja prosessi jatkaa suoritustaan. Jos sen sijaan lupaa ei ole myönnettävissä, prosessin suoritus keskeytetään ja se voi jatkaa vasta, kun lupa voidaan myöntää. Suoritettuaan kriittisen vaiheensa prosessi vapauttaa saamansa luvan viestittämällä semaforille, jolloin vapautettu lupa voidaan myöntää toiselle lupaa odottavalle prosessille..

Semaforin operaatioita merkitään nimillä

- *WAIT(S)* tai *P(S)* (tulee hollanninkielisestä sanasta *proberen*, testaa), jolla siis pyydetään semaforilta lupaa ja tarvittaessa odotetaan, kunnes lupa voidaan myöntää.
- *SIGNAL(S)* tai *V(S)* (tulee hollanninkielisestä sanasta *verhogen*, kasvata), jolla vapautetaan saatu lupa, ts kasvatetaan myönnettävissä olevien lupien määrää.

Operaatioita käytetään esimerkiksi seuraavasti:

```
ei-kriittinen vaihe
wait(S)
kriittinen vaihe
signal(S)
ei-kriittinen vaihe jatkuu
```

Semaforit pystytään jakamaan käyttötavoiltaan. Yleiset semaforit (*general semaphore*) ovat niitä tavallisesti käytettyjä semaforeja, jossa kokonaislukumuuttuja on mikä tahansa ei-negatiivinen luku. Semafori, jonka kokonaislukumuuttuja saa arvokseen vain 0 tai 1, kutsutaan binäärisemaforiksi (*binary semaphore*). Näitä käytetään yleensä kahden prosessin synkronoinnissa. Jaetuiksi semaforeiksi (*split semaphore*) puolestaan kutsutaan semaforien käyttötapaa, jossa käytetään vähintään kahta semaforia, joiden arvojen summa on aina vakio.

Yleinen semafori on tyypiltään ns. *heikko suspend-and-wait* -semafori. Heikko semafori ei määrittele, missä järjestyksessä odottavat prosessit pääsevät jatkamaan suoritustaan, minkä seurauksena nälkiintyminen on mahdollista. Ns. vahvassa semaforissa ei tätä ongelmaa ole, koska luvat jaetaan pyytämisjärjestyksessä. Jos lupaa pyytävälle prosessille ei välittömästi voida antaa lupaa, *suspend-and-wait* -semafori siirtää prosessin odottavaan tilaan, ts aiheuttaa prosessinvaihdon. Ns. *busy-wait* -semafori ei puolestaan aiheuta prosessinvaihtoa, vaan tarkistaa koko ajan silmukassa, onko lupaa saatavilla.

Semaforien käyttö Javalla

Java 1.5:n *Semaphore*-luokka (*java.util.concurrent* -paketissa) toteuttaa semaforit. Javan semaforit on toteutettu hyvin yleiskäyttöisiksi, mistä johtuen toteutus poikkeaa hieman yleisestä semaforista. Yleiskäyttöisyyden ansiosta semaforien eri käyttötekniikat ovat helposti sovellettavissa Semaphore-luokkaa käyttäen.

Uutta semaforia alustettaessa sille annetaan parametrina lupien määrä, esim.

```
import java.util.concurrent.Semaphore;
// ...
Semaphore kissa = new Semaphore(42);
```

alustaa semaforin, jolla on 42 lupaa. Erillistä binäärisemaforia ei Javassa ole olemassa, vaan sama toiminnallisuus saadaan aikaiseksi alustamalla semaforin arvo ykköseksi. Toisin kuin Dijkstran määrittelemät semaforit, Javan toteutuksessa semaforin arvo voidaan alustaa myös negatiiviseksi, jolloin lupia täytyy aluksi ”palauttaa” (signal), ennen kuin semafori voi niitä myöntää (ts. suoritusta voidaan jatkaa wait-operaatiosta).

Lupien määrän lisäksi semafori voidaan alustushetkellä määrittää joko heikoksi tai vahvaksi, esim.

```
import java.util.concurrent.Semaphore;
// ...
Semaphore kissa = new Semaphore(42, true);
```

luo uuden vahvan semaforin. Vastaavasti totuusarvo *false* loisi heikon semaforin. Vahvoja semaforeja käytettäessä on syytä huomioida, että vielä wait-operaation sisälläkin saattaa tapahtua säikeenvaihto, ennen kuin wait-operaatiota kutsunut säie on lisätty lupaa odottavien säikeiden jonoon. Tämän seurauksena jokin toinen säie saattaa päästä ”etuilemaan” lupaa odottavien säikeiden jonossa, vaikka olisikin kutsunut wait-operaatiota ensin mainittua säiettä myöhemmin.

Semaphore-luokan ilmentymien voitaneen olettaa olevan yleisen semaforin tapaan *suspend-and-wait* -tyyppisiä. Javan API:ssa tähän ei kuitenkaan oteta kantaa.

Semaphore-luokan operaatioiden nimet poikkeavat perinteisistä: *acquire()*-metodi vastaa p/wait -operaatiota, *release()*-metodi vastaa v/signal -operaatiota. Koska acquire()-metodissa odottaminen saattaa keskeytyä myös säikeen saaman ulkopuolisen keskeytyksen (*InterruptedException*) seurauksena, on metodia kutsuttava *try-catch* -lohkon sisällä. Näin ollen edellä alustetun semaforin käyttö tapahtuu esim. seuraavasti:

```
// ei-kriittinen vaihe
try {
  kissa.acquire(); // wait-operaatio
} catch (InterruptedException e) {
}
// kriittinen vaihe
kissa.release(); // signal-operaatio
// ei-kriittinen vaihe jatkuu...
```

Samalla kertaa voidaan pyytää tai vapauttaa useampi lupa antamalla lupien määrä acquire()- ja release()-metodien parametrina, esim. kissa.acquire(7) tai kissa.release(7). Tällöin acquire()-metodi palaa vasta, kun kaikki luvat on saatu.

Perusoperaatioiden lisäksi Semaphore-luokan *tryAcquire()*-metodilla voidaan pyytää lupaa (tai useita lupia) jäämättä odottamaan, jos pyydettyä lupaa ei voida samantien myöntää. Meto-

di antaa heti pyydetyt luvat, vaikka lupaa odottavien säikeiden jonossa olisikin muita säikeitä. Se ei siis huomioi, onko semafori määritelty reiluksi vai ei. Pyydettäessä useita lupia samanaikaisesti metodi ei anna yhtäkään lupaa, jos kaikkia pyydettyjä lupia ei voida myöntää. Metodia kutsuttaessa voidaan antaa myös aikakatkaisuparametri *(timeout),* jonka ajan lupia enintään odotetaan, mikäli niitä ei heti voida myöntää.

Toinen erikoisempi metodi on *drainPermits(),* joka antaa kaikki heti saatavissa olevat luvat, ts. asettaa lupien määrän nollaksi, mikäli se oli positiivinen. Lisäksi Semaphore-luokka tarjoaa useita metodeja semaforin tilan tutkimiseksi. Näiden kaikkien metodien kuvaukset löytyvät luokan API:sta.

Käytännön esimerkkejä

On olemassa useita tunnettuja rinnakkaisuuteen liittyviä ongelmia, joiden ratkaisemiseksi on monia työkaluja ja algoritmeja. Alla on esitelty kaksi tunnettua ongelmaa ja niihin yhdenlaiset ratkaisut, joissa ratkaisuun käytetään Javan semaforeja. Esimerkkikoodina on esitetty ratkaisun kannalta vain oleellisimmat kohdat, kokonaiset toimivat ohjelmakoodit ovat jäljempänä.

Aterioivat filosofit

Aterioivien filosofien ongelma on klassinen esimerkki rinnakkaisuuden hallintaongelmista. Viisi filosofia istuu pyöreän pöydän ääreen ajattelemaan ja syömään. Pöydässä on yhteensä viisi haarukkaa, siis yksi haarukka kunkin filosofin kummallakin puolella. Aikansa ajateltuaan filosofi haluaa syödä, johon hän tarvitsee haarukat molemmilta puoliltaan. Koska vierekkäiset filosofit eivät voi käyttää samaa haarukkaa samanaikaisesti, on algoritmin huolehdittava vuorottelusta siten, että kaikki filosofit saavat aikanaan syödä. Syötyään filosofi palaa ajattelemaan, kunnes hän tulee jälleen nälkäiseksi.

Alla on yksi esimerkki, kuinka aterioivien filosofien ongelma voidaan ratkaista Javalla käyttäen semaforeja. Filosofi-luokan ilmentymät ovat kukin omia säikeitään mallintaen yhden filosofin toimintaa. Konstruktorissa filosofi valitsee, missä järjestyksessä hän ottaa haarukat käteensä. Lukkiutumisvaara voidaan poistaa haarukoiden valintametodeissa käyttämällä epäsymmetristä valintaa. Metodissa run kuvataan filosofin toiminta: ajatteleminen, haarukoiden ottaminen, syöminen ja haarukoiden laskeminen takaisin pöydälle.

```
public class Filosofi implements Runnable {
  private int haarukkaEka;
  private int haarukkaToka;

  public Filosofi(Poyta poyta) {
    this.haarukkaEka = this.poyta.valitseEkaHaarukka(this.paikka);
    this.haarukkaToka = this.poyta.valitseTokaHaarukka(this.paikka);
  }
```

```
  public void run() {
    while (true) {
      ajattele();

      this.poyta.otaHaarukka(this.haarukkaEka);
      this.poyta.otaHaarukka(this.haarukkaToka);

      syo();

      this.poyta.laskeHaarukka(this.haarukkaEka);
      this.poyta.laskeHaarukka(this.haarukkaToka);
    }
  }
}
```

Edellä Filosofi-luokassa ei otettu kantaa, kuinka haarukan ottaminen ja palauttaminen käytännössä tapahtuu, vaan yksityiskohdat on kapseloitu Poyta-luokan sisään. Poyta-luokassa haarukat ovat binäärisemaforeja, jotka luodaan konstruktorissa. Näin ratkaistaan helposti ongelma, että vain yksi filosofi kerrallaan saa käyttää haarukkaa. Nälkiintymisen mahdollisuus estetään käyttämällä vahvoja semaforeja, jolloin filosofit saavat haarukkansa pyytämisjärjestyksessä. Haarukkaa otettaessa kutsutaan haarukan acquire()-metodia (wait-operaatio), jolloin säie jää odottamaan, kunnes haarukka on vapaa. Haarukkaa laskettaessa takaisin pöydälle kutsutaan release()-metodia (signal-operaatio), jolloin seuraava haarukkaa kaipaava filosofi voi ottaa sen.

```
import java.util.concurrent.Semaphore;

public class Poyta {
  private Semaphore haarukat[];

  public Poyta(int paikkaLkm) {
    this.haarukat = new Semaphore[paikkaLkm];
    for (int i = 0; i < this.haarukat.length; ++i) {
      this.haarukat[i] = new Semaphore(1, true); // binäärisemafori
    }
  }

  public void otaHaarukka(int haarukka) {
    try {
      this.haarukat[haarukka].acquire();
    } catch (InterruptedException e) {
    }
  }

  public void laskeHaarukka(int haarukka) {
    this.haarukat[haarukka].release();
  }
}
```

Nukkuva parturi

Toinen yleisesti tunnettu ongelma on nukkuvan parturin ongelma. Parturiliikkeessä on parturi, parturin tuoli sekä tuoleja vuoroaan odottaville asiakkaille. Kun asiakkaita ei ole lainkaan, parturi nukkuu tuolissaan. Asiakkaan saapuessa parturiliikkeeseen asiakas herättää parturin, ja parturi ryhtyy töihin. Parturin ollessa jo työssään asiakas istuu odottamaan vuoroaan. Jos kaikki odotustuolit on jo varattu, asiakas lähtee pois.

Alla olevassa Java-esimerkissä ongelmassa esiintyvien kilpailutilanteiden hallinta on ratkaistu semaforeja käyttäen. Luokka NukkuvaParturi kuvaa parturiliikettä ja siinä alustetaan tarvitta-

vat semaforit. Aterioivien filosofien ratkaisun tapaan tässäkin ratkaisussa käytetään vahvoja semaforeja.

```
import java.util.concurrent.Semaphore;

public class NukkuvaParturi {
  Semaphore asiakkaat;
  Semaphore parturi;
  Semaphore tuolit;
  int vapaidenPaikkojenLkm = 5;

  public NukkuvaParturi() {
    this.asiakkaat = new Semaphore(0, true);
    this.parturi = new Semaphore(0, true);
    this.tuolit = new Semaphore(1, true); // mutex
  }
}
```

Jokainen asiakas on oma säikeensä. Ensimmäiseksi asiakas varaa itselleen odotuspaikan vähentämällä vapaiden paikkojen lukumäärää. Koska vain yksi säie saa kerrallaan muuttaa vapaiden paikkojen määrää, suojataan tämä kriittinen vaihe tuolit-semaforia käyttäen. Seuraavaksi asiakas kertoo parturille olevansa jonossa "signaloimalla" asiakkaat-semaforia, ts. suorittaa asiakkaat.release()-metodikutsun. Lopuksi asiakas jää odottamaan vuoroaan kutsumalla parturi.acquire()-metodia. Metodi palaa, kun parturi on käsitellyt edellä olevat asiakkaat ja kutsuu sitten parturi.release()-metodia.

```
public class Asiakas implements Runnable {
  private NukkuvaParturi parturiliike;

  public void run() {
    try {
      this.parturiliike.tuolit.acquire();
    } catch (InterruptedException e) {
    }
    if (this.parturiliike.vapaidenPaikkojenLkm > 0) {
      --this.parturiliike.vapaidenPaikkojenLkm;
      this.parturiliike.asiakkaat.release();
      this.parturiliike.tuolit.release();
      try {
        this.parturiliike.parturi.acquire();
      } catch (InterruptedException e) {
      }
    } else { // ei vapaita paikkoja - lähdetään pois.
      this.parturiliike.tuolit.release();
    }
  }
}
```

Parturi työskentelee omana säikeenään. Aluksi parturi jää odottamaan asiakasta kutsumalla asiakkaat-semaforin acquire()-metodia. Suoritus jatkuu, kun asiakas on ilmoittanut saapumisestaan kutsumalla asiakkaat.release()-metodia, jolloin parturi palauttaa asiakkaan käyttämän odotuspaikan kasvattamalla vapaiden paikkojen määrää yhdellä. Vapaiden paikkojen määrän muuttaminen tehdään jälleen tuolit-mutexin suojaamana. Parturi päästää seuraavan vuoroaan odottaneen asiakas-säikeen jatkamaan suoritustaan parturi.release()-kutsulla ja ryhtyy leikkaamaan.

```
public class Parturi implements Runnable {
  public void run() {
    while (true)
    {
      try {
        this.parturiliike.asiakkaat.acquire();
      } catch (InterruptedException e) {
      }
      try {
        this.parturiliike.tuolit.acquire();
      } catch (InterruptedException e) {
      }
      ++this.parturiliike.vapaidenPaikkojenLkm;
      this.parturiliike.parturi.release();
      this.parturiliike.tuolit.release();
      leikkaa();
    }
  }
}
```

Kokonaiset ohjelmakoodit

Aterioivat filosofit

Luokka Filosofit

```
/**
 * Luokka käynnistää viiden filosofin aterioinnin.
 */
public class Filosofit {
  private static final int filosofienLkm = 5;


  /**
   * Rakentaa pöydän viidelle ja kutsuu filosofit mietiskelemään ja
   * aterioimaan.
   *
   * @param args
   *          Komentoriviparametrit (ei käytössä)
   */
  public static void main(String[] args) {
    Poyta poyta = new Poyta(filosofienLkm);
    for (int i = 0; i < filosofienLkm; ++i) {
      (new Thread(new Filosofi(poyta))).start();
    }
  }

}
```

Luokka Filosofi

```
/**
 * Luokka toteuttaa filosofin toiminnot eli ajattelemisen ja syömisen ja
 * niiden välisen vuorottelun.
 */
public class Filosofi implements Runnable {
  private static final long AJATTELUAIKA_MIN = 500;
  private static final long AJATTELUAIKA_MAX = 1000;
  private static final long SYOMISAIKA_MIN = 500;
  private static final long SYOMISAIKA_MAX = 1000;
  private static final char TILA_ALKUTILA = '_';
  private static final char TILA_AJATTELEE = 'a';
  private static final char TILA_ODOTTAA = 'o';
  private static final char TILA_SYO = 's';
  private static final char TILA_VALMIS = 'v';
  private Poyta poyta;
  private int paikka;
  private int haarukkaEka;
  private int haarukkaToka;
  private Character tila = TILA_ALKUTILA;


  /**
   * Luo uuden filosofin ja istuttaa sen parametrina annettuun pöytään.
   *
   * @param poyta
   *          Pöytä, johon uusi filosofi istahtaa.
   */
  public Filosofi(Poyta poyta) {
    this.poyta = poyta;
    Integer paikka = this.poyta.otaPaikka(this);
    if (paikka != null) {
      this.paikka = paikka;
    } else {
```

```
      throw new RuntimeException("Pöytä on jo täynnä");
    }

    // Valitaan filosofin käyttämät haarukat
    this.haarukkaEka = this.poyta.valitseEkaHaarukka(this.paikka);
    this.haarukkaToka = this.poyta.valitseTokaHaarukka(this.paikka);
  }


  /**
   * Filosofi ajattelee.
   */
  private void ajattele() {
    try {
      Thread.sleep((long) (Math.random()
          * (AJATTELUAIKA_MAX - AJATTELUAIKA_MIN) + AJATTELUAIKA_MIN));
    } catch (InterruptedException e) {
    }
  }


  /**
   * Filosofi syö.
   */
  private void syo() {
    try {
      Thread.sleep((long) (Math.random() * (SYOMISAIKA_MAX -
          SYOMISAIKA_MIN) + SYOMISAIKA_MIN));
    } catch (InterruptedException e) {
    }
  }


  /**
   * Metodi huolehtii ajattelun ja syömisen vuorottelusta. Ennen ruokailua
   * otetaan luonnollisestikin haarukat kauniisiin käsiin.
   */
  public void run() {
    while (true) {
      this.tila = TILA_AJATTELEE;
      System.out.println(this.poyta);
      ajattele();

      this.tila = TILA_ODOTTAA;
      System.out.println(this.poyta);
      this.poyta.otaHaarukka(this.haarukkaEka);
      this.poyta.otaHaarukka(this.haarukkaToka);

      this.tila = TILA_SYO;
      System.out.println(this.poyta);
      syo();

      this.tila = TILA_VALMIS;
      this.poyta.laskeHaarukka(this.haarukkaEka);
      this.poyta.laskeHaarukka(this.haarukkaToka);
    }
  }


  /**
   * Palauttaa filosofin tilan.
   */
  public String toString() {
    return this.tila.toString();
  }
}
```

Luokka Poyta

```
import java.util.concurrent.Semaphore;

/**
 * Luokka simuloi pöytää, jonka äärellä filosofit ajattelevat ja syövät.
 * Haarukat kuuluvat pöydälle.
 */
public class Poyta {
  private Filosofi filosofit[];
  private int varatutPaikatLkm;
  private Semaphore haarukat[];


  /**
   * Luo uuden pöydän filosofeille ja kattaa haarukat pöytään.
   *
   * @param paikkaLkm
   *          Pöydän paikkojen ja haarukoiden lukumäärä.
   */
  public Poyta(int paikkaLkm) {
    this.filosofit = new Filosofi[paikkaLkm];
    this.varatutPaikatLkm = 0;
    this.haarukat = new Semaphore[this.filosofit.length];
    for (int i = 0; i < this.haarukat.length; ++i) {
      this.haarukat[i] = new Semaphore(1, true); // binäärisemafori
    }
  }


  /**
   * Varaa filosofille paikan pöydästä.
   *
   * @param filosofi
   *          Filosofi, jolle paikka varataan.
   * @return Varatun paikan numero tai null, jos pöytä on jo täynnä.
   */
  public Integer otaPaikka(Filosofi filosofi) {
    if (this.varatutPaikatLkm >= this.filosofit.length) {
      return null;
    } else {
      int paikka = this.varatutPaikatLkm;
      ++this.varatutPaikatLkm;
      filosofit[paikka] = filosofi;
      return paikka;
    }
  }


  /**
   * Valitaan filosofin ensimmäiseksi varaama haarukka. Viimeinen filosofi
   * ottaa haarukan toiselta puoleltaan kuin muut.
   *
   * @param paikka
   *          Filosofin istumapaikka.
   * @return Palauttaa haarukan numeron.
   */
  public int valitseEkaHaarukka(int paikka) {
    return Math.min(paikka, (paikka + 1) % this.filosofit.length);
  }


  /**
   * Valitaan filosofin toiseksi varaama haarukka. Viimeinen filosofi ottaa
   * haarukan toiselta puoleltaan kuin muut.
   *
   * @param paikka
   *          Filosofin istumapaikka.
   * @return Palauttaa haarukan numeron.
   */
```

```
  public int valitseTokaHaarukka(int paikka) {
    return Math.max(paikka, (paikka + 1) % this.filosofit.length);
  }


  /**
   * Filosofi ottaa haarukan käteensä semaforin avulla. Metodi (semafori)
   * palaa vasta, kun haarukka on vapaa käytettäväksi.
   *
   * @param haarukka
   *          Käteen otettavan haarukan numero.
   */
  public void otaHaarukka(int haarukka) {
    try {
      this.haarukat[haarukka].acquire();
    } catch (InterruptedException e) {
    }
  }


  /**
   * Filosofi laskee haarukan takaisin pöydälle vapauttamalla semaforin.
   *
   * @param haarukka
   *          Pöydälle laskettavan haarukan numero.
   */
  public void laskeHaarukka(int haarukka) {
    this.haarukat[haarukka].release();
  }


  /**
   * Palauttaa pöydässä istuvien filosofien tilakoodit merkkijonona.
   *
   * @return Pöydässä istuvien filosofien tilat.
   */
  public String toString() {
    String str = "";
    for (int i = 0; i < this.varatutPaikatLkm; ++i) {
      str += this.filosofit[i].toString();
    }
    return str;
  }
}
```

Nukkuva Parturi

Luokka NukkuvaParturi

```
import java.util.concurrent.Semaphore;

public class NukkuvaParturi {
  private static final int SAAPUMISVALI_MIN = 100;
  private static final int SAAPUMISVALI_MAX = 3000;
  Semaphore asiakkaat;
  Semaphore parturi;
  Semaphore tuolit;
  int vapaidenPaikkojenLkm = 5;


  public NukkuvaParturi() {
    this.asiakkaat = new Semaphore(0, true);
    this.parturi = new Semaphore(0, true);
    this.tuolit = new Semaphore(1, true); // mutex
  }
```

```
  public static void main(String[] args) {
    NukkuvaParturi parturiliike = new NukkuvaParturi();
    (new Thread(new Parturi(parturiliike))).start();

    while (true) {
      try {
        Thread.sleep((long) (Math.random()
            * (SAAPUMISVALI_MAX - SAAPUMISVALI_MIN) + SAAPUMISVALI_MIN));
      } catch (InterruptedException e) {
      }
      (new Thread(new Asiakas(parturiliike))).start();
    }

  }

}
```

Luokka Asiakas

```
import java.util.concurrent.Semaphore;

public class Asiakas implements Runnable {
  private static int asiakkaitaYht = 0;
  private static Semaphore asiakkaitaYhtSem = new Semaphore(1, true);
  private NukkuvaParturi parturiliike;
  private int asiakasNro;


  public Asiakas(NukkuvaParturi parturiliike) {
    this.parturiliike = parturiliike;
    try {
      asiakkaitaYhtSem.acquire();
    } catch (InterruptedException e) {
    }
    this.asiakasNro = asiakkaitaYht;
    ++asiakkaitaYht;
    asiakkaitaYhtSem.release();
  }


  public void run() {
    System.out.println("Asiakas " + this.asiakasNro
        + " saapuu (vapaita paikkoja "
        + this.parturiliike.vapaidenPaikkojenLkm
        + ").");
    try {
      this.parturiliike.tuolit.acquire();
    } catch (InterruptedException e) {
    }
    if (this.parturiliike.vapaidenPaikkojenLkm > 0) {
      --this.parturiliike.vapaidenPaikkojenLkm;
      this.parturiliike.asiakkaat.release();
      this.parturiliike.tuolit.release();
      System.out.println("Asiakas " + this.asiakasNro
          + " odottaa parturia.");
      try {
        this.parturiliike.parturi.acquire();
      } catch (InterruptedException e) {
      }
      System.out.println("Asiakas " + this.asiakasNro
          + " pääsee hiustenleikkuuseen.");

    } else { // ei vapaita paikkoja - lähdetään pois.
      this.parturiliike.tuolit.release();
      System.out.println("Asiakas " + this.asiakasNro + " lähtee.");
    }
  }
}
```

Luokka Parturi

```
public class Parturi implements Runnable {
  private static final int LEIKKUUAIKA_MIN = 500;
  private static final int LEIKKUUAIKA_MAX = 2500;
  NukkuvaParturi parturiliike;


  public Parturi(NukkuvaParturi parturiliike) {
    this.parturiliike = parturiliike;
  }


  public void run() {
    while (true)
    {
      try {
        this.parturiliike.asiakkaat.acquire();
      } catch (InterruptedException e) {
      }
      try {
        this.parturiliike.tuolit.acquire();
      } catch (InterruptedException e) {
      }
      ++this.parturiliike.vapaidenPaikkojenLkm;
      this.parturiliike.parturi.release();
      this.parturiliike.tuolit.release();
      leikkaa();
    }
  }


  private void leikkaa() {
    try {
      Thread.sleep((long) (Math.random()
          * (LEIKKUUAIKA_MAX - LEIKKUUAIKA_MIN) + LEIKKUUAIKA_MIN));
    } catch (InterruptedException e) {
    }
    System.out.println("Hiustenleikkuu valmis.");
  }

}
```