Projekti 1

Säikeet ja kriittisen vaiheen kontrollointi javalla

Lasse Leino ja Marko Kahilakoski
Helsingin Yliopisto
Tietojenkäsittelytieteen laitos
Rinnakkaisohjelmointi

18. joulukuuta 2006

Sisältö

1 Säikeet ja kriittisen vaiheen kontrollointi javalla **1**
1.1 Yleistä . . . 1
1.2 Keskeytykset . . . 2
1.3 Säikeen odottaminen . . . 3
1.4 Synkronisointi . . . 3
1.5 Producer-Consumer . . . 6
1.5.1 Producer . . . 6
1.5.2 Consumer . . . 7
1.5.3 Storage . . . 8
1.5.4 Producer-Consumer test . . . 11

1 Säikeet ja kriittisen vaiheen kontrollointi javalla

1.1 Yleistä

Javassa kaikilla ohjelmilla on yksi säie, eli pääohjelma (main thread), jossa pystytään luomaan uusia säikeitä. Tämä voidaan toteuttaa kahdella tavalla.

Rajapintaluokka Runnable määrittelee metodin run(). Luokka voidaan laittaa toteuttamaan rajapintaluokka Runnable, jonka jälkeen tehdään luokkaan metodi run(), joka sisältää säikeen ohjelmakoodin. Tämän jälkeen Runnable-olio annetaan parametrina Thread:in konstruktorille.

Esimerkiksi:

```
public class Kivaa implements Runnable {

    public void run() {
        System.out.println("Rio on kivaa!");
    }

    public static void main(String args[]) {
        (new Thread(new Kivaa())).start();
    }

}
```

Toinen vaihtoehto on tehdä luokasta, Thread:in aliluokka. Thread toteuttaa itse rajapintaluokan Runnable, mutta sen run-metodi on tyhjä. Run-metodi toteutetaan taas itse luokassa.

Esimerkiksi:

```
public class Kivaa extends Thread {

    public void run() {
        System.out.println("Rio on kivaa!");
    }

    public static void main(String args[]) {
        (new Kivaa()).start();
    }
}
```

Molemmissa tapauksissa säie käynnistetään metodilla Thread.start. Ensimmäinen tapa säikeiden tekoon on yleisempi ja joustavampi, koska luokka voi samalla toimia jonkun muun kuin Thread-luokan aliluokkana.

1.2 Keskeytykset

Säikeen suoritus voidaan asettaa suspend-tilaan metodilla Thread.sleep. Sleep ottaa parametreinaan taukoajan joko milli- tai nanosekunteina. Ei voida olla kuitenkaan varmoja, että säikeen suoritus keskeytyy juuri annetuksi ajaksi, koska toiminnon tarkkuus riippuu käyttöjärjestelmästä, jolla ohjelmaa ajetaan.

Seuraava esimerkki tulostaa annetun tekstin kymmenen kertaa yhden sekunnin välein.

```
public class Kivaa {
    public static void main(String args[]) throws InterruptedException {
        for (int i = 0; i < 10; i++) {
            Thread.sleep(1000);
            System.out.println("Rio on kivaa!");
        }
    }
}
```

Sleep-metodi heittää poikkeuksen InterruptedException, jos jokin toinen säie keskeyttää nukkuvan säikeen. Jos halutaan tehdä jotain erityistä, kun säie keskeytetään, käsitellään InterruptedException poikkeus esimerkiksi seuraavalla tavalla:

```
public class Kivaa {
    public static void main(String args[]) {
        for (int i = 0; i < 10; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Keskeytys :I");
                return;
            }
            System.out.println("Rio on kivaa!");
        }
    }
}
```

Jos säie tekee paljon töitä (vaikka numeronmurskausta) eikä käynnistä yhtään metodia, joka heittäisi poikkeuksen InterruptedException, voidaan tarkistaa myös itse onko keskeytystä tullut. Tämä tehdään Thread.interrupted metodia käyttäen.

Esimerkiksi:

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

Kun säie keskeytetään, sille asetetaan keskeytystila, joka kertoo, että säie on keskeytetty. Tämä poistuu, kun käytetään edeltävää metodia. Jos halutaan vain tarkistaa keskeytyksen tila, voidaan käyttää metodia Thread.isInterrupted, joka

jättää keskeytystilan ennalleen.

1.3 Säikeen odottaminen

Join-metodi mahdollistaa tilanteen, jossa säikeen on odotettava toisen säikeen suorituksen valmistumista. Parametrina voi antaa myös ennalta määritellyn odotusajan, kuten Sleep-metodilla, mutta odotusaikojen luotettavuudessa on samat ongelmat.

Esimerkiksi: säie odottaa toisen säikeen valmistumista.

```
toinenSäie.joins();
```

1.4 Synkronisointi

Esitetään yksikertainen esimerkki laskurista, jolla voi joko lisätä lukua yhdellä tai vähentää sitä yhdellä.

Esimerkiksi:

```
public class Laskuri {
    private int luku = 0;

    public void lisää() {
        luku++;
    }

    public void poista() {
        luku--;
    }

    public int arvo() {
        return luku;
    }
}
```

Ongelmia seuraa, jos meillä on monta säiettä suorittamassa samaa koodia, koska ne voivat mennä kriittisesti päällekäin. Esimerkiksi komento luku++ voidaan jakaa kolmeen osaan: haetaan luvun nykyinen arvo, lisätään luvun arvoa yhdellä ja talletetaan luvun arvoksi saatu arvo.

Seuraava skenaario on siis mahdollinen:

- Säie A hakee luvun nykyisen arvon (luku = 0)
- Säie B hakee luvun nykyisen arvon (luku = 0)
- Säie A lisää arvoa yhdellä (luku = 1)
- Säie B lisää arvoa yhdellä (luku = 1)

- Säie A tallentaa uuden arvon (luku = 1)
- Säie B tallentaa uuden arvon (luku = 1)

Edellinen skenaario on siis virheellinen, koska luvun arvon pitäisi kahden lisäyksen jälkeen olla kaksi.

Ongelma poistuu, kun käytetään synkronoituja metodeja. Muutetaan esimerkkiä:

```
public class Laskuri {
    private int luku = 0;

    public synchronized void lisää() {
        luku++;
    }

    public synchronized void poista() {
        luku--;
    }

    public synchronized int arvo() {
        return luku;
    }
}
```

Kun säie A on suorittamassa synkronoitua metodia lisää, muut säikeet, jotka haluavat suorittaa saman metodin joutuvat odottamaan vuoroaan. Nyt edellisen esimerkin ongelmaa ei enää ole. Javassa tälläinen lukko on toteutettu monitoreilla. Synkronoidun metodin lukko varaa metodin olion ja kun metodin suoritus päättyy, lukko vapautuu.

Jos on tilanne, jossa kasvatetaan kahta lukua, mutta niiden kasvattaminen ei riipu ollenkaan toisistaan, niin synkronoidun metodin lukko varaisi yhden säikeen suorituskerralla koko luokan itselleen ja estäisi näin lukujen yhtäaikaisen kasvattamisen. Tähän voidaan käyttää synkronoituja lauseita, joissa täytyy antaa parametrina, mitä oliota käytetään lukkona.

Esimerkiksi:

```
public class toinenLaskuri {
    private long luku = 0;
    private long toinenLuku = 0;
    private Object lukko = new Object();
    private Object toinenLukko = new Object();

    public void kasvataEnsimmäistä() {
        synchronized(lukko) {
            luku++;
```

```
        }
    }

    public void kasvataToista() {
        synchronized(toinenLukko) {
            toinenLuku++;
        }
    }
}
```

Nyt molempien lukujen kasvattaminen on säieturvallista ja niitä voidaan kasvattaa keskenään rinnakkain.

Metodi joka odottaa tietyn ehdon täyttymistä ennen kuin se jatkaa suoritustaan voidaan laittaa odottamaan metodilla Object.wait. Säie ei jatka suoritustaan ennen kuin jokin toinen säie antaa sille keskeytyksen. Huomaa, että tämä keskeytys ei välttämättä ole se mitä metodi odottaa, joten pitää tarkistaa, täyttyykö haluttu ehto.

Esimerkiksi:

```
public synchronized odota() {
        while (odota) {
                try {
                        wait();
                } catch (InterruptedException e) {}
        }
}
```

Kun säie suorittaa metodin wait(), se luopuu lukostaan ja siirtyy suspend-tilaan ja jää odottamaan. Kaikille lukkoa odottaville säikeille voidaan ilmoittaa muutoksesta:

```
public synchronized lopetaOdotus() {
    odota = false;
    notifyAll();
}
```

Jos halutaan herättää vain yksi säie, voidaan käyttää metodia notify.

Esitellään vielä lopuksi kurssilla tutuksi tullut Producer-Consumer ongelma toteuttettuna javalla seuraavassa kappaleessa.

1.5 Producer-Consumer

1.5.1 Producer

```
/** Tuottajaluokka, Thread luokan aliluokka **/
public class Producer extends Thread {

    private Storage storage;
    private int number;      // Uniikki kuluttajanumero
    private int puts = 0;    // Tuottokerrat

    /**
    * Konstruktori, asettaa tuottajan Storageen
    * @param c
    * @param number
    */
    public Producer(Storage c, int number) {
        storage = c;
        storage.addProducers();
        this.number = number;
        System.out.format("Created producer #%d\n", number);
    }

    /**
    * Threadin ajometodi.
    * Ajetaan kunnes suoritusaika saavutetaan, tai kunnes viimeinenkin Consumer on
    * lopettanut
    */
    public void run() {
        do {
            int temp = (int)(Math.random() * 1000); // Tuotettava elementti

            System.out.format("Producer %d trying to put %d. %d consumers active ...\n",
                              number, temp, storage.getNumConsumers());

            storage.put(number, temp); // Asetetaan elementti Storageen
            puts++;                    // Kasvatetaan tuottoja
            try {
                sleep((int)(Math.random() * 1000)); // Koetetaan nukkua (järkevästi)
                                                    // satunnainen aika
            }
            catch (InterruptedException e) { }      // Saadaan keskeytys, voidaan jatkaa

        } while (storage.getFinishTime() > System.currentTimeMillis()
                 && storage.getNumConsumers() > 0);
        storage.retireProducers(); // Jäädään eläkkeelle.
        System.out.format("Producer #%d retiring. Made %d put operations...\n",
                          number, puts);
    }
}
```

1.5.2 Consumer

```
/** Kuluttajaluokka, Thread luokan aliluokka **/
public class Consumer extends Thread {
    private Storage storage;
    private int number;      // Uniikki kuluttajanumero
    private int gets = 0;    // Hakukerrat

    public Consumer(Storage c, int number) { // Kuluttajakonstruktori
        storage = c;
        storage.addConsumers(); // Lisätään kuluttajien yhteismäärää
        this.number = number;
        System.out.format("Created consumer #%d\n", number);
    }

    public void run() { // Threadin ajometodi
        int value = 0;
        do { // Loopataan, kunnes suoritusaika saavutetaan, tai kunnes viimeinenkin
             // Tuottaja on lopettanut
            value = storage.get(number); // Haetaan varastosta ja kasvatetaan hakuja
            gets++;
            try { // Koetetaan nukkua (järkevästi) satunnainen aika
                sleep((int)(Math.random() * 1000));
            }
            // Saadaan keskeytys, voidaan jatkaa
            catch (InterruptedException e) { }

        } while (storage.getFinishTime() > System.currentTimeMillis()
                 && storage.getNumProducers() > 0);
        storage.retireConsumers(); // Jäädään eläkkeelle.
        System.out.format("Consumer #%d retiring. Made %d get operations...\n",
                          number, gets);
    }
}
```

1.5.3 Storage

```
/**
* Storage on Producer/Consumer dilemman ydinluokka, joka tarjoaa aksessoreita
* tuottajille ja kuluttajille.
*/
public class Storage {
    private int contents;               // Tuotettu elementti
    private boolean available = false;  // Vapaa/varattu kytkin
    long finishTime = 0;                // Suoritusaika (lopetusaika)
    int producers = 0, consumers = 0;   // Olemassa olevat Producerit ja
                                        // Consumerit

    /**
    * Konstruktori, asettaa ajoajan
    * @param secondsToRun
    */

    public Storage(long secondsToRun) {
        this.finishTime = secondsToRun*1000 + System.currentTimeMillis();
    }

    /**
    * Lisää tuottajan
    */
    public synchronized void addProducers() {
        producers++;
    }

    /**
    * Vähentää tuottajan ja herättää nukkuvat kuluttajat (ja tuottajat)
    */
    public synchronized void retireProducers() {
        producers--;
    if(producers == 0) // Mikäli ei ole enää aktiivisia tuottajia,
                       // kerrotaan siitä:
        notifyAll();   // Herätys kuluttajille deadlockin välttämiseksi
    }

    /**
    * Palauttaa tuottajien määrän
    */
    public synchronized int getNumProducers() {
        return producers;
    }

    /**
    * Lisää kuluttajan
    */
    public synchronized void addConsumers() {
```

```
    consumers++;
}

/**
* Vähentää kuluttajan ja herättää tuottajat (ja kuluttajat)
*/
public synchronized void retireConsumers() {
    consumers--;
if(consumers == 0) // Mikäli ei ole enää aktiivisia kuluttajia,
                   // kerrotaan siitä:
    notifyAll();   // Herätys tuottajille deadlockin välttämiseksi
}

/**
* Palauttaa kuluttajien määrän
*/
public synchronized int getNumConsumers() {
    return consumers;
}

/**
* Palauttaa lopetusajan
*/
public long getFinishTime() {
    return this.finishTime;
}

/**
* Hakumetodi, palauttaa tuottajan tuottaman elementin tai lopettaa.
* @param who
*/
public synchronized int get(int who) {
    while (available == false) {
        try {
            wait(); // Odotellaan, kunnes huomautetaan, että kulutettavaa
                    // on tarjolla
        }

        catch (InterruptedException e) { }
        // Saatiin keskeytys, voidaan jatkaa.
        if(producers == 0) {
            return 0; // Mikäli ei ole enää tuottajia, ilmoitetaan siitä
        }
    }

    available = false;
    System.out.format("Consumer %d got: %d%n", who, contents);
    notifyAll(); // Herätetään nukkujat, saadaan "kriittinen vaihe"
                 // valmiiksi.
    return contents;
```

```
    }

    /**
    * Tuottometodi: Kuka tuottaa ja minkä elementin
    * @param who
    * @param value
    */
    public synchronized void put(int who, int value) {
        while (available == true) {
            try {
                wait(); // Odotellaan, että saadaan tuottovuoro
            } catch (InterruptedException e) { }
        // Saatiin keskeytys, voidaan jatkaa
        if(consumers == 0)
            return; // Mikäli ei ole kuluttajille joille tuottaa, on turha
                    // tuottaa.
        }
                    // Muutoin tuotetaan elementti, ja asetetaan saatavilla
                    // -lippu
        contents = value;
        available = true;
        System.out.format("Producer %d put: %d%n", who, contents);
        notifyAll(); // Kerrotaan edistyksestä odottaville prosesseille
    }
}
```

1.5.4 Producer-Consumer test

```
import java.util.Vector;

public class ProducerConsumerTest {
    public static void main(String[] args) {
        int n, m;

        long secondToRunApp = 10; // Kauanko ajetaan?

        n = Integer.parseInt(args[0]); // Oletetaan parametreinä tuottajat
        m = Integer.parseInt(args[1]); // ja kuluttajat

        Vector<Producer> producers = new Vector<Producer>(); // Luodaan säilöt
        Vector<Consumer> consumers = new Vector<Consumer>();

        Storage storage = new Storage(secondToRunApp); // Ja tuotto/kulutus
                                                       // -ympäristö

        System.out.format("Creating:\n  %d consumers\n  %d producers\n\n",m, n);

        for(int i = 0; i < n; i++) // Lisätään tuottajat
            producers.add(new Producer(storage, i+1));

        for(int i = 0; i < m; i++) // ... ja kuluttajat
            consumers.add(new Consumer(storage, i+1));

        System.out.println("\nStarting threads...\n");

        for(Producer p : producers) // Käynnistetään säikeet
                                          // (Huom. Java:n foreach)
            p.start();

        for(Consumer c : consumers)
            c.start();

    }
}
```