Javan semaforit

Joel Rybicki, Aleksi Nurmi, Jara Uitto
16.12.2007
Helsingin yliopisto
Tietojenkäsittelytieteen laitos
Tätä ohjetta saa käyttää ja jatkokehittää opetustarkoituksiin.

Javan semaforitoteutus

Semafori on olio, joka ratkaisee kriittisen vaiheen ongelman ohjelmoijan kannalta helpommin. Kriittistä vaihetta vartioimaan luodaan semafori. Jokaisessa prosessissa kriittisen vaiheen suoritus ympäroidään semaforiin liittyvillä funktiokutsuilla. Funktiokutsuja on kaksi ja näistä käytetään useita nimityksiä, kuten *wait*/*signal*, *P*/*V*, *acquire*/*release*, *pend*/*post* tai *up*/*down*. Semafori pitää huolen, että kriittiseen vaiheeseen pääsee kerrallaan vain ohjelmoijan asettama määrä prosesseja. Semaforin nimi tulee rautatieliikenteessä käytetyistä siipiopastimista, joilla ratkaistiin sama ongelma käytännössä.

Semaforin luonti

Javassa semafori on toteutettu luokkana Semaphore. Tämä alustetaan pitkälti samoin kuten luentojen ja kirjan esimerkkisemaforit. Aluksi määritellään semaforin "alkutila" eli kuinka monta prosessia kerrallaan pääsee semaforista läpi. Binäärisemafori eli muteksi voidaan toteuttaa alustamalla semaforin "lupalappujen" (*permits*) määräksi yksi. Konstruktoreita on kaksi.

Semaphore(int permits)

Creates a Semaphore with the given number of permits and nonfair fairness setting.

Semaphore(int permits, boolean fair)

Creates a `Semaphore` with the given number of permits and the given fairness setting.

Semaphore-luokan konstruktorille voidaan antaa myös boolean-tyyppinen *fair*-parametri. Tämä määrittelee onko semafori **vahva** eli reilu vai **heikko** eli epäreilu. Vahvan semaforin odotusjono on first in, first out -tyyppinen (jono). Kun *fair*-parametri on false, odotusjono on järjestämätön joukko. Tällöin odottavien prosessien joukosta suoritukseen valitaan mikä tahansa joukkoon kuuluva prosessi. Tällöin semafori on **heikko** ja nälkiintyminen on mahdollista. Mikäli *fair*-parametria ei anneta, semafori on oletuksena heikko.

Wait ja signal

Kirjan ja luentojen *wait* ja *signal* -operaatioita vastaavat Javassa *acquire* ja *release*. Nämä metodit toimivat vastaavasti kuin luennon esimerkeissä. *Acquire* vähentää semaforin laskurista yhden ja laittaa prosessin jonoon mikäli laskurin arvo meni alle nollan. *Release* toimii kuten *signal* eli mikäli semaforin jonossa ei ole ketään, lisätään laskuriin yksi, muuten päästetään yksi prosessi jonosta jatkamaan. Oleellisin ero on kirjan ja luentojen vastaaviin operaatioihin on, että *acquire*-metodi heittää InterruptedException-poikkeuksen, joka tulee kaapata (koodista tulee hiukan pidempää).

void acquire() throws InterruptedException

Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.

void release()

Releases a permit, returning it to the semaphore.

Javan API ei erikseen määrittele, onko Semaphore-luokan toteutus busy-wait -tyyppinen. Nykyisin Java käyttää sisäisesti käyttöjärjestelmien moniajopalveluita, joten semafori varmastikin asettaa säikeet suspend-tilaan. Eksoottisimmista laite- ja käyttöjärjestelmäympäristöistä tai hyvin vanhoista Java-versioista ei kuitenkaan voi mennä takuuseen.

Esimerkit

Esimerkki: Mutex-lukko binäärisemaforilla

```
public Process extends Thread {
    // Binäärisemafori, joka toimii mutex-lukkona
    private static Semaphore mutex = new Semaphore(1);

    public void run() {
        while(true) {
            // Rinnakkaistettua laskentaa..
            // ..
            // ..

            // Prosessi haluaa siirtyä kriittiseen vaiheeseen
            try {
                mutex.acquire();
            } catch (InterruptedException e) { }
```

```
            // Kriittinen vaihe!
            // Vain yksi prosessi kerrallaan voi suorittaa tätä.
            criticalSection();


            // Kriittinen vaihe ohi. Vapautetaan mutex-lukko
            mutex.release();

            // Ja jatketaan laskentaa..
        }
    }
}
```

Esimerkki: Java-toteutus Ben-Arin kirjan algoritmista 6.11

```
/*
Java adaption of algorithm 6.11 (Ben-Ari).
*/

import java.util.concurrent.Semaphore;
import java.util.Random;

public class Philosopher extends Thread {
        // The amount of philosophers (and forks)
        protected static final int PHILOSOPHER_COUNT = 4;
        protected static final int FORK_COUNT = PHILOSOPHER_COUNT;

        // Maximum time for thinking or eating
        protected static final int MAX_IDLE_TIME = 5000;

        // Semaphores
        protected static Semaphore[] forks = new Semaphore[FORK_COUNT];
        protected static Semaphore room = new Semaphore(FORK_COUNT);

        protected static Random rng = new Random(); // Used by spendTime()

        // Philosopher data
        protected int id;

        public Philosopher(int id) {
                this.id = id;
        }

        /* Tell the world what I am currently doing. */
        protected void Iam(String s) {
                System.out.println("Philosopher #"+id+" is " + s);
        }

        /* Spends at least 'max' milliseconds doing nothing */
        protected void spendTime(int max) {
                int time = (int)(rng.nextFloat()*max);
                try {
                        sleep( max );
                } catch(InterruptedException e) {}
        }

        /* Spend some time thinking */
```

```
        protected void think() {
                Iam("thinking.");
                spendTime(MAX_IDLE_TIME);
                Iam("hungry.");
        }

        /* Get into the dining room then grab forks on the left and right side.
         * Spend some time eating then release the forks.
        */
        protected void eat() {
                Iam("entering the room.");
                // Semaphore.acquire() may throw InterruptedException!
                try {
                        room.acquire(); // Get into the dining room
                        grabFork(id);
                        grabFork(id+1);
                } catch (InterruptedException e) {}

                Iam("eating.");
                spendTime(MAX_IDLE_TIME);
                Iam("done eating.");
                // Signal the semaphores
                returnFork(id);
                returnFork(id+1);
                room.release(); // Get out of the room
        }


        /* Grab a fork */
        protected void grabFork(int i) throws InterruptedException {
                if (i == FORK_COUNT) i = 0;

                forks[i].acquire();
                Iam("holding fork " + i + ".");
        }

        /* Return a fork */
        protected void returnFork(int i) {
                if (i == FORK_COUNT) i = 0;

                Iam("releasing fork " + i + ".");
                forks[i].release();
        }

        /* Repeat forever.. */
        public void run() {
                while (true) {
                        think();
                        eat();
                }
        }

// ------------------------ main ------------------------------------------

        public static void main(String[] args) {
                // Initialize the semaphores
                room = new Semaphore(PHILOSOPHER_COUNT - 1, true);
                for(int i = 0; i<forks.length; i++) {
```

```
                        forks[i] = new Semaphore(1);
                }

                // Create dining philosophers
                for(int j = 0; j<PHILOSOPHER_COUNT; j++) {
                        System.out.println("Philosopher #"+j+" is born!");
                        Philosopher p = new Philosopher(j);
                        p.start();
                }
        }
}
```

Esimerkki: Itsetehty busy-wait -semafori

```
// Esimerkkitoteutus busy-wait -semaforista Javalla
// Toteutus on heikko eli epäreilu, sillä virtuaalikoneen
// skeduloija päättää suoritusjärjestyksen.
public class BusyWaitSemaphore {
// ------------------------------------------------------
    // Apuluokka, joka hoitaa laskurin synkronoinnin
    protected class SynchronizedPermitCounter {
        int value;
        public SynchronizedPermitCounter(int value) {
                this.value = value;
        }
        public synchronized void increase() {
                this.value++;
        }
        public synchronized void decrease() {
                this.value--;
        }
        public synchronized int getValue() {
                return this.value;
        }
    }
// ------------------------------------------------------

    protected SynchronizedPermitCounter permits;

    public BusyWaitSemaphore(int permits) {
        this.permits = new SynchronizedPermitCounter(permits);
    }

    // Synchronized päästää vain yhden säikeen kerrallaan suorittamaan
    // tätä metodia.
    // Samanaikaiset kutsujat blokataan (siirtyvät suspend-tilaan).
    // Toteutus on jossain määrin reilu. Ensimmäinen odottaja pääsee
    // ensimmäisenä läpi,
    // sillä vain yksi säie voi olla kerrallaan suorittamassa metodia (ja while-
    // silmukkaa).
    public synchronized void acquire() {
        while ( this.permits.getValue() <= 0 ) {
            // Busy wait!
        }
        this.permits.decrease();
    }
```

```
    public void release() {
            this.permits.increase();
    }
}
```

Esimerkki: Aterioivat filosofit itsetehdyllä busy-wait -semaforilla

```
/*
Java adaption of algorithm 6.11 (Ben-Ari).
*/

import java.util.Random;

public class BusyPhilosopher extends Thread {
        // The amount of philosophers (and forks)
        protected static final int PHILOSOPHER_COUNT = 5;
        protected static final int FORK_COUNT = PHILOSOPHER_COUNT;

        // Maximum time for thinking or eating
        protected static final int MAX_IDLE_TIME = 5000;

        // BusyWaitSemaphores
        protected static BusyWaitSemaphore[] forks = new
BusyWaitSemaphore[FORK_COUNT];
        protected static BusyWaitSemaphore room = new
BusyWaitSemaphore(FORK_COUNT);

        protected static Random rng = new Random();

        // Philosopher data
        protected int id;

        public BusyPhilosopher(int id) {
                this.id = id;
        }

        /* Tell the world what I am currently doing. */
        protected void Iam(String s) {
                System.out.println("Philosopher #"+id+" is " + s);
        }

        /* Spends at least 'max' milliseconds doing nothing */
        protected void spendTime(int max) {
                int time = (int)(rng.nextFloat()*max);
                try {
                        sleep( max );
                } catch(InterruptedException e) {}
        }

        /* Spend some time thinking */
        protected void think() {
                Iam("thinking.");
                spendTime(MAX_IDLE_TIME);
                Iam("hungry.");
        }

        /* Get into the dining room then grab forks on the left and right side.
```

```
         * Spend some time eating then release the forks.
        */
        protected void eat() {
                Iam("entering the room.");
                room.acquire(); // Get into the dining room
                grabFork(id);
                grabFork(id+1);
                Iam("eating.");

                spendTime(MAX_IDLE_TIME);

                Iam("done eating.");
                returnFork(id);
                returnFork(id+1);
                room.release(); // Get out of the room
        }


        /* Grab a fork */
        protected void grabFork(int i) {
                if (i == FORK_COUNT) {
                        i = 0;
                }
                forks[i].acquire();
                Iam("holding fork " + i + ".");
        }

        /* Return a fork */
        protected void returnFork(int i) {
                if (i == FORK_COUNT) {
                        i = 0;
                }

                Iam("releasing fork " + i + ".");
                forks[i].release();
        }

        /* Repeat forever.. */
        public void run() {
                while (true) {
                        think();
                        eat();
                }
        }

// ------------------------- main ----------------------------------------------

        public static void main(String[] args) {
                // Initialize the semaphores
                room = new BusyWaitSemaphore(PHILOSOPHER_COUNT - 1);
                for(int i = 0; i<forks.length; i++) {
                        forks[i] = new BusyWaitSemaphore(1);
                }

                // Create dining philosophers
                for(int j = 0; j<PHILOSOPHER_COUNT; j++) {
                        System.out.println("Philosopher #"+j+" is born!");
                        BusyPhilosopher p = new BusyPhilosopher (j);
```

```
                        p.start();
                }
        }
}
```

Lisätietoa

Semaphore-luokan API:
http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Semaphore.html

Javan säikeistyksestä: http://www.jguru.com/faq/view.jsp?EID=143462

Semaphore-luokka ei ole ainoa keino hallita rinnakkaisuutta Javassa. Katso myös avainsana *synchronized* ja rajapinnan Lock toteuttavia luokkia.