Monitorit -projekti

Rinnakkaisohjelmointi
13.12.2006

Jaakko Louhio, Lari Sorvo

3. MONITORIT

a. Menetelmän käyttötarkoitus ja sovellusalue

Monitori on synkronointitapa jolla kontrolloidaan useamman prosessin pääsyä tiettyyn resurssiin. Kyseessä olevaa resurssia käyttäessä on ehdottoman tärkeää että vain yksi prosessi kerrallaan pääsee käyttämään sitä. Tämän vuoksi tarvitaan ”valvojaa” joka pitää huolen siitä ettei useampi prosessi käytä tiettyä resurssia yhtä aikaa. Monitori on juuri tällainen valvoja.

Monitori-käsite on yleistys käyttöjärjestelmän kernelistä, tai sen valvojasta. Normaalit sovellusohjelmat pyytävät käyttöjärjestelmältä apua kriittisten asioiden suoritukseen, esimerkiksi muistin allokoimiseen. Käyttöjärjestelmän toimivuuden kannalta on äärimmäisenä tärkeää että operaatio suoritetaan oikein, ja ettei muistia varata päällekkäin eri ohjelmille.

Yksinkertaisemmassa tapauksessa mille tahansa objektille tai objektiryhmälle voidaan määritellä monitori joka takaa synkronoinnin tämän/näiden välille. Jos useampi prosessi kutsuu saman monitorin tiettyä operaatiota, takaa monitorin toteutus sen että prosessin suorituksessa on ratkaistu poissulkemisongelma.

b. Menetelmän perusidea

Ohjelmoitaessa rakenteita joissa useampi prosessi käyttää samoja resursseja, on pidettävä huoli siitä etteivät eri prosessit yhtä aikaa käytä jotain tiettyä kriittistä resurssia. Tämän ongelman ratkaisemiseksi on kehitetty useita eri menetelmiä. Yksi pitkään käytössä ollut menetelmä on semafori, jolla taataan ettei kyseistä ongelmaa synny. Semafori on kuitenkin matalan tason synkronointiprimitiivi ja mikäli isompaa kokonaisuutta ohjelmoitaessa käytettäisiin vain semaforeja, olisi toteuttajilla todella iso vastuu ohjelman toimivuuden kannalta käyttää semaforeja oikein. Yksi ainoa lipsahdus toteutusvaiheessa takaisi sekä rikkinäisen rakenteen että ongelman vaikean paikallistamisen.

Monitorit ovat yleistys nykyään laajalti käytetyistä olioista eri olio-ohjelmointikielissä. Monitorin toteutuksessa käytetään kapselointia jossa rakenteen sisäinen toteutus on piilotettu ulkopuolisilta, ja ainoastaan rakenteen rajapinta on käytettävissä ulkopuolisille. Monitorin määrittelyyn kuuluu myös se

tosiasia että siihen pääsee käsiksi vain yksi prosessi kerrallaan. Nämä kaksi piirrettä takaavat että monitori toimii johdonmukaisesti.

Monitorin toteutus eri ohjelmointikielissä voi vaihdella suuresti, mutta perus toimintaperiaate on aina sama. Monitori toimii ”vahtina” tiettyyn resurssiin, ja resurssiin ei pääse mitään muuta kautta käsiksi. Kaikki monitorin operaatiot toimivat atomisesti. Juuri tämä takaa rakenteen oikean toimivuuden. Rakenne takaa ratkaisun poissulkemisongelmaan, mutta nälkiintymistä järjestelmässä saattaa kuitenkin tapahtua.

Perusperiaate monitorissa on se että monitorin rakenne määritetään etukäteen tarkasti. Monitoriin voidaan määrittää mitä muuttujia ja rakenteita tahansa, mutta niihin ei pääse monitorin ulkopuolelta käsiksi. Monitori päästää ainoastaan yhden prosessin kerrallaan suoritukseen, eikä uusi prosessi pääse monitoriin ennen kuin edellinen on sieltä poistunut.

Riippuen ratkaistavasta ongelmasta, monitoreissa saatetaan joskus tarvita ehto-muuttujia. Tunnetussa tuottaja-kuluttuja –ongelmassa käytetään puskuria jonka kautta kulutettavaa dataa siirretään tuottajalta kuluttajalle. Joskus puskuri saattaa olla täynnä, joskus taas tyhjä. Näihin tilanteisiin tarvitaan monitorin sisäiseen rakenteeseen yllämainittuja ehto-muuttujia joilla tarkastetaan onko puskuri täynnä/tyhjä; kuluttajaa ei voida palvella jos puskuri on tyhjä, eikä dataa voida tuottaa lisää mikäli puskuri on jo ennestään täynnä.

c. Menetelmän yhteneväisyydet/eroavaisuudet kurssilla esitettyyn perustapaukseen verrattuna

Monitori eroaa aikaisemmin käsitellyistä rakenteista siinä mielessä että sen käyttötarkoitus on hieman erilainen. Vaikka rakenteen idea onkin sama kuin esimerkiksi semaforeissa, mm. ratkaista poissulkemisongelma, on sen käyttökohde hieman erilainen. Monitorin käsite on korkeammalla tasolla verrattuna muihin aikaisemmin esitettyihin tapauksiin. Myöskin itse monitorin toteutuskuvaus on löyhä, eikä mitään tarkkaa määrittelyä sen toteutustavalle ole määritelty; eri ohjelmointikielissä rakenne voidaan toteuttaa hyvinkin erilaisilla tyyleillä.

Vaikka monitoreita käytetään nimenomaan samanaikaisuuden tuottamien ongelmien ratkaisuun, ei itse monitorirakenne tue minkäänlaista rinnakkaissuoritusta monitorin rakenteen sisällä. Ainoastaan yksi prosessi kerrallaan on monitorin sisällä suorituksessa. Monitori on myös täysin staattinen, eikä se muutu luomisensa jälkeen mitenkään. Näin ollen monitorissa ei ole mitään varsinaista dynaamisuutta vaikka sen käyttö joskus antaisi päinvastaisen kuvan.

d. Kaksi käyttöesimerkkiä menetelmästä

1. TuotaKuluta.java

```
/**
 * Esimerkissä on ratkaistu tuottajien ja kuluttajien ongelma
 * käyttämällä javan synkronoituja metodeja, jotka siis ajavat
 * monitorien aseman (paremmin kuin monitorit). Olion sisällä
 * synkronoitujaa metodeita pääsee suorittamaan vain yksi säie
 * kerrallaan ja synkronointi varmistaa, että metodin tulos
 * tulee näkyviin seuraaviin synkoroitujen metodien suorituksiin.
 *
 * Ohjelma sisältää tuota() ja kuluta() metodit ja run() metodin,
 * jolla luodaan tuottajia ja kuluttajia "systeemiin".
 *
 * Ohjelman voi suorittaa ja katsoa kuinka tuottajat ja kuluttajat
 * käyttävät puskuria. Huomaa että jokaisella suoritus kerralla ei
 * välttämättä tule yhtään tuottajaa tai kuluttajaa systeemiin.
 * Joskus puskuri on jatkuvasti täynnä ja välillä pääasiassa tyhjänä.
 * /
public class TuotaKuluta implements Runnable{

    /**
     * Puskuriin "tuottaja" tuottaa kokonaislukuja, joita
     * "kuluttajat" käyvät kuluttamassa.
     */
    int[] puskuri = new int[30];
    /**
     * Tyhjä ja täysi -muuttujilla merkitään puskurin tilaa.
     * Kuluttaja ei voi kuluttaa jos puskuri on tyhjä ja
     * tuottaja ei voi tuottaa jos puskuri on täynnä.
     */
    boolean tyhjä = true;
    boolean täysi = false;
    /**
     * Tuonti ja vienti -muuttujat pitävät yllä mihin kohtaan
     * puskuria ollaan seuraavaksi lisäämässä luku ja mistä
     * kohdasta seuraavaksi lukemassa.
     */
    int tuonti = 0;
    int vienti = 0;

    /**
     * Tuota() metodi tuottaa tuonti -muuttujan osoittamaan
     * paikkaan kokonaisluvun ja kasvattaa tuonti -muuttujan
```

```
 * osoittamaan seuraavaa puskurin paikkaa.
 * Sen jälkeen tulostetaan puskurin tila (jotta lelu olisi
 * hauskempi). Jos puskuri on täysi, niin laitetaan tuottaja
 * säie nukkumaan. Kun tuottaja on saanut tuotettua luvun,
 * herättää se kaikki nukkuvat säikeet.
 * Puskuri on aina epätyhjä, kun tuottaja on tuottanut sinne
 * luvun ja täysi aina kun se on tuottanut luvun puskuriin ja
 * puskurin seuraava paikka ei ole tyhjä.
 */
synchronized void tuota(){
    //Nukutetaan jos ei voida tuottaa.
    while(täysi){
        try {
            wait();
        } catch (InterruptedException e) {}
    }

    //Lisätään luku puskuriin.
    puskuri[tuonti] = tuonti;
    tuonti = (tuonti + 1) % 30;

    //Katsotaan tuliko puskuri täyteen.
    if(puskuri[tuonti] != 0)
        täysi = true;
    tyhjä = false;

    //Tulostetaan puskurin sisältö.
    System.out.print("Puskurissa on: .");
    for(int i = 0; i < 30; i++)
        System.out.print(puskuri[i] + ".");
    System.out.println();

    //Herätetään nukkuvat säikeet.
    notifyAll();
}

/**
 * Metodi kuluta() kuluttaa puskurista muuttujan vienti
 * osoittamasta paikasta kokonaisluvun. Tämä luku otetaan
 * talteen palautetaan lopuksi kutsujalle. Puskuriin asetetaan
 * tämän jälkeen arvo 0 merkitsemään, että paikka puskurissa
 * on nyt tyhjä.
 * Jos puskuri on tyhjä, niin ei voida kuluttaa ja säie
 * laitetaan nukkumaan. Kun kuluttaja on kuluttanut luvun, niin
 * se herättää kaikki nukkuvat säikeet.
 * Puskuri on aina epätäysi, kun kuluttaja on kuluttanut sieltä
 * luvun ja tyhjä aina kun kuluttaja on kuluttanut sieltä luvun
 * ja puskurin seuraava paikka on tyhjänä.
 */
synchronized int kuluta(){
    int kulutettu = 0;

    //Nukutaan jos ei ole kulutettavaa puskurissa.
    while(tyhjä){
       try {
           wait();
       } catch (InterruptedException e) {}
```

```
        }

        //Otetaan luku pois puskurista.
        kulutettu = puskuri[vienti];
        puskuri[vienti] = 0;
        vienti = (vienti + 1) % 30;

        //Tarkistetaan tyhjenikö puskuri.
        if(puskuri[vienti] == 0)
            tyhjä = true;
        täysi = false;

        //Kerrotaan mikä luku kulutettiin.
        System.out.println("Kulutin luvun: " + kulutettu);

        //Herätetään nukkuvat prosessit.
        notifyAll();
        return kulutettu;
    }

    /**
     * Run() metodilla käynnistetään säikeitä, jotka arpovat ovatko
     * ne tuottajia vai kuluttajia ja suorittavat sen mukaan
     * tuottamista tai kuluttamista.
     */
    public void run(){
        int tyyppi = (int)(Math.random() * 5);
        while(true){
            if(tyyppi == 0){
                tuota();
                try {
                    Thread.sleep(1200);
                } catch (InterruptedException e) {}
            }else{
                kuluta();
                try {
                    Thread.sleep(4500);
                } catch (InterruptedException e) {}
            }
        }
    }

    public static void main(String[] args){

        TuotaKuluta olio = new TuotaKuluta();
        Thread[] säikeet = new Thread[10];
        for(int i = 0; i < 10; i++){
            säikeet[i] = new Thread(olio);
            säikeet[i].start();
        }
    }

}
```

2. LueKirjoita.java

```
/**
 * Tämän luokan tarkoituksena on esitellä monitorin toteutusta javalla.
 * Tässä on totetutettu kriittisen vaiheen suojaus perinteisenä ja
 * klassisena monitori toteutuksena. Javan "sychnorized" voimasana
 * mahdollistaisi paremmankin toteutuksen.
 *
 * Ongelmana on kurssikirjassakin esitelty lukijoiden ja kirjoittajien
 * ongelma. Tässä monitorissa on lukijalle ja kirjoittajalle omat pre-
 * ja postprotokollat, jotka ne suorittavat ennen ja jälkeen kriittisen
 * vaiheen.
 */
public class LueKirjoita implements Runnable{

    /**
     * Arvon "arvo" lukeminen ja kirjoittaminen on tässä ns. kriittistä.
     */
    volatile int arvo = 1;
    /**
     * Arvo "lukijoita" pitää yllä kuinka monta lukijaa on lukemassa
     * "arvo":a. "Kirjoittaa" taas kertoo onko joku kirjoittajista
     * varannut "arvo":n kirjoittamista varten.
     */
    volatile int lukijoita = 0;
    volatile boolean kirjoittaa = false;

    /**
     * aloitaLuku() on lukijoiden preprotokolla. Niin kauan kuin on
     * kirjoittaja kirjoittamassa ei päästetä lukijoita lukemaan.
     * wait() käskyllä suorittava prosessi laitetaan nukkumaan, jotta
     * se ei suorittaisi turhaan. Kun prosessi pääsee lukemaan, se
     * kasvattaa lukijoiden määrää ja herättää kaikki odottavat prosessit
     * notifyAll() käskyllä. Käskyä notifyAll() on syytä käyttää
     * notify():n asemasta, sillä nyt halutaan päästää kaikki lukijat
     * kerralla lukemaan.
     */
    synchronized void aloitaLuku(){
        while(kirjoittaa){
            try{
                wait();
            }catch(InterruptedException e){}
        }
        lukijoita++;
        notifyAll();
    }

    /**
     * lopetaLuku() on lukijoiden postprotokolla, joka ilmoittaa
     * monitorille, että lukeminen on loppu vähentämällä lukijoiden
     * määrää. Jos prosessi oli viimeinen lukija, niin herätetään
     * kaikki prosessit, jolloin nukkuvat prosessit pääsevät tarkistamaan
     * jälleen pääsisivätkö suorittamaan kriittistä vaihetta
     * (while -silmukka) vai menevätkö takaisin nukkumaan (joku muu ehti
     * ensin).
     */
    synchronized void lopetaLuku(){
```

```
    lukijoita--;
    if(lukijoita == 0){
        notifyAll();
    }
}

/**
 * aloitaKirjoitus() on kirjoittajien preprotokolla, joka
 * kirjoittajan on läpäistävä päästäkseen kirjoittamaan. Jos
 * lukijoita tai kirjoittaja on jo päästetty kriittiselle alueelle,
 * laitetaan kirjoittaja nukkumaan. Kun kirjoittaja lopulta pääsee
 * suorittamaan kriittistä vaihetta, asetetaan "kirjoittaa" muuttujan
 * arvoksi tosi, jolloin kukaan muu ei pääse kriittiseen vaiheeseen.
 */
synchronized void aloitaKirjoitus(){
    while(kirjoittaa || (lukijoita > 0)){
        try{
            wait();
        }catch(InterruptedException e){}
    }
    kirjoittaa = true;
}

/**
 * lopetaKirjoitus() on kirjoittajan postprotokolla, jolla
 * kirjoittaja ilmoittaa saaneensa kirjoituksen valmiiksi. Tämä
 * ilmoitetaan asettamalla muuttujan "kirjoita" arvoksi epätosi
 * ja herättämällä nukkuvat proesessit.
 */
synchronized void lopetaKirjoitus(){
    kirjoittaa = false;
    notifyAll();
}

/**
 * run() metodilla käynnistetään säikeitä. Ensiksi arvotaan ollaanko
 * kirjoittaja vaiko lukija, jonka jälkeen suoritetaan säiettä, joko
 * lukien arvoa tai arpoen sille uuden arvon. Suorittamalla ohjelmaa
 * voi nähdä kuinka arvot vaihtuvat (ja kriittisen vaiheen suojaus
 * toimii). Säikeen suorittaminen jatkuu kunnes kirjoitetaan tai
 * luetaan nolla. Huomioi että aina ei välttämättä tule kirjoittajia
 * tai lukijoita ollenkaan.
 */
public void run() {
    int id = (int)(Math.random() * 5);
    boolean stop = false;
    while(true){
        if(id == 0){

            //preprotokolla
            aloitaKirjoitus();
            //kriittinen vaihe
            System.out.println("Kirjotin uuden arvon.");
            arvo = (int)(Math.random() * 10);
            if(arvo == 0)
                stop = true;
            //preprotokolla
```

```
                lopetaKirjoitus();

            }else{

                //preprotokolla
                aloitaLuku();
                //kriittinen vaihe
                System.out.println("Arvo on " + arvo +".");
                if(arvo == 0)
                    stop = true;
                //preprotokolla
                lopetaLuku();

            }
            if(stop)
                break;
           //Nukutaan vähän aikaa ettei tulosteta lukuja liian nopeasti.
            try {
                Thread.sleep(250);
            } catch (InterruptedException e) {}
        }
    }

    public static void main(String args[]) throws InterruptedException{
        LueKirjoita otus = new LueKirjoita();
        Thread eka = new Thread(otus);
        Thread toka = new Thread(otus);
        Thread kolmas = new Thread(otus);
        Thread neljäs = new Thread(otus);
        eka.start();
        toka.start();
        kolmas.start();
        neljäs.start();
    }
}
```

e. Käyttöohje

Kun useampi säie käyttää yhteistä muuttujaa samanaikaisesti muuttujan käsittelyssä saattaa tulla päällekkäisyyttä. Tämä aiheuttaa yleensä virheitä ohjelman suorituksessa. Yleensä nämä tilanteet hoidetaan monitoreilla mutta Javassa ei varsinaisesti ole monitoreita vaan synkronointi, joka on jopa voimakkaampi käsite. Synkronointia käytetään Javassa kun halutaan estää tällaiset virheet tai varmistaa että ensin suoritetun metodin muutokset näkyvät muissa synkronoiduissa metodeissa.

Tietoa miten yllä mainittuun Javan synkronointiin pääsee käsiksi:

-Object-luokassa on määritelty synkronointiin vaadittavat välineet ja näin ollen kaikki luokat perivät nämä välineet.

-Käytännössä näitä välineitä käytetään silloin kun suoritetaan useampaa säiettä samanaikaisesti. Luokat jotka suorittavat useampaa säiettä samanaikaisesti toteuttavat Runnable-luokan määrittelemän rajapinnan (`class Oma implements Runnable`).

-Synkronoitavat metodit määritellään sanalla `synchronized`. Huomioi, että `synchronized` määreen käyttäminen konstruktorissa aiheuttaa syntaksi virheen!

-Object-luokka tarjoaa metodin `wait()`, jolla säie laitetaan nukkumaan. Lisäksi object luokalta peritään metodit `notify()`, joka herättää jonkun nukkuvan `wait()`:lla nukutetun säikeen, ja `notifyAll()`, joka herättää kaikki `wait()`:ssa odottavat säikeet.

-Javassa on jokaisella oliolla nk. lukko, joka säikeen on saatava haltuunsa päästäkseen suorittamaan synkronoituja metodeja. Lukko voi olla vain yhden säikeen hallussa kerrallaan. Käytännössä tämä tarkoittaa, että synkronoituja metodeja pääsee suorittamaan vain yksi säie kerrallaan.

-Synkronointi tarjoaa Javassa helpon menetelmän samanaikaisuuden hallintaan.