JAVA MONITORIT

Monitorien käyttötarkoitus ja sovellusalue

Monitorit ovat Java –ohjelmointikielen sisäänrakennettu mekanismi säikeiden rinnakkaisuuden hallintaan. C.A.R. Hoare esitti ensimmäistä kertaa monitorin käsitteen vuonna 1974. Rinnakkaisuuden hallinta on monimutkaista ja virhealtista: monitorit tarjoavat tähän semaforeja luotettavamman ja helppokäyttöisemmän rakenteellisen työkalun.

Java monitorien perusidea

Javassa monitorina voi toimia mikä tahansa olio. Monitoriolion avulla kriittiset, suojaamista tarvitsevat toiminnot voidaan kapseloida olion sisään niin, että monitori kontrolloi näihin toimintoihin pääsyä. Monitorilla on aina lukko (lock) ja odotusjoukko (wait set). Monitorin kontrolloimat metodit ja koodilohkot varustetaan avainsanalla synchronized. Monitorien käyttöön liittyvät myös Object –luokan metodit wait(), notify() ja notifyAll().

Kun säie haluaa suorittaa suojattua toimintaa, sen pitää saada haltuunsa monitorin lukko, jonka voi omistaa vain yksi säie kerrallaan. Lukon avulla voidaan siis suojata kriittiset toiminnot rinnakkaisesti suoritettavilta säikeiltä. Ne säikeet jotka odottavat pääsyä suojattuun osioon, sijoitetaan monitorin odotusjoukkoon. Odotusjoukossa olevat säikeet odottavat lukon vapautumista ja oikeutta suorittaa monitorin valvomia toimintoja. Kun lukkoa hallussaan pitävä säie luopuu siitä, se antaa ilmoituksen (notify) lukon vapautumisesta odotusjoukon säikeille. Ilmoituksen saanut säie odottaa, kunnes ilmoittaja poistuu monitorista ja alkaa tämän jälkeen suorittaa toimintojaan. Ilmoittaminen voidaan kohdistaa yhteen satunnaisesti valittuun odotusjoukon säikeeseen (notify) tai kaikkiin odottaviin säikeisiin (notifyAll).

Javassa on myös luokkakohtaisia static-metodeita, joihin ei liity olioita. Luokkametodit voidaan myös määritellä synkronoiduiksi luokkakohtaisilla lukoilla. Luokkakohtainen ja

oliokohtainen lukko ovat toisistaan riippumattomia, joten nämä voivat olla suorituksessa yhtä aikaa.

Monitorin ja Java-monitorin yhteneväisyydet ja eroavaisuudet

Java monitorien perusidea on sama kuin perusmonitorienkin, mutta Javassa ei ole erillistä monitor-luokkaa vaan kuten edellä mainittiin jokainen olio voi toimia monitorina. Java-monitori toteuttaa monitoreille asetettua vaatimusta, että kääntäjän avulla tarkistetaan koodin oikeellisuutta ja näin vähentää virheiden määrää. Se täyttää kapselointi odotukset eli palvelun kutsujan ei tarvitse tietää miten yksittäinen palvelu on toteutettu eikä monitorin tekijän tarvitse tietää kuka, missä, miten ja milloin sen tuottamia palveluita kutsuu. Java-monitorit eivät myöskään salli viittauksia ulkopuolisiin muuttujiin, eikä samanaikaista saman objektin palveluiden kutsua (automaattinen poissulkeminen).

Java-monitorin suoritukseen päässyt säie käyttää Ilmoita ja Jatka -signalointisemantiikkaa. Toinen signalointisemantiikka vaihtoehto Ilmoita ja Keskeytä ei ole mahdollinen Javassa. Toisaalta taas Java-monitorissa on vaihtoehtoina ilmoittaa kaikille (notifyAll()) tai vain yhdelle satunnaisesti valitulle (Notify()) säikeelle monitorin vapautumisesta. Kaikille säikeille ilmoittaminen (NotifyAll()) johtaa kaikkien säikeiden vapautumiseen ja Java-monitorin sisälle pääsyehtojen uudelleen tarkistamiseen kaikkien säikeiden kohdalla. Perusmonitoreissa yleensä vain yksi prosessi odottaa ilmoitusta sisään pääsystä, kun taas kaikki Javan käynnistämä säikeet odottavat whilen kaltaisessa luupissa. Tämä johtaa yleensä säikeiden lukumäärän kasvaessa tehottomuuteen ja raskauteen. Javassa on myös mahdollista antaa säikeille kolme erilaista prioriteettia: min, max tai normaali.

Perusmonitoreissa on ehtomuuttujia käyttäen mahdollista keskeyttää tietyn palvelun toiminto odottamaan tietyn toiminnan valmistumista ja vapauttaa näin monitorin muut palvelut muiden käyttöön. Java-monitorit eivät toteuta täysin perusmonitorien IRR-ominaisuutta (immediate resumption requirement), niillä on vain yksi odotusjono monitorin ulkopuolella yhtä objektia kohden (E = W < S). Lisäksi tämä odotusjono ei ole

perusmonitorien tapaan FIFO-jono, niin se ei myöskään takaa sitä, että ensimmäisenä luotu säie pääse suoritukseen myös ensimmäisenä. Tämä tarkoittaa, että nälkiintyminen on täysin mahdollista Java-monitorissa.

Monitorien käyttöesimerkit

Aterioivat filosofit

Edsger Dijkstra esitti vuonna 1965 tietojenkäsittelyn rinnakkaisuuteen liittyvän aterioivien filosofien ongelman. Dijkstran esittämässä ongelmassa viisi filosofia istuu pyöreän pöydän ympärillä. Jokaisella filosofilla on edessään lautasellinen spagettia ja yksi haarukka lautasen vasemmalla puolella. Syödäkseen jokainen filosofi tarvitsee kuitenkin kaksi haarukkaa, sekä oman haarukkansa, että oikealla puolella istuvan filosofin haarukan. Filosofien elämä koostuu ajattelemisesta ja syömisestä. Filosofin tullessa nälkäiseksi hän yrittää saada itselleen lautasen vasemmalla ja oikealla puolella olevat haarukat. Mikäli filosofi onnistuu saamaan molemmat haarukat, syö hän jonkin aikaa kunnes tulee kylläiseksi ja laskee sitten haarukat takaisin pöydälle ja jatkaa miettimistä. Tavoitteena on luoda filosofeille toimintaohje, jonka avulla kukin filosofi pääsee syömään eikä ohjelma pääse lukkiutumaan.

Seuraavassa on esitetty ohjelman toteutus Java monitorien avulla:

```
class DiningMonitor
      {
            //ohjelman attribuutit
            //filosofien lukumäärän kertova attribuutti
            private final int numberOfPhils;
            // olio-taulukko, jonka jokainen olio toimii //yhden
            filosofin monitorina
            private Object[] monitorBuffer;
```

```
//haarukoiden järjestysnumerot sisältävä //taulukko
private int[] forkBuffer;
// Boolean muuttuja, jonka avulla testataan //filosofien
nälkiintymistilanne
private boolean checkStarvation = true;
// filosofien nälkiintymistilannetta ylläpitävä
//taulukko
private boolean[] starvationBuffer = null;


public DiningMonitor(int numPhils) throws
IllegalArgumentException
{
	int i;
	// Asetetaan parametrina saatu //filosofien määrä
	numberOfPhils-//muuttujan arvoksi ja luodaan uudet
	//taulukot monitoreja ja haarukoita //varten
	numberOfPhils = numPhils;
	monitorBuffer = new Object[numberOfPhils];
	forkBuffer = new int[numberOfPhils];

	// Käydään läpi yllä luodut uudet //taulukot ja..
	for(i=0;i < numberOfPhils;i++)
	{
		//asetetaan jokaisen haarukan //alkuarvoksi
		1 (käytettävissä)
		forkBuffer[i] = 1;
		monitorBuffer[i] = new Object();
	}
	//Tarkistetaan nälkiintymistilanne
	if(checkStarvation)
	{
		//uusi taulu nälkiintymistilanteen
		//ylläpitämiseksi
		starvationBuffer = new
				boolean[numberOfPhils];
		//Asetetaan aluksi jokainen //filosofi
		kylläiseksi
		for(i=0;i < numberOfPhils;i++)
		{
			starvationBuffer[i] = false;
		}
	}
}
```

```
// pyydetään haarukoita
public void takeForks(int i)
{
	//monitoroitu olio (ohjelmalohko)
	synchronized(monitorBuffer[i])
	{
		try
		{
			// pyydetään lupaa ottaa //haarukat
			testAvailable() -//metodilta
			while( !testAvailable(i) )
				monitorBuffer[i].wait();
		}
		catch(InterruptedException ie)
		{
			;
		}
	}
}
// vapautetaan käytetyt haarukat muiden //ruokailijoiden
käytettäviksi
public synchronized void putForks(int i)
{
	forkBuffer[i]=1;
	forkBuffer[(i+1)%numberOfPhils]=1;

	indicate((i+numberOfPhils-1)%numberOfPhils);
	indicate((i+1)%numberOfPhils);

}
// tarkistetaan, onko tarvittavat haarukat
//käytettävissä
//monitoroitu metodi..
private synchronized boolean testAvailable(int i)
{
	boolean cond,starveCond = true;

	// testataan, onko haarukat vapaina
	cond = (forkBuffer[i]==1) &&
			(forkBuffer[(i+1)%numberOfPhils]==1);

	if(checkStarvation)
	{
```

```
                        // Tarkistetaan, onko //naapurifilosofit
                        nälkiintymässä
                        starveCond =
                        !starvationBuffer[(i+numberOfPhils-1)
                                          %numberOfPhils] &&
                        !starvationBuffer[(i+1)%numberOfPhils];

                        if(!cond)
                        {
                              // asetetaan haarukoita //pyytänyt
                              filosofi //nälkiintyväksi
                              if(starveCond)
                                    starvationBuffer[i] = true;
                        }

                  }

                  if(cond && starveCond)
                  {
                        // merkitään haarukat varatuiksi..
                        forkBuffer[i]=0;
                        forkBuffer[(i+1)%numberOfPhils]=0;

                        //ja nollataan nälkiintymistilanne
                        if(checkStarvation)
                              starvationBuffer[i] = false;
                  }

                  return(cond && starveCond);
            }

            // Vapautetaan haarukat muiden käytettäviksi
            private void indicate(int i)
            {
                  //monitoroitu olio
                  synchronized(monitorBuffer[i])
                  {
                        //vapautetaan lukko odotusjoukon
                        //säikeille
                        monitorBuffer[i].notify();
                  }
            }
      }
```

Ohjelmassa jokaiselle filosofille annetaan aluksi järjestysnumero (metodi DiningMonitor). Järjestysnumeron perusteella kontrollioliot tietävät filosofin paikan pöydässä (monitorBuffer, forkBuffer, starvationBuffer). Annettujen järjestysnumeroiden avulla luodaan uudet taulukot filosofien monitorien, haarukoiden ja nälkiintymistilanteen hallintaa varten. Filosofien määrä määrittelee taulukkojen koot. Jokaiseen taulukkoon lisätään kullekin filosofille oma alkio. Kunkin taulukon alkio alustetaan alussa ruokailuun pyrkimisen kannalta suotuisaan tilaan eli filosofit asetetaan kylläisiksi, kaikki haarukat saataville ja monitorien valvomat toiminnot suoritettaviksi.

Filosofin tullessa nälkäiseksi kutsuu ohjelma DiningMonitor -luokan metodia takeForks(), jolle välitetään parametrina nälkäisen filosofin järjestysnumero. TakeForks() metodin monitorina toimii yksi oliotaulukon monitorBuffer alkio. TakeForks() metodi välittää ruokailemaan pyrkivän filosofin järjestysnumeron edelleen testAvailable() metodille, joka tarkistaa aluksi filosofin järjestysnumeron perusteella, onko filosofin molemmin puolin löytyvät haarukat vapaana käytettäviksi. Seuraavaksi metodi tarkistaa, ettei filosofin kumpikaan naapuri ole nälkiintymässä. Jos haarukat ovat käytettävissä eivätkä naapurit ole nälkiintymässä, merkitsee metodi haarukat varatuiksi, nollaa filosofin nälkiintymistilanteen ja antaa takeForks() -metodille luvan päästää filosofin syömään välittämällä tälle arvon true.

Filosofin tavoitellessa haarukoita, voi hän ottaa haarukan vain, jos hänen kumpikaan naapuri ei ole nälkiintymässä. Jos haarukat eivät ole käytettävissä eikä filosofin naapurit ole nälkiintymässä, asettaa metodi haarukoita pyytäneen filosofin nälkiintyväksi ja välittää takeForks() metodille vastaukseksi false, jolloin säie siirtyy odotustilaan ja monitorBuffer[i] -oliotaulukon odotusjoukkoon. Niin ikään, jos filosofin pyytämät haarukat eivät ole saatavilla ja filosofin naapureista

toinen tai molemmat ovat nälkiintymässä, palautetaan takeForks() –metodille false eikä ruokailemaan pyrkinyttä filosofia aseteta nälkiintyväksi. Näin jokainen filosofi pääsee jossakin vaiheessa syömään, eikä yksikään pääse nälkiintymään.

Koska metodi takeForks() on synkronoitu vain oliotaulukon monitorBuffer alkion suhteen, voi metodia suorittaa useampi säie yhtä aikaa. Tämän vuoksi metodin testAvailable() on oltava synkronoitu koko DiningMonitor –olion suhteen, jotta haarukkataulukkoa ei päästäisi muuttamaan useammasta säikeestä yhtä aikaa.

Filosofin päästyä syömään kutsuu ohjelma seuraavaksi putForks() metodia, joka vapauttaa haarukat muiden filosofien käytettäviksi. PutForks() metodille välitetään niin ikään ruokailleen filosofin järjestysnumero, jonka perusteella metodi merkitsee ruokailleen filosofin haarukat vapaiksi asettamalle niiden arvoksi 1. Metodi kutsuu vielä säikeen lukon vapauttamisesta viestittävää metodia indicate(), joka ilmoittaa haarukoiden vapautumisesta niitä odottaville filosofeille (odotusjoukon säikeille). Koska jokaisella filosofilla on oma monitori, jonka odotusjoukossa se on (monitorBuffer[i]), voidaan haarukoiden vapautumisesta viestittää suoraan niitä filosofeja, joita tämä saattaa koskea.

Nukkuva parturi

Nukkuva parturi –ongelma on niin ikään Edsger Dijkstran keksimä rinnakkaisuudenhallintaa kuvaava ongelma. Parturiliikkeessä on yksi parturi, yksi parturin tuoli ja joukko odotustuoleja parturin asiakkaita varten. Kun parturilla ei ole yhtään asiakasta, parturi istuu tuolissansa ja nukkuu. Kun liikkeeseen saapuu uusi asiakas, tämä joko herättää nukkuvan parturin, tai jos parturi leikkaa parhaillaan toisen asiakkaan hiuksia, istuutuu odotusaulan vapaalle tuolille odottamaan omaa vuoroaan. Mikäli kaikki odotusaulan tuolit

ovat varattuja, poistuu viimeksi liikkeeseen tullut asiakas paikalta saamatta palvelua.

Seuraavassa on esitetty nukkuva parturi –ongelma Java monitorien avulla:

```
//Asiakkaiden toimintaa ohjaava proseduuri
procedure customer_itinerary {
        boolean succeeded;
        succeeded = Customers.TryHaircut();
        if (!succeeded) {
           no_haircut();
        }
}
//Parturin toimintaa ohjaava proseduri
procedure barber_itinerary {
        while (1) {
            Barber.NextCustomer();
            CutHair();
            Barber.CustomerDone();
        }
}

//Customer-luokka
Class Customers {
      //asetetaan parturintuoli vapaaksi ja
      //vapaita odotustuoleja viisi kappaletta
      boolean barber_chair_free = TRUE;
      int free_waiting_chairs = 5;

       //monitoroitu ohjelmalohko
       synchronized boolean TryHaircut {
             //onko yhtään odotustuolia vapaana
                if (free_waiting_chairs == 0) {
                        return FALSE;
                 }
             // Tarkistetaan, onko parturilla asiakas tai
             // asiakkaita odottamassa odotustuoleilla, jos
             //parturilla on asiakas ja odotustuoleilla on //vielä
             tilaa istutaan odotustuolille
                 if ((barber_chair_free == FALSE)
                   ||  (free_waiting_chairs < 5)) {
             //vähennetään vapaiden odotustuolien määrää
```

```
                free_waiting_chairs--;
              //odotetaan omaa vuoroa (asetetaan säie //odottavaan
              tilaan..)
                 wait();
                 //oman vuoron tultua vapautetaan odotustuoli
                 free_waiting_chairs++;

              }
            //varataan parturin tuoli
            barber_chair_free = FALSE;
            //ilmoitetaan parturi-luokan metodilla DoNotify
            //parturille, että tuoliin on istuutunut asiakas
            barber.DoNotify();
            //kutsutaan Barber-luokan metodia WaitForCutToFinish
            barber.WaitForCutToFinish();
            //asetetaan parturituoli vapaaksi
            barber_chair_free = TRUE;
            //
            barber.DoNotify();
            //ilmoitetaan parturituolinvapautumisesta //jonossa
            oleville odotusjoukonsäikeille
                  notify()
            //hiustenleikkaus suoritettu
            return TRUE;
      }
      //monitorilla suojattu metodi
      synchronized boolean CheckChairFree() {
                return barber_chair_free;
      }
}
//Barber-luokka
Class Barber {
      //Monitoriksi määritetty metodi
      synchronized void NextCustomer {
               //Tarkistetaan customer-luokan metodilta, //istuuko
               parturintuolissa asiakas
               if (Customers.CheckChairFree() == TRUE) {
              // jos tuoli on vapaana, odotetaan,  että joku
            //istuutuu parturin tuoliin
                 wait();
               }

        }
```

```
        //monitoriksi määritetty metodi
        synchronized void CustomerDone() {
        //ilmoitetaan, että hiustenleikkaus on valmis
                    notify();
                    wait();
          }
        //monitoriksi määritetty metodi
        synchronized void DoNotify() {
                    notify();
        }
        //monitoriksi määritetty metodi
        synchronized void WaitForCutToFinish () {
        //odotetaan, että hiustenleikkaus saadaan valmiiksi
                    wait();
        }

}
```

Parturin saapuessa töihin kutsuu parturin toimintaa ohjaava proseduuri barber_itinerary Barber luokan metodia NextCustomer, joka tarkistaa, istuuko parturintuolissa jo asiakas odottamassa palvelua. Jos tuoli on tyhjä, asetetaan metodia suorittava säie odottavaan tilaan wait()-komennolla. Näin säie luovuttaa monitorin hallinnan pois ja jää odottamaan säikeen etenemisen mahdollistavaa notify-viestiä joltakin toiselta säikeeltä. Eli parturi ryhtyy nukkumaan ja odottamaan seuraavaa palveltavaa asiakasta.

Asiakkaan saapuessa parturiin kutsuu asiakkaiden toimintaa ohjaava proseduuri customer_itinerary Customer-luokan metodia TryHaircut. Metodi tarkistaa aluksi, onko parturin odotusaulan kaikki tuolit varattuja. Jos kaikki odotustuolit ovat varattuja, palauttaa ohjelma metodia kutsuneelle proseduurille paluuarvona FALSE. Asiakas ei onnistunut saamaan palvelua. Jos odotustuoleilla on vielä tilaa, tarkistaa ohjelma seuraavaksi, onko parturi varattuna tai odottavia asiakkaita odotusaulan tuoleilla. Parturin tai osan odotustuoleista ollessa varattuna, varaa ohjelma asiakkaalle yhden odotusaulan tuoleista vähentämällä tuolitilannetta ylläpitävän attribuutin free_waiting_chairs arvoa yhdellä.

Vastaavasti jos odotusaula on tyhjä ja parturin tuoli on vapaa, merkitsee ohjelma parturin tuolin varatuksi asettamalla attribuutin barber_chair_free arvoksi false. Asiakas istuutuu parturin tuoliin. Tämän jälkeen ohjelma herättää nukkuvan parturin kutsumalla Barber luokan metodia DoNotify, joka antaa Notify-viestin Barber-luokan NextCustomer metodissa odottavalle säikeelle. Säie pääsee nyt jatkamaan suoritusta. Parturi ryhtyy näin leikkaamaan asiakkaan hiuksia.

Seuraavaksi Customer proseduuri suorittaa Barber-luokan metodin WaitForCutToFinish eli asiakas odottaa, että parturi saa hiukset leikattua. Metodin säie asetetaan odotustilaan. Parturi proseduuri kutsuu metodia CustomerDone, joka antaa WaitForCutToFinish metodissa odottavalle säikeelle luvan jatkaa eteenpäin antamalla tälle viestin notify. Metodia CustomerDone suorittava säie asetetaan odotustilaan odottamaan viestiä siitä, että asiakas on poistunut tuolista. Asiakasproseduuri ilmoittaakin seuraavaksi parturituolinvapautumisesta asettamalla attribuutin barber_chair_free arvoksi TRUE ja kutsumalla Barber-luokan metodia DoNotify, joka vapauttaa CustomerDone metodissa odottavan säikeen. Tämän jälkeen Customer proseduuri antaa seuraavalle odotustuolilla odottavalle asiakkaalle viestin notify päästäen tämän parturin tuolille. Ennen tuolille istuutumista vapauttaa asiakas vielä varaamansa odotustuolin kasvattamalla free_waiting_chairs attribuutin arvoa yhdellä. Tämän jälkeen ohjelma etenee edellä esitetyn mukaisesti.

LÄHTEET

Ben-Air, M, Principals *of concurrent and distributed programming*. Addison-Wesley, 2006

Löflund, J. 2002. Edsger Wybe Dijkstra. Helsingin Yliopisto. [WWW-dokumentti]. http://www.cs.helsinki.fi/u/kerola/tkhist/k2002/alustukset/Dijkstra/dijkstra.html

OSU Oregon State University. [WWW-dokumentti]. *http://classes.engr.oregonstate.edu/eecs/fall2002/cs411/barbershop.txt*

Oulun yliopiston tietojenkäsittelytieteen laitos. Monitorit. [WWW-dokumentti]. http://www.tol.oulu.fi/~avesanen/Rinn_Ohjelm/Luennot/Monitorit.html

Sun Developer Network. Monitors. http://java.sun.com/developer/Books/performance2/chap4.pdf

Tampereen teknillinen yliopisto: Ohjelmointikielten periaatteet – Luennot [WWW-dokumentti].. *http://www.cs.tut.fi/~okp/luennot/kalvot/rinnan06.pdf#search=%22java%20monitorit%22*

Universität Paderborn. Monitors in general and in Java [WWW-dokumentti].. *http://ag-kastens.uni-paderborn.de/lehre/material/ppje/folien/comment18-37.2.pdf*