Helsingin Yliopisto, tietojenkäsittelytieteen laitos

Rinnakkaisohjelmointi (syksy 2006)

Liite 1. Projektin tulokset (Semaforit Javassa)

Jukka Hyvärinen
Aleksanteri Aaltonen

a. Käyttötarkoitus ja sovellusalue

Semafori on käyttöjärjestelmän tai laitteiston tasolla toteutettu rakenne prosessien synkronointiin. Semaforeja voi käyttää moniin tehtäviin, mutta yksinkertaisimmillaan ne toimivat mutual-exclusion -takeena eli mutexina tarjoten tehokkaan ja yksinkertaisen tavan ohjelmakoodin kriittisen alueen suojaukseen: vain yksi prosessi pääsee suorittamaan tiettyä koodilohkoa kerrallaan. Tämänkaltaista semaforia kutsutaan mutexin lisäksi binäärisemaforiksi, koska sillä voi olla ainoastaan kaksi arvoa: vapaa tai käytössä. Yleisellä semaforilla voi olla suurempiakin arvoja, jotka kuvastavat käytössä olevien resurssien määrää. Nämä resurssit voidaan yleensä jakaa useamman prosessin kesken, siten että varattuna olevien resurssien määrä pysyy aina samana tai pienempänä kuin kaikkien käytössä olevien resurssien määrä, tai yhdistelemällä useita semaforeja oikealla tavalla voidaan kaikki resurssit varata kerrallaan valitulle määrälle (esim. yhdelle) prosesseja. Esimerkkinä tästä on lukijoiden ja kirjoittajien ongelma, jossa lukijoita voi olla aktiivisena kuinka monta tahansa, kunhan kirjoittajia ei ole. Toisaalta yhden kirjoittajan ollessa aktiivisena mitään muuta prosessia ei saa päästä kirjoittamaan tai lukemaan.

Yksinkertaisuudestaan huolimatta semaforien avulla on mahdollista toteuttaa melkein mikä tahansa kriittisen alueen ongelma, jossa on käytössä yhteistä muistia, mutta monimutkaisemmissa tapauksissa on järkevämpää käyttää jotain erikoistuneempaa rakennetta.

b. Perusidea

Semaforin perusidea on yksinkertainen: se tarjoaa atomisen laskurin kasvattamis- tai pienentämisoperaation, sekä toiminnallisuuden prosessien nukuttamiseen tai herättämiseen. Semaforeilla on aina 2 perusoperaatiota: WAIT ja SIGNAL. Prosessin kutsuessa WAIT-käskyä semafori atomisesti laskee laskurin arvoa yhdellä, tai jos laskurin arvo on ei-positiivinen, nukuttaa prosessin asettamalla sen SUSPENDED -tilaan. SIGNAL vastaavasti katsoo, onko semaforilla nukutettuja prosesseja, ja jos on, se herättää niistä yhden asettamalla sen READY TO RUN -tilaan. Muussa tapauksessa se kasvattaa laskurin arvoa yhdellä. Semaforin implementaatio määrää, millä perusteella herätettävä prosessi valitaan. Vahva semafori herättää aina sen prosessin, joka nukutettiin

ensimmäisenä, kun taas heikko semafori herättää satunnaisen semaforissa nukkuvan prosessin. On myös olemassa busy-wait -tyyppisiä semaforeja, jotka eivät muuta prosessin tilaa mitenkään. Sen sijaan ne käyttävät silmukaa, jossa prosessi luuppaa, kunnes laskurin arvo on taas positiivinen. Busy-wait -semafori on kevyempää toteuttaa, mutta toisaalta luuppaava prosessi syö kokoajan laskentatehoa prosessorilta ja sen takia niitä ei kannatakaan käyttää muissa kuin moniprosessorikoneissa, eikä kovin usein niissäkään.

Semaforin alkuarvon määrittää sen käyttäjä. Useimmissa implementaatioissa hyväksyttyjä arvoja ovat kaikki ei-negatiiviset kokonaisuluvut (binäärisemaforin tapauksessa 0 ja 1), mutta Javan toteutus hyväksyy epästandardisti myös negatiiviset alkuarvot. Tämä tarkoittaa, että SIGNAL -komentoa täytyy kutsua tietyn monta kertaa, ennenkuin yksikään prosessi voi ohittaa semaforin. Tämä voi olla hyödyllistä esim. tuottaja/kuluttaja -tapauksessa, jossa kuluttaja lukee kerrallaan useita resurssialkioita. Perusoperaatioiden lisäksi monissa toteutuksissa semaforit tukevat suoraan useiden resurssialkioiden varaamista ja vapauttamista kerrallaan, sekä vapaiden alkioiden määrän tarkistamista.

c. Yhteneväisyydet / eroavaisuudet kurssilla esitettyyn perustapaukseen

Javassa semaforeja varten on selkeästi nimetty luokka, Semaphore. Se löytyy java.util.concurrent –paketista. Javan semaforien komentojen terminologia eroaa hieman normaalista, WAIT-käskyä vastaa acquire (tämä johtunee siitä, että Javassa wait on jo varattu monitoreille), kun taas SIGNALia vastaa release. Acquiren seurauksena nukutettu säie voidaan herättää keskeyttämällä säie interrupt -metodilla. Jos tätä ei haluta, voidaan käyttää acquireUninterruptibly-metodia.

Javassa voi myös hakea suoraan listan semaforissa odottavista säikeistä, sekä tarkistaa, kuinka paljon resursseja on vapaana. Lisäksi mukana on muutama atominen lisäoperaatio, jotka voisi toteuttaa yllämainittujen yhdistelmänäkin, kuten tryAcquire -metodi, joka ei pysäytä prosessia missään vaiheessa, vaan palauttaa false siinä tapauksessa että resursseja ei ole vapaana. tryAcquiresta on variantti, joka odottaa määritellyn ajan semaforin vapautumista ennen falsen palauttamista. Vapaiden resurssien määrää voi myös vähentää mielivaltaisesti reducePermits -metodilla.

Javan semaforit eivät oletuksena ole vahvoja (Javan oma termi on ”reiluja”), mutta semaforilla on vaihtoehtoinen konstruktori, jolla tällaisen vahvan semaforin voi luoda.

Muilta osin Javan semaforien toteutus on kohtalaisen standardinomainen. Toisin kuin monitorien tapauksessa, semaforeilla ei ole omistajasäiettä, joten semaforin vapauttavaa release-käskyä voi kutsua mistä tahansa säikeestä.

Semaphore-luokan lisäksi javassa useita lukkoluokkia java.util.concurrent.locks -paketissa. Nämä ovat oikeastaan monitoreita, mutta semaforien kannalta kiinnostava on erityisesti ReentrantReadWriteLock, joka ratkaisee yhden perinteisesti semaforeilla toteutetuista rinnakkaisuusongelmista, nimittäien lukijoiden ja kirjoittajien ongelman. Kyseinen lukko mahdollistaa siis useiden lukijoiden tai yhden kirjoittajan kerrallaan aktiivisena olemisen. Käytännössä lukko lukemista varten varataan readLock().lock() -metodilla ja vapautetaan vastaavalla unlock() -metodilla. Kirjoittamista varten lukko varataan writeLockin kautta. Lukijoita ja kirjoittajia ei luonnollisesti päästetä suoritukseen samaan aikaan.

d. Esimerkit

Aterioivien filosofien ongelma

Aterioivien filosofien ongelmassa on 5 filosofia, joilla jokaisella on 2 tehtävää: ensin filosofi ajattelee ja sitten syö, jonka jälkeen hän siirtyy taas ajattelemaan syvällisiä asioita. Filosofeilla on käytössään pyöreä pöytä, jossa on 5 lautasta ja 5 haarukkaa, mutta jokainen filosofi tarvitsee 2 haarukkaa syödäkseen. Ongelmana on jakaa haarukat siten, että jokainen haarukka on enintään yhden filosofin käytössä kerrallaan (kriittisen alueen suojaus) ja ettei kukaan filosofi nälkiinny, eikä tilanne jää jumiin.

Ongelman ratkaiseva esimerkkikoodi on tiedostossa s06_philosophers.java

PlusMinus-ohjelma laskuharjoitustehtävistä

PlusMinus on ohjelma, jossa on yhteisessä muistissa käytettävissä kuusi nollalla alustettavaa muuttujaa: X1, X2, X3, X4, Ctrl ja Sum. PlusMinus-ohjelmassa on viisi laskentaprosessia jotka suorittavat 50 kertaa silmukan, jonka kriittinen vaihe on:

- Jos muuttujan Ctrl arvo on pariton, vähennetään kaikista X-muuttujista luku 1. Jos muuttujan Ctrl arvo on parillinen, lisätään kaikkiin X-muuttujiin luku 2.
- Lopuksi lisätään muuttujaan Ctrl arvo 1 ja lasketaan kaikkien X-muuttujien summa muuttujaan Sum.

Esimerkkikoodissa kriittisen vaiheen suojaus on toteutettu Javan semaforeilla.

PlusMinus-ohjelmiston koodi on tiedostossa s06_PlusMinus.java

e. Käyttöohje

Javan semaforien käyttö on helppoa. Käyttö aloitetaan luomalla ensin semafori-olio new-käskyllä. Konstruktorille annetaan resurssien määrä alkutilanteessa sekä haluttaessa optionaalinen tieto siitä, halutaanko semaforin toimivan reilusti vai epäreilusti. Tarvittava Semaphore-luokka on java.util.concurrent-luokan aliluokka.

```
public class MyApp implements Runnable {

        // Luodaan reilu binäärisemafori ja alustetaan se alkuarvolla 1.
    private static Semaphore semaphore = new Semaphore(1, true);

    //  Sisältää kriittistä koodia, jota voi päästä suorittamaan vain 1 säie kerrallaan
    public void importantFunction() {

            try {
                // Semafori varataan suorittavalle säikeelle acquire-komennolla, jonka täytyy
                olla poikkeuksen varalta try-catch -lohkossa.
                semaphore.acquire();

            } catch (InterruptedException e) {
                    // Säie keskeytettiin ja tämä keskeytys voitaisiin käsitellä.
            }
            // Nyt vain yksi säie pääsee suorittamaan tätä koodia
            doImportantStuff();

        // Semafori vapautetaan lopuksi release-komennolla.
            semaphore.release();
      }

      public void run() {
            // Suoritetaan tärkeä funktio
            importantFunction();
      }

    public static void main(String[] args) {
            // Luodaan 10 säiettä ja päästetään ne vuorollaan tärkeään funktioon
            for (int i = 0; i < 10; ++i) {
                    new Thread(MyApp()).start();
            }

    }
}
```