Semaforit Javassa

Mari Kononow, Eveliina Mattila, Sindi Poikolainen

13.12.2008

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Johdanto

Semaforeja käytetään yhtenä ratkaisumenetelmänä kriittisen vaiheen ongelmaan. Semafori ratkaisee poissulkemisongelman seuraamalla ja rajoittamalla kriittiseen vaiheeseen pääsevien prosessien määrää, mutta ei estä prosessien lukkiutumista. Mikäli ohjelmoijan semaforille asettama maksimiraja ei vielä ole täynnä, päästää semafori prosessin etenemään. Jos sallittujen prosessien määrä on jo täyttynyt, joutuu prosessi ensin odottamaan semaforilta etenemislupaa siihen saakka, että jokin aiemmin kriittiseen vaiheeseen etenemään päässyt prosessi luovuttaa omansa. Käyttö tapahtuu prosessissa siten, että kriittisen vaiheen edelle ja jälkeen lisätään semaforin operaatioiden kutsut. Idean alun perin kehittänyt Dijkstra käytti näistä nimityksiä P/V. Operaatioista käytetään yleisesti myös nimiä wait/signal.

Prosessissa kriittistä vaihetta edeltävä kutsu P tai wait pyytää semaforilta etenemislupaa. Jos semaforille asetettuja etenemislupia on vielä jäljellä, vähentää semafori tästä määrästä yhden ja sallii prosessin edetä. Muussa tapauksessa prosessi joutuu odottamaan vuoroaan. Kriittisestä vaiheesta poistuttaessa kutsuttava operaatio V tai signal johtaa semaforin etenemislupien määrän lisäämiseen, jolloin mahdollisesti odottamassa olevat prosessit voidaan päästä jatkamaan.

Semaforien käyttö Javassa

Javassa semaforit on toteutettu luokan java.util.concurrent.Semaphore ilmentyminä, jotka alustetaan kutsulla `Semaphore(int permits) tai Semaphore(int permits, boolean fair)`. Integer-arvo määrittää semaforin tarjoamien lupien maksimimäärän, totuusarvomuuttuja `fair` taas kertoo, suositaanko pisimpään odottamaan joutuneita prosesseja. Semafori, jonka tarjoamien lupien maksimimäärä on rajoitettu kymmeneen, luodaan ja alustetaan esimerkiksi kutsulla `Semaphore sem = new Semaphore(10);`

Prosessi pyytää semaforilta etenemislupaa metodia `void acquire()`, joka jää odottamaan luvan saamista. Mikäli ohjelmoija ei halua, että prosessi jää odottamaan, voidaan käyttää aksessoria boolean `tryAcquire()`, joka yrittää saada semaforilta

luvan ja palauttaa arvon `false`, mikäli lupia ei ole jäljellä. Jos odotukseen halutaan käyttää vain rajoitettu aika, metodille `tryAcquire(long time, TimeUnit unit)` määritetään time-parametrillä miten kauan lupaa odotetaan. Odotusajan täyttyessä metodi palauttaa arvon `false`, jos lupia ei ollut jäljellä eikä niitä ehtinyt vapautua.

Yleinen Semafori

Yleisessä semaforissa voi saada arvokseen minkä tahansa ei-negatiivisen arvon. Yleisiä semaforeija käytetään laskemaan mm. Kuinka monta prosessia voi päästä suorittamaan kriittistä vaihetta, alustamalla semaforin arvo halutulla ei-negatiivisella arvolla (n). Kun n-määrä prosesseja on suorittamassa kriittistä vaihettaan, semaforin arvo on 0, ja muut prosessit joutuvat odottamaan “blocked”-tilassa vuoroaan, kunnes jokin suorittavista prosesseista päättää oman suorituksensa ja signalloi odottavaa.

Yleistä semaforia voi myös käyttää samalla tavalla kuin counting-semaforia. Sen avulla voi pitää kirjaa saatavilla olevien resurssien määrästä. Counting-semafori ja yleinen semafori mielletään usein samoiksi.

Alla on yleinen semafori toteutettu käyttämällä Javan wait() ja notify()-metodeita. Toteutuksessa semaforille annetaan alkuarvo, joka rajoittaa suoritukseen pääsevien prosessien määrää.

```
class semaphore {

  protected int value = 0 ;

  protected semaphore() { value = 0 ; }

  protected semaphore(int initial) { value = initial ; }

  public synchronized void P() {
    value-- ;
    if (value < 0)
      try { wait() ; } catch(  InterruptedException e ) { }
  }

  public synchronized void V() {
    value++ ; if (value <= 0) notify() ;
  }
}
```

Counting- Semafori

Counting-semaforin idea on hallita saatavilla olevia resursseja. Counting-semafori on toteutettu yksinkertaisten lukkojen pohjalta siten, että semafori on säikeistä vapaa laskuri. Se saa alkuarvokseen käytettävissä olevien resurssien määrän. Esimerkiksi tietokantaa päivättävässä ohjelmistossa, laskurin määrä olisi saatavilla olevien tietokantayhteyksien määrä. Kun prosessi saa semaforin haltuun, saatavilla olevien tietokantayhteyksien määrää vähennetään yhdellä. Kun resurssia käytetään semafori vapautetaan ja laskuri kasvaa yhdellä. Mikäli prosessi yrittää saada semaforin haltuun kun kaikki resurssit ovat käytössä, se joutuu odottamaan vuoroaan “blocked”-tilassa.

Javalla counting-semafori käyttämällä Javan wait()- ja notify()-operaatioita on toteutettu seuraavasti:

```
public class CountingSemaphore {
  private int signals = 0;

  public synchronized void take() {
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
  }

}
```

Binäärisemafori

Binäärisemafori eroaa yleisestä ja “counting” -semforeista siten, että sillä on totuusarvomuuttuja kokonaislukumuuttujan sijaan. Binäärisemaforin alkuarvo astetaan 1:ksi eli todeksi. Semaforin odotusoperaatio odottaa kunnes muuttujan arvo on 1 (true) ja sen

jälkeen asettaa sen 0:ksi (false). Signallointioperaatio muuttaa arvon 1:ksi (true) herättäen suorititettavan prosessin. Binäärisemaforin käyttö on tehokas mutexiin. Mutexin käyttö suojaa ainoastaan mutual exclusionin kun taas binäärisemaforia käytetään sekä synkronointiin, että mutual exclusioniin.

Ben-ari binäärisemaforin avulla toteutettu mutual exclusion [BenA06]):

Algorithm 6.1: Critical section with semaphores	
binary semaphore ← (1,Ø)	
p	q
loop forever p1: non-critical section p2: wait (S) p3: critical section p4: signal(S)	loop forever q1: non-critical section q2: wait (S) q3: critical section q4: signal(S)

Javassa binäärisemafori toteutetaan parametroimalla Semaphore-luokan konstruktori arvolla 1. Lupia suoritukseen on siis 0 tai 1 kappaletta (tosi, epätosi). Semaforeilla ei ole tietoa omistuksesta, ja näin ollen binäärisemaforin lukon voi aukaista muukin säie kun sen omistaja. Binäärisemaforin käyttö voi olla tehokas tapa esimerkiksi lukkiutumisesta toipuessa.

Javalla toteutettu binäärisemafori:

```
public class BinarySemaphore {
   private boolean locked = false;

   public AnotherBinarySemaphore() {}  // constructors
   public AnotherBinarySemaphore(boolean initial) {locked = initial;}
   public AnotherBinarySemaphore(int initial) {
      if (initial < 0 || initial > 1)
         throw new IllegalArgumentException("initial<0 || initial>1");
      locked = (initial == 0);
   }
```

```
    public synchronized void P() {
       while (locked) {
          try { wait(); } catch (InterruptedException e) {}
       }
       locked = true;
    }

    public synchronized void V() {
       if (locked) notify();
       locked = false;
    }
}
```

Busy-wait -semafori

Busy-wait -semafori eroaa muista semaforeista siten, että se ei sisällä sisäistä prioriteettijonoa kutsuvista prosesseista. Odotus on toteutettu busy-wait -silmukalla, jossa prosessi pyörii niin kauan kunnes se pääsee semaforin ohi. Prosessi ei siis mene suspended-tilaan, vaan käyttää suoritinta koko odotuksen ajan, mikä vie turhaan suorittimen laskentatehoa. Busy-wait -semaforin käyttö ei tämän vuoksi ole suositeltavaa, ainakaan mikäli odottavilla prosesseilla ei ole omaa prosessoria käytettävissään.

Javassa busy-wait -semaforin käyttö ei juuri eroa binäärisemaforin käytöstä, sillä jos kutsussa lupien määräksi annetaan 1 ja fair -parametri jätetään pois, toimii semafori poissulkemisongelman ratkaisevan busy-wait -semaforin kaltaisesti. Semaforin sisäinen toteutus voidaan toteuttaa myös itse esimerkiksi pelkällä while -loopilla, jossa odotetaan semaforin arvon kasvamista nollaa suuremmaksi. Toteutuksessa voidaan myös mahdollistaa kutsujärjestyksen mukainen suoritus käyttämällä kutsuttavan metodin määrityksessä Synchronized -luokkaa, joka sallii vain yhden prosessin pääsyn kerrallaan busy-wait -silmukkaan.

Jaetut semaforit (Split semaphores)

Jaetuilla semaforeilla tarkoitetaan kahta tai useampaa semaforia, joiden S.value – arvojen summa on vakio. Nämä avulla saadaan prosessi odottamaan jonkin toisen prosessin toiminnon päättymistä. Jaetut semaforit on usein esitelty tuottaja-kuluttaja-ongelman yhteydessä, jossa tuottajaprosessin tulee tarkistaa, ettei tule ”ylituotantoa” ja kuluttajan tulee tarkistaa, että tuotetta on jäljellä.

Ben-Arin esitys tuottaja-kuluttaja -ongelmasta jaettuja semaforeja käyttäen (Alg.6.8 [BenA06]):

Algorithm 6.8: producer-consumer (finite buffer, semaphores)	
finite queue of dataType buffer ← empty queue semaphore notEmpty ← (0, Ø) semaphore notFull ← (N, Ø)	
producer	consumer
DataType d loop forever p1: d ← produce p2: wait(notFull) p3: append(d, buffer) p4: signal(notEmpty)	DataType d loop forever q1: wait(notEmpty) q2: d ← take(buffer) q3: signal(notFull) q4: consume(d)

Tässä esimerkissä semaforit notEmpty ja notFull muodostavat jaettujen semaforien kokonaisuuden. Niiden arvojen summa on aina N, mikä voidaan ilmaista myös invariantilla:

```
notEmpty + notFull = N.
```

Tavallisissa jaetuissa semaforeissa $N \geq 2$, mutta N voi myös olla 1, jolloin puhutaan *binäärisistä jaetuista semaforeista*. Tällöin kahden semaforin tapauksessa toisen semaforin arvo olisi aina 1 ja toisen 0.

Jaettujen semaforien java-toteutus on melko yksinkertainen. Luodaan vain tarvittavan monta tavallista semaforia, joiden arvojen summa on aina haluttu N.

Javalla toteutettuna:

```
Semaphore notEmpty = new Semaphore(0, true);
Semaphore notFull = new Semaphore(N, true);
```

```
/*** Producer ***/
while(true) {
      /* non-critical section */
      try {
            notFull.aquire();
      } catch(InterruptedException ie) { /* error handling */ }

      /** critical section **/

      notEmpty.release();
}

/*** Consumer ***/
while(true) {
      /* non-critical section */
      try {
            notEmpty.aquire();
      } catch(InterruptedException ie) { /* error handling */ }

      /** critical section **/

      notFull.release();
}
```

Negatiiviset arvot semaforeissa

Yleensä semaforien alkuarvoksi asetetaan aina ei-negatiivinen kokonaisluku, mutta java mahdollistaa myös negatiivisten alkuarvojen käytön. Tämä tarkoittaa sitä, että semaforin “signal”-operaatiota pitää kutsua $|N|$ kertaa (N = semaforin negatiivinen alkuarvo), ennen kuin mikään prosessi pääsee semaforin “wait”-käskyn ohi.

Tuottaja-kuluttaja -tapauksessa tätä voisi soveltaa esimerkiksi siten, että tuotetta tulee tuottaa k kappaletta valmiiksi varastoon, ennenkuin kuluttaja voi kuluttaa yhtään tuotetta. Silloin koodi näyttäisi seuraavalta:

```
Semaphore notEmpty = new Semaphore(-k, true);
Semaphore notFull = new Semaphore(N, true);

/*** Producer ***/
while(true) {
      /* non-critical section */
      try {
            notFull.aquire();
      } catch(InterruptedException ie) { /*error handling*/ }

      /** critical section **/

      notEmpty.release();
}

/*** Consumer ***/
while(true) {
      /* non-critical section */
```

```
        try {
              notEmpty.aquire();
        } catch(InterruptedException ie) { /*error handling*/ }

        /** critical section **/

        notFull.release();
}
```

(Tulee huomata, että tässä esimerkissä notEmpty ja notFull eivät enää ole jaettuja semaforeja, sillä niiden kokonaisarvo ei ole vakio!)

Tämä on tietenkin hieman epätavallinen tilanne, mutta helpommin ymmärrettävä esimerkki olisi, että tietty määrä operaatioita (merk. *Y*) tulee suorittaa ennen erästä tiettyä operaatiota *W*. Alustetaan semaforin S alkuarvoksi |*Y*|. Tällöin voidaan suorittaa käsky signal(S) jokaisen *Y*-operaation jälkeen, ja kun kaikki nämä operaatiot on suoritettu ja semaforin arvo on 1, pääsee operaatio *W* semaforin läpi suoritukseen.

Negatiivisen arvon avulla on myös helppoa selvittää odottavien prosessien määrä, eikä tarvitse esimerkiksi käydä koko prioriteettijonoa läpi.

Liite 1: Esimerkki semaforien käytöstä Javassa: Aterioivien filosofien ongelma

```
import java.util.concurrent.Semaphore;

public class Philosopher extends Thread {

   private int id;              // Every philosopher has a unique id
   public static final int COUNT = 4;   // The count of philosophers
and forks

   // A (fair binary) semaphore for each fork
   private static Semaphore[] forks = { new Semaphore(1, true),
                          new Semaphore(1, true),
                          new Semaphore(1, true),
                          new Semaphore(1, true),
                          new Semaphore(1, true) };

   // Semaphore for the room, only 4 allowed in the room at the same
time
   private static Semaphore room = new Semaphore(COUNT-1, true);

   public Philosopher(int id) { // Constructor with philosophers id
       this.id = id;
   }

   private void think() {           // Method for thinking
       System.out.println("Philosopher #"+this.id+" thinking...");
   }

   private void eat() {             // Method for eating
       try {
           room.acquire();                 // Entering the room
           forks[this.id].acquire();       // Reserving the forks
           forks[(this.id + 1)%COUNT].acquire();
       }
       catch(InterruptedException ie) {   // Catched if interrupted
           System.out.println("Philosopher #"+this.id+" has been
terminated.");
           return;
       }

       System.out.println("Philosopher #"+this.id+" is eating...");

       forks[this.id].release();          // Releasing the forks
       forks[(this.id + 1)%COUNT].release();
       room.release();              // Leaving the room

       System.out.println("Philosopher #"+this.id+" has finished
eating.");
   }

   public void run() {        // The main program of a philosopher
       System.out.println("Philosopher #"+this.id+" has been
born!");
       while(true) {                // An eternal loop
           think();
           eat();
       }
   }
```

```
}

public class Diner {    // The main class creating the philosophers

   public Diner() {
       // Creating COUNT philosophers
       for(int i = 0; i < Philosopher.COUNT; i++) {
          new Philosopher(i).start();
       }
   }

   public static void main(String[] args) {
   // Main program of Diner-class
       new Diner();
   }
}
```

Liite 2: Kertaustehtävät

Luento 5:

Linkki tehtävään:

http://db.cs.helsinki.fi/%7Ekerola/php/practice.php?
file=/home/fs/exmattil/public_html/Rio/lukkiutuminen_Dijkstra_s08_fi.xml

<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

<task id="TKTS10011" type="singleselection" language="fi" author="Pikkojoulu">

<!-- taskin attribuutit

type = kyselyn tyyppi (singleselection, multiselection, range, text)

language = kieli (en,fi,sw)

-->

<course>Rinnakkaisohjelmointi</course>

<topic>Luento 5 - Lukkiutuminen</topic>

<question>Mikä on Dijkstran lukkiutumisen havaitsemisalgoritmin toimintaperiaate?</question>

<!-- question voi sisältää <IMG> kuva-elementtejä -->

<selection iscorrect="False">

<answer>Algoritmi laskee lukkiutuimsen todennäköisyyden prosessien maksimiresurssitarpeiden perusteella. </answer>

<explanation>Pankkiirinalgoritmi (Banker's Algorithm) käyttää apuna prosessien maksimiresurssitarpeita, mutta ei sekään laske lukkiutuimsen todennäköisyyttäniiden perusteella. </explanation>

</selection>

<selection iscorrect="False">

<answer>Prosesseja suoritetaan kunnes havaitaan, että prosessin suoritus kestää pidempään kuin oletettu suoritusaika. </answer>

<explanation>Prosesseille ei ole mahdollista laskea oletettua suoritusaikaa. Prosessi voi joutua esimerkiksi odottamaan käyttäjän syötettä. </explanation>

</selection>

<selection iscorrect="True">

<answer>Algoritmin avulla etsitään kaikki prosessit joiden resurssitarpeet voidaan tyydyttää ja

joiden suoritus voi loppua.</answer>

<explanation>Algoritmi etsii prosessit joiden sen hetken resurssitarpeet voidaan tyydyttää ja oletetaan, että prosessin suoritus loppuu. Prosessien suorituksen päättyessä vapautuvat kaikki sen käyttämät resurssit. Vaihetta toistetaan kunnes ei enää löydy prosesseja joiden suoritus loppuu. Jäljelle jäävät ainoastaan ne prosessit jotka voivat lukkiutua.</explanation>

</selection>

<selection iscorrect="False">

<answer>Jos lukkiutuminen tapahtuu niin palataan lukkiutumista edeltävään tilaan ja tallennetaan tieto niistä prosesseista jotka seuraavassa tilassa lukkiutuvat.</answer>

<explanation>Lukkiutumista edeltävään tilaan palaamista voidaan käyttää lukkiutumisen purkamiseen, mutta sen avulla ei voi havaita lukkiutuvia prosesseja. </explanation>

</selection>

</task>

Luento 9:

Linkki tehtävään:

http://db.cs.helsinki.fi/%7Ekerola/php/practice.php?file=/home/fs/exmattil/public_html/Rio/yleist%E4_luento9_s08_fi.xml

<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

<task id="TKTS10011" type="singleselection" language="fi" author="Pikkojoulu">

<!-- taskin attribuutit

type = kyselyn tyyppi (singleselection, multiselection, range, text)

language = kieli (en,fi,sw)

-->

<course>Rinnakkaisohjelmointi</course>

<topic>Luento 9 - Samanaikaisuuden hallinta hajautetussa ympäristössä</topic>

<question>Mikä seuraavista väitteistä pitää paikkansa?</question>

<!-- question voi sisältää <IMG> kuva-elementtejä -->

<selection iscorrect="False">

<answer>Rendezvous-palvelinmallissa vastaanottavan prosessin tulee tietää kutsuvan prosessin id. </answer>

<explanation>Vastaanottajan ei tarvitse tietää kutsuvan prosessin id:tä, vaan päinvastoin kutsuvan prosessin tulee tietää vastaanottajan id. </explanation>

</selection>

<selection iscorrect="False">

<answer>RPC-palvelinmallissa ei koskaan voi olla useita samanaikaisia kutsuja </answer>

<explanation>Kutsuja voi olla useampi aktiivisena samaan aikaan, mikäli mutex-ongelma on ratkaistu kutsuttavan palvelun sisällä eikä kutsuvassa prosessissa.</explanation>

</selection>

<selection iscorrect="False">

<answer>Synkronoidussa kommunikaatiossa vain vastaanottava prosessi odottaa kommunikaation valmistumista.</answer>

<explanation>Synkronoidussa kommunikaatiossa kummankin prosessin tulee odottaa kunnes kommunikaatio valmistuu.</explanation>

</selection>

<selection iscorrect="True">

<answer>Vastaanottajan odotussemantiikkana voi olla toiminnan jatkaminen, vaikka viestiä ei olisi vielä saapunut.</answer>

<explanation>Vastaanottajan ei välttämättä tarvitse saada viestiä toiminnan jatkamiseksi, vaikka viestin odottaminen onkin tavallisempaa.</explanation>

</selection>

</task>