Κατανεμημένα Συστήματα

5ο Φυλλάδιο εργαστηρίου (30/03/2017)

ΣΥΓΧΡΟΝΙΣΜΟΣ ΝΗΜΑΤΩΝ ΣΤΗ JAVA

1. Συγχρονισμός νημάτων

Στις περισσότερες περιπτώσεις, όταν πολλά νήματα εκτέλεσης διαχειρίζονται το ίδιο σύνολο αντικειμένων, είναι επιθυμητό να υπάρχει συγχρονισμός στις εργασίες που επιτελούνται. Αυτό είναι απαραίτητο, γιατί διαφορετικά τα αποτελέσματα ενδέχεται να είναι καταστροφικά. Αν, για παράδειγμα, ένα νήμα εκτέλεσης διαβάζει μια μεταβλητή ενός αντικειμένου την ίδια στιγμή που ένα άλλο νήμα εκχωρεί μια τιμή σε αυτή, το αποτέλεσμα και των δύο ενεργειών είναι εντελώς απρόβλεπτο. Το πρόβλημα του συγχρονισμού παράλληλων διεργασιών είναι εξαιρετικά δύσκολο, αφού εκτός από το να παρέχει τη δυνατότητα αποκλειστικής χρήσης στα αντικείμενα, ένας μηχανισμός συγχρονισμού πρέπει να εξασφαλίζει την αποφυγή αδιεξόδων.

Η Java παρέχει έναν αρκετά απλό και εύχρηστο μηχανισμό για το συγχρονισμό των νημάτων εκτέλεσης. Ο μηχανισμός αυτός βασίζεται στην έννοια του κλειδώματος (lock). Κάθε αντικείμενο της Java μπορεί να θεωρηθεί ότι διαθέτει ένα κλειδί. Το ίδιο συμβαίνει και για κάθε κλάση. Προκειμένου να επιτευχθεί αποκλειστική πρόσβαση σε ένα σύνολο μεθόδων ενός αντικειμένου, αυτές δηλώνονται ως **synchronized** (συγχρονισμένες). Στη συνέχεια για να κληθεί μια συγχρονισμένη μέθοδος ενός αντικειμένου, πρέπει να αποκτηθεί το κλειδί του αντικειμένου από το νήμα εκτέλεσης που πραγματοποιεί την κλήση. Το κλειδί επιστρέφεται μετά την εκτέλεση της μεθόδου.

Ας θεωρήσουμε για παράδειγμα την ακόλουθη κλάση SafeVariable. Η κλάση αυτή ορίζει δύο συγχρονισμένες μεθόδους **get** και **put**, που χρησιμοποιούνται για την ανάκτηση και την αποθήκευση μιας ακέραιας τιμής από ένα αντικείμενο.

```
public class SafeVariable{

	private int value;

	public synchronized int get() {return value; }
	public synchronized void put(int v) { this.value=v;}
}
```

Με τη χρήση των μεθόδων αυτών εξασφαλίζεται ότι για κάθε αντικείμενο αυτής της κλάσης, το πολύ μια μέθοδος θα εκτελείται για κάθε χρονική στιγμή. **Τονίζεται πως η συγκεκριμένη λειτουργία αφορά μεθόδους που έχουν δηλωθεί synchronized στο ίδιο αντικείμενο. Κάτι τέτοιο δηλαδή δεν αποκλείει την ταυτόχρονη εκτέλεση μεθόδων από διαφορετικά αντικείμενα**. Τα κλειδιά των κλάσεων χρησιμοποιούνται για στατικές συγχρονισμένες μεθόδους, κατά τρόπο παρόμοιο με τα κλειδιά των αντικειμένων.

Εκτός από τη δήλωση συγχρονισμένων μεθόδων, είναι δυνατός ο συγχρονισμός αυθαίρετων τμημάτων κώδικα, με χρήση και πάλι της λέξης **synchronized**. Για παράδειγμα, στο παρακάτω τμήμα κώδικα, προκειμένου να εκτελεστεί το εσωτερικό της εντολής πρέπει να αποκτηθεί το

κλειδί του αντικειμένου obj. Αυτό το τμήμα κώδικα μπορεί όμως να ανήκει σε οποιαδήποτε μέθοδο, όχι απαραίτητα του αντικειμένου αυτού.

```
synchronized(obj){
      //manipulate obj in some way
}
```

1ο Παράδειγμα Synchronized block

```
class Table {

 void printTable(int n) {
   synchronized(this){  //synchronized block
     for(int i=1;i<=10;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }
}

class MyThread1 extends Thread{
Table t;

MyThread1(Table t){
this.t=t;
}

public void run(){
      System.out.println("Thread : " + getName());
      t.printTable(10);
    }
}


class MyThread2 extends Thread{
Table t;

MyThread2(Table t){
this.t=t;
}

public void run(){
    System.out.println("Thread : " + getName());
    t.printTable(100);
    }
}


public class Main {
```

```
public static void main(String args[]){
      Table obj = new Table(); // Ένα κοινό αντικείμενο obj
      MyThread1 t1=new MyThread1(obj);
      MyThread2 t2=new MyThread2(obj);
      t1.start();
      t2.start(); } }
```

Εκτός από τη χρήση της λέξης synchronized, είναι απαραίτητη η χρήση των μεθόδων **wait()** και **notify()** της κλάσης Object για τον συγχρονισμό των λειτουργιών στην Java. Οι μέθοδοι αυτές ορίζονται για κάθε αντικείμενο, καθώς όλες οι κλάσεις κληρονομούν την Object. Οι κλήσεις σε αυτές τις μεθόδους πραγματοποιούνται πάντα μέσα από synchronized μεθόδους ή synchronized τμήματα κώδικα. Με την κλήση της μεθόδου **wait** για κάποιο αντικείμενο, το τρέχον νήμα εκτέλεσης επιστρέφει το κλειδί του αντικειμένου. Αν το νήμα εκτελεί μια synchronized μέθοδο αυτού του αντικειμένου, η επιστροφή του κλειδιού έχει ως αποτέλεσμα να σταματήσει προσωρινά η εκτέλεση του νήματος. Αυτό είναι επιθυμητό στην περίπτωση που η εκτέλεση της εργασίας δεν είναι δυνατό να ολοκληρωθεί λόγω έλλειψης πληροφοριών, για τις οποίες πρέπει το τρέχον νήμα να περιμένει.

Για να συνεχίσει η εκτέλεση θα πρέπει να συμβούν δύο πράγματα : αφενός να, ενημερωθεί το νήμα εκτέλεσης ότι μπορεί να συνεχίσει, μέσω της κλήσης της μεθόδου **notify** του αντικειμένου από ένα άλλο νήμα, και αφετέρου να αποκτηθεί και πάλι το κλειδί. Για κάθε κλήση της μεθόδου notify ενός αντικειμένου, ο διερμηνέας της Java ενημερώνει μόνο μια μέθοδο που έχει εκτελέσει τη μέθοδο wait γι' αυτό το αντικείμενο. Αν πρέπει να ενημερωθούν όλες οι μέθοδοι που βρίσκονται σε αναμονή, μπορεί να χρησιμοποιηθεί η μέθοδος **notifyAll**.

2ο Παράδειγμα Επίλυσης Race Conditions (Συνθήκες ανταγωνισμού) με synchronized

Χρήση κοινού τραπεζικού λογαριασμού με synchronized:

```
public class Account {
         private double balance;

        public Account(double initialDeposit) {
                balance = initialDeposit;
        }


        // Μέθοδος για την κατάθεση ενός ποσού στο λογ/σμό
        public synchronized void deposit(double amount) {
                balance += amount;
        }

        // Μέθοδος για την ανάληψη ενός ποσού από το λογ/σμό
        public synchronized void withdraw(double amount) {
                 // Δεν επιτρέπεται αρνητικό υπόλοιπο
                if ( balance >= amount ) { balance -= amount; }
        }
```

```
}


public class AccountOwner extends Thread
{
        private Account acc;

        public AccountOwner (Account a) {
                acc = a;
        }

        public void run() {
                acc.withdraw(100);
        }
}


public class ThreadSync
{
        public static void main(String[] args)
        {
                Account accnt = new Account(150);

                // Δημιουργία δύο νημάτων ή κατόχων λογαριασμών
                AccountOwner Bob = new AccountOwner(accnt);
                AccountOwner Alice  = new AccountOwner(accnt);

                Bob.start();
                Alice.start();
        }
}
```

3ο Παράδειγμα – Πρόβλημα αδιεξόδου (deadlock)

Ένα από τα πιο συνήθη προβλήματα που εμφανίζονται κατά τη διαχείριση νημάτων είναι αυτό του αδιεξόδου. Πρόβλημα αδιεξόδου έχουμε όταν ένα νήμα περιμένει ένα κλειδί αντικειμένου το οποίο είναι δεσμευμένο από κάποιο άλλο νήμα που με τη σειρά του περιμένει ένα κλειδί αντικειμένου από το αρχικό νήμα. Ο ακόλουθος κώδικας μπορεί να δημιουργήσει πρόβλημα αδιεξόδου.

```
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          double amount) {
    synchronized (fromAccount) {
      synchronized (toAccount) {
        if (fromAccount.getBalance()>=amount) {
          fromAccount.withdraw(amount);
          toAccount.deposit(amount);
        }
      }
    }
  }
```

Αυτό μπορεί να συμβεί όταν ένα νήμα Α καλεί την μέθοδο ως:

```
transferMoney(account1, account2, 100);
```

και παράλληλα το νήμα Β καλεί την μέθοδο :

```
transferMoney(account2, account1, 300);
```

Σε αυτή την περίπτωση τα δύο νήματα προσπαθούν να αποκτήσουν τα ίδια κλειδιά αντικειμένων αλλά με διαφορετική σειρά. Καθώς το καθένα από αυτά έχει δεσμεύσει το ένα κλειδί και περιμένει το δεύτερο, θα επέλθουμε σε κατάσταση αδιεξόδου.

Η πιο σημαντική τεχνική για την πρόληψη αδιεξόδου είναι να αποφευχθεί ο άσκοπος συγχρονισμός.

Το καλύτερο που μπορεί να γίνει στη γενική περίπτωση είναι η προσεκτική εξέταση του αν δημιουργείται αδιέξοδο κατά το σχεδιασμό του κώδικα μας. Εάν πολλά αντικείμενα χρειάζονται το ίδιο σύνολο κοινών πόρων για τη λειτουργία τους, καλό θα είναι να υπάρχει η διαβεβαίωση ότι θα το ζητήσουν με την ίδια σειρά. Για παράδειγμα, εάν η κλάση Α και η κλάση Β χρειάζονται αποκλειστική πρόσβαση στα αντικείμενα Χ και Υ, θα πρέπει να υπάρχει διαβεβαίωση ότι και οι δύο κλάσεις θα ζητήσουν πρώτα το Χ και μετά το Υ. Αν καμία δεν ζητήσει το Υ έστω και αν ήδη κατέχει το Χ, το αδιέξοδο δεν αποτελεί πρόβλημα.

Γενικά η απλή προσθήκη της δήλωσης synchronized σε όλες τις μεθόδους δεν αποτελεί λύση για όλα τα προβλήματα συγχρονισμού. Πρώτον, δημιουργεί προβλήματα απόδοσης επιβραδύνοντας τον κώδικα (μετατρέπεται σε σειριακός). Δεύτερον, αυξάνει δραματικά τις πιθανότητες αδιεξόδου. Τρίτον και σημαντικότερο, δεν είναι πάντα αυτό καθαυτό το αντικείμενο που πρέπει να προστατεύεται από την ταυτόχρονη τροποποίηση ή πρόσβαση και ο συγχρονισμός με το στιγμιότυπο της κλάσης της μεθόδου δεν μπορεί να προστατεύσει το αντικείμενο που πραγματικά πρέπει να προστατευτεί.

4ο Παράδειγμα – Πρόβλημα παραγωγού καταναλωτή

Να υλοποιήσετε το παράδειγμα του Producer/CubbyHole/Consumer με συγχρονισμό νημάτων. Το αντικείμενο Producer παράγει έναν ακέραιο αριθμό μεταξύ 0 και 9 και τον αποθηκεύει σε ένα αντικείμενο CubbyHole, και τυπώνει τον παραγόμενο αριθμό. Για να καταστήσει το πρόβλημα συγχρονισμού πιο ενδιαφέρον, ο Producer "κοιμάται" για ένα τυχαίο χρονικό διάστημα μεταξύ 0 και 100 χιλιοστών του δευτερολέπτου πριν την παραγωγή του επόμενου αριθμού της επανάληψης. Η διαδικασία επαναλαμβάνεται για 10 φορές. Το αντικείμενο Consumer, καταναλώνει τον ακέραιο αριθμό από το CubbyHole (το ίδιο ακριβώς αντικείμενο στο οποίο ο Producer έβαλε τον ακέραιο αριθμό) τόσο γρήγορα όσο διατίθενται. Ο συγχρονισμός μεταξύ αυτών των δύο threads υλοποιείται μέσα από δύο μεθόδους get() και put() του αντικειμένου CubbyHole.

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
```

```
    private int id;

    public Producer(CubbyHole c, int id)
        {
            cubbyhole = c;
            this.id = id;
        }

    public void run() {
    for (int i = 0; i < 10; i++) {
        cubbyhole.put(i);
        System.out.println("Producer #" + this.id + " put: " + i);
        try {
            sleep((int)(Math.random() * 1000));
        }catch (InterruptedException e) { }
      }
      }
}

public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int id;
    public Consumer(CubbyHole c, int id) {
        cubbyhole = c;
        this.id = id;
    }


    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.id + " got: " + value);
        }
    }
}


public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
                } catch (InterruptedException e) { }
        }
        available = false;
        System.out.println("in get");
        notifyAll();
        return contents;
     }

    public synchronized void put(int value) {
```

```
        while (available == true) {
            try {
                wait();
            }catch (InterruptedException e) { }
        }

        contents = value;
        available = true;
        System.out.println("in put");
        notifyAll();
    }
}


public class Main {
public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
```

Όταν γίνει κλήση της **wait()**, το thread που έκανε την κλήση απελευθερώνει μόνο το lock του αντικειμένου που περιμένει (και όχι τυχόν άλλα lock που διαθέτει για άλλα αντικείμενα) και μπαίνει σε κατάσταση αναμονής. Παραμένει σε αυτή την κατάσταση είτε αν λήξει η χρονική διάρκεια σε περίπτωση που ορίζεται ως παράμετρο στην wait είτε αν υπάρχει ειδοποίηση. Ειδοποίηση συμβαίνει όταν κάποιο άλλο thread καλεί την μέθοδο **notify()** ή **notifyAll()** στο αντικείμενο που περιμένει το thread.

Η μέθοδος **notifyAll()** προτιμάται στην περίπτωση που τα threads που περιμένουν ένα αντικείμενο είναι περισσότερα από ένα, επειδή δεν υπάρχει τρόπος επιλογής των threads που θα πρέπει να ειδοποιηθούν. Όταν ειδοποιηθούν όλα τα threads που περιμένουν, θα βγουν από τη κατάσταση αναμονής τους και θα προσπαθήσουν να πάρουν το lock του αντικειμένου. Ωστόσο, μόνο ένα μπορεί να το πετύχει αμέσως, το οποίο και συνεχίζει την εκτέλεση του. Τα υπόλοιπα μένουν μπλοκαρισμένα μέχρι το πρώτο να απελευθερώσει το lock. Αν τα threads που περιμένουν για το ίδιο αντικείμενο είναι πάρα πολλά, τότε ο χρόνος που θα απαιτηθεί μέχρι και το τελευταίο να πάρει με τη σειρά του το lock του αντικειμένου και να συνεχιστεί, είναι αρκετά σημαντικός.
Μέσα σε αυτό το χρόνο, είναι απολύτως πιθανό, το αντικείμενο, για το οποίο περίμενε το thread, να έχει βρεθεί για άλλη μια φορά σε μη αποδεκτή κατάσταση. **Στην περίπτωση αυτή, απαραίτητο θα ήταν, η κλήση της μεθόδου wait() να μπει μέσα σε βρόγχο για να ελέγχει την τρέχουσα κατάσταση του αντικειμένου.** Το γεγονός ότι υπήρξε ειδοποίηση για το αντικείμενο δεν σημαίνει κατ' ανάγκη και ότι το αντικείμενο είναι πλέον σε αποδεκτή κατάσταση. Αν δεν υπάρχει εγγύηση, ότι ένα αντικείμενο που μπήκε σε αποδεκτή κατάσταση, δεν υπάρχει περίπτωση να αλλάξει και πάλι κατάσταση, ο συνεχής έλεγχός της είναι ο πιο ενδεδειγμένος τρόπος.

1η εργαστηριακή άσκηση

Να επεκτείνετε το παραπάνω παράδειγμα επίλυσης του προβλήματος παραγωγού-καταναλωτή θεωρώντας ότι το αντικείμενο της κλάσης CubbyHole διαθέτει πέντε θέσεις αποθήκευσης. Σημειώστε ότι στη περίπτωση αυτή θα χρειαστείτε να ελέγχετε δυο καταστάσεις. Ο παραγωγός δεν μπορεί να αποθηκεύσει άλλον αριθμό όταν είναι όλες οι θέσεις γεμάτες και ο καταναλωτής δεν μπορεί να πάρει έναν αριθμό όταν όλες οι θέσεις είναι άδειες.

Απάντηση

```
public class CubbyHole {
    private int[] contents = {0,0,0,0,0};
    private boolean bufferEmpty = true;
    private boolean bufferFull = false;

    private final int size = 5;
    private int counter = 0;

    public synchronized int get() {
        while (bufferEmpty == true) {
            try {
                wait();
                } catch (InterruptedException e) { }
        }

        counter--;
        int value=contents[counter];
        System.out.println("The consumer removes the value : " + value + " by the
cubbyhole");
        bufferFull=false;
        if (counter==0)
            {
            bufferEmpty = true;
            System.out.println("The buffer is empty");
            }

        // αφύπνιση του παραγωγού που πιθανόν να έχει εμποδιστεί
       notifyAll();
       return value;
     }


    public synchronized void put(int value) {
        while (bufferFull == true) {
            try {
                wait();
            }catch (InterruptedException e) { }
        }
        bufferEmpty=false;

        System.out.println("The producer adds the value " + value + " in the
cubbyhole");
```

```
        contents[counter] = value;
        counter++;

         // ελέγχει αν έχει γεμίσει το cubbyhole
        if (counter==size)
             {
             bufferFull = true;
             System.out.println("The cubbyhole is full");
        }

         // αφύπνισε τον καταναλωτή που πιθανόν να περιμένει
        notifyAll();
    }
}


public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int id;

    public Producer(CubbyHole c, int id)
        {
            cubbyhole = c;
            this.id = id;
        }

    public void run() {
    for (int i = 0; i < 10; i++) {
        cubbyhole.put(i);
        System.out.println("Producer #" + this.id + " put: " + i);
        try {
            sleep((int)(Math.random() * 1000));
        }catch (InterruptedException e) { }
      }
}
}


public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int id;
    public Consumer(CubbyHole c, int id) {
        cubbyhole = c;
        this.id = id;
    }


    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.id + " got: " + value);
        }
    }
}
```

```
public class Main {
public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
```

Εναλλακτικός Τρόπος Συγχρονισμού

Όπως έχουμε πει, μία synchronized μέθοδος έμμεσα κλειδώνει το στιγμιότυπο της κλάσης (αντικείμενο) πριν να εκτελέσει την μέθοδο. Το JDK 1.5 (και μετά) επιτρέπει στον προγραμματιστή να δημιουργήσει άμεσα τα locks. Αυτό δίνει περισσότερη ευελιξία για τον έλεγχο και το συγχρονισμό threads. Αυτά τα αντικείμενα **Lock**, όπως και τα απλά κλειδώματα, μπορούν να κατέχονται μόνο από ένα νήμα κάθε στιγμή. Όμως, επιπλέον υποστηρίζουν το μηχανισμό *αναμονή* και *ειδοποίηση* **wait/notify**, μέσω αντικειμένων **Συνθήκης** (αντικείμενα **Condition**).

Ένα lock είναι ένα στιγμιότυπο κλάσης που υλοποιεί την διεπαφή java.util.concurrent.locks.Lock. Οι μέθοδοι που ορίζονται στη διεπαφή αφορούν τη δημιουργία και απελευθέρωση locks. Επίσης, παρέχεται η μέθοδος newCondition() για την δημιουργία προϋποθέσεων (αντικειμένων τύπου Condition) για την επικοινωνία και το συντονισμό των threads. Η πιο χρησιμοποιούμενη κλάση που υλοποιεί την Lock διεπαφή είναι η **ReentrantLock** που επιβάλει μία δίκαιη πολιτική locks ώστε να μην παρουσιάζονται προβλήματα λιμοκτονίας (starvation) (π.χ., threads που περιμένουν περισσότερο έχουν προτεραιότητα).

Το μεγαλύτερο πλεονέκτημα των αντικειμένων Lock σε σχέση με τα απλά κλειδώματα είναι ότι παρέχουν μεθόδους που έχουν τη δυνατότητα να απεμπλέκονται από μια προσπάθεια κατάληψης ενός κλειδώματος. Η μέθοδος ***tryLock*** εγκαταλείπει τη προσπάθεια κατάληψης ενός κλειδώματος εάν αυτό είναι κατειλημμένο ή αν περάσει ένα προκαθορισμένο χρονικό διάστημα (χρονοδιακοπή). Η μέθοδος ***lockInterruptibly*** εγκαταλείπει τη προσπάθεια κατάληψης ενός κλειδώματος εάν ένα άλλο νήμα στείλει σήμα διακοπής.

Ακολουθεί ενδεικτικό παράδειγμα χρήσης της διεπαφής lock για την επίλυση του προβλήματος παραγωγού-καταναλωτή.

```
import java.util.concurrent.locks.*;

public class CubbyHole {
    private int[] contents = {0,0,0,0,0};
    private boolean bufferEmpty = true;
    private boolean bufferFull = false;

    private Lock lock = new ReentrantLock();
    private Condition full = lock.newCondition();
```

```
    private Condition empty = lock.newCondition();

    private final int size = 5;
    private int counter = 0;

    public int get() {
       int value=0;
        lock.lock();
        try {
        while (bufferEmpty == true)
                empty.await();

        counter--;
        value=contents[counter];
        System.out.println("The consumer removes the value : " + value + " by the
cubbyhole");
        bufferFull=false;
        if (counter==0)
            {
            bufferEmpty = true;
            System.out.println("The buffer is empty");
            }

       full.signal();
        }
        catch (InterruptedException ex){}
        finally{
          lock.unlock();
          return value;
        }
     }


    public void put(int value) {
        lock.lock();
        try{

        while (bufferFull == true)
            full.await();

        bufferEmpty=false;

        System.out.println("The producer adds the value " + value + " in the
cubbyhole");
       contents[counter] = value;
       counter++;

        // ελέγχει αν έχει γεμίσει το cubbyhole
       if (counter==size)
            {
            bufferFull = true;
            System.out.println("The cubbyhole is full");
       }

       empty.signal();
```

```
        }
        catch (InterruptedException ex){}

        finally{
            lock.unlock();
}
    }
}
```