Lucrarea 10
Fire de execuție

Cuprins

Definiții 1
Construcții de limbaj pentru declararea firelor de execuție 2
Sincronizarea firelor de execuție 4
Temă 8

Definiții

Firele de execuție *(threads)* reprezintă porțiuni de cod ale aceluiaşi program care se pot executa în paralel una față de alta. Programele de până acum au fost de tip *single-threaded*, adică au fost compuse fiecare din câte un singur fir de execuție. Firele de execuție reprezintă modalitatea prin care limbajul Java suportă programarea concurentă. Execuția paralelă a firelor depinde practic de suportul de execuție al programelor, hardware şi software. În general, dacă se dispune de un sistem cu mai multe procesoare firele vor fi distribuite pe acestea.

Dacă avem un singur procesor execuția paralelă se simulează folosind diferite tehnici. În principiu se cunosc două asemenea tehnici care constituie, de fapt, politici de planificare a firelor. Primele trei versiuni ale interpretorului Java (pentru Solaris, Windows 95 şi Windows NT) aplică următoarea tehnică: un fir de execuție este lăsat să ruleze până când se termină sau până când ajunge în stare de aşteptare, moment în care este preluat un fir cu prioritate mai mare pregătit pentru execuție. Versiunile de interpretor realizate pentru 32 de biți aplică tehnica numită *time-slicing* (divizarea timpului). Practic, fiecărui fir i se alocă câte un interval de timp în care poate ocupa procesorul. În felul acesta, fiecare fir ajunge, prin rotație, să ocupe câte puțin procesorul, iar utilizatorul are impresia că firele rulează în paralel.

Divizarea în fire de execuție este avantajoasă mai ales în cazul operațiilor care necesită timp mare de rulare sau care forțează procesorul să aştepte pentru a accesa discul sau rețeaua. Constituirea unor asemenea operații ca fire separate conduce la:

— creşterea vitezei de execuție a programului în ansamblu, deoarece procesorul nu mai pierde vremea aşteptând după resurse, ci ia la rând alte fire până când resursele devin disponibile;

— creşterea vitezei de răspuns a programului, deoarece interfața utilizator poate avea propriul ei fir de execuție care să preia intrările utilizatorului chiar în timpul cât celelalte fire sunt "ocupate".

Desigur că toate aceste avantaje au un preț: întrucât mai multe fire de execuție aparțin aceluiaşi pogram, rezultă că ele pot accesa în comun anumite structuri de date ale programului. Aici se pune problema sincronizarii accesului, astfel încât să se asigure consistența datelor, lucru care complică puțin proiectarea programului.

Construcții de limbaj pentru declararea firelor de execuție

Pentru ca o secvență de program să fie considerată fir de execuție distinct, sunt necesare următoarele:

— construirea unei clase care extinde clasa predefinita `Thread`;

— redefinirea în acea clasă a metodei `run()` care are prototipul:

```
public void run();
```

Codul acestei metode constituie un fir de execuție. Altfel spus, în metoda `run()` se plasează secvența care trebuie rulată în paralel cu alte fire. Se precizează că metoda `run()` definită în clasa `Thread` nu face nimic.

— crearea unuia sau a mai multor obiecte ale clasei definite mai înainte şi apelarea pentru ele a metodei `start()` (moştenită de la `Thread`); metoda `start()` creează un nou flux de control (pe baza datelor obiectului) şi lansează în execuție firul apelând metoda `run()` a acestuia.

Exemplu considerat creează trei fire de execuție cu rol de numărătoare:

```
import java.io.*;
class Fir extends Thread
{
    private String nume;
    public Fir(String nume) {this.nume = nume;}
    public void run()
    {
        // aici pun codul firului de executie
        for(long i=0; i <= 10000; i++)
            System.out.println("Firul "+nume+" iteratia "+i);
    }
    public static void main(String[] args)
    {
        new  Fir("Dandanache  A.").start();  //  se  lanseaza  in
            executie un fir
        new  Fir("Catavencu  N.").start();  //  se  lanseaza  in
            executie al 2-lea fir
        new Fir("Coana Joitica").start(); // s.a.m.d
        System.out.println("Aici se termina metoda main.");
    }
}
```

Cele trei fire execută, de fapt, acelaşi lucru, ele fiind create prin intermediul unor obiecte ale aceleiaşi clase, `Fir`. Programul prezentat are în realitate patru fire de execuție: metoda `main()` constituie firul principal care va continua să se execute în paralel cu cele trei fire pe care le lansează. Pe lângă cele patru fire „vizibile“ ale programului mai există un fir care corespunde *Garbage Collector*-ului.

Dacă politica de planificare a firelor nu este cea de *time-slicing*, atunci este foarte probabil ca cele trei fire din exemplul de mai sus să nu se execute interclasat, ci unul după altul. Acest lucru se datorează faptului că un fir este lăsat să ruleze până fie se termină, fie ajunge în stare de aşteptare. Firele din exemplu, nu ajung în stare de aşteptare deoarece nu aşteaptă după nici o resursă. În acest caz, ca să putem vedea totuşi o execuție „paralelă“ va trebui să forțăm noi

intrarea în aşteptare. Cel mai simplu mod de a simula aşteptarea este acela de a lăsa firul să „doarmă“ o bucată de timp. „Adormirea“ se realizează cu metoda `sleep()` a clasei `Thread`, care primeşte ca parametru numărul de milisecunde reprezentând durata „somnului“. Cu acestea, programul de mai sus poate fi rescris astfel:

```
import java.io.*;
class Fir extends Thread
{
    private String nume;
    public Fir(String nume) {this.nume = nume;}
    public void run()
    {
        // aici pun codul firului de executie
        try
        {
            for(long i=0; i <= 10000; i++)
            {

                System.out.println("Firul "+nume+" iteratia "+i);
                sleep(100); // dorm 100 milisec.
            }
        }
        catch (InterruptedException e) { return; }
    }
    public static void main(String[] args)
    {
        new Fir("Dandanache").start(); // se lanseaza in executie
            un fir
        new Fir("Catavencu").start(); // se lanseaza in executie
            al 2-lea fir
        new Fir("Coana Joitica").start(); // s.a.m.d
        System.out.println("Aici se termina metoda main.");
    }
}
```

Introducerea blocului `try-catch` în metoda `run()` a fost necesară deoarece, pe de o parte, metoda `sleep()` are o clauză `throws InterruptedException`, iar pe de altă parte metoda `run()` nu are clauza `throws`, deci suntem nevoiți să captăm eventuala excepție emisă de metoda `sleep()`. S-a expus mai sus o variantă prin care putem defini fire de execuție. Această variantă ar putea fi dezavantajoasă atunci când clasa care conține metoda `run()` ar trebui să extindă şi o altă clasă decât `Thread`. Deoarece Java nu permite moştenire multiplă, în asemenea cazuri putem folosi şi o altă variantă, în care ne bazăm pe interfața predefinită `Runnable` (pe care clasa `Thread` o implementează de altfel) şi pe faptul că unul dintre constructorii clasei `Thread` acceptă ca parametru o referință la `Runnable`. Varianta cu `Runnable` a programului anterior este următoarea:

```
import java.io.*;
class Fir implements Runnable
{
    private String nume;
    public Fir(String nume) {this.nume = nume;}
```

```
    public void run()
    {
        // aici pun codul firului de executie
        for(long i=; i <= 10000; i++)
            System.out.println("Firul "+nume+" iteratia "+i);
    }
    public static void main(String[ ] args)
    {
        Fir f1 = new Fir("Dandanache A."); // se creeaza un fir
        Fir f2 = new Fir("Catavencu N."); // se creaza al 2-lea
            fir
        Fir f3 = new Fir("Coana Joitica"); // s.a.m.d
        new Thread(f1).start(); // se lanseaza in executie un fir
        new Thread(f2).start(); // se lanseaza in executie al 2-
            lea fir
        new Thread(f3).start(); // s.a.m.d
        System.out.println("Aici se termina metoda main().");
    }
}
```

Sincronizarea firelor de execuție

În exemplele din paragraful precedent firele de execuție erau secvențe absolut independente una de cealaltă, în sensul că nu accesau date comune. Un caz tipic în care două fire de execuție folosesc în comun structuri de date şi când este necesară sincronizarea lor, este exploatarea **în regim producător-consumator** a unei zone de date.

Presupunem că avem un tablou de numere întregi, de dimensiune limitată. Unul dintre fire, producătorul depune valori în tablou, câte una o dată. Celălalt fir, consumatorul, citeşte câte o valoare din tablou. Ambele operații realizează parcurgerea circulară a tabloului (adică, după ce s-a accesat ultimul element al tabloului, la următorul acces se va prelucra primul element al tabloului). Sincronizarea este necesară, în primul rând, pentru asigurarea consistenței datelor: producătorul să nu încerce să introducă valori când tabloul este plin, iar consumatorul să nu încerce să citească dintr-un tablou gol.

De asemenea, foarte important este ca atât operația de introducere a unei valori cât şi cea de citire a unei valori din tablou să se desfăşoare fără întrerupere sau în regim de **excludere mutuală**. Ce înseamnă aceasta? Să presupunem că cele două operații arată că în secvența de mai jos:

```
// introducerea
void Pune(int v)
{
    while(nr_elem == tablou.length)
    {
        //daca tabloul este plin, astept
    }
    iP = (iP + 1) % tablou.length; // iP va indica urmatoarea
        pozitie libera
    tablou[ iP ] = v;
    nr_elem++; // incrementez numarul de elemente
```

```
}

// extragerea
int Scoate()
{
    while (nr_elem == 0)
    {
        // daca tabloul este gol, astept
    }
    iS = (iS + 1) % tablou.length; // iS indica urmatorul element
        de citit
    nr_elem--;
    return tablou[ iS ];
}
```

Să presupunem acum că este în curs de execuție metoda de introducere. Ea apucă să modifice indicele `iP` şi, înainte ca valoarea `v` să se depună în tablou, expiră cuanta de timp a firului producător. În acest moment intră în acțiune firul consumator care, în cel mai bun caz va citi o valoare pe care a mai citit-o o dată, dacă nu cumva se blochează pe motiv că tabloul este gol. Probleme similare pot fi întâlnite şi în cazul extragerii. Soluția unor astfel de probleme este aceea de a impune ca pe durata execuției uneia dintre operații nici un alt fir să nu poată accesa tabloul. Limbajul Java oferă în acest sens posibilitatea de a declara anumite metode ca fiind de tip `synchronized`.

Fie următoarea secvență:

```
class Resursa
{
    public synchronized void oMetoda ( ) { ... }
    // alte metode
}
```

Faptul că `oMetoda()` este declarată ca fiind sincronizată înseamnă că pe durata în care un fir execută această metodă, obiectul receptor implicat intră în starea „blocat“, astfel încât nici un alt fir nu mai poate executa vreo altă metodă sincronizată pentru obiectul respectiv. Dacă un fir apelează o metodă sincronizată pentru un obiect aflat în starea blocat, firul respectiv intră într-o coadă de aşteptare ataşată obiectului. Ieşirea din blocaj a obiectului se realizează la terminarea execuției metodei sincronizate.

Sincronizarea metodelor reprezintă mecanismul prin care se asigură accesul firelor la resurse comune în regim de excludere mutuală. Dacă analizăm metodele `Pune()` şi `Scoate()` de mai sus constatăm ca operațiile propriu-zise de scriere/citire se pot efectua doar în anumite condiții. Spre exemplu, nu putem citi dintr-un tablou gol. Ca urmare, dacă tabloul este gol, firul consumator trebuie să aştepte ca un eventual producător să depună ceva în tablou. Aici intervine problema **comunicării între fire**. Comunicarea presupune un mecanism prin care un fir care a ținut ocupată o resursă să anunțe celelalte fire în momentul eliberării resursei.

Intrarea în aşteptare şi anunțarea eliberării unei resurse se realizează cu ajutorul metodelor `wait()`, respectiv `notifyAll()` pe care toate clasele le moştenesc de la `Object`. Metoda `wait()` permite firului care execută la un moment dat o metodă sincronizată asupra unui obiect să intre în stare de aşteptare, obiectul ieşind din starea de blocaj. Ca urmare, un alt fir poate acționa asupra obiectului respectiv. Ieşirea din starea de aşteptare se va face când unul dintre

firele active apelează metoda `notifyAll()`. Aceasta înştiinţează firele aflate în aşteptare că obiectul dorit de ele iese din blocaj. Modul în care se folosesc metodele `wait()` şi `notifyAll()` este ilustrat în secvenţa de mai jos, în care sunt rescrise metodele `Pune()` şi `Scoate()`, împreuna cu clasa din care ele fac parte:

```
class Resursa
{
    private int[] tablou;
    private int nr_elem=0;
    private int iP=0;
    private int iS=0;
    public Resursa (int n) { tablou=new int[n]; }
    // introducerea
    public synchronized void Pune(int v)
    {
        while(nr_elem == tablou.length)
        {
            // tablou plin
            try
            {
                wait(); // asteapta
            }
            catch (InterruptedException e) {
            System.out.println(e);}
        }
        iP = (iP + 1) % tablou.length; // iP va indica urmatoarea
            pozitie libera
        tablou[ iP ] = v;
        nr_elem++; // incrementez numarul de elemente
        notifyAll(); // anunt firele in asteptare ca am eliberat
            resursa
    }
    // extragerea
    public synchronized int Scoate() {
        while (nr_elem == 0)
        {
            // tablou gol
            try
            {
                wait(); // asteapta
            }
            catch (InterruptedException e)
                {System.out.println(e);}
        }
        iS = (iS + 1) % tablou.length; // iS indica urmatorul
            element de citit
        nr_elem--;
        notifyAll(); // anunt firele in asteptare ca am eliberat
            resursa
        return tablou[ iS ];
    }
```

```
}
```

Să vedem acum cum scriem firele producător şi consumator propriu-zise:

```
class Producator extends Thread
{
    private Resursa r;
    private int nr_valori;
    public Producator (Resursa r, int n)
        { this.r = r; nr_valori = n; }
    public void run()
    {
        for(int i=0; i < nr_valori; i++)
        {
            r.Pune(i);
            System.out.println("Am pus valoarea "+i);
        }
    }
}

class Consumator extends Thread
{
    private Resursa r;
    private int nr_valori;
    public Producator (Resursa r, int n)
        { this.r = r; nr_valori = n; }
    public void run()
    {
        for(int i=0; i < nr_valori; i++)
        {
            int j = r.Scoate();
            System.out.println("Am scos valoarea "+j);
        }
    }
}

class Client
{
    public static void main(String[] args)
    {
        Resursa r = new Resursa(10);
        Producator p = new Producator(r,40);
        Consumator c = new Consumator(r,40);
        p.start();
        c.start();
    }
}
```

În secvența de mai sus am folosit variabila membru `nr_valori` în ambele fire pentru a reprezenta numărul total de valori care vor fi manipulate în procesul de scriere/citire. Este obligatoriu ca la crearea unei perechi de obiecte producator-consumator care manipulează aceeaşi resursă `r` să se dea aceeaşi valoare parametrului `n` din constructor.

Dacă la consumator `nr_valori` este mai mare decât la producător, vom ajunge în situația în care firul consumator nu mai poate ieşi din starea de aşteptare, după ce a citit ultima valoare creată de producător. O altă variantă ar fi utilizarea unui fanion pe care producătorul să-l seteze când a terminat de creat valorile, iar consumatorul să-şi termine execuția în funcție de acest fanion.

Temă

Se consideră o parcare pentru maşini, cu *n* intrări şi o singură ieşire. Activitățile din parcare sunt dirijate de către un paznic. Acesta autorizează intrările (respectiv ieşirile) maşinilor în (respectiv din) parcare şi memorează numărul *s* de locuri libere. Înainte de a încerca să intre sau să iasă din parcare, fiecare şofer îi comunică paznicului decizia sa. Acesta, fiind un om simplu, nu poate discuta simultan cu mai multe persoane, deci nici un şofer nu se poate adresa paznicului în intervalul de timp în care acesta dialoghează cu un alt şofer. Există două modalități prin care paznicul discută cu şoferii:

— un şofer îl anunță pe paznic că doreşte să intre în parcare. Paznicul verifică dacă există locuri libere şi, în caz afirmativ, îi permite şoferului accesul. Maşina ocupă unul dintre locurile libere, valoarea lui *s* se micşorează cu o unitate, iar dialogul se încheie. Dacă nu mai sunt locuri în parcare, paznicul îi spune şoferului că nu poate intra deoarece parcarea este plină, dar că a notat numărul maşinii, invitându-l să fie gata să intre în parcare atunci când se va elibera un loc. Dialogul se încheie.

— un şofer doreşte să iasă din parcare şi începe dialogul cu paznicul pentru a-l informa de această intenție. Dacă există şoferi care aşteaptă să intre în parcare, atunci paznicul îi permite şoferului care a inițiat dialogul să plece şi, concomitent (pentru operativitate) îl cheamă pe unul dintre şoferii aflați pe lista de aşteptare să ocupe locul eliberat. Dialogul se încheie. În caz contrar, paznicul îi permite şoferului, care a inițiat dialogul, să plece, măreşte cu o unitate valoarea lui *s* şi încheie dialogul.

Să se scrie un program pentru vizualizarea modului în care se realizează dialogul dintre paznic şi şoferi.