THREAD-URI (fire de execuţie)

In viaţa reală asistăm in permanenţă la mai multe acţiuni ce se desfăşoară in acelaşi timp. Acest fapt implică şi calculatoarele moderne să realizeze in acelaşi timp mai multe acţiuni (atât prin resurse software cât şi prin resurse hardware)

Posibilitatea de a lansa şi gestiona mai multe *fire de execuţie (procese)* este principala noţiune nou introdusă de sistemul de operare Windows.

Ac'iunile ce se desfăşoară concomitent pot fi foarte variate: programe „traditionale", secvente de animatie, sunet etc fapt ce subliniază importanţa firelor de execuţie.

Crearea firelor de execuţie

- **Clasa Thread**

O modalitate de a crea şi lansa in execuţie fire (de execuţie) este de a folosi clasa Thread din pachetul java.lang:

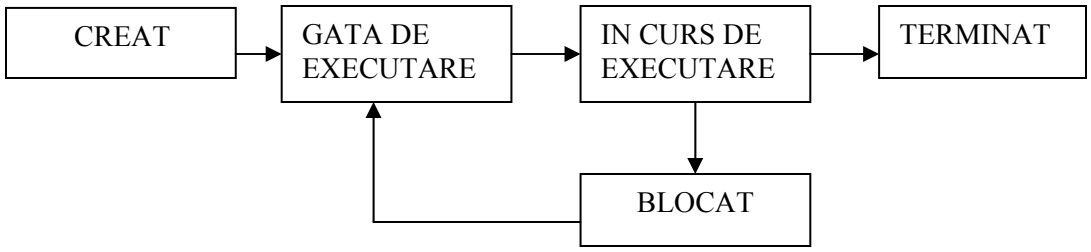
Public class Thread extends Objects implements Runnable

(despre interfaţa Runnable vom vorbi mai jos).

Pentru a crea un fir de execuţie , trebuie să cream un obiect de tipul Thread:

Thread fir=new Thread();

După ce obiectul este creat, el poate fi configurat şi apoi se poate trece la execuţia sa. Configurarea cuprinde elemente ca de exemplu precizarea unui nume, a unei priorităţi etc. Când firul este gata de execuţie, se invoca metoda sa start fără argumente. Apoi firul de execuţie incepe să execute metoda sa run.
Când metoda run se incheie, firul de execuţie se termină.
Următoarea schema va prezenta stările in care se poate afla un proces (fir de execuţie) .



Un fir de execuţie este gata de execuţie dacă indeplineşte toate condiţiile pentru a se trece la execuţiea sa, dar nu i s-a alocat incă un procesor; când un procesor liber preia firul, acesta trece in starea in curs de execuţie. Blocarea unui fir de execuţie poate fi realizată pe baza unei condiţii (de exemplu un semafor); ca urmare firul trece in starea

blocat. La deblocarea sa, realizată prin mecanisme dintre care unele vor fi prezentate in continuare, el revine in starea gata de execuție, aşteptând ca un procesor să devina liber şi să reia execuția sa.

Exemplu:

Să considerăm următorul program :

```
class Tip extends Thread{
int k;boolean continua=true;
      public void run(){
      while(continua) IO.write(" "+ k++);
            IO.writeln("Fir");
      }
      public void terminare(){continua=false;}
}
class P1 {
      public static void main(String[] w){
      Tip Ob=new Tip();Ob.start();
      while (IO.readch() != '\n' );
      Ob.terminare();IO.writeln("main");
 }
}
```

Pe baza obiectului Ob este lansat un fir de execuție. Activitatea acestuia constă in a tipări, atata timp cât variabila booleană *continua* are valoarea true, şirul numerelor naturale. Firul de execuție principal prevede citirea repetată de caractere până când este introdus <Enter>; în acel moment obiectul Ob execută metoda *terminare*, al cărei efect este modificarea valorii lui *continua,* ceea ce va determina oprirea firului de execuție lansat pe baza obiectului Ob. Firul de execuție principal se termină şi el. Să remarcăm că nu este previzibil dacă mesajul „*Fir*" va fi tiparit inainte sau după mesajul *„main"*.

Utilizarea interfetei *Runnable* constituie o alternativă la extinderea clasei Thread. Avantajul constă in primul rând în însusi faptul că este o interfață: o clasă oarecare poate implementa *Runnable* şi extinde o altă clasă (pe când o clasă ce extinde Thread nu mai poate extinde vreo altă clasă).

Vom utiliza exemplul următor pentru a prezenta interfața Runnable:

```
class Execut implements Runnable {
      int k;boolean continua=true;
      public void run() {
            while (continua) IO.write(" " + k++);
            IO.writeln("Fir");
      }
      public void terminare() {continua=false; }
}
class RunFir {
      public static void main(String[] arg) {
```

```
            Execut Ob=new Execut();
            Thread fir=new Thread(Ob);
            fir.start();
            while (IO.readch() != '\n');
            Ob.terminare();IO.writeln("main");
        }
}
```

Un mod de sincronizare

Acest mod are o largă aplicabilitate in transcrierea în Java a algoritmilor paraleli. Dacă un fir de execuție lansează unul sau mai multe fire de execuție, care prin natura problemei, trebuie să aştepte terminarea activităților acestora inainte de a intreprinde o noua acțiune (in lipsa unei astfel de măsuri, firul principal îşi continuă activitatea in paralel cu firele pe care le-a lansat) o soluție ar fi metoda *join* a clasei Thread. Ea are următoarele două forme:

public final void join() throws InterruptedException
public final void join(long mili) throws InterruptedException

Prima formă face ca firul de execuție care a lansat la rândul său un fir de execuție să aştepte un timp nelimitat, inainte de a trece la o acțiune următoare prevăzută in program. A doua formă limitează aşteptarea la un numar specificat de milisecunde.

Să considerăm următorul program:

```
class Tip extends Thread {
        static int k=1;
        public void run() {
                for (int i=1 ;i<500;i++) IO.write(k+" ");
        }
}
class P2 {
        public static void main(String[] sir)
                throws InterruptedException {
                Tip Ob=new Tip();Ob.start();
                Thread.sleep(50);
                Ob.k=2;Ob.join();
                IO.writeln("***");
                }
}
```

După ce lansează un nou fir de execuție pe baza obiectului Ob, firul principal îşi intrerupe activitatea 50 milisecunde, apoi modifică in 2 valoarea câmpului k al clasei Tip şi aşteaptă ca acesta să îşi incheie activitatea, inainte de a tipari trei asteriscuri şi de a-şi termina la rândul sau execuția. Drept urmare la ieşire va apare un sir de 1, urmat de un sir de 2.

Firul principal nu este cu nimic deosebit de cele pe care le lansează; in particular, acesta înseamnă că firul principal, după ce lansează un fir, îşi poate încheia activitatea (se poate termina), întreaga aplicație încheindu-se atunci când firul nou lansat se termină. Afirmația de mai sus este valabilă pentru orice fir (nu numai pentru cel principal) ce lansează alte fire.

Construirea unui fir de execuție cronometru

Un *cronometru* este un obiect care apelează la un alt obiect dupa un interval regulat de timp. Un obiect cronometru este foarte util pentru a construi un ceas animat sau a crea intervalele dintre cadrele unui film. Un cronometru utilizează o interfață care permite specificarea obiectului care trebuie apelat după fiecare interval de timp. Interfetele Java permit crearea echivalentului unei functii *callback*. Următorul program ,*cronometru.java*, prezintă un cronometru sportiv digital care afişează durata rulării programului:

```
import java.applet.*;
import java.util.*;
import java.awt.*;
interface obCron
{
	public void actualiz();
}
class unCeas extends Canvas implements obCron {
int zero;
int secunde;
public unCeas()
{
	Date acum=new Date();
	zero=(acum.getHours()*3600+acum.getMinutes()*60+acum.getSeconds());
}
public void actualiz()
{
	Date acum=new Date();
	secunde=acum.getHours()*3600+acum.getMinutes()*60+acum.getSeconds()-zero;
	repaint();
}
public void paint(Graphics g)
{
	String buffer=new String(Integer.toString(secunde));
	g.drawString(buffer,20,20);
}
}
class unCronom extends Thread {
int timp;
obCron cronom_ob;
```

```
public unCronom(obCron to ,int t)
{
cronom_ob=to;
timp=t;
}
public void run()
{
        while (true)
        {
        try {
                sleep(timp);
           }
        catch(InterruptedException e){ }
        cronom_ob.actualiz();
        }
}
}
public class cronometru extends Applet {
        unCeas unceas=new unCeas();
        unCronom uncronom=new unCronom(unceas,1000);
        public void init()
        {
                setLayout(new GridLayout(2,3));
                add(unceas);
                uncronom.start();
        }
        public void repaint()
        {
        }
}
```

Primitive de sincronizare
Problema Producator -Consumator

Principalele primitive de sincronizare puse la dispozitie de Java sunt următoarele metode publice ale clasei rădăcină Object:

final void wait()
final void wait(long t)
final void notify()
final void notifyAll()

Inainte de a trece la prezentarea lor, specificăm că ele:

-pot lansa excepția IllegalMonitorStateException dacă firul curent nu deține controlul asupra monitorului reprezentat de obiectul curent ;

-trebuie să fie invocate din interiorul unei metode sincronizate sau a unui bloc sincronizat.

Metodele notify şi notifyAll

Metoda *notify* - "trezeşte" unul din eventualele fire din mulțimea de aşteptare asociată monitorului care a invocat metoda. Acesta devine candidat la a prelua controlul asupra monitorului conform mecanismului descris mai sus.

Metoda *notifyAll* diferă de *notify* prin aceea că „trezeşte" toate eventualele fire din mulțimea de aşteptare asociată monitorului care a invocat metoda.

Reamintim enunțul problemei: *un producător şi un consumator îşi desfăşoara in acelaşi timp activitatea, folosind in comun o bandă de lungime* lung. *Producătorul produce câte un obiect şi il plasează la un capat al benzii. Consumatorul ia câte un obiect de la celălalt capăt al benzii şi îl consumă.*

Dificultatea problemei constă în faptul că producătorul şi consumatorul pun/iau obiecte pe/de pe banda in ritmuri imprevizibile, ceea ce conduce la următoarele situații limită:

-producatorul incearcă să pună un obiect in banda plină;

-consumatorul incearcă să ia un obiect de pe banda vidă;

```
import java.util.*;
class Time extends Random
{
        int delay() { return (int) (100.0f*nextFloat());}
}
class Banda
{
        int lung=3 ,n,p,u=lung-1;
        char[] coada =new char[lung];
        static Time ObT=new Time();

        synchronized char ia()
        {
                char ch;
                while (n==0)
                {
                        try {wait();}
                        catch (InterruptedException e){}
                }
                ch=coada[p];p=(p+1)%lung;n--;
                IO.write(" I"+ch);notify();return ch;
        }

        synchronized void pune(char ch)
        {
                while (n==lung)
                {
                        try {wait();}
                        catch(InterruptedException e){}
```

```
                }
                        u=(u+1)%lung;coada[u]=ch;n++;
                        IO.write("  P"+ch);notify();

        }
}
class Prod implements Runnable
{
        Banda b;
        Prod(Banda bb){b=bb;}

        public void run()
         {
                char ch;
                for (ch='a';ch<='z';ch++)
                {
                b.pune(ch);
                try {Thread.sleep(Banda.ObT.delay());}
                catch(InterruptedException e){}
                }
         }
}
class Cons implements Runnable
{
        Banda b;
        Cons(Banda bb){b=bb;}

        public void run()
         {
                char ch;
                do {
                  ch=b.ia();
                  try {Thread.sleep(Banda.ObT.delay());}
                  catch(InterruptedException e){}
                  }
                while (ch!='z');
         }
}
class PC
{
        public static void main(String[] w)
        {
                Banda b=new Banda();
                Prod P=new Prod(b);Cons C=new Cons(b);
                Thread FirP=new Thread(P);FirP.start();
                Thread FirC=new Thread(C);FirC.start();
```

```
            try { FirP.join();FirC.join();}
            catch(InterruptedException e){}
        }
}
```

Clasa *Time* furnizează un alt mod de accentuare a nedeterminismului: metoda *delay* intoarce un număr intreg cuprins intre 0 şi 100. Această modalitate va fi folosită pentru a întârzia un timp aleatoriu operaţiile de punere şi luare de/pe pe bandă. In metoda *delay* a fost folosită funcţia *nextFloat* a clasei *Random* din pachetul *java.util* , funcţie ce intoarce o valoare de tip *float* din intervalul [0,1).

Clasa *Banda* ţine evidenţa cozii corespunzătoare benzii: vectorul *coada* are lungimea *lung*, primul său element este pe poziţia p, ultimul pe poziţia u, iar numărul de elemente de pe bandă este n. Metoda *ia* realizează extragerea unui element de pe bandă, iar metoda *pune* corespunde introducerii unui nou element pe bandă; elementele sunt caractere. Metodele *ia şi pune* sunt declarate cu modificatorul *synchronized*.

Obiectul P de tipul *Prod*, clasa ce implementează interfaţa *Runnable,* lansează un fir de execuţie corespunzător producatorului; sunt produse literele mici ale alfabetului şi se incearcă plasarea lor pe bandă prin invocarea metodei *pune* .

Obiectul C de tipul *Cons* , clasa ce implementează şi ea interfaţa *Runnable*, lansează un fir de execuţie corespunzator consumatorului; prin invocarea repetată a metodei *ia* se incearcă preluarea unui caracter de pe bandă, până când s-a obţinut caracterul ‚z'.

Cele două fire de execuţie folosesc acelaşi obiect b de tipul *Banda* pentru invocarea metodelor *ia şi pune*.

Sincronizarea este asigurată prin folosirea metodelor *wait şi notify.*

Utilizarea grupului de fire de execuţie implicit

De fapt toate firele de execuţie sunt membre ale unui grup de fire implicit. In cadrul programelor putem utiliza grupul de fire de execuţie implicit pentru a enumera toate firele programului. Obţinem grupul de fire implicit apelând metodele statice *currentThread* şi *getThreadGroup* ale clasei Thread. Pentru a obţine firul implicit trebuie să parcurgem ierarhia grupului de fire. Următorul program ,"firele.java", arată cum sunt enumerate toate firele dintr-un applet:

```
import java.applet.*;
class unFir extends Thread
{
        public unFir (String nume)
        {
        super(nume);
        }
        public void run(){}
}

public class firele extends Applet
{
```

```
        ThreadGroup top_grup;
        ThreadGroup parinte;
        Thread un_fir=new unFir("firul meu");
        public void print_fir(ThreadGroup g,String indent)
        {
                int cnt_fire=g.activeCount();
                int cnt_grup=g.activeGroupCount();
                Thread fire[]=new Thread[cnt_fire];
                ThreadGroup grupuri[]=new ThreadGroup[cnt_grup];
                System.out.println("Grup fire:"+g.getName());
                g.enumerate(fire,false);
                for (int i=0;i<cnt_fire;i++)
                if (fire[i]!=null)
                System.out.println(indent+"Fir"+fire[i].getName());
                g.enumerate(grupuri,false);
                for (int i=0;i<cnt_grup;i++)
                 if (grupuri[i]!=null)
                        print_fir(grupuri[i],indent+" ");

        }
        public void init()
        {
                top_grup=Thread.currentThread().getThreadGroup();
                while ((parinte=top_grup.getParent())!=null)
                        top_grup=parinte;
                print_fir(top_grup,"");
        }

}
class Plimbare implements Runnable
{
        public void run()
        {
                //aici plasati instructiunile firului
        }
}
```

Clasa **Thread** mai permite lucrul cu semafoare, rezolvarea problemei filozofilor la masă, intoarcerea firului curent (final String getName()), modificarea firului curent (static void setName()), întoarcerea firului curent (Thread currentThread()),…etc.