Laboratorul 8

8.1. Fire de execuţie (Thread-uri)

În programarea concurentă există două unităţi de execuţie de bază: *procesele* şi *firele de execuţie*.

Programarea concurentă presupune execuţia în paralel a mai multor task-uri, acest lucru fiind posibil doar pe un sistem de operare multitasking (Unix, Windows). Procesele pot fi văzute ca fiind aplicaţii sau programe care la rândul lor pot comunica cu alte procese.

Un fir de execuţie este o succesiune secvenţială de instrucţiuni care se execută în cadrul unui proces.

Într-un program putem avea mai multe fire de execuţie în paralel, iar sistemul de operare va aloca pe rând cuante din timpul procesorului fiecărui fir de execuţie până la terminarea acestora.

8.1.1. Crearea şi pornirea unui fir de execuţie.

În Java, fiecare fir de execuţie are asociată o instanţă a clasei *Thread*.

Există două moduri de a crea un fir de execuţie:

- Implementarea interfeţei ***Runnable;***
- Extinderea clasei ***Thread***.

Atât interfaţa *Runnable*, cât şi clasa *Thread*, au definită metoda ***run()***, care conţine codul executat de firul de execuţie. Pentru a lansa un fir de execuţie se apelează metoda ***start()*** a clasei *Thread*, metodă care nu face alt ceva decât să pornească un nou fir de execuţie şi să apeleze metoda *run()*.

1) Crearea unui fir de execuţie prin implementarea interfeţei Runnable.

```
public class FirDeExecutieRunnable implements Runnable {

 public FirDeExecutieRunnable() {}

 @Override
 public void run() {
 // codul executat de firul de executie
 }

 public static void main(String[] args) {
  // crearea unei instante a clasei FirDeExecutieRunnable
  FirDeExecutieRunnable f = new FirDeExecutieRunnable();
  // crearea unei instance a clasei Thread ce primeste ca parametru un
obiect Runnable
  Thread t = new Thread(f);
  // lansarea firului de executie
  t.start();
 }
}
```

2) Crearea unui fir de execuţie prin extinderea clasei Thread.

```
public class FirDeExecutieThread extends Thread {

 public FirDeExecutieThread() {}
```

```
 public FirDeExecutieThread(String name) {
  super(name);
  // name - numele firului de executie
 }

 @Override
 public void run() {
  // codul executat de firul de executie
 }

 public static void main(String[] args) {
  // crearea unei instante a clasei FirDeExecutieThread
  FirDeExecutieThread t = new FirDeExecutieThread();
  // lansarea firului de executie
  t.start();
 }
}
```

8.1.2. Oprirea temporară a unui fir de execuţie

Pentru opri temporar execuţia unui thread se poate folosi metoda ***sleep(…)***. Durata pentru care este întrerupt firul de execuţie depinde şi de precizia timer-elor sistemului de operare.

```
public static void sleep(long millis) throws InterruptedException
public static void sleep(long millis, int nanos) throws InterruptedException
```

De exemplu:

```
  try {
   // se face o pauza de o secunda
   Thread.sleep(1000);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
```

8.1.3. Fire de execuţie de tip „daemon”

Un „daemon” reprezintă un fir de execuţie care se termină automat la terminarea aplicaţiei. De obicei firele „daemon” pun la dispoziţia celorlalte fire de execuţie anumite servicii.

Atunci când singurele fire de execuţie rămase sunt cele „daemon” aplicaţia îşi termină execuţia.

Pentru a seta un fir ca fiind „daemon” se foloseşte metoda ***setDaemon(true)***, înainte ca firul să fie lansat.

8.1.4. Ciclul de viaţă al unui fir de execuţie

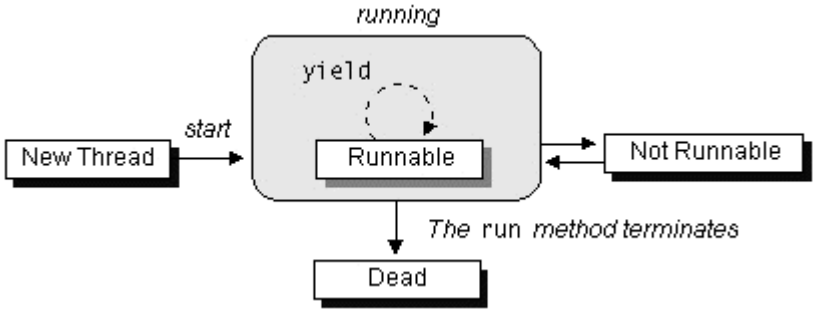

Figura 8-1 Ciclul de viaţă al unui fir de execuţie
(sursa: http://www.iam.ubc.ca/guides/javatut99/essential/threads/lifecycle.html)

Stările în care se poate afla un fir de execuţie la un moment dat:

- *New Thread*
  - Este creată o instanţă a clasei Thread sau a unei subclase a acesteia.
  - În această stare nu se execută încă nimic în paralel, firul de execuţie este „vid”.
- *Runnable*
  - Thread-ul este lansat în execuţie prin apelul metodei ***start()*** din clasa Thread.
  - Se alocă resursele de sistem necesare şi se apelează metoda ***run()***.
  - În această stare un fir de execuţie poate să execute instrucţiuni sau să-şi aştepte rândul la procesor.
- *Not Runnable* – se poate ajunge în această stare dacă:
  - Se apelează metoda **Thread.sleep(…)**.
  - Se apelează metoda ***wait()*** (din clasa object).
  - Firul este blocat într-o operaţie de intrare/ieşire.
- *Dead*
  - În această stare se ajunge când firul şi-a terminat execuţia.
  - Un fir de execuţie nu poate fi oprit din program prin apelarea unei metode (în Java versiunea 1.0 exista metoda ***stop()*** pentru a se opri un fir, dar aceasta a fost marcată ca fiind *deprecated* din motive de securitate).

8.1.5. Cazurile în care un fir de execuţie poate ajunge din starea *Not Runnable* în starea *Runnable*

- Dacă a fost apelată metoda ***sleep()***, atunci firul execuţie ajunge în starea Runnable după scurgerea intervalului de timp specificat.
- Dacă a fost apelată metoda ***wait()***, atunci un alt fir de execuţie trebuie să-l informeze dacă acea condiţie este îndeplinită sau nu (folosind metodele ***notify()*** şi ***notifyAll()*** din clasa *Object*).
- Dacă firul este blocat într-o operaţie de intrare/ieşire, atunci firul redevine *Runnable* când operaţiunea respectivă s-a terminat.

8.1.6. Metoda *isAlive*()

Această metodă returnează:

- ***true***, dacă firul de execuţie a fost pornit şi nu a fost oprit (este în starea *Runnable* sau *Not Runnable*)
- ***false***, dacă firul de execuţie este fie în starea *New Thread*, fie în starea *Dead*.

8.1.7. Terminarea unui fir de execuţie

Pentru ca un fir de execuţie să se termine trebuie ca metoda *run()* să-şi termine execuţia. Există două metode pentru terminarea unui fir de execuţie:

- Metoda *run()* îşi termină execuţia în mod natural: dacă avem instrucţiuni cu timp de execuţie finit sau bucle finite.
- Se foloseşte o variabilă de terminare: dacă metoda *run()* trebuie să execute o buclă infinită. Periodic se interoghează variabila respectivă şi dacă este îndeplinită condiţia de terminare se iese din buclă.

8.1.8. Priorităţile de execuţie

Priorităţile de execuţie ale unui thread se stabilesc la crearea unui nou fir de execuţie şi se pot schimba pe parcursul programului folosind metoda ***setPriority(…)*** a clasei *Thread*.

În general, când există mai multe fire de execuţie cu priorităţi diferite, sunt preferate firele care au prioritate mai mare, dar nu există nici o garanţie în acest sens deoarece sistemul de operare poate interveni în alocarea cuantelor de timp.

Apelul metodei ***yield()*** din clasa *Thread* are ca efect oprirea temporară a execuţiei thread-ului pentru a permite celorlalte thread-uri să se execute.

Prioritatea de execuţie este un număr întreg cuprins între MIN_PRIORITY şi MAX_PRIORITY, valoarea implicită fiind NORM_PIORITY. Aceste constante statice sunt definite în clasa *Thread*:

- `public static final int MAX_PRIORITY = 10;`
- `public static final int NORM_PRIORITY = 5;`
- `public static final int MIN_PRIORITY = 1;`

8.2. Sincronizarea

Firele de execuţie comunică între ele prin intermediul unor resurse comune (câmpuri). Acest lucru poate da naştere la interferenţa firelor de execuţie sau la erori de consistenţă a memoriei. Pentru a se evita această situaţie se poate folosi sincronizarea metodelor sau sincronizarea blocurilor.

8.2.1. Sincronizarea metodelor

Sincronizarea metodelor se realizează cu ajutorul cuvântului cheie **synchronized**. Atunci când un thread execută o anumită metodă a unui obiect, celelalte fire care vor să execute aceiaşi metodă sincronizată pentru acelaşi obiect trebuie să aştepte ca primul fir să termine de executat metoda respectivă. La un moment dat, o metodă sincronizată poate să fie executată doar de un singur fir de execuţie.

```
public class ContorSincronizat {
	private int contor = 0;

	public synchronized void incrementeaza() {
		contor++;
	}

	public synchronized void decrementeaza() {
		contor--;
	}

	public synchronized int getContor() {
		return contor;
	}
}
```

8.2.2. Sincronizarea blocurilor

Spre deosebire de sincronizarea metodelor, în cazul sincronizării blocurilor trebuie specificat obiectul pentru care se realizează sincronizarea. Sintaxa este: ***synchronized*** *(obiect) { ... }*.

```
public class ContorSincronizat {
	private int contor = 0;

	public void incrementeaza() {
		synchronized (this) {
			contor++;
		}
	}

	public void decrementeaza() {
		synchronized (this) {
			contor--;
		}
	}

	public int getContor() {
		synchronized (this) {
			return contor;
		}
	}
}
```

8.3. Exemplu

```
import java.io.IOException;

public class BuclaInfinita extends Thread {

 private boolean stopProgram;
 private int   count;

 public BuclaInfinita(String nume) {
  super(nume);
  this.stopProgram = false;
  this.count = 0;
 }

 @Override
 public void run() {
  while (!stopProgram) {
   try {
    Thread.sleep(1000);
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
   count++;
  }
 }

 public void stopProgram() {
  this.stopProgram = true;
 }

 public int getCount() {
  return count;
 }

 public static void main(String[] args) throws IOException {
  BuclaInfinita t = new BuclaInfinita("Firul meu");
  t.start();
  System.in.read();
  t.stopProgram();
  while (t.isAlive()) {
   System.out.println("Firul de executie inca nu s-a terminat");
   try {
    Thread.sleep(1000);
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
  }
  System.out.println("Au trecut " + t.getCount() + " secunde de la apasarea
tastei");
 }

}
```

8.4. Temă

Realizaţi o aplicaţie ce va permite citirea simultană din 3 fişiere text (cuvânt cu cuvânt) şi scrierea cuvintelor pe măsură ce sunt citite într-un al patrulea fişier. Pentru fiecare citire din fişier se va folosi câte un fir de execuţie, iar scrierea cuvintelor se va face sincronizat.