**Fire de executie**

Sistemele de operare moderne folosesc multitaskuri preventive pentru a aloca timp pe procesor pentru aplicatii. Avem doua tipuri de task-uri ce pot fi programate pentru executie:

* Procesele: un proces este o zona de memorie ce contine cod si date. Un proces are un fir de executie ce este programat sa primeasca procesorul pentru portiuni de timp. Portiunea de timp (time slice) este de obicei masurata in milisecunde. Cand timpul a expirat taskul este in mod fortat indepartat de pe procesor si un alt task ii ia locul
* Fire: un fir este o executie programata a unui proces. Firele pot fi concurente. Toate firele unui proces impart aceleasi date de memorie, dar pot urmari diferite cai catre o sectiune a codului

Pentru a executa un program cat mai repede cu putinta trebuie sa evitam blocajele (bottleneck). Exista cateva situatii care descriu aceste blocaje:

* Conexiuni la resurse: doua sau mai multe task-uri asteapta pentru utilizarea exclusiva a unei resurse
* Blocarea din cauza operatiilor de I/O: pe timpul acestor operatii se asteapta transferul datelor
* Subutilizarea procesorului: o aplicatie intr-un singur fir utilizeaza un singur procesor

Chiar daca nu scriem explicit cod pentru a crea noi fire de executie, codul poate rula intr-un mediu multifir. Trebuie asadar sa fim avertizati cum lucreaza firele si cum sa scriem cod thread-safe. Spre exemplu, cand scriem code care ruleaza intr-un alt software (middleware sau server de aplicatie) trebuie avuta in vedere documentatia produsului pentru a descoperii daca firele sunt create automat. Intr-un server de aplicatie Java EE exista o componenta numita servlet utilizata pentru manipularea cererilor HTTP. Servlet-urile trebuie sa fie intotdeauna thread-safe deoarece serverul porneste un nou fir la fiecare cerere HTTP.

**Clasa Thread**

Este folosita pentru a crea si porni fire. Codul ce urmeaza a fi executat de un fir trebuie plasat intr-o clasa ce face una din urmatoarele:

* Extinde clasa Thread (codarea este mai simpla). Va trebui sa suprascriem metoda run().
* Implementeaza interfata Runnable (mult mai flexibil)

Exemplu:

**public** **class** ExampleThread **extends** Thread {

@Override

**public** **void** run() {

**for**(**int** i = 0; i < 100; i++) {

System.*out*.println("i:" + i);

}

}

}

Codul de executat in noul fir trebuie plasat in metoda run(). Trebuie sa evitam apelul lui run() direct. Apelul lui run() nu va porni un nou fir, efectul fiind acelasi al apelului unei metode obisnuite.

Dupa crearea firului, el trebuie pornit prin apelul metodei start():

**public** **class** ExtendingThreadMain {

**public** **static** **void** main(String[] args) {

ExampleThread t1 = **new** ExampleThread();

t1.start();

}

}

Cand apelam start() se programeaza pentru apel metoda run(). Un fir poate fi pornit doar o data.

Implementand Runnable suntem obligati sa furnizam cod pentru run(). Beneficiul implementarii lui Runnable este acela ca inca putem extinde o alta clasa.

**public** **class** ExampleRunnable **implements** Runnable {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 100; i++) {

System.*out*.println("i:" + i);

}

}

}

Dupa ce am creat un Runnable trebuie sa il trimitem constructorului lui Thread. Doar Thread poate porni firul:

**public** **class** ImplementingRunnableMain {

**public** **static** **void** main(String[] args) {

ExampleRunnable r1 = **new** ExampleRunnable();

Thread t1 = **new** Thread(r1);

t1.start();

}

}

Campurile statice sau campurile instanta sunt potential impartite intre fire.

**public** **class** ExampleSharedRunnable **implements** Runnable {

**private** **int** i;

@Override

**public** **void** run() {

**for** (i = 0; i < 100; i++) {

System.*out*.println("i:" + i);

}

}

}

Obiectele care sunt referite de mai multe fire au campurile accesate concurent.

**public** **class** SharedDataMain {

**public** **static** **void** main(String[] args) {

ExampleSharedRunnable r1 = **new** ExampleSharedRunnable();

Thread t1 = **new** Thread(r1);

t1.start();

Thread t2 = **new** Thread(r1);

t2.start();

}

}

In exemplu am trimis aceasi instanta Runnable instantelor multiple Thread. Instantele Thread impart campurile instanta Runnable. Acelasi comportament il au si campurile statice.

Datele share-uite trebuie accesate cu grija. Campurile instanta si cele statice:

* Sunt create intr-o zona de memorie cunoscuta ca spatiul heap
* Pot fi potential impartite intre orice fire
* Pot fi modificate concurent de mai multe fire. Nu exista erori de compilare sau avertizari ale IDE-urilor. Accesul sigur al campurilor share-uite sta in grija programatorului

Debugging-ul firelor poate fi dificil deoarece frecventa si durata in care procesorul este alocat firului pot varia din multiple motive:

* Programatorul firului de executie este gestionat de un sistem de operare si sistemele de operare pot utiliza diversi algoritmi de programare
* Masinile pot avea diverse viteze ale procesorului
* Alte aplicatii pot incarca sistemul

De aceea este posibil ca o aplicatie sa functioneze corect la dezvoltare si sa aiba probleme in productie.

Unele variabile nu sunt vreodata share-uite, ele sunt in permanenta type-safe: variabilele locale, parametrii metodelor, parametrii manipulatorilor de exceptie. De asemenea, datele imutabile, precum obiectele string sau campurile finale, sunt thread-safe deoarece ele pot fi citite dar nu pot fi scrise.

**Operatii atomice**

O operatie atomica functioneaza ca o singura operatie. Spre exemplu, o instructiune in Java nu este intotdeauna atomica. Spre exemplu:

i++; - creaza o copie temporara a lui i, incrementeaza copia si apoi scrie noua valoare inapoi in i.

l = 0xffff\_ffff\_ffff\_ffff; - este o variabila de 64 de biti ce poate fi accesata folosind doua operatii separate pe 32 de biti.

O problema posibila cu doua fire de executie ce incrementeaza acelasi camp este ca modificarea poate fi pierduta. Sa presupunem ca doua fire de executie citesc valoarea 41 dintr-un camp, incrementeaza valoarea cu unu si scriu rezultatul inapoi in camp. Ambele fire incheie incrementarea dar rezultatul este doar 42. Dependent de cum este implementata JVM si de tipul de procesor utilizat, putem obtine aceasta situatie rar sau niciodata, dar trebuie avut in vedere ca ea se poate intampla.

Daca avem un numar lung 0xffff\_ffff\_ffff\_ffff si il incrementam cu 1, rezultatul ar trebui sa fie 0x0000\_0001\_0000\_0000. Din cauza operatiei ce se face in doua etape putem avea 0x0000\_0001\_ffff\_ffff sau 0x0000\_0000\_0000\_0000 dependent de care biti se modifica primii. Daca un al doilea fir are permisiunea sa citeasca un camp de 64 de biti in timp ce este modificat de un alt fir, o valoare incorecta poate fi obtinuta.

Operatiile efectuate intr-un fir pot sa nu apara la executie in ordine daca vom observa rezultatele dintr-un alt fir. Aceasta deoarece firele opereaza pe copii cache ale variabilelor sharuite.

Ca sa ne asiguram de consistenta comportamentului in fire, trebuie sa sincronizam actiunile lor, adica sa ne asiguram ca o actiune se intampla inaintea alteia, respectiv sa golim cache-ul variabilelor share-uite.

Fiecare fir are o memorie de lucru in care isi tine copiile de lucru ale variabilelor. Cand firul executa un program opereaza pe aceste copii. Exista cateva actiuni ce sincronizeaza memoria de lucru a firului cu memoria principala:

* Citirea sau scrierea volatila a unei variabile
* Blocarea sau deblocarea
* Prima si ultima actiune a unui fir
* Actiuni ce pornesc un fir sau detecteaza ca un fir s-a terminat

Un camp poate fi declarat ca volatile. Scrierea sau citirea unui camp volatil va determina ca firul sa-si sincronizeze memoria de lucru cu memoria principala. Volatil nu inseamna, insa, atomic, adica operatia in care este implicat campul nu este thread safe.

Un fir se opreste prin incheierea metodei run().

Vom da un exemplu de folosire a campurilor volatile pentru oprirea unui fir de executie.

**public** **class** ExampleVolatileRunnable **implements** Runnable {

**public** **boolean** timeToQuit = **false**;

@Override

**public** **void** run() {

System.*out*.println("Thread started");

**while** (!timeToQuit) {

System.*out*.println("thread running");

}

System.*out*.println("Thread finishing");

}

}

**public** **class** ThreadStoppingMain {

**public** **static** **void** main(String[] args) {

ExampleVolatileRunnable r1 = **new** ExampleVolatileRunnable();

Thread t1 = **new** Thread(r1);

t1.start();

// ...

r1.timeToQuit = **true**;

}

}

Metoda main() in aplicatiile Java SE este executata intr-un fir, numit si firul principal, ce este dreat automat de JVM. Ca orice fir si firul principal, atunci cand scrie in campul timeToQuit este important ca scrierea sa fie vazuta de firul t1. Daca timeToQuit nu ar fi fost volatil nu exista garantia ca scrierea s-ar fi vazut imediat.

Observatie: este foarte posibil ca si fara declararea volatila aplicatia sa functioneze perfect, dar nu exista garantia ca aceasta s-ar fi intamplat tot timpul.

Cuvantul synchronized este utilizat pentru a crea blocuri de cod thread-safe. Un bloc sincronizat:

* Determina firul sa scrie toate modificarile in memoria principala atunci cand s-a atins sfarsitul firului. Acest comportament este similar cu volatile
* Este utilizat pentru gruparea blocurilor de cod in vederea utilizarii exclusive de catre un fir. Acest comportament rezolva si problema atomicitatii.

Fie urmatorul exemplu:

**public** **class** Item {

**private** **int** id;

**private** String description;

**private** **double** price;

**public** Item() {

}

**public** Item(**int** id, String description, **double** price) {

**this**.id = id;

**this**.description = description;

**this**.price = price;

}

/\*\*

\* **@return** the id

\*/

**public** **int** getId() {

**return** id;

}

/\*\*

\* **@param** id

\* the id to set

\*/

**public** **void** setId(**int** id) {

**this**.id = id;

}

/\*\*

\* **@return** the description

\*/

**public** String getDescription() {

**return** description;

}

/\*\*

\* **@param** description

\* the description to set

\*/

**public** **void** setDescription(String description) {

**this**.description = description;

}

/\*\*

\* **@return** the price

\*/

**public** **double** getPrice() {

**return** price;

}

/\*\*

\* **@param** price

\* the price to set

\*/

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

}

**public** **class** ShoppingCart {

**private** List<Item> cart = **new** ArrayList<>();

**public** **synchronized** **void** addItem(Item item) {

cart.add(item);

}

**public** **synchronized** **void** removeItem(**int** index) {

cart.remove(index);

}

**public** **synchronized** **void** printCart() {

Iterator<Item> ii = cart.iterator();

**while**(ii.hasNext()) {

Item i = ii.next();

System.*out*.println("Item:" + i.getDescription());

}

}

// public void printCart() {

// StringBuilder sb = new StringBuilder();

// synchronized (this) {

// Iterator<Item> ii = cart.iterator();

// while (ii.hasNext()) {

// Item i = ii.next();

// sb.append("Item:");

// sb.append(i.getDescription());

// sb.append("\n");

// }

// }

// System.out.println(sb.toString());

// }

}

In obiectul ShoppingCart putem apela doar o metoda la un moment dat, deoarece toate metodele sunt sincronizate. Doua instante ShoppingCart nu pot fi utilizate concurent. Practic, sincronizarea se extinde intregului obiect.

Daca metodele nu ar fi fost sincronizate, apelul lui removeItem() in timp ce printCart() itereaza prin colectia de Item poate crea rezultate impredictibile. Un iterator care esueaza va arunca un java.util.ConcurrentModificationException, o exceptie runtime, daca iteratorul colectiei este modificat in timp ce este utilizat (vezi metoda comentata pentru exemplu de folosire a unui bloc sincronizat).

Sincronizarea in aplicatii multifir asigura un comportament consistent . Deoarece blocurile si metodele sincronizate sunt folosite pentru a restrictiona o sectiune de cod pentru a fi utilizata de un singur fir, putem afecta performanta prin ceea ce se numeste bottleneck. Blocurile sincronizate pot fi utilizate in locul metodelor sincronizate pentru a reduce numarul de linii utilizate exclusiv de un singur fir.

Trebuie sa utilizam sincronizarea cat se poate de putin, pentru ca reduce performanta, dar cat este nevoie pentru a garanta consistenta.

Orice obiect in Java este asociat unui monitor, pe care un fir il poate bloca sau debloca. Metodele sincronizate utilizeaza monitorul pentru obiectul this. Metodele statice sincronizate utilizeaza monitorul claselor, iar blocurile sincronizate pot specifica ce monitor de obiect sa blocheze sau sa deblocheze.

Blocurile sincronizate pot fi incuibate. Prin blocuri sincronizateputem bloca mai multe monitoare simultan.

**Intreruperea unui fir**

Intreruperea reprezinta o alta modalitate pentru a cere ca un fir sa-si opreasca executia. Cand un fir este intrerupt depinde de programator sa decida ce actiune se va produce in continuare. Aceasta poate fi returnarea din metoda run() sau continuarea executiei codului.

Orice fir are metodele interrupt() si isInterrupted(). Iata un exemplu de folosire a lor:

**public** **class** ExampleInterruptedRunnable **implements** Runnable {

@Override

**public** **void** run() {

System.*out*.println("Thread started");

**while**(!Thread.*interrupted*()) {

// ...

}

System.*out*.println("Thread finishing");

}

}

**public** **class** ThreadInterruptingMain {

**public** **static** **void** main(String[] args) {

ExampleInterruptedRunnable r1 = **new** ExampleInterruptedRunnable();

Thread t1 = **new** Thread(r1);

t1.start();

// ...

t1.interrupt();

}

}

Intreruperea este un mod convenabil de a opri un fir. Intreruperea se poate face si asupra firelor blocate.

Un fir isi poate opri executia pentru o perioada de timp.

**public** **class** ThreadSleepMain {

**public** **static** **void** main(String[] args) {

**long** start = System.*currentTimeMillis*();

**try** {

Thread.*sleep*(4000);

} **catch** (InterruptedException ex) {

// What to do?

}

**long** time = System.*currentTimeMillis*() - start;

System.*out*.println("Slept for " + time + " ms");

}

}

In exemplul anterior am oprit firul curent pentru patru secunde. Dupa patru secunde firul este programat din nou pentru executie. Asta nu inseamna ca firul porneste imediat dupa 4 secunde. Durata de „adormire” este afectata de hardware, sistemul de operare si incarcarea sistemului.

Daca apelam interrupt() pe un fir adormit, metoda sleep() va arunca un InterruptedException ce trebuie manipulata. Daca intreruperea a insemnat doar intreruperea somnului si nu executia firului atunci putem ignora exceptia. Altfel, va trebui sa rearucam exceptia sau sa incheiem petoda run().

**Alte metode ale clasei Thread**

* setName(String), getName() si getId()
* isAlive(), returneaza true daca firul nu s-a incheiat
* isDaemon() si setDaemon(boolean). JVM poate sa-si incheie executie in timp ce firele daemon ruleaza
* join(), firul curent asteapta ca alte fire sa se incheie
* Thread.currentThread(), instantele Runnable pot returna instante Thread ce se executa in monentul curent

Si clasa Object are metode ce sunt in legatura cu firele de executie: wait(), notify() si notifyAll(). Firele pot merge la culcare pentru o perioada nedeterminata de timp, se trezesc doar cand obiectul cerut este disponibil, iar firul adormit este instiintat printr-o notificare.

Firele daemon sunt fire background ce sunt mai putin importante decat firele normale. Pentru ca firul principal nu este un fir daemon, toate firele pe care le vom crea nu vor fi fire daemon. Orice fir nondaemon ce ruleaza va retine JVM de la a se inchide, chiar daca metoda main() s-a incheiat. Daca un fir nu poate preveni JVM de la a se inchide, atunci acesta este un fir daemon.

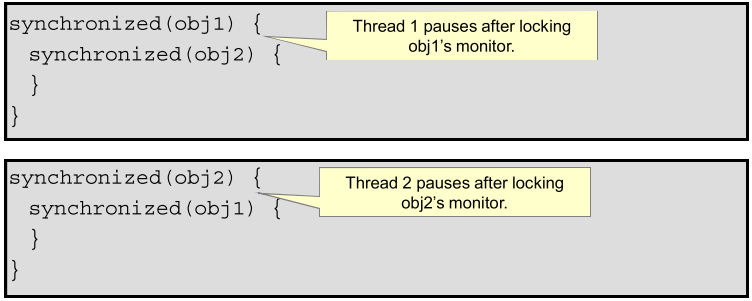
Urmatoarele metode ar trebui evitate:

* setPriority(int) si getPriority(), in primul rand pentru ca e foarte posibil sa nu aiba vreun impact

Urmatoarele metode sunt invechite si nu ar trebui folosite: destroy(), resume(), suspend(), stop().

**Deadlock**

Blocarea mortala apare atunci cand doua sau mai multe fire sunt blocate definitiv, asteptand unul dupa altul.



Alte probleme ce pot apare relativ la firele de executie sunt starvation si livelock.

Starvation descrie o situatie in care un fir este incapabil sa obtina acces la resurse share-uite si nu este capabil sa progreseze. Aceasta se intampla atunci cand resursele impartite sunt indisponibile pentru o lunga perioada de timp din cauza firelor „lacome”. De exemplu, presupunem ca un obiect furnizeaza o metoda sincronizata careia ii ia mult timp pentru a se incheia. Daca un fir invoca aceasta metoda frecvent, alte fire ce necesita acces sincronizat frecvent la acelasi obiect vor fi deseori blocate.

Deseori un fir actioneaza ca raspuns la actiunea unui alt fir. Daca actiunea celuilalt fir este tot un raspuns la primul apare livelock. Ca si deadlock, in livelock firele nu sunt capabile sa faca progrese. Firele nu sunt blocate, dar sunt permanent ocupate raspunzandu-si unul altuia.

Vom examina particularitatile programarii multifir prin problema celor cinci filosofi, enuntata astfel: cinci filosofi sunt asezati la o masa rotunda incercand sa manance avand la dispozitie 5 betigase. Un betigas este plasat intre fiecare doi filosofi. Un filosof are nevoie de doua betigase pentru a putea minca.

Rezolvarea aplicatiei din exemplul de mai sus prin fire de executie, determina gestionarea “starii” unui filosof printr-un fir de executie. Cele cinci fire nu pot opeara independent pentru ca impart un numar prea mic de betigase si deci un obiect Betigas este folosit la un moment dat de un sngur fir. In plus, fiecare filosof trebuie sa afle cand un betigas este eliberat de catre un alt filosof, astfel incat sa-l poata folosi.

Clasa ce modeleaza starea unui betigas este:

class Betigas {

boolean betigasDetinut = false ;

public synchronized void eliberareBetigas(){

betigasDetinut = false ;

notifyAll () ; // betigasul este eliberat

}

public synchronized void luareNetigas() throws InterruptedException {

while (betigasDetinut) wait() ; // pina betigasul este eliberat

betigasDetinut = true ;

}

}

Atributul de tip boolean al clasei da starea obiectului, mai precis daca betigasul este detinut de catre cineva sau nu. Initiat betigasul este disponibil.

Cele doua metode simuleaza actiunile de modificare a starii: eliberareBetigas() si luareNetigas(). Ambele sunt sincronizate, prima notifica celelalte fire despre eliberarea betigasului, prin metoda notifyAll(), a doua blocheaza firul curent pina ce betigasul este eliberat, prin metoda wait().

Clasa care modeleaza comportamentul unui filosof o vom numi Filosof si implementeaza Runnable. Clasa are doua atribute, indicele betigaselor cu care maninca filosoful, i cel din dreapta respectiv j cel din stanga. Initial fiecare filosof are un singur betigas de indice egal cu cel al sau, din sirul de filosofi. Metoda run() va contine un ciclu infinit in care starile filosofului se modifica. Initial firul este intrerupt pentru o perioada aleatoare cuprinsa intre 0 si 3 secunde, considerata perioada in care filosoful gandeste (de aia e filosof !). Apoi filosoful incearca sa ia un betigas prin aplelul metodei luareBetigas(). Dupa aceasta incearca obtinerea celui de-al doilea betigas prin apelul aceleiasi metode. Avand cele doua betigase filosoful maninca un timp aleator curins intre 1 si 2 secunde. Perioada mesei este simulata prin intreruperea firului pe o durata limitata. Dupa masa filosoful elibereaza ambele betigase prin intermediul metodei eliberareBetigas(). Cum metoda luareBetigas() este sincronizata nu este posibil ca doi filosofi sa o apeleze simultan. Deci, un singur fir este notificat la un moment dat de eliberarea obiectului. Folosirea in acest caz a apelului lui notify() sau notifyAll() este indiferenta.

Problema mare ce apare in cazul exemplului nostru este evitarea punctului mort. Iata un scenariu. Presupunem ca suntem la inceputul mesei si fiecare filosof sa ia betigasul din dreapta sa. Nu poate manca doar cu unul si de aceea incearca sa il ia pe al doilea. Fiecare va apela la nesfarsit metoda wait(). Cum niciun filosof nu are doua betigase metoda notifyAll() nu va fi apelata vreodata. Programul este in acest moment intr-un punct mort. Noi am construit rezolvarea prin intreruperea initiala, aleatorie, a fiecarui fir de excutie, perioada denumita de gandire. Este putin probabil ca fiecare filosof sa gandeasca la fel de mult, deci punctul mort nu se va instala inca de la inceput. Dar nu este exclus sa se instaleze pe parcurs. Vom evita instalarea punctului mort prin faptul ca ne asiguram ca la un moment dat un filosof nu are doar un singur betigas, ci poate sa il ia si pe al doilea. In cod am evitat blocarea prin faptul ca un filosof este obligat ca mai intai sa-si aleaga betigasul de indice minim si dupa ce a facut rost de acesta il va lua pe celalalt, care sigur exista. Spre exemplu, daca filosoful 0 si 1 doresc in acelasi timp sa manince (0 are nevoie de betigasele 0 si 5, iar 1 de 0 si 1) vor astepta ambii dupa betigasul 0 (de indice minim). Daca filosoful 1 va primi betigasul 0 atunci il va lua si pe 1 care este disponibil, altfel 0 il va lua pe 5. Conform acestei strategii 5 va fi intotdeauna disponibil ca al doilea betigas.

Vom retine obiectele Filosof intr-un sir de obiecte si vom crea pentru fiecare un fir de executie. De asemenea, vom crea un sir de obiecte Betigas, initial fiecare filosof detine un astfel de obiect, de indice egal cu al sau.

Codul claselor Filosof si Test este dat in continuare:

**public** **class** Test{

Filosof[] filosofi ;

Thread[] fir ;

Betigas[] betigase ;

**public** Test () {

filosofi = **new** Filosof[5] ;

fir = **new** Thread[5] ;

betigase = **new** Betigas[5] ;

}

**public** **void** porneste () {

// construieste 5 instante ale claselor Filosof si Betigas

**for** (**int** i = 0; i<5 ; i++ ) {

betigase[i] = **new** Betigas() ;

filosofi[i] = **new** Filosof (i) ;

fir[i] = **new** Thread ( filosofi[i] ) ;

fir[i].start() ;

}

}

**public** **static** **void** main (String args[]) {

Test x = **new** Test () ;

x.porneste();

}

**class** Filosof **implements** Runnable {

**int** i, j ;

**public** Filosof (**int** myNumber) {

i = j = myNumber ;

}

**public** **void** run () {

**while** (**true**) {

System.*out*.println("filosoful " + i + " gandeste"); // incepe prin a gandi

**try** {

// timp de gandire aleator intre 0 si 3 secunde

**double** r = Math.*random*() \* 3.0 ;

Thread.*sleep*((**long**)(r \* 1000)) ;

// alege ca indice al betigasului stang 4 daca indicele filosofului este 0

**if**(i - 1 == -1) j=4; **else** j=(i - 1) ;

// aici se previne punctul mort

betigase[Math.*min*(i,j)].luareBetigas();

System.*out*.println("filosoful "+i+" are betigasul "+Math.*min*(i,j));

// acum incearca sa ia al doilea betigas

betigase[Math.*max*(i,j)].luareBetigas();

// maninca un timp intre 1 si 2 secunde

System.*out*.println("filosoful "+i+" are si betigasul "+Math.*max*(i,j)+" si maninca");

Thread.*sleep*((**long**)(r \* 2000)) ;

// dupa masa elibereaza betigasele

betigase[i].eliberareBetigas();

betigase[j].eliberareBetigas();

}

**catch** (InterruptedException e) {

System.*out*.println("filosoful " + i + " gandeste");

**return** ;

}

}

}

}

**class** Betigas {

**boolean** betigasDetinut = **false** ;

**public** **synchronized** **void** eliberareBetigas() {

betigasDetinut = **false** ;

notifyAll () ; // anunta eliberarea betigasului

}

**public** **synchronized** **void** luareBetigas() **throws** InterruptedException {

**while** (betigasDetinut) wait() ; // asteapta eliberarea betigasului

betigasDetinut = **true** ;

}

}

}

Un exemplu tipic de blocare este prezentat in exemplul urmator:

**class** A {

**synchronized** **void** met1(B b) {

String nume = Thread.*currentThread*().getName();

System.*out*.println(nume + " apelul metodei A.met1");

**try** {

Thread.*sleep*(1000);

} **catch**(Exception e) {

System.*out*.println("A este intrerupta");

}

System.*out*.println(nume + " incearca sa apeleze B.met2()");

b.met2();

}

**synchronized** **void** met2() {

System.*out*.println("apelul metodei A.met2");

}

}

**class** B {

**synchronized** **void** met1(A a) {

String name = Thread.*currentThread*().getName();

System.*out*.println(name + " apelul metodei B.met1");

**try** {

Thread.*sleep*(1000);

} **catch**(Exception e) {

System.*out*.println("B este intrerupta");

}

System.*out*.println(name + " incearca sa apeleze A.met2()");

a.met2();

}

**synchronized** **void** met2() {

System.*out*.println("apelul metodei A.met2");

}

}

**public** **class** Deadlock **implements** Runnable {

A a = **new** A();

B b = **new** B();

Deadlock() {

Thread.*currentThread*().setName("Firul Principal");

Thread t = **new** Thread(**this**, "Firul Concurent");

t.start();

a.met1(b);

System.*out*.println("inapoi in firul principal");

}

**public** **void** run() {

b.met1(a);

System.*out*.println("inapoi in firul concurent");

}

**public** **static** **void** main(String args[]) {

**new** Deadlock();

}

}