Fire de executie (thread-uri)

Exemple de implementari de mecanisme clasice bazate pe fire de executie (thread-uri)

Fire de executie (thread-uri): implementare variabile lock

Lock.java

```
public class Lock {

    private Thread owner = null;

    public Lock() {
        super();
    }

    public synchronized void lock() {
        Thread who = Thread.currentThread();
        if(owner == who) return; //owner holds the lock
        else if(owner == null) {
            owner = who;
        }
        else /* owner != who && owner != null */ {
            while (owner != null) {
                try {
                    wait();
                }
                catch(InterruptedException e) {}
                owner = who;
                return;
            }
        }
    }

    public synchronized void unlock() throws IllegalMonitorStateException {
        Thread who = Thread.currentThread();
        if(owner == null) return;
        else if(owner != who) {
            throw new IllegalMonitorStateException();
        } else /* owner == who */ {
            owner = null;
            notify();
        }
    }
}
```

Asupra unei variabile de tip lock nu se pot executa decat doua operatii: lock() si unlock(); Orice alt fir de executie care incearca sa mai execute operatii legate de lock se blocheaza pentru ca zavorul este detinut de firul de executie care detine lock-ul.

Fire de executie (thread-uri): implementare semafoare

Semaphore.java

```
public class Semaphore {
    private int value = 0;

    public Semaphore(){
        value = 0;
    }
    public Semaphore(int initial){
        if(initial < 0) throw new IllegalArgumentException("innitial < 0");
        System.out.println("START, val = " + value);
        value = initial;
    }
    public synchronized void P() {
        value--;
        System.out.println("P(), val = " + value);
        if(value < 0) {
            while (true) {
                try {
                    wait();  // waiting notify()
                    System.out.println("P(), waiting, val = " + value);
                    break;  //it was notify()
                }
                catch(InterruptedException e) {
                    if(value >= 0) {
                        System.out.println("P(),stop waiting, val = " + value);
                        break; // stop waiting
                    }
                    else continue;      // it was not V()
                }
            }
        }
    }
    public synchronized void V(){
        value++;
        System.out.println("V(), val = " + value);
        if(value <= 0) {
            System.out.println("V(), notify, val = " + value);
            notify();
        }
    }
    public synchronized int getValue(){
        return value;
    }
    public static void main(String args[]) {
        Semaphore s1 = new Semaphore();
        s1.V();
        s1.P();
    }
}
```

Daca valoare < 0 => exista thread-uri blocate datorita operatiei P()
In metoda P() s-a facut tratarea situatiei in care in timp ce un thread este blocat datorita executiei operatiei P(), este intrerupt prin executia metodei interrupt().
Sa presupunem ca este un singur thread blocat pentru ca a executat operatia P() si ca un alt thread incepe executia unei operatii V() pentru acelasi semafor, in timp ce al treilea thread apeleaza metoda interrupt() pentru thread-ul blocat. Efectul acestui apel este ca thread-ul din blocat devine gata de executie si cand ii va veni randul va continua executia in ciclul de asteptare. Pentru a evita un deadlock in cazul in care se executa notify() si nimeni nu mai este blocat, se introduce in secventa de tratare a intreruperii portiunea de cod in care se verifica daca nu cumva conditiile de incheiere a executiei operatiei P() sunt indeplinite (if (value >= 0) break);) Daca exista mai multe thread-uri blocate pentru ca au executat operatia P() atunci nu se poate pierde aparitia semnalului de trezire produs de metoda notify().

Fire de executie (thread-uri): implementare bariere

BarrierExample.java | Barrier.java

```
class BarrierExample
{
    static class MyThread1 implements Runnable {
        public MyThread1(Barrier barrier){
            this.barrier = barrier;
        }
        public void run(){
            try {
                Thread.sleep(1000);
                System.out.println("MyThread1 waiting on barrier");
                barrier.block();
                System.out.println("MyThread1 has been released");
            } catch (InterruptedException ie){
                System.out.println(ie);
            }
        }
        private Barrier barrier;
    }

    static class MyThread2 implements Runnable {
        Barrier barrier;
        public MyThread2(Barrier barrier){
            this.barrier = barrier;
        }
        public void run(){
            try {
                Thread.sleep(3000);
                System.out.println("MyThread2 releasing blocked threads");
                barrier.release();
                System.out.println("MyThread1 releasing blocked threads");
            } catch (InterruptedException ie) {
                System.out.println(ie);
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Barrier BR = new Barrier();
        Thread t1 = new Thread(new BarrierExample.MyThread1(BR));
        Thread t2 = new Thread(new BarrierExample.MyThread2(BR));
        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }
}
```

Barrier.java | BarrierExample.java

```
public class Barrier
{
    public synchronized void block() throws InterruptedException
    {
        wait();
    }

    public synchronized void release() throws InterruptedException
    {
        notify();
    }

    public synchronized void releaseAll() throws InterruptedException
    {
        notifyAll();
    }

}
```

Se utilizeaza, in aceste exemplu, 2 clase care folosesc bariera pentru sincronizare. Primul thread care soseste asteapta la bariera, al doilea deblocheaza primul thread, urmat de primul, cu aceeasi actiune, apoi este deblocat al doilea thead

```
MyThread1 waiting on barrier
MyThread2 releasing blocked threads
MyThread1 releasing blocked threads
MyThread1 has been released
```

Fire de executie (thread-uri): server web simplificat

WebServer.java | Handler.java | TestClientWebServer.java

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class WebServer implements Runnable {
    static final int PORT = 1042;
    final int TIMEOUT = 30000;
    Handler handler = new Handler();
    ServerSocket socket = null;

    public void run() {
        try {
            socket = new ServerSocket(PORT);
            socket.setSoTimeout(TIMEOUT);
            System.out.println("[SERVER]Socket: " + socket.getLocalPort() + " " + socket.getSoTimeout());
            System.out.println("[SERVER] Web server started and waiting for requests ...");
            while(true){
                final Socket connection = socket.accept();
                new Thread(new Runnable() {
                    public void run() {
                        System.out.println("[SERVER] N waiting for requests ...");
                        handler.process(connection);
                    }
                }).start();
            }
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
    public static void main (String args[]) throws IOException{
        WebServer webServer = new WebServer();
        webServer.run();
    }
}
```

Se implementeaza un server de web simplificat; se creeaza un socket folosind portul 1042. Se stabileste si un timeout de 30 de sec. Pentru fiecare accesare a socket-ului, se porneste un nou thread in care se apeleaza metoda process a handler-ului.
Vom vedea in urmatorul exemplu cum se poate optimiza aceasta solutie, prin utilizarea unui pool de thread-uri prealocate

Fire de executie (thread-uri): server web simplificat

Handler.java

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;

public class Handler {

    void process(Socket s) {
        DataInputStream  in = null;
        DataOutputStream out = null;
        try {
            in = new DataInputStream(s.getInputStream());
            out = new DataOutputStream(s.getOutputStream());
            int request = in.readInt();
            System.out.println("[SERVER HANDLER]: read from input stream: " + request);
            int result = -request;
            out.writeInt(result);
            System.out.println("[SERVER HANDLER]: write to output stream: " + result);

        }
        catch (IOException ex) {}
        finally {
            try { if (in != null)  in.close();
            } catch(IOException ignore) {};
            try { if (out != null) out.close();
            } catch(IOException ignore) {};
            try { s.close();
            } catch(IOException ignore) {System.out.println("[SERVER HANDLER] Close port: Error closing port");};
        }
    }
}
```

Metoda process a handlerului Handler citeste un intreg din fluxul de date de intrare deschis pe socket si scrie un intreg (egal cu opusul valorii citite);

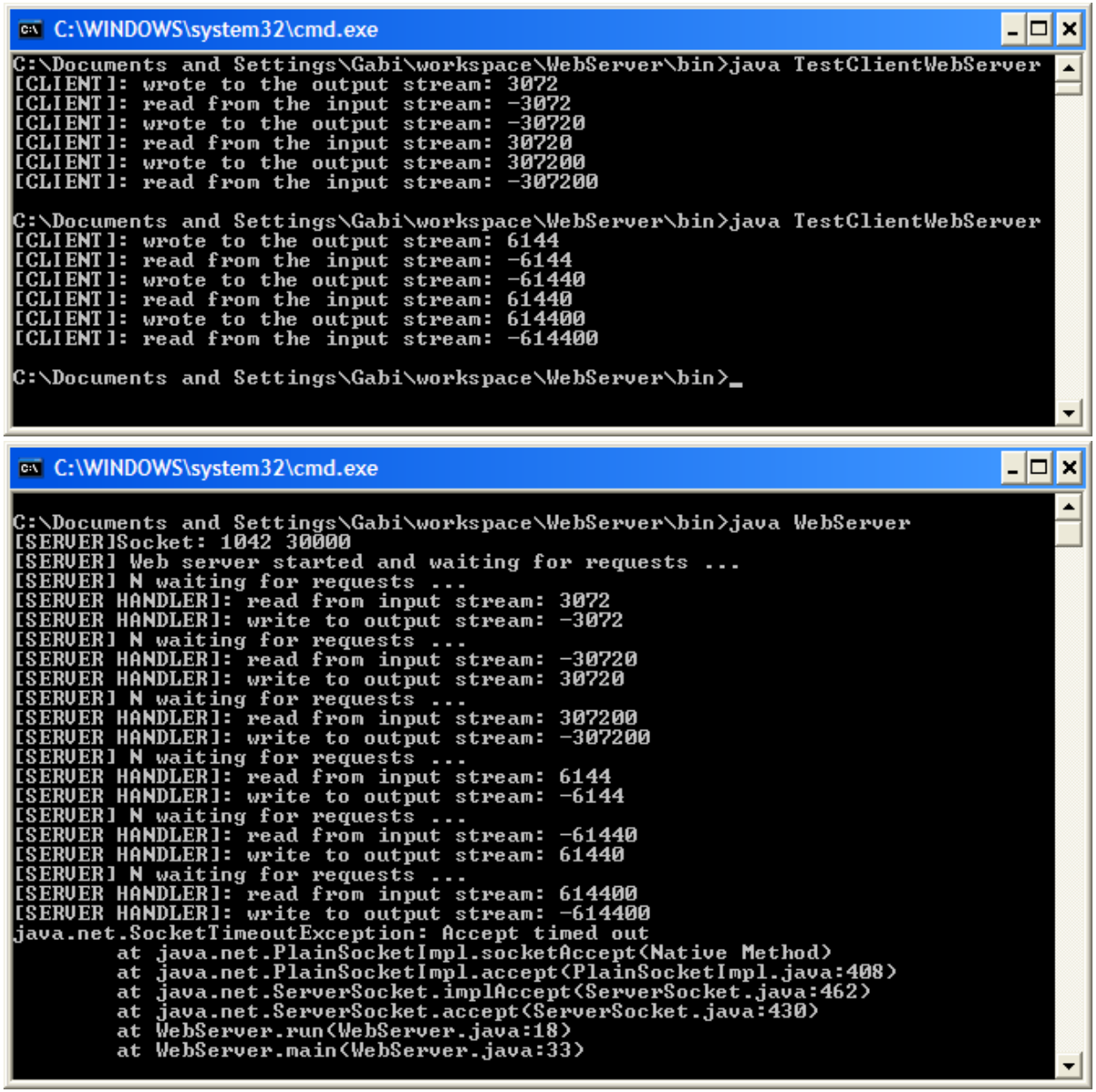
Fire de executie (thread-uri): server web simplificat

TestClientWebServer.java

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;


public class TestClientWebServer {

    static Socket socket = null;
    static DataInputStream  in = null;
    static DataOutputStream out = null;
    static int request = -1;
    static int response = -1;
    static final int PORT = 1042;
    static final int REQ = 1024;

    public static void main (String args[]) throws IOException {
        for(int i = 0; i < 3; i++) {
            socket = new Socket("localhost",PORT);
            try {
                in = new DataInputStream(socket.getInputStream());
                out = new DataOutputStream(socket.getOutputStream());
                if(response == -1) {
                    request = (REQ * (int)(10 * Math.random()));
                }
                else {
                    request = 10 * response;
                }
                out.writeInt(request);
                System.out.println("[CLIENT]: wrote to the output stream: " + request);
                response = in.readInt();
                System.out.println("[CLIENT]: read from the input stream: " + response);
            }
            catch (IOException ex) {}
            finally {
                try { if (in != null)  in.close();
                } catch(IOException ignore) {};
                try { if (out != null) out.close();
                } catch(IOException ignore) {};
                try { socket.close();
                } catch(IOException ignore) {System.out.println("[SERVER HANDLER] Close port: Error closing port");};
            }
        }


    }

}
```

Clientul trimite un intreg la server, scriind valoarea intr-un DataOutputStream care primeste ca parametru output Stream-ul socketului; citeste apoi raspunsul de la server, dintr-un DataInputStream care primeste ca parametru input Stream-ul socket-ului; il inmulteste cu 10 si il trimite din nou. Aceasta se repeta de 3 ori in total.

Fire de executie (thread-uri): server web simplificat



Dupa pornirea WebServer, se asteapta un request de la client; odata primit request-ul, se trimite un raspuns cu valoarea (-request); clientul citeste la randul sau acest raspuns, il inmulteste cu 10 si il trimite din nou. Aceasta se repeta de 3 ori in total. Cele doua console arata functionarea clientilor si serverului pentru 2 apeluri de la client succesive. Serverul va arunca o exceptie SocketTimeoutException dupa 30 de secunde.

Fire de executie (thread-uri): pool de threaduri

ThreadPool.java

```
abstract class JobQueueModel {
    abstract void put(Runnable r) throws InterruptedException;
    abstract Runnable get() throws InterruptedException;
}
interface Executor {
    void execute(Runnable r);
}
public class ThreadPool implements Executor {
    protected final JobQueueModel  jobQueue;
    public void execute(Runnable r) {
        try {
            jobQueue.put(r);
        }
        catch(InterruptedException ie) {
            //postpone response
            Thread.currentThread().interrupt();
        }
    }
    public ThreadPool(JobQueueModel queue, int nWorkers) {
        jobQueue = queue;
        for(int i = 0; i < nWorkers; i++) {
            activation();
        }
    }

    protected void activation() {
        Runnable rx = new Runnable() {
            public void run() {
                try {
                    while (true) {
                        Runnable r = (Runnable) (jobQueue.get());
                        r.run();
                    }
                }
                catch(InterruptedException ie) {} // ready
            }
        };
        new Thread(rx).start();
    }
}
```

Acest model de calcul este de tip “replicated workers”. Se creeaza un grup de thread-uri care stiu sa preia joburi dintr-o coada de joburi. Pentru introducerea joburilor in coada se utilizeaza metoda execute(). In functie de caracteristicile aplicatiei, pot exista diferite variante in functie de:

- Politica de organizare a cozii (pot fi job-uri ordonate dupa prioritati, pot sa fie luate intotdeauna primele joburi adaugate in coada …);
- Selectarea si eventual modificarea numarului de thread-uri active;
- Tratarea situatiei in care nu mai exista thread-uri disponibile;
- Tratarea situatiei in care exista thread-uri care nu au de lucru (starving);
- Modalitatea de tratare a erorilor;