

**aprile 2013**

13

Implementazione del costrutto Monitor

Sistemi operativi e programmazione distribuita

Matteo Tamburini

AA: 2012/2013

Matricola: 426878

Email: mattetambu@gmail.com

**Università di Pisa – Corso di Laurea Magistrale in Ingegneria Informatica**

Sommario

[Specifiche 1](#_Toc353297068)

[Specifiche di progetto 1](#_Toc353297069)

[Specifica dei requisiti 1](#_Toc353297070)

[Seconda parte 2](#_Toc353297071)

[Organizzazione dei files 3](#_Toc353297072)

[Cartelle e sottocartelle 3](#_Toc353297073)

[Packages 3](#_Toc353297074)

[Implementazione 4](#_Toc353297075)

[Classe Queue 4](#_Toc353297076)

[Classe Semaphore 5](#_Toc353297077)

[Classe FairLock 6](#_Toc353297078)

[Problemi di sincronizzazione 8](#_Toc353297079)

[Mailbox 8](#_Toc353297080)

[Classe Main 8](#_Toc353297081)

[Classe Mailbox 8](#_Toc353297082)

[Classe Sender 10](#_Toc353297083)

[Classe Receiver 11](#_Toc353297084)

[Output 11](#_Toc353297085)

[Realizzazione Java5 13](#_Toc353297086)

[Output 15](#_Toc353297087)

[Resource manager 16](#_Toc353297088)

[Classe Main 16](#_Toc353297089)

[Classe Resource\_Manager 16](#_Toc353297090)

[Classe Requester 18](#_Toc353297091)

[Output 18](#_Toc353297092)

[Realizzazione Java5 19](#_Toc353297093)

[Output 21](#_Toc353297094)

[File manager 22](#_Toc353297095)

[Classe Main 23](#_Toc353297096)

[Classe File\_Manager 23](#_Toc353297097)

[Classe Writer 25](#_Toc353297098)

[Classe Reader 26](#_Toc353297099)

[Output 26](#_Toc353297100)

[Realizzazione Java5 28](#_Toc353297101)

[Output 31](#_Toc353297102)

# Specifiche

## Specifiche di progetto

La prima parte del progetto consiste nella implementazione del meccanismo di sincronizzazione tipico dei monitor (mutua esclusione e variabili condition) che, con riferimento alla primitiva signal, utilizzi la semantica signal-and urgent.

Per l’implementazione del meccanismo devono essere utilizzati esclusivamente i costrutti di sincronizzazione offerti da Java 1.4.2 (e cioè i metodi e/o i blocchi synchronized e i metodi wait(), notify() e notifyAll() offerti dalla classe Object).

In altri termini, si chiede di implementare un meccanismo simile a quello direttamente offerto dalle versioni più recenti di java ma con una diversa semantica.

In particolare, il meccanismo direttamente offerto dalle versioni più recenti di java consente di utilizzare oggetti di tipo Lock per programmare esplicitamente la mutua esclusione fra le esecuzioni dei metodi di accesso ad un oggetto condiviso fra più threads. Inoltre, oggetti di tipo Lock consentono anche di dichiarare variabili di tipo Condition su cui eseguire le operazioni primitive c.await(), c.signal() e c.signalAll() usate rispettivamente per bloccare un thread sulla variabile Condition c, per svegliare un thread bloccato sulla variabile Condition c, o per svegliare tutti i thread sospesi su tale variabile. In questo caso però la semantica offerta da java è la signal-and-continue.

## Specifica dei requisiti

Nel progetto, si chiede di fornire un analogo meccanismo cambiando però la semantica offerta, secondo, le specifiche indicate di seguito.

REQUISITO N.1: implementare un meccanismo di mutua esclusione da utilizzare per programmare esplicitamente la mutua esclusione fra le esecuzioni dei metodi di accesso ad un oggetto condiviso fra più threads. Cioè, a differenza della mutua esclusione implicita offerta dai blocchi synchronized, la garanzia di mutua esclusione è demandata al programmatore e cioè programmata esplicitamente.

In particolare: implementare la classe FairLock che offra i due metodi, lock() ed unlock() da usare per programmare esplicitamente gli accessi esclusivi ad un oggetto condiviso.

REQUISITO N.2: a differenza del wait set di un oggetto (contenente i thread sospesi per mutua esclusione nell’accesso ad un blocco synchronized) che, come noto, non prevede nessun tipo di gestione, i thread che si sospendono nel tentativo di accedere ad un oggetto condiviso eseguendo il metodo mutex.lock() (dove mutex rappresenta un’istanza della classe FairLock utilizzata per garantire la mutua esclusione negli accessi all’oggetto condiviso) devono essere risvegliati in ordine FIFO.

REQUISITO N.3: implementare la classe Condition che offre i metodi await() e signal() usati, rispettivamente, per sospendere un thread su una variabile Condition e per risvegliare il primo che si è sospeso (se c’è) fra i thread presenti sulla variabile Condition. Al solito, se nessun thread è sospeso sulla variabile Condition, il metodo signal() non esegue nessuna operazione. Quindi la Condition deve fornire una coda FIFO di thread sospesi. La semantica della signal deve essere la signal-and-urgent. Per questo motivo, non è prevista l’implementazione del metodo signalAll().

REQUISITO N.4: Come noto dalla teoria dei monitor, una variabile Condition è sempre locale ad un monitor e le operazioni primitive await() e signal() possono essere quindi utilizzate solo all’interno dei metodi di accesso ad un monitor. Ciò implica che ogni istanza della classe Condition deve essere intrinsecamente legata ad un lock (oggetto della classe FairLock). Per questo motivo la classe FairLock deve offrire, oltre ai metodi lock() ed unlock() prima visti, anche il metodo newCondition(). In questo modo, se ad esempio mutex è un istanza della classe FairLock, l’operazione mutex.newCondition()restituisce un’istanza di Condition legata allo specifico oggetto mutex di tipo Lock.

## Seconda parte

Scegliere uno fra i vari problemi di sincronizzazione visti anche a lezione e che preveda soluzioni diverse a seconda della semantica della signal utilizzata.

Quindi, risolvere il problema utilizzando il meccanismo implementato nella prima parte del progetto.

Successivamente, risolvere lo stesso problema utilizzando però gli strumenti di sincronizzazione offerti dalle versioni più recenti di java (Lock , variabili Condition e i metodi await(), signal() e signalAll()) che utilizzano la semantica signal-and-urgent, e verificare come deve essere modificata la soluzione rispetto a quella che utilizza il meccanismo di sincronizzazione implementato nel progetto.

# Organizzazione dei files

## Cartelle e sottocartelle

I files sono organizzanti secondo la seguente struttura:

* la directory *Monitor* contiene i sorgenti delle classi principali del progetto, quelle necessarie per la realizzazione del costrutto Monitor, ovvero i files *Fairlock.java, Semaphore.java* e *Queue.java.*
* la directory *Examples* contiene gli esempi richiesti nella seconda parte del progetto. Questa cartella è composta da tre sotto-directories ognuna contenente i sorgenti di un dato esempio suddivisi in base all’utilizzo del codice della cartella *Monitor* o dei costrutti offerti da *Java5*.

La directory principale, *Source\_files*, che include le due directories *Monitor* e *Examples,* contiene tre files batch utili per semplificare le operazioni di compilazione ed esecuzione:

* *compile.bat*: esegue i comandi di compilazione per tutti gli esempi realizzati.
* *run.bat*: contiene i comandi di esecuzione dei vari esempi.
* *run\_whit\_redirect.bat*: contiene i comandi di esecuzione degli esempi eseguendo la redirezione dell’output risultante in appositi files di testo.

## Packages

Gli esempi realizzati fanno tutti riferimento ad un unico package chiamato *Examples* strutturato in un insieme di sotto-packages per distinguere i vari esempi e le relative implementazioni. Ogni esempio apparterrà quindi al package

*Examples.<tipo\_esempio>*

e per ognuno, le varie implementazioni apparterranno al package

*Examples.<tipo\_esempio>.<tipo\_implementazione>*

Le classi *Fairlock.java, Semaphore.java* e *Queue.java* sono racchiuse in un package chiamato *Monitor* e quindi il relativo codice può essere incluso negli esempi che ne fanno uso attraverso la stringa

import Monitor.\*;

# Implementazione

Di seguito si riporta il codice relativo alla realizzazione del costrutto monitor e delle classi di appoggio necessarie. E’ presente la classe Queue che implementa una coda di long-int, la classe Semaphore che definisce un semaforo FIFO e la classe FairLock che fornisce gli strumenti principali di un monitor sfruttando la semantica signal-and-urgent.

## Classe Queue

La classe Queue contenuta nel file Queue.java implementa una coda di long-int per la gestione dei PID dei threads. I metodi forniti dalla classe sono:

* *empty ():* restituisce **true** se la coda è vuota (**false** altrimenti).
* *first ():* restituisce il primo elemento della coda senza estrarlo dalla lista.
* *contains (long ID):* restituisce **true** se il PID passato come parametro è presente nella lista (**false** altrimenti).
* *insert(long ID):* inserisce il PID passato come paramentro in coda alla lista.
* *extract():* restituisce l’elemento in testa estraendolo dalla lista.

Il codice relativo risulta:

package Monitor;public class Queue { private class elem { private long info; private elem next; } private elem head = null; private elem tail = null; public Queue () { } public boolean empty () { return (head == null); } public long first () { return (!empty())? head.info: -1; } public boolean contains (long ID) { elem p = head; while (p != null && p.info != ID) p = p.next; return (p != null && p.info == ID); } public void insert (long ID) { elem p = new elem(); p.info = ID; p.next = null; if (empty()) head = tail = p; else tail = tail.next = p; } public long extract () { long ID = -1; if (!empty()) { ID = head.info; head = head.next; if (head == null) tail = null; } return ID; }}

## Classe Semaphore

La classe Semaphore contenuta nel file Semaphore.java implementa una semaforo che, sfruttando un’istanza della classe Queue, gestisce i thread in ordine FIFO. Il valore di inizializzazione del semaforo deve essere passato come parametro al costruttore della classe (e rappresenta il numero massimo di thread che possono eseguire consecutivamente il metodo *P()* senza bloccarsi). Questa classe fornisce gli strumenti di sincronizzazione di base che saranno sfruttati in tutto il progetto attraverso i metodi:

* *P ():* sospende l’esecuzione del thread attraverso la primitiva *wait()* se il semaforo è ‘rosso’ (contatore associato <= 0).
* *V (): risveglia un thread in attesa (se presente)*.

Il codice relativo risulta:

package Monitor;public class Semaphore { private Queue p\_suspended = new Queue(); private int p\_count; public Semaphore (int num) { p\_count = num; } public synchronized void P () throws InterruptedException { p\_count--; if (p\_count < 0) { p\_suspended.insert(Thread.currentThread().getId()); while(p\_suspended.contains(Thread.currentThread().getId())) wait(); } } public synchronized void V () { p\_count++;

if (p\_count <= 0) {

p\_suspended.extract();

notifyAll();

}

}

}

Usando la notifyAll() nel metodo *V()* si risvegliano tutti i thread bloccati sul semaforo attraverso la wait(), ma soltanto quello estratto dalla coda (il primo) potrà proseguire l’esecuzione. Questo meccanismo è necessario in quanto l’utilizzo della *notify()* porterebbe al risveglio di un thread facente parte del wait-set del semaforo potenzialmente diverso dal thread in testa alla lista *p\_suspended* (che quindi testando la condizione tornerebbe nuovamente a bloccarsi generando una situazione di deadlock).

## Classe FairLock

Per la realizzazione della classe FairLock secondo la semantica signal-and-urgent saranno necessari due semafori di tipo Semaphore: il primo, *mutex,* che rappresenta un semaforo di mutua esclusione, e il secondo, *urgent*, per garantire la priorità ai processi segnalanti (rispetto a tutti gli altri nella entry-queueu) una volta che il thread segnalato abbia terminato. Per rappresentare il fatto che una variabile Condition deve essere sempre locale ad un monitor, la classe Condition è stata programmata come interna alla classe Fairlock che ne permette l’istanziamento attraverso un apposito metodo. I metodi forniti dalla classe FairLock sono:

* *lock ():* permette di acquisire il lock su un oggetto (utilizzo esclusivo).
* *unlock ():* permette di rilasciare il lock su un oggetto.
* *newCondition():* restituisce un’istanza della classe Condition.

I metodi forniti dalla classe Condition risultano:

* *await ():* causa l’interruzione dell’esecuzione del thread che quindi si blocca sulla Condition e, in accordo alla semantica signal-and-urgent, risveglia l’eventuale thread nella coda urgent o rilascia la mutua esclusione.
* *signal ():* se è presente almeno un thread in attesa sulla Condition lo risveglia e pone il thread attualmente in esecuzione in attesa nella coda urgent.

Il codice relativo risulta:

package Monitor;

public class FairLock {

private int urgent\_count = 0;

private Semaphore mutex = new Semaphore(1);

private Semaphore urgent = new Semaphore(0);

public FairLock () {

}

public void lock () throws InterruptedException {

mutex.P();

}

public void unlock () {

if (urgent\_count > 0) urgent.V();

else mutex.V();

}

public Condition newCondition () {

return new Condition();

}

public class Condition {

private int cond\_count = 0;

private Semaphore cond\_sem = new Semaphore (0);

public Condition () {

}

public void await () throws InterruptedException {

cond\_count++;

if (urgent\_count > 0) urgent.V();

else mutex.V();

cond\_sem.P();

cond\_count--;

}

public void signal () throws InterruptedException {

if (cond\_count > 0) {

urgent\_count++;

cond\_sem.V();

urgent.P();

urgent\_count--;

}

}

public boolean empty () {

return (cond\_count == 0);

}

};

}

# Problemi di sincronizzazione

## Mailbox

Con il termine Mailbox si fa riferimento al problema dei produttori/consumatori, ovvero al problema della comunicazione fra thread mittenti e thread riceventi sfruttando un buffer di dimensioni limitate.

### Classe Main

Questa è la classe in cui viene creato un oggetto di tipo *Mailbox* in grado di contenere un massimo di 3 messaggi e una coppia di produttori (*Sender*) e consumatori (*Receiver*) che si scambiano un totale di 8 messaggi.

package Examples.Mailbox.Using\_Monitor;

public class Main {

public static void main (String[] args) {

Mailbox mailbox = new Mailbox(3);

Receiver r1 = new Receiver(mailbox, 4);

Sender s1 = new Sender(mailbox, 3);

Sender s2 = new Sender(mailbox, 5);

Receiver r2 = new Receiver(mailbox, 4);

}

};

### Classe Mailbox

Questa è la classe principale dell’esempio che regola l’interazione tra i produttori e i consumatori gestendo le sincronizzazioni e i messaggi scambiati. Attraverso un parametro intero passato al costruttore viene definita la dimensione della mailbox, ovvero il limite massimo di messaggi che può contenere. La mutua esclusione per l’accesso ai metodi della classe è garantita sfruttando i metodi forniti dalla classe *FairLock* della quale è presente un’istanza con l’attributo *mutex*. Due variabili *Condition, not\_full* e *not\_empty,* sono utilizzate rispettivamente per bloccare i produttori che tentano di inviare un messaggio quando la mailbox è completamente occupata e i consumatori quando tentano di estrarre un messaggio dalla mailbox vuota. Il codice relativo risulta:

package Examples.Mailbox.Using\_Monitor;

import Monitor.\*;

public class Mailbox {

private final int N;

private int [] buffer;

private int count = 0, head = 0, tail = 0;

private FairLock mutex = new FairLock();

private FairLock.Condition not\_full = mutex.newCondition();

private FairLock.Condition not\_empty = mutex.newCondition();

public Mailbox(int dim) {

N = dim;

buffer = new int[dim];

System.out.println(" -----------------------------------------------\n -- Created Mailbox --");

}

public void send (int message) throws InterruptedException {

try {

mutex.lock();

System.out.println(" Request to send the message " + message);

if (count == N) {

System.out.println(" Waiting for not-full buffer");

not\_full.await();

}

buffer[tail] = message;

tail = (tail + 1) % N;

count++;

System.out.println(" Message " + message + " sent");

not\_empty.signal();

}

finally {

mutex.unlock();

}

}

public int receive () throws InterruptedException {

int message;

try {

mutex.lock();

System.out.println(" Request for receiving a message");

if (count == 0) {

System.out.println(" Waiting for not-empty buffer");

not\_empty.await();

}

message = buffer[head];

head = (head + 1) % N;

count--;

System.out.println(" Message " + message + " received");

not\_full.signal();

}

finally {

mutex.unlock();

}

return message;

}

}

### Classe Sender

Questa classe rappresenta i produttori di dati che inviano messaggi alla mailbox sfuttando l’apposito metodo *mailbox.send().* Il costruttore della classe effettua l’associazione del produttore alla mailbox e, attraverso *start(),* avvia il thread richiamando il metodo *run().* Il numero di messaggi da inviare viene passato come parametro al costruttore. Viene simulata l’eleborazione dei dati da parte del thread attraverso la chiamata alla primitiva *sleep().*

package Examples.Mailbox.Using\_Monitor;

public class Sender extends Thread {

private long ID;

private int send\_times;

private Mailbox mailbox;

public Sender (Mailbox mailbox, int times) {

send\_times = times;

this.mailbox = mailbox;

System.out.println(" -- Created Sender -- ");

start();

}

public void run () {

ID = Thread.currentThread().getId();

try {

for (int i = 0; i < send\_times; i++) {

mailbox.send((int) ID\*(i+1));

sleep(100);

}

}

catch (Exception e) {

System.out.println("\n\t############## ERROR ##############\n\t" + e.toString() + "\n");

}

}

}

### Classe Receiver

Questa classe rappresenta i consumatori di dati che estraggono messaggi dalla mailbox sfuttando l’apposito metodo *mailbox.receive().* Essendo questa classe duale rispetto alla classe *Sender* valgono le stesse considerazioni già effettuate.

package Examples.Mailbox.Using\_Monitor;

public class Receiver extends Thread {

private int receive\_times;

private Mailbox mailbox;

public Receiver (Mailbox mailbox, int times) {

receive\_times = times;

this.mailbox = mailbox;

System.out.println(" -- Created Receiver -- ");

start();

}

public void run () {

int messagge;

try {

for (int i = 0; i < receive\_times; i++) {

messagge = mailbox.receive();

sleep(100);

}

}

catch (Exception e) {

System.out.println("\n\t############## ERROR ##############\n\t" + e.toString() + "\n");

}

}

}

### Output

L’output di questo esempio è composto da un insieme di stringhe che permettono di verificare, tramite ispezione visiva, la correttezza della soluzione proposta. Le stringhe fornite in output per descrivere le principali interazioni risultano:

* *Request for receiving a message:* inizio dell’esecuzione del metodo *receive* da parte di un consumatore a seguito dell’acquisizione del lock sulla mailbox.
* *Waiting for not-empty buffer:* buffer vuoto e conseguente interruzione dell’esecuzione del metodo *receive*.
* *Message <messagge> received*: esecuzione del metodo *receive* completata con successo.
* *Request to send the message <messagge>:* inizio dell’esecuzione del metodo *send* da parte di un produttore a seguito dell’acquisizione del lock sulla mailbox.
* *Waiting for not-full buffer:* buffer pieno e conseguente interruzione dell’esecuzione del metodo *send*.
* *Message <messagge> sent*: esecuzione del metodo *sent* completata con successo.

L’output ottenuto dall’esecuzione della classe *Main* risulta:

-----------------------------------------------

-- Created Mailbox --

-- Created Receiver --

Request for receiving a message

Waiting for not-empty buffer

-- Created Sender --

-- Created Sender --

-- Created Receiver --

Request to send the message 10

Message 10 sent

Message 10 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 11

Message 11 sent

Message 11 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 22

Message 22 sent

Message 22 received

Request to send the message 20

Message 20 sent

Request for receiving a message

Message 20 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 30

Message 30 sent

Message 30 received

Request to send the message 33

Message 33 sent

Request for receiving a message

Message 33 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 44

Message 44 sent

Message 44 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 55

Message 55 sent

Message 55 received

### Realizzazione Java5

La realizzazione della Mailbox sfruttando i costrutti offerti da Java5 risulta praticamente identica alla soluzione precedentemente analizzata, in cui le classi *Main,* *Sender* e *Receiver* non subiscono alcuna modifica (in realtà la classe *Main* viene modificata in modo tale da generare un solo produttore e due consumatori per semplificare la fase di analisi dei risultati) mentre la classe *Mailbox* deve essere aggiornata per l’utilizzo delle classi *Lock*  e *Condition* presenti nel package *java.util.cuncurrent.locks*. Il codice che ne deriva risulta:

package Examples.Mailbox.Using\_Java5;

import java.util.concurrent.locks.\*;

public class Mailbox {

…

private Lock mutex = new ReentrantLock();

private Condition not\_full = mutex.newCondition();

private Condition not\_empty = mutex.newCondition();

…

}

L’output che si ottiene considerando questa soluzione risulta però (sfruttando un solo produttore e due consumatori che si scambiano 8 messaggi):

-----------------------------------------------

-- Created Mailbox --

-- Created Receiver --

-- Created Receiver --

Request for receiving a message

Waiting for not-empty buffer

Request for receiving a message

Waiting for not-empty buffer

-- Created Sender --

Request to send the message 11

Message 11 sent

Message 11 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 22

Message 22 sent

**Message 22 received**

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 33

Message 33 sent

**Message 33 received**

Request to send the message 44

Message 44 sent

Request for receiving a message

**Message 44 received**

**Message 22 received**

Request for receiving a message

**Message 33 received**

Request for receiving a message

**Message 44 received**

Request to send the message 55

Message 55 sent

Request to send the message 66

Message 66 sent

Request for receiving a message

Message 55 received

Request to send the message 77

Message 77 sent

Request to send the message 88

Message 88 sent

Come si evince dalle parti in grassetto nell’output l’esecuzione non è avvenuta in modo corretto, si nota infatti che i messaggi *22, 33* e *44* vengono ricevuti dai consumatori due volte ciascuno e questo è causato dalle continue letture dei consumatori (accettate in modo scorretto) sul buffer circolare che portano a leggere ripetutamente gli stessi messaggi. Questo comportamento anomalo è dovuto alla differenza di semantica utilizzata negli oggetti di tipo *Lock* e *Condition* rispetto agli oggetti *FairLock e Condition* presenti nel package *Monitor*: si passa infatti dall’utilizzo della semantica *signal-and-urgent* alla semantica *signal-and-continue* che non garantisce che la condizione di sincronizzazione valida alla riattivazione di un thread risulti ancora valida quando questo passi all’esecuzione. Tutto ciò si traduce nella necessità di sostituire i costrutti condizionali *if-else* per il test delle condizioni di sincronizzazione in costrutti ciclici *while-then* ottenendo:

package Examples.Mailbox.Using\_Java5;

import java.util.concurrent.locks.\*;

public class Mailbox {

…

private Lock mutex = new ReentrantLock();

private Condition not\_full = mutex.newCondition();

private Condition not\_empty = mutex.newCondition();

…

public void send (int message) throws InterruptedException {

…

while (count == N) {

System.out.println(" Waiting for not-full buffer");

not\_full.await();

}

…

}

public int receive () throws InterruptedException {

…

while (count == 0) {

System.out.println(" Waiting for not-empty buffer");

not\_empty.await();

}

...

}

}

#### Output

L’output ottenuto a seguito della correzione risulta:

-----------------------------------------------

-- Created Mailbox --

-- Created Receiver --

Request for receiving a message

Waiting for not-empty buffer

-- Created Sender --

-- Created Sender --

-- Created Receiver --

Request to send the message 10

Message 10 sent

Message 10 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 11

Message 11 sent

Message 11 received

Request to send the message 20

Message 20 sent

Request for receiving a message

Message 20 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 22

Message 22 sent

Message 22 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 30

Message 30 sent

Message 30 received

Request to send the message 33

Message 33 sent

Request for receiving a message

Message 33 received

Request to send the message 44

Message 44 sent

Request for receiving a message

Message 44 received

Request for receiving a message

Waiting for not-empty buffer

Request to send the message 55

Message 55 sent

Message 55 received

## Resource manager

Il Resource\_Manager è un gestore di risorse il cui scopo è risolvere il problema della condivisione di un pool limitato di risorse equivalenti fra più threads che concorrono per l’utilizzo di queste.

### Classe Main

Questa è la classe in cui viene creato un oggetto di tipo *Resource\_Manager,* che gestisce un pool di 2 risorse equivalenti, e un insieme di 3 utilizzatori (*Requester*) che condividono l’utilizzo delle risorse offerte.

package Examples.Resource\_Manager.Using\_Monitor;

public class Main {

public static void main (String[] args) {

Resource\_Manager resource\_manager = new Resource\_Manager(2);

Requester r1 = new Requester(resource\_manager, 3);

Requester r3 = new Requester(resource\_manager, 5);

Requester r2 = new Requester(resource\_manager, 4);

}

};

### Classe Resource\_Manager

Questa è la classe principale dell’esempio che regola l’accesso alle risorse da parte degli utilizzatori che possono richiedere e rilasciare una risorsa attraverso i metodi *request()* e *release().* La variabile *Condition resourse\_avaiable* èutilizzata per bloccare i *Requester* che richiedono di utilizzare una risorsa quando non ne è disponibile alcuna. Il codice risulta:

package Examples.Resource\_Manager.Using\_Monitor;

import Monitor.\*;

public class Resource\_Manager {

private final int N;

private boolean [] available;

private int available\_count;

private FairLock mutex = new FairLock();

private FairLock.Condition resourse\_available = mutex.newCondition();

public Resource\_Manager(int dim) {

N = available\_count = dim;

available = new boolean[N];

for (int i = 0; i < N; i++) available[i] = true;

System.out.println(" -----------------------------------------------\n -- Created Resource\_Manager --");

}

public int request () throws InterruptedException {

int i = 0;

try {

mutex.lock();

System.out.println(" Request for using a resource");

if (available\_count == 0) {

System.out.println(" Waiting for an available resource");

resourse\_available.await();

}

while (!available[i]) i++;

available[i] = false;

available\_count--;

System.out.println(" Resource number " + i + " now busy");

}

finally {

mutex.unlock();

}

return i;

}

public void release (int i) throws InterruptedException {

try {

mutex.lock();

available[i] = true;

available\_count++;

System.out.println(" Resource number " + i + " now available");

resourse\_available.signal();

}

finally {

mutex.unlock();

}

}

}

### Classe Requester

Questa classe rappresenta i thread che effettuano richieste di utilizzo delle risorse al *Resource\_manager,* e il numero di richieste da effettuare viene passato come parametro al costruttore.

package Examples.Resource\_Manager.Using\_Monitor;

public class Requester extends Thread {

private int request\_times;

private Resource\_Manager resource\_manager;

public Requester (Resource\_Manager resource\_manager, int times) {

request\_times = times;

this.resource\_manager = resource\_manager;

System.out.println(" -- Created Requester -- ");

start();

}

public void run () {

int num;

try {

for (int i = 0; i < request\_times; i++) {

num = resource\_manager.request();

sleep(100);

resource\_manager.release(num);

}

}

catch (Exception e) {

System.out.println("\n\t############## ERROR ##############\n\t" + e.toString() + "\n");

}

}

}

### Output

Le stringhe fornite in output per descrivere le principali interazioni risultano:

* *Request for using a resource:* inizio dell’esecuzione del metodo *request* da parte di un richiedente a seguito dell’acquisizione del lock sulla mailbox.
* *Waiting for an available resource:* nessuna risorsa disponibile e conseguente interruzione dell’esecuzione del metodo *request*.
* *Resource number <resource> now busy*: esecuzione del metodo *request* completata con successo e risorsa allocata al richiedente.
* *Resource number <resource> now available:* esecuzione del metodo *release* completata con successo e risorsa deallocata correttamente.

L’output ottenuto dall’esecuzione della classe *Main* (con un totale di 12 richieste) risulta:

-----------------------------------------------

-- Created Resource\_Manager --

-- Created Requester --

-- Created Requester --

-- Created Requester --

Request for using a resource

Resource number 0 now busy

Request for using a resource

Resource number 1 now busy

Request for using a resource

Waiting for an available resource

Resource number 1 now available

Resource number 1 now busy

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

Request for using a resource

Waiting for an available resource

Resource number 1 now available

Resource number 1 now busy

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

Request for using a resource

Waiting for an available resource

Resource number 1 now available

Resource number 1 now busy

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

Request for using a resource

Waiting for an available resource

Resource number 0 now available

Resource number 0 now busy

Resource number 1 now available

Request for using a resource

Resource number 1 now busy

Resource number 1 now available

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

Request for using a resource

Resource number 1 now busy

Resource number 1 now available

Resource number 0 now available

### Realizzazione Java5

Come già visto per la classe *Mailbox*, le classi *Main* e *Requester* non subiscono alcuna modifica mentre la classe *Resource\_Manager* deve essere adattata all’utilizzo delle classi *Lock*  e *Condition* presenti nel package *java.util.cuncurrent.locks*. Il codice che ne deriva risulta:

package Examples.Resource\_Manager.Using\_Java5;

import java.util.concurrent.locks.\*;

public class Resource\_Manager {

…

private Lock mutex = new ReentrantLock();

private Condition resourse\_available = mutex.newCondition();

…

}

L’output che si ottiene considerando questa soluzione risulta però scorretto, ovvero:

-----------------------------------------------

-- Created Resource\_Manager --

-- Created Requester --

-- Created Requester --

Request for using a resource

-- Created Requester --

Resource number 0 now busy

Request for using a resource

Resource number 1 now busy

Request for using a resource

Waiting for an available resource

Resource number 1 now available

Request for using a resource

Resource number 1 now busy

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

Resource number 1 now available

Request for using a resource

Resource number 1 now busy

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

**############## ERROR ##############**

**java.lang.ArrayIndexOutOfBoundsException: 2**

Resource number 1 now available

Request for using a resource

Resource number 1 now busy

Resource number 0 now available

Resource number 1 now available

Request for using a resource

Resource number 0 now busy

Resource number 0 now available

Come si evince dalle parti in grassetto nell’output l’esecuzione non è avvenuta in modo corretto, si nota infatti che viene generata un eccezione di *OutOfBounds* nell’accesso al vettore *available[]* dovuto ad un’accettazione scorretta di una richiesta di allocazione da parte del gestore. Questo comportamento anomalo è ancora dovuto alla differenza di semantica utilizzata negli oggetti di tipo *Lock* e *Condition* rispetto agli oggetti *FairLock e Condition* presenti nel package *Monitor.* Eseguendo la stessa correzione operata nel caso della *Mailbox*, sostituiamo i costrutti condizionali *if-else* per il test delle condizioni di sincronizzazione con costrutti ciclici *while-then* ottenendo:

package Examples.Resource\_Manager.Using\_Java5;

import java.util.concurrent.locks.\*;

public class Resource\_Manager {

…

private Lock mutex = new ReentrantLock();

private Condition resourse\_available = mutex.newCondition();

…

public int request () throws InterruptedException {

…

while (available\_count == 0) {

System.out.println(" Waiting for an available resource");

resourse\_available.await();

}

…

}

…

}

#### Output

L’output ottenuto a seguito della correzione risulta:

-----------------------------------------------

-- Created Resource\_Manager --

-- Created Requester --

-- Created Requester --

-- Created Requester --

Request for using a resource

Resource number 0 now busy

Request for using a resource

Resource number 1 now busy

Request for using a resource

Waiting for an available resource

Resource number 1 now available

Request for using a resource

Resource number 1 now busy

Waiting for an available resource

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

Waiting for an available resource

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

Resource number 1 now available

Request for using a resource

Resource number 1 now busy

Waiting for an available resource

Resource number 0 now available

Resource number 1 now available

Request for using a resource

Resource number 0 now busy

Resource number 1 now busy

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

Resource number 1 now available

Request for using a resource

Resource number 1 now busy

Resource number 0 now available

Resource number 1 now available

Request for using a resource

Resource number 0 now busy

Resource number 0 now available

Request for using a resource

Resource number 0 now busy

Resource number 0 now available

## File manager

Il File\_Manager è un gestore di file il cui scopo è risolvere il problema dell’accesso condiviso ad un file da parte di più lettori e scrittori evitando condizioni di starvation.

Considerando le interazioni di thread lettori con il file, risulta evidente che questi possano accedervi concorrentemente senza generare condizioni di errore in quanto tali accessi non comportano alcuna alterazione dello stato della risorsa. I thread scrittori, invece, dovranno accedere al file in mutua esclusione in quanto le operazioni di scrittura comportano modifiche della risorsa. Tutto ciò si traduce nella necessità di imporre una politica di accesso che privilegi prima l’una poi l’altra parte per evitare condizioni di attesa senza fine: se il file è occupato da uno scrittore si offre la priorità ai thread lettori per il successivo accesso, mentre se sono in corso operazioni di lettura la priorità per la prossima operazione viene associata ad una scrittura (un thread lettore non potrà iniziare una lettura in presenza di letture attive se è presente almeno uno scrittore in attesa).

### Classe Main

Questa è la classe in cui viene creato un oggetto di tipo *File\_Manager,* che gestisce le richieste di due scrittori e altrettanti lettori che condividono lo stesso file.

package Examples.File\_Manager.Using\_Monitor;

public class Main {

public static void main (String[] args) {

File\_Manager file\_manager = new File\_Manager();

Reader r1 = new Reader(file\_manager, 5);

Reader r2 = new Reader(file\_manager, 4);

Writer w1 = new Writer(file\_manager, 2);

Writer w2 = new Writer(file\_manager, 3);

}

};

### Classe File\_Manager

Questa è la classe principale dell’esempio che regola gli accessi al file accettando o meno le richieste che provengono dai lettori, attraverso l’invocazione del metodo *start\_reading()*, e degli scrittori, attraverso l’invocazione del metodo *start\_writing()*. Le variabili *Condition ok\_readers* e *ok\_writers* sono utilizzate rispettivamente per bloccare i lettori e gli scrittori quando questi effettuano richieste che non possano essere soddisfatte. Il codice risulta:

package Examples.File\_Manager.Using\_Monitor;

import Monitor.\*;

public class File\_Manager {

private int active\_readers = 0;

private boolean active\_writer = false;

private FairLock mutex = new FairLock();

private FairLock.Condition ok\_readers = mutex.newCondition();

private FairLock.Condition ok\_writer = mutex.newCondition();

public File\_Manager () {

System.out.println(" -----------------------------------------------\n -- Created File\_Manager --");

}

public void start\_reading () throws InterruptedException {

try {

mutex.lock();

System.out.println(" Read request");

if (active\_writer || !ok\_writer.empty()) {

System.out.println(" Waiting for readers turn");

ok\_readers.await();

}

System.out.println(" Start reading");

active\_readers++;

ok\_readers.signal();

}

finally {

mutex.unlock();

}

}

public void end\_reading () throws InterruptedException {

try {

mutex.lock();

active\_readers--;

System.out.println(" End reading - " + active\_readers + " readers are still reading");

if (active\_readers == 0) ok\_writer.signal();

}

finally {

mutex.unlock();

}

}

public void start\_writing () throws InterruptedException {

try {

mutex.lock();

System.out.println(" Write request");

if (active\_writer || active\_readers > 0) {

System.out.println(" Waiting for my turn");

ok\_writer.await();

}

System.out.println(" Start writing");

active\_writer = true;

}

finally {

mutex.unlock();

}

}

public void end\_writing () throws InterruptedException {

try {

mutex.lock();

System.out.println(" End writing");

active\_writer = false;

if (!ok\_readers.empty()) ok\_readers.signal();

else ok\_writer.signal();

}

finally {

mutex.unlock();

}

}

}

### Classe Writer

Questa classe definisce gli attori che effettuano accessi in scrittura al file. Questi richiedono al *File\_Manager* il permesso di accesso al file attraverso l’apposito metodo *file\_manager.start\_writing()* e, una volta terminate le operazioni di modifica (simulate attraverso la chiamata alla primitiva *sleep()*)*,* liberano il file attraverso l’invocazione del metodo *file\_manager.end\_writing()*. Il numero di scritture da effettuare viene passato come parametro al costruttore.

package Examples.File\_Manager.Using\_Monitor;

public class Writer extends Thread {

private int write\_times;

private File\_Manager file\_manager;

public Writer (File\_Manager file\_manager, int times) {

write\_times = times;

this.file\_manager = file\_manager;

System.out.println(" -- Created Writer -- ");

start();

}

public void run () {

try {

for (int i = 0; i < write\_times; i++) {

file\_manager.start\_writing();

sleep(100);

file\_manager.end\_writing();

}

}

catch (Exception e) {

System.out.println("\n\t############## ERROR ##############\n\t" + e.toString() + "\n");

}

}

}

### Classe Reader

Questa classe rappresenta gli attori che effettuano accessi in lettura al file. Questi richiedono al *File\_Manager* il permesso per iniziare la lettura attraverso l’apposito metodo *file\_manager.start\_reading()* e segnalano l’avvenuta conclusione delle operazioni attraverso l’invocazione del metodo *file\_manager.end\_reading()*. Il numero di letture da effettuare viene passato come parametro al costruttore.

package Examples.File\_Manager.Using\_Monitor;

public class Reader extends Thread {

private int read\_times;

private File\_Manager file\_manager;

public Reader (File\_Manager file\_manager, int times) {

read\_times = times;

this.file\_manager = file\_manager;

System.out.println(" -- Created Reader -- ");

start();

}

public void run () {

try {

for (int i = 0; i < read\_times; i++) {

file\_manager.start\_reading();

sleep(100);

file\_manager.end\_reading();

}

}

catch (Exception e) {

System.out.println("\n\t############## ERROR ##############\n\t" + e.toString() + "\n");

}

}

}

### Output

Le stringhe fornite in output per descrivere le principali interazioni risultano:

* *Read request:* inizio dell’esecuzione del metodo *start\_reading* da parte di un lettore a seguito dell’acquisizione del lock sull’oggetto.
* *Waiting for readers turn:* accesso al file negato (perché una scrittura è in atto o in quanto esiste almeno uno scrittore in attesa) con conseguente interruzione dell’esecuzione del metodo *start\_reading*.
* *Start reading*: esecuzione del metodo *start\_reading* completata con successo e inizio lettura del file da parte del thread richiedente.
* *End reading - <active\_readers> readers are still reading:* invocazione del metodo *end\_reading* da parte di un lettore che ha completato il suo accesso al file. Viene inoltre segnalato il numero di lettori ancora attivi che utilizzano la risorsa.
* *Write request:* inizio dell’esecuzione del metodo *start\_writing* da parte di uno scrittore a seguito dell’acquisizione del lock sull’oggetto.
* *Waiting for my turn:* accesso al file negato (perché è in atto una lettura o una scrittura) con conseguente interruzione dell’esecuzione del metodo *start\_writing*.
* *Start writing*: esecuzione del metodo *start\_writing* completata con successo e inizio scrittura del file da parte del thread richiedente.
* *End writing:* invocazione del metodo *end\_writing* da parte di un scrittore che ha completato il suo accesso al file.

L’output ottenuto dall’esecuzione della classe *Main* (con un totale di 12 richieste) risulta:

-----------------------------------------------

-- Created File\_Manager --

-- Created Reader --

-- Created Reader --

Read request

Start reading

Read request

Start reading

-- Created Writer --

-- Created Writer --

Write request

Waiting for my turn

Write request

Waiting for my turn

End reading - 1 readers are still reading

End reading - 0 readers are still reading

Start writing

Read request

Waiting for readers turn

Read request

Waiting for readers turn

End writing

Start reading

Start reading

Write request

Waiting for my turn

End reading - 1 readers are still reading

End reading - 0 readers are still reading

Start writing

Read request

Waiting for readers turn

Read request

Waiting for readers turn

End writing

Start reading

Start reading

Write request

Waiting for my turn

End reading - 1 readers are still reading

End reading - 0 readers are still reading

Start writing

Read request

Waiting for readers turn

Read request

Waiting for readers turn

End writing

Start reading

Start reading

End reading - 1 readers are still reading

End reading - 0 readers are still reading

Start writing

Read request

Waiting for readers turn

End writing

Start reading

Write request

Waiting for my turn

End reading - 0 readers are still reading

Start writing

End writing

### Realizzazione Java5

Le classi *Main* e *Reader* e *Writer* non subiscono alcuna modifica mentre la classe *File\_Manager* deve essere adattata all’utilizzo delle classi *Lock* e *Condition* presenti nel package *java.util.cuncurrent.locks*. Tale modifica comporta la necessità di sostituire il metodo *empty() (*fornito dalla classe *FairLock.Condition* e utilizzato per verificare l’assenza di thread in attesa sulla variabile *Condition*) con il metodo *boolean hasWaiters()* offerto dalla classe *ReentrantLock* e definito con semantica opposta rispetto al metodo *empty()*.

Consideriamo inoltre la differenza di semantica offerta dai costrutti *Java5* e operiamo la sostituzione, già operata per correggere i precedenti due esempi, dei costrutti condizionali *if-else* per il test delle condizioni di sincronizzazione con costrutti ciclici *while-then.* Il codice che ne deriva risulta:

package Examples.File\_Manager.Using\_Java5;

import java.util.concurrent.locks.\*;

public class File\_Manager {

…

private ReentrantLock mutex = new ReentrantLock();

private Condition ok\_readers = mutex.newCondition();

private Condition ok\_writer = mutex.newCondition();

…

public void start\_reading () throws InterruptedException {

…

while (active\_writer || mutex.hasWaiters(ok\_writer)) {

System.out.println(" Waiting for readers turn");

ok\_readers.await();

}

…

}

public void end\_reading () throws InterruptedException {

…

}

public void start\_writing () throws InterruptedException {

…

while (active\_writer || active\_readers > 0) {

System.out.println(" Waiting for my turn");

ok\_writer.await();

}

…

}

public void end\_writing () throws InterruptedException {

…

}

}

Questa soluzione risulta però scorretta infatti tentando l’esecuzione si giunge ad uno stato di deadlock, ovvero:

-----------------------------------------------

-- Created File\_Manager --

-- Created Reader --

-- Created Reader --

Read request

Start reading

Read request

Start reading

-- Created Writer --

-- Created Writer --

Write request

Waiting for my turn

**Write request**

**Waiting for my turn**

End reading - 1 readers are still reading

Read request

Waiting for readers turn

End reading - 0 readers are still reading

Read request

Waiting for readers turn

Start writing

End writing

**Waiting for readers turn**

Write request

Start writing

End writing

**Waiting for readers turn**

STOPPED

Analizzando l’output si nota che:

* vengono completate un totale di due scritture che appartengono entrambe al *Writer w1* (si può verificare inserendo in ogni output il *PID* del thread che lo genera) che così conclude la sua esecuzione.
* lo scrittore *w2,* effettuando una richiesta di scrittura, si pone in attesa trovando il file occupato (*Waiting for my turn* evidenziato in grassetto).
* entrambi i lettori, eseguendo una richiesta di lettura, sono costretti a bloccarsi a causa dello scrittore *w2* in attesa (*Waiting for readers turn* evidenziato in grassetto).

L’esecuzione, quindi, non potrà mai proseguire in quanto tutti i thread non ancora terminati sono bloccati. Questa situazione di deadlock è causata da una errata definizione delle condizioni di sincronizzazione, che causa il bloccaggio dei lettori risvegliati anche quando la loro esecuzione dovrebbe proseguire. La condizione di sincronizzazione per i lettori deve essere quindi modificata come segue:

package Examples.File\_Manager.Using\_Java5;

import java.util.concurrent.locks.\*;

public class File\_Manager {

…

public void start\_reading () throws InterruptedException {

…

if (active\_writer || mutex.hasWaiters(ok\_writer)) {

do {

System.out.println(" Waiting for readers turn");

ok\_readers.await();

} while (active\_writer);

}

…

}

…

}

#### Output

L’output ottenuto a seguito della correzione risulta:

-----------------------------------------------

-- Created File\_Manager --

-- Created Reader --

-- Created Reader --

Read request

Start reading

Read request

Start reading

-- Created Writer --

-- Created Writer --

Write request

Waiting for my turn

Write request

Waiting for my turn

End reading - 1 readers are still reading

Read request

Waiting for readers turn

End reading - 0 readers are still reading

Read request

Waiting for readers turn

Start writing

End writing

Write request

Start writing

Waiting for readers turn

End writing

Start reading

Start reading

End reading - 1 readers are still reading

Read request

Waiting for readers turn

End reading - 0 readers are still reading

Read request

Start reading

Waiting for my turn

Start reading

End reading - 1 readers are still reading

Read request

Waiting for readers turn

End reading - 0 readers are still reading

Read request

Start reading

Waiting for my turn

Start reading

End reading - 1 readers are still reading

End reading - 0 readers are still reading

Read request

Start reading

Waiting for my turn

End reading - 0 readers are still reading

Start writing

End writing

Write request

Start writing

End writing

Write request

Start writing

End writing