

**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](#_Toc353030193)

[Specifiche di progetto 1](#_Toc353030194)

[Specifica dei requisiti 1](#_Toc353030195)

[Organizzazione dei files 3](#_Toc353030196)

[Cartelle e sottocartelle 3](#_Toc353030197)

[Packages 3](#_Toc353030198)

[Implementazione 4](#_Toc353030199)

[Classe FairLock 4](#_Toc353030200)

[Classe Semaphore 5](#_Toc353030201)

[Classe Queue 4](#_Toc353030202)

[Problemi di sincronizzazione 6](#_Toc353030203)

[Mailbox 6](#_Toc353030204)

[Classe Main 6](#_Toc353030205)

[Classe Mailbox 6](#_Toc353030206)

[Classe Sender 6](#_Toc353030207)

[Classe Receiver 6](#_Toc353030208)

[Output 6](#_Toc353030209)

[Realizzazione Java5 6](#_Toc353030210)

[Output 6](#_Toc353030211)

[Resource manager 7](#_Toc353030212)

[Classe Main 7](#_Toc353030213)

[Classe Resource\_Manager 7](#_Toc353030214)

[Classe Requester 7](#_Toc353030215)

[Output 7](#_Toc353030216)

[Realizzazione Java5 7](#_Toc353030217)

[Output 7](#_Toc353030218)

[File manager 8](#_Toc353030219)

[Classe Main 8](#_Toc353030220)

[Classe File\_Manager 8](#_Toc353030221)

[Classe Writer 8](#_Toc353030222)

[Classe Reader 8](#_Toc353030223)

[Output 8](#_Toc353030224)

[Realizzazione Java5 8](#_Toc353030225)

[Output 8](#_Toc353030226)

# 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

## Packages

Corpo del testo*.*

# 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

Corpo del testo*.*

### Classe Main

Corpo del testo.

### Classe Mailbox

Corpo del testo.

### Classe Sender

Corpo del testo.

### Classe Receiver

Corpo del testo.

### Output

Corpo del testo.

### Realizzazione Java5

Corpo del testo.

#### Output

Corpo del testo.

## Resource manager

Corpo del testo*.*

### Classe Main

Corpo del testo.

### Classe Resource\_Manager

Corpo del testo.

### Classe Requester

Corpo del testo.

### Output

Corpo del testo.

### Realizzazione Java5

Corpo del testo.

#### Output

Corpo del testo.

## File manager

Corpo del testo*.*

### Classe Main

Corpo del testo.

### Classe File\_Manager

Corpo del testo.

### Classe Writer

Corpo del testo.

### Classe Reader

Corpo del testo.

### Output

Corpo del testo.

### Realizzazione Java5

Corpo del testo.

#### Output

Corpo del testo.

Corpo del testo.

Codice

Corpo del testo.

Codice2

Corpo del testo.

Codice3

Corpo del testo.

Codice Latex

Corpo del testo.