

La Programmazione MT safe in Java

La diffusione di Java come linguaggio/piattaforma enterprise per lo sviluppo di applicazioni e servizi anche server-side ha reso centrale il tema della programmazione concorrente in Java.

Il tema della concorrenza è presente in tanti componenti della piattaforma e dei frameworks Java, per esempio:

Servlets e JavaServer Pages. Le Servlet spesso richiedono accesso a delle informazioni di stato condivise con altre servlets.

Nella programmazione **RMI** differenti threads attivati sul lato dell'oggetto server remoto possono avere problemi di concorrenza.

Swing e AWT sono tipicamente single threaded (per semplificare problemi di concorrenza) ma ci sono tantissime problematiche relative alla concorrenza da gestire.

La programmazione concorrente in Java - 1

Costrutti per la sincronizzazione in Java

Java ha molti meccanismi di sincronizzazione, a differenti livelli di astrazione.

Alcuni costrutti di base (**synchronized**, **wait()**, **notify()** ...,) sono presenti dalla nascita del linguaggio.

Il package **java.util.concurrent.atomic** contiene classi con le relative operazioni che sono garantite essere atomiche.

Successivamente, a partire da Java versione 5 (2004), sono state introdotte le “**concurrency utilities**”, a un livello di astrazione più alto per migliorare: *facilità di uso, prestazioni, affidabilità, produttività, leggibilità (e quindi maintenance) del codice*.

La programmazione concorrente in Java - 2

Interferenza tra threads: corsa critica nel bank account (1/7)

Ricordiamo l'interferenza nel caso del “bank account” (in cui due threads devono effettuare 30 versamenti di 100€ sullo stesso conto corrente).

```
import java.lang.*;
public class ProblemaConcorrenza
{
    public static void main(String argv[])
    {
        BankAccount ba = new BankAccount(0);
        Employee e1 = new Employee("Mason", ba);
        Employee e2 = new Employee("Steinberg", ba);
        e1.start();
        e2.start();
    }
}
```

La programmazione concorrente in Java - 3

Interferenza tra threads: corsa critica nel bank account (2/7)

I threads degli impiegati che eseguono i versamenti:

```
class Employee extends Thread {
    private String _name; private BankAccount _conto;
    private static final int NUM_OF_DEPOSITS = 30;
    private static final int AMOUNT_PER_DEPOSIT = 100;

    Employee(String name, BankAccount conto)
    { _name = name; _conto = conto; }

    public void run() {
        try{
            for (int i = 1; i <= NUM_OF_DEPOSITS; ++i) {
                Thread.sleep(500);
                _conto.increase(AMOUNT_PER_DEPOSIT);
            }
            System.out.println ("Impiegato " + _name + " ha versato un totale
di " + NUM_OF_DEPOSITS * AMOUNT_PER_DEPOSIT);
        } catch (InterruptedException e) {}
    }
}
```

La programmazione concorrente in Java - 4

Interferenza tra threads: corsa critica nel bank account (3/7)

L'oggetto conto corrente:

```
class BankAccount {  
    BankAccount(int initialValue) {  
        _value = initialValue;  
    }  
    void increase(int amount) {  
        int temp = _value;  
        Simulate.HWinterrupt();  
        _value = temp + amount;  
        System.out.println("Nuovo saldo: " + _value);  
    }  
    private int _value;  
}
```

Dove “Simulate” simula l'arrivo di un interrupt hardware che deschedula il thread.

La programmazione concorrente in Java - 5

Interferenza tra threads: corsa critica nel bank account (4/7)

Un possibile risultato dell'esecuzione del codice:

```
Alex@Alex:~# java ProblemaConcorrenza  
[...]  
Nuovo saldo: 4900  
Impiegato Mason ha versato un totale di 3000  
Impiegato Steinberg ha versato un totale di 3000
```

A fronte di un versamento totale di 6000€ ci ritroviamo con un conto di 4900€!!!! C'è un problema di interferenza!

Qual è l'origine del problema di interferenza??

La programmazione concorrente in Java - 6

Interferenza tra threads: corsa critica nel bank account (5/7)

```
class BankAccount {  
    BankAccount(int initialValue) {  
        _value = initialValue;  
    }  
    void increase(int amount) {  
        int temp = _value;  
        Simulate.HWinterrupt();  
        _value = temp + amount;  
        System.out.println("Nuovo saldo: " + _value);  
    }  
    private int _value;  
}
```

Corsa critica ...

...nell'accesso alla variabile di stato condivisa.

Attenzione allo **stato condiviso (shared e mutable)!!!**

La programmazione concorrente in Java - 7

Interferenza tra threads: corsa critica nel bank account (6/7)

Nell'esempio bank account, se anche scriviamo
value = value + amount

tale istruzione continua a non essere atomica e genera quindi delle possibili corse critiche in situazione multi threaded. (si noti che anche **cont++** NON è atomica).

Ci sono varie soluzioni a questo problema di corsa critica.

- 1) Notiamo che, in questo caso, lo stato (**shared e mutable**) del conto corrente che deve essere protetto è il valore del conto, un intero. Allora si potrebbe usare un **value** della classe **AtomicLong**, un intero lungo che fornisce delle operazioni atomiche per operare, come il metodo **addAndGet(value)**

Questa **non è una soluzione generalizzabile**, vale solo quando lo stato è composto da un solo dato semplice (qui è un intero). Non si può garantire atomicità su azioni che coinvolgono DUE oggetti diversi (anche se sono entrambi atomici) perché la relazione invariante in quel caso li coinvolge entrambi.

La programmazione concorrente in Java - 8

Interferenza tra threads: corsa critica nel bank account (7/7)

- 2) Una soluzione più generale allora richiede l'utilizzo dei costrutti di sincronizzazione del linguaggio Java, come la **keyword synchronized**.

```
class BankAccount {  
    ...  
    void synchronized increase(int amount) {  
        int temp = _value;  
        Simulate.HWinterrupt();  
        _value = temp + amount;  
        ...  
    }  
    ...
```

Corsa critica

Studieremo il **synchronized** tra poco.

La programmazione concorrente in Java - 9

Interferenza tra threads: visibilità degli update (1/5)

Attenzione: la complessità dell'ambiente di esecuzione Java Multi Threaded richiede molta attenzione, perché ci sono problemi molto meno evidenti della "semplice" interferenza nel bank account.

Il problema della **visibilità degli update** (la memory visibility). Quando un thread modifica lo stato di un oggetto, "ci aspettiamo" che gli altri threads vedano lo stato appena modificato.

La programmazione concorrente in Java - 10

Interferenza tra threads: visibilità degli update (2/5)

```
public class NoVisibility {  
    private static boolean ready;  
    private static int number;  
  
    private static class ReaderThread extends Thread {  
        public void run() {  
            while (!ready)  
                Thread.yield();  
            System.out.println(number);  
        }  
    }  
  
    public static void main(String[] args) {  
        new ReaderThread().start();  
        number = 42;  
        ready = true;  
    }  
}
```

Che risultato ci aspettiamo?
Cosa verrà stampato?

Esempio dal Goetz, et alii: Java Concurrency in Practice

La programmazione concorrente in Java - 11

Interferenza tra threads: visibilità degli update (3/5)

Si noti che nell'esempio ci sono due thread, un "reader" che aspetta di vedere che la variabile ready diventi true, per poi stampare number (42).

Purtroppo, il thread reader potrebbe:

1. finire correttamente e stampare 42.
2. non finire mai.
3. finire e stampare 0 (problema di "reordering").

Che non sono comportamenti "intuitivi".

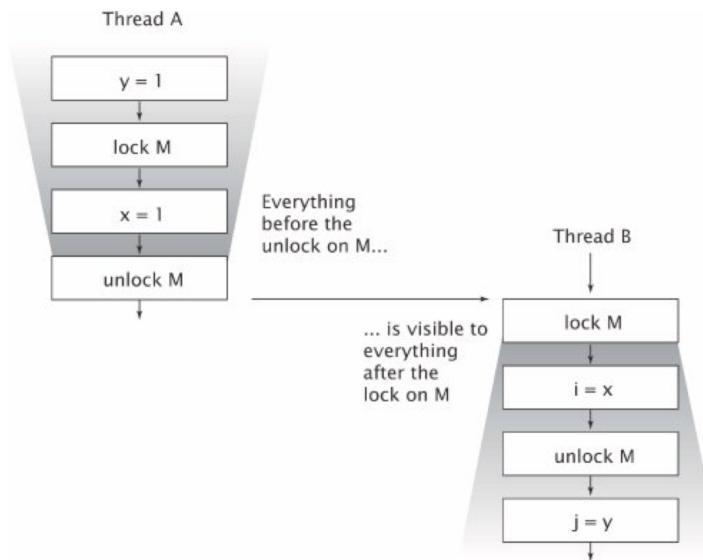
Il Java Memory Model è cambiato nel corso del tempo, è diventato più sofisticato e complesso (anche per sfruttare meglio la potenza dei nuovi processori multicore). Per ottimizzare accesso a memoria, il compilatore può permettere a un thread di mantenere il valore di una variabile in una memoria locale (registri) invece che in memoria centrale, oppure in una cache dedicata a un processore... Quindi, se più threads accedono a tale variabile, potrebbero vedere risultati inattesi ("stale" data).

La programmazione concorrente in Java - 12

Interferenza tra threads: visibilità degli update (4/5)

Come evitare che ci siano dati “stale”? Con la sincronizzazione.

Per fortuna la sincronizzazione non risolve solo i problemi di vera e propria **sincronizzazione**, ma anche quelli di **visibilità**.



La programmazione concorrente in Java - 13

Interferenza tra threads: visibilità degli update (5/5)

Sincronizzazione e visibilità:

- 1) utilizzare i costrutti di sincro del linguaggio (come synchronized)
- 2) usare la keyword volatile?

Quando un campo è dichiarato **volatile** il compilatore e il runtime sono avvisati che si tratta di un oggetto shared mutable e quindi non va mantenuto nelle cache o nei registri (dei vari processori).

Per vari motivi, primo fra tutti la leggibilità del codice, è meglio utilizzare i costrutti di sincronizzazione, invece della etichetta volatile.

Inoltre, volatile risolve la visibilità, **non l'atomicità**, per esempio, l'istruzione `count++` non è atomica neanche se `count` è indicata come volatile....

La programmazione concorrente in Java - 14

Costrutti per la sincronizzazione in Java

Partiamo dai meccanismi di sincronizzazione di base, presenti in Java fin dalla nascita (a partire dalla keyword `synchronized`).

Implicit Monitor Lock (keyword `synchronized`) (1/3)

Java supporta la **mutua esclusione** nell'accesso a risorse condivise tramite la keyword `synchronized`.

Ogni oggetto Java ha associato un **lock** (in Java spesso riferito come “*implicit monitor lock*”) che viene acquisito quando si invocano metodi o istruzioni `synchronized`:

- blocco di istruzioni
`synchronized (shared-obj) { sezione critica }`
- singolo metodo (1)
`type metodo() { synchronized (this) {sezione critica} }`
- singolo metodo (2)
`synchronized type metodo() { sezione critica }`

La programmazione concorrente in Java - 15

Implicit Monitor Lock (keyword `synchronized`) (2/3)

Un (implicit monitor) lock può essere in possesso di **UN SOLO** thread alla volta.

Un thread che non riesce ad acquisire un **lock rimane sospeso** sulla richiesta della risorsa (in una coda chiamata “**entry set**”) fino a quando il **lock** diventa disponibile.

Il **lock** è automaticamente **rilasciato** quando il thread esce dalla sezione `synchronized`, o se viene interrotto da un’eccezione.

Un singolo oggetto con molti metodi o blocchi `synchronized` ha comunque **un solo lock** associato, con conseguenze:

- ➔ due thread diversi non possono accedere contemporaneamente a **due sezioni synchronized diverse di uno stesso oggetto**
- ➔ tutti i thread sospesi sono sulla stessa coda (“**entry set**”)

I lock sono assegnati su base thread e sono **rientranti** (o ricorsivi). Lo stesso thread può accedere a stessa procedura `synchronized` in modo ricorsivo, poiché dopo la prima volta detiene il lock. Attenzione al rilascio perché il lock viene rilasciato solo al termine della prima invocazione.

La programmazione concorrente in Java - 16

Implicit Monitor Lock (keyword synchronized) (3/3)

Non ci sono particolari relazioni tra lo stato di un oggetto e il suo “implicit monitor lock”. Anche se spesso si costruiscono oggetti che encapsulano tutto lo stato shared mutable e si sincronizzano **tutti** i metodi per l’accesso.

Se una **relazione invariante** è un’azione composta che coinvolge più variabili di stato, attenzione che **tutte le variabili** devono essere **protette da uno stesso lock**.

Non si confonda questi “implicit monitor lock” con gli oggetti “lock” delle concurrency utilities che vedremo nel seguito.

La programmazione concorrente in Java - 17

ESEMPIO - Il Bank Account Sincronizzato

```
public static void main(String[] args) {  
    SynchronizedBankAccount sba = new SynchronizedBankAccount (0);  
    Employee e = new Employee ("Mason", sba);  
    Customer c = new Customer ("Steinberg", sba);  
    e.start();  
    c.start();  
}
```

La programmazione concorrente in Java - 18

ESEMPIO - Il Bank Account Sincronizzato

```
public class SynchronizedBankAccount {  
    private int _value;  
    SynchronizedBankAccount (int initialValue) { _value = initialValue; }  
    synchronized public void increase (int amount) {  
        int temp = _value;  
        Simulate.HWinterrupt();  
        _value = temp + amount;  
    }  
    synchronized public int withdraw (int amount) {  
        if (_value >= amount) {  
            int temp = _value;  
            Simulate.HWinterrupt();  
            _value = temp - amount;  
            return amount;  
        }  
        return 0;  
    }  
    synchronized void printBalance() {  
        System.out.println ("Saldo attuale: " + _value);  
    }  
}
```

La programmazione concorrente in Java - 19

Azioni composte e atomicità (1/3)

Ritorniamo sul punto, detto sopra, che “se una relazione invariante è un’azione composta che coinvolge più variabili di stato, attenzione che **tutte le variabili devono essere protette da uno stesso lock**”.

Questo vale in termini più generali anche quando applicato alle azioni composte su oggetti **atomici**, come i Vector dell’esempio sotto.

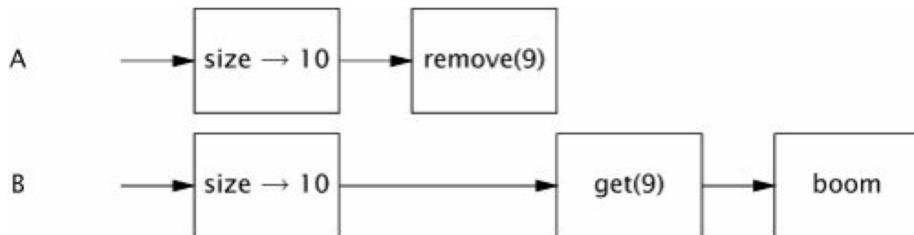
```
public static Object getLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    return list.get(lastIndex);  
}  
  
public static void deleteLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex);  
}
```

La programmazione concorrente in Java - 20

Azioni composte e atomicità (2/3)

Le singole operazioni della classe Vector sono garantite essere atomiche, ma un'azione composta come quella svolta da due thread, A e B, può generare situazioni non consistenti, come in figura sotto.

```
public static Object getLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    return list.get(lastIndex);  
}  
  
public static void deleteLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex);  
}
```



Ancora una volta è richiesta una sincronizzazione...

La programmazione concorrente in Java - 21

Azioni composte e atomicità (3/3)

Possiamo quindi utilizzare l'implicit monitor lock, utilizzando il synchronized...

```
public static Object getLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    return list.get(lastIndex);  
}  
  
public static void deleteLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex);  
}
```

Dove metto il synchronized?



La programmazione concorrente in Java - 22

Costrutti di sincronizzazione in Java

Con i metodi `synchronized` si garantisce la **mutua esclusione**.

Come faccio a realizzare i meccanismi di sincronizzazione con guardia (condizione di sincronizzazione ed eventuale sospensione)? Quindi, ci sono strumenti paragonabili ai **semafori privati** e alle **variabili condizione** viste per i Monitor?

Posso usare una forma come quella sotto?

```
while (not condizione-sincronizzazione); //spin  
<accesso a risorsa>
```

Questa soluzione è da scartare:

- tipicamente while dovrebbe essere interno a metodo sincronizzato ...
- è comunque un'attesa attiva (e se scheduling è FIFO?)

Servono primitive per **sospendere e risvegliare i thread** una volta giunti all'interno di una regione sincronizzata (come le wait e signal viste per i semafori e i monitor).

La programmazione concorrente in Java - 23

Costrutti di sincronizzazione in Java (wait)

La primitiva `wait()` sospende il thread che la invoca.

Ogni oggetto Java (istanza di una sottoclasse qualsiasi della classe Object) è dotato di una **wait-queue** (un **wait set**), una coda dove vanno i thread che si sospendono (primitiva `wait()`). Si può parlare di una “**intrinsic condition variable**” o intrinsic condition queue, intrinseca per ogni oggetto (o anche “implicit condition variable”).

Attenzione: una wait-queue può essere manipolata **SOLO se si è in possesso del lock** associato a quell'oggetto (e quindi se si è all'interno di codice `synchronized`). Altrimenti viene lanciata un'eccezione.

Un thread entra nella wait-queue invocando:

- `wait()` che sospende il thread sulla wait-queue e in maniera **atomica** rilascia il lock (“implicit monitor lock”).

Viene rilasciato il lock anche se acquisito più volte ricorsivamente. Non sono rilasciati altri lock eventualmente detenuti (attenzione alle chiamate innestate a monitor).

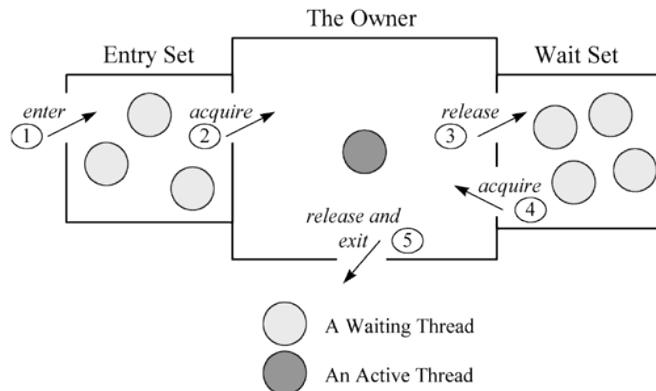
Versioni di `wait()` dotate di timeout.

La programmazione concorrente in Java - 24

Costrutti di sincronizzazione in Java

Il monitor associato a ogni oggetto Java ha due code associate:

- Su “entry set” si sospendono i thread in seguito a `synchronized` (“implicit monitor lock”)
- Su “wait set” (o “wait queue”) si sospendono i thread che eseguono una `wait()` (“intrinsic o implicit condition variable”)



La programmazione concorrente in Java - 25

Costrutti di sincronizzazione in Java (notify)

I thread sono risvegliati dalla wait-queue quando sono invocate:

- `notify()`

invocata su un oggetto, risveglia UN SOLO thread della wait-queue. **Scelta arbitraria del thread**, dipende da implementazione JVM.

- `notifyAll()`

invocata su un oggetto, risveglia tutti i thread della wait-queue. È d'obbligo quando ci sono più condizioni di sincronizzazione all'interno dello stesso oggetto e devono quindi essere risvegliati tutti i thread sospesi su diverse condizioni di sincronizzazione (sempre sullo stesso oggetto). Ovviamente i thread svegliati entrano nel monitor **uno alla volta**...

I thread possono essere risvegliati anche alla fine del timeout di `wait()` o per interruzioni ed eccezioni.

Ricordarsi che `wait`, `notify` e `notifyAll` devono essere chiamati all'interno di una regione `synchronized`.

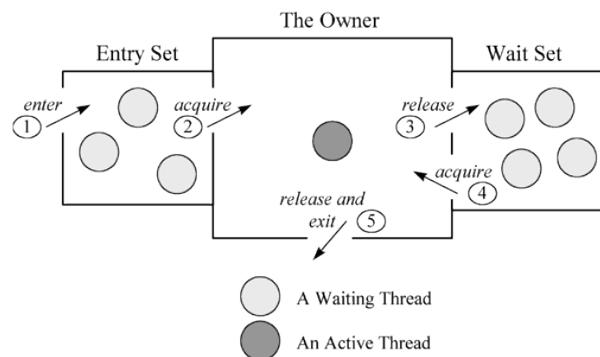
La programmazione concorrente in Java - 26

Costrutti di sincronizzazione in Java (semantica notify)

Quando un thread esegue la **notify()** il processo **risvegliato** può ripartire (dall'istruzione dopo la wait).

MA, attenzione, Java adotta la semantica **signal and continue** per cui il thread **risvegliato** deve aspettare che il thread **segnalante** esca dal monitor e liberi quindi il lock (implicit monitor lock).

Quando il thread segnalante esce, il thread **risvegliato** compete per il lock con altri eventuali thread (sia in wait set che entry set).



Considerata questa semantica **signal and continue**, cosa deve fare un thread quando viene risvegliato??

La programmazione concorrente in Java - 27

Produttore e Consumatore (guarded bounded buffer)

I processi produttore e consumatore:

```
class Producer implements Runnable {
    Buffer buf; String alphabet= "abcdefghijklmnopqrstuvwxyz";
    Producer(Buffer b) {buf = b;}
    public void run() {
        try { int ai = 0;
            while(true) {
                ...
                buf.put(new Character(alphabet.charAt(ai)));
                ai=(ai+1) % alphabet.length();
                ...
            } } catch (InterruptedException e){ } } }

class Consumer implements Runnable {
    Buffer buf;
    Consumer(Buffer b) {buf = b;}
    public void run() {
        try { while(true) {
            ...
            Character c = (Character)buf.get();
            ...
        } } catch(InterruptedException e ){ } } }
```

La programmazione concorrente in Java - 28

Produttore/Consumatore (guarded bounded buffer)

```
public class BufferImpl implements Buffer {  
    protected Object[] buf; int in=0; int out=0; int count=0; int size;  
  
    public BufferImpl(int size) {  
        this.size = size;  
        buf = new Object[size];  
    }  
  
    public synchronized void put(Object o) throws InterruptedException {  
        while (count==size) wait();  
        buf[in] = o;  
        ++count; in=(in+1) % size;  
        notify();  
    }  
  
    public synchronized Object get() throws InterruptedException {  
        while (count==0) wait();  
        Object o =buf[out];  
        buf[out]=null;  
        --count; out=(out+1) % size;  
        notify();  
        return (o);  
    }  
}
```

Guardie

La programmazione concorrente in Java - 29

Guardie con sospensione in Java

Quando un processo si sospende (`wait()` ma anche `sleep()`) si deve sempre usare il try-catch per essere sicuri di raccogliere un'eventuale eccezione che possa arrivare e forzare il risveglio del processo (`InterruptedException`).

SOSPENSIONE:

```
synchronized (obj) {  
    while (!condition) {  
        try {  
            obj.wait();  
        } catch(...){...}  
    }  
    <accesso a sezione critica>  
}
```

RISVEGLIO:

```
synchronized (obj) {  
    condition = true;  
    obj.notifyAll(); // oppure obj.notify()  
}
```

La programmazione concorrente in Java - 30

Guardie con sospensione in Java

Principale limitazione dei meccanismi di basso livello di Java è la presenza di un'unica **wait-queue** per un oggetto sincronizzato, con conseguenze importanti.

Il test della condizione di sincronizzazione deve essere **dentro un while**, perché ci possono essere processi di diverse tipologie (si pensi al caso lettori/scrittori) sospesi sulla stessa coda. In questo caso, non possiamo risvegliare selettivamente un processo specifico. I processi risvegliati devono quindi ritestare la condizione.

Il while è imposto *anche* dal linguaggio perché ci possono essere **risvegli spuri**. La JVM specifica esplicitamente che si possono verificare situazioni in cui un thread viene risvegliato dalla wait anche in assenza di una corrispondente notify.

In pratica l'unica wait queue non ci permette la soluzione di problemi come abbiamo visto con i semafori privati o le condition variables dei monitor.

La programmazione concorrente in Java - 31

Produttore/Consumatore con Semafori Privati (binari)

```
void produttore(void) {  
    while(TRUE) {  
        produzione-msg (&m);  
        wait(pieno);  
        wait(mutex);  
        put-msg(m);  
        signal(mutex);  
        signal(vuoto);  
    }  
}  
  
void consumatore(void) {  
    while(TRUE) {  
        wait(vuoto);  
        wait(mutex);  
        get-msg(&m);  
        signal(mutex);  
        signal(pieno);  
        consumo-msg(m);  
    }  
}
```

La programmazione concorrente in Java - 32

Esempio Produttore/consumatore in Java

```
public synchronized void put(Object o)
    throws InterruptedException {
    while (count==size) wait();
    buf[in] = o;
    ++count;
    in=(in+1) % size;
    notify();
}

public synchronized Object get()
    throws InterruptedException {
    while (count==0) wait();
    Object o =buf[out];
    buf[out]=null;
    --count;
    out=(out+1) % size;
    notify();
    return (o);
}
```

Dove si sospendono?

Chi viene notificato?

La programmazione concorrente in Java - 33

Esempio Lettori/Scrittori in Java

```
public synchronized void inizio_lettura()throws...{
    while (writing) wait();
    ++readers;
}

public synchronized void inizio_scrittura()throws...{
    while (readers>0 || writing) wait();
    writing = true;
}

public synchronized void fine_lettura () {
    --readers;
    if(readers==0) notify() (OPPURE notifyAll() ?)
}

public synchronized void fine_scrittura() {
    writing = false;
    notify() (OPPURE notifyAll() ?);
}
```

Quale politica è qui definita? E' Fair? Ci sono problemi safety?

La programmazione concorrente in Java - 34

Esempio Lettori/Scrittori in Java

```
public synchronized void inizio_lettura() throws...{  
    while (writing) wait();  
    ++readers; }  
  
public synchronized void inizio_scrittura() throws...{  
    while (readers>0 || writing) wait();  
    writing = true; }  
  
public synchronized void fine_lettura () {  
    --readers;  
    if(readers==0) notify() OPPURE (?) notifyAll();}  
  
public synchronized void fine_scrittura() {  
    writing = false;  
    notify() OPPURE (?) notifyAll();}
```

Questo esempio è safe? Fair? confrontare con semafori privati qui sotto:

```
inizio_lettura() {  
    if (scrittori){  
        attesaLettori++; wait(semLettori); attesaLettori--;  
    }  
    if (attesaLettori>0) signal(semLettori);  
    contLettori++; }
```

while o if?

Chi viene notificato?

La programmazione concorrente in Java - 35

```
inizio_scrittura() {  
    if (contLettori>0||scrittori){ attesaScrittori++;  
        wait(semScrittori); attesaScrittori--;  
    }  
    scrittori=true;  
}  
  
fine_lettura() {  
    contLettori--;  
    if (contLettori==0 && attesaScrittori >0) signal(semScrittori);  
}  
  
fine_scrittura() {  
    scrittori=false;  
    if (attesaLettori>0) signal(semLettori);  
    else if (attesaScrittori>0)  
        signal(semScrittori);  
}
```

La programmazione concorrente in Java - 36

Esempio Lettori/Scrittori in Java: problemi dei meccanismi di basso livello

Una sola **wait-queue** su cui si sospendono tutti i lettori e tutti gli scrittori.

Ogni volta risveglia **tutti** i thread, ogni thread ritesta la condizione, chi entra?

È una soluzione live? Si può avere starvation di un tipo di processi? Sono possibili problemi di fairness?

Uso di **notify** invece di **notifyAll** migliora l'efficienza???

Servono costrutti di livello più alto, come i semafori privati e le variabili condizione dei monitor visti precedentemente.

La programmazione concorrente in Java - 37

Possiamo costruire dei semafori in Java?

Utilizzando i costrutti primitivi di Java si potrebbero realizzare meccanismi ad-hoc di più alto livello, come semafori e anche monitor. Per esempio, ricordando la definizione di semaforo, si potrebbe realizzare una classe:

```
public class Semaforo {  
    private int value;  
    public Semaforo (int initial){ value = initial; }  
  
    synchronized public void wait() throws InterruptedException {  
        while (value == 0) wait();  
        --value;  
    }  
  
    synchronized public void signal() {  
        ++value;  
        notify();  
    }  
}
```

Sono proprio uguali ai semafori tradizionali?

La programmazione concorrente in Java - 38

Produttore/Consumatore con semafori

Si ricordi lo schema:

```
void produttore(void) {  
    while(TRUE) {  
        produzione-msg (&m);  
        wait(pieno);    valore iniziale N  
        wait(mutex);  
        put-msg(m);  
        signal(mutex);  
        signal(vuoto);  
    } }  
  
void consumatore(void) {  
    while(TRUE) {  
        wait(vuoto); valore iniziale 0  
        wait(mutex);  
        get-msg(&m);  
        signal(mutex);  
        signal(pieno);  
        consumo-msg(m);  
    } }  
}
```

La programmazione concorrente in Java - 39

Produttore/Consumatore con (pseudo)semafori in Java

```
class SemaBuffer implements Buffer {  
    Semaforo full;      //conta il numero di elementi  
    Semaforo empty;    //conta i posti liberi  
    SemaBuffer(int size) {  
        this.size=size; buf=new Object[size];  
        full = new Semaforo(0);  
        empty= new Semaforo(size);  
    } }  
    synchronized public void put(Object o) throws InterruptedException {  
        empty.wait();  
        buf[in] = o; ++count; in=(in+1)%size;  
        full.signal();  
    }  
    synchronized public Object get() throws InterruptedException{  
        full.wait();  
        Object o =buf[out]; buf[out]=null;  
        --count; out=(out+1)%size;  
        empty.signal();  
        return (o);  
    }
```

Funziona tutto bene??? È proprio uguale allo schema sopra?

La programmazione concorrente in Java - 40

Produttore/Consumatore con (pseudo)semafori in Java

L'esempio sopra non garantisce liveness, perché ci può essere **deadlock**:

- se il buffer è vuoto
- consumatore fa una get
- consumatore chiama full.wait () e si blocca (e rilascia il lock sul semaforo full MA non quello preso con get)
- produttore invoca put ma viene sospeso sul lock del buffer, che non è stato rilasciato dal consumatore.

In questo caso il deadlock è causato da chiamate innestate (**nested monitor**).

La programmazione concorrente in Java - 41

Esempio corretto di Prod/Cons con (pseudo)semafori in Java

```
public void put(Object o) throws InterruptedException {
    empty.wait();
    synchronized(this){
        buf[in] = o; ++count; in=(in+1)%size;
    }
    full.signal();
}
```

In questo caso, il deadlock può essere evitato solo attraverso un progetto attento, sincronizzando la sezione critica di accesso al buffer **solo dopo** il passaggio del semaforo.

Esigenza di costrutti di alto livello disponibili a livello di linguaggio.

La programmazione concorrente in Java - 42

I costrutti di alto livello in Java: le Concurrency Utilities

La difficoltà di usare strumenti a basso livello ha indotto a inserire nella versione 5 di Java le “**Concurrency Utilities**”, un insieme di meccanismi di sincronizzazione di alto livello che facilitano la realizzazione di applicazioni concorrenti.

Si consulti: <http://docs.oracle.com/javase/7/docs/api/>

Strumenti presenti nelle Concurrency Utilities:

- gestione dei thread (Executor framework, thread pool)
- classi ottimizzate per accesso concorrente (per esempio dati come AtomicLong e strutture dati tipo Queue)
- meccanismi di sincronizzazione (come semaphore, mutex, barriers, ...)
- lock (evoluzione dei blocchi synchronized)
- variabili condizione (associate ai lock)

Vantaggi offerti dall'utilizzo delle Concurrency Utilities:

- facilità di uso (usiamo classi standard, invece di svilupparle)
- prestazioni (fondamentali per applicazioni lato server)
- affidabilità (abbiamo visto negli esempi sopra la difficoltà di evitare deadlock etc.)
- produttività (utilizzo di classi standard ovviamente aumenta la produttività)
- facile leggibilità del codice e quindi facilità di gestione (maintenance)

La programmazione concorrente in Java - 43

Le Concurrency Utilities: la classe Semaphore

La classe **Semaphore** introduce i semafori, con una semantica identica a quella vista nei semafori tradizionali, quindi:

- un oggetto **Semaphore** è una struttura dati composta da un contatore (il numero di “permessi” disponibile) e da una coda su cui si possono sospendere i thread.
- i metodi principali sono **acquire()** e **release()** che corrispondono alle `wait()` e `signal()` viste sui semafori.

Nella terminologia Java, un semaforo viene inizializzato (mediante il costruttore) con un numero iniziale di “permessi”.

Ogni **acquire()** decrementa il numero di permessi oppure blocca il processo se il numero di permessi è zero.

Ogni **release()** incrementa il numero di permessi oppure risveglia un eventuale thread sospeso (cioè rilascia un permesso che viene acquisito da un thread sospeso).

Semantica signal and continue.

La programmazione concorrente in Java - 44

La classe Semaphore: particolarità

Il **costruttore** `Semaphore()` crea un semaforo con un certo numero iniziale di permessi e con un parametro per settare la **fairness dei risvegli**: si può cioè richiedere esplicitamente una politica fair (FIFO) che però ha un costo in termini di prestazioni.

Un semaforo con UN SOLO permesso è un semaforo binario.

Ci sono metodi `acquire()` e `release()` che permettono di acquisire/rilasciare n permessi in una volta sola. Se non sono disponibili TUTTI i permessi richiesti il thread viene sospeso (attenzione alla starvation...).

Molti altri metodi disponibili:

- Metodo `tryAcquire()` per tentare di acquisire i permessi del semaforo in modo **non bloccante** (acquisisce permessi se disponibili, altrimenti ritorna subito il controllo).
- Metodo `hasQueuedThread()` per verificare se ci sono threads sospesi su semaforo.

La programmazione concorrente in Java - 45

(sort of) Produttore/Consumatore con Java Semaphore (da API) 1/2

```
class Pool {  
  
    private static final int MAX_AVAILABLE = 100;  
    private final Semaphore available =  
        new Semaphore(MAX_AVAILABLE, true);  
  
    public Object getItem() throws InterruptedException {  
        available.acquire();  
        return getNextAvailableItem();  
    }  
  
    public void putItem(Object x) {  
        if (markAsUnused(x))  
            available.release();  
    }  
}
```

La programmazione concorrente in Java - 46

(sort of) Produttore/Consumatore con Java Semaphore (da API) 2/2

```
protected Object[] items = ... whatever kinds of items
protected boolean[] used = new boolean[MAX_AVAILABLE];

protected synchronized Object getNextAvailableItem() {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
        if (!used[i]) {
            used[i] = true;
            return items[i];
        }
    } return null; // not reached
}

protected synchronized boolean markAsUnused(Object item) {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
        if (item == items[i]) {
            if (used[i]) {
                used[i] = false;
                return true;
            } else return false;
        }
    } return false;
} }
```

La programmazione concorrente in Java - 47

Le Concurrency Utilities: il package locks

Tra le Concurrency Utilities si trova il package `java.util.concurrent.locks` che introduce importanti costrutti per il locking e per le variabili condizione.

Sono previste interfacce e varie implementazioni in classi, tra cui, in particolare:

Interfacce: Lock, ReadWriteLock e Condition.

Classi: ReentrantLock, ReentrantReadWriteLock, etc.

La programmazione concorrente in Java - 48

Le Concurrency Utilities: i Lock

Un lock permette di controllare l'accesso a una risorsa condivisa tra più thread.

Tradizionalmente, i Lock sono stati introdotti per garantire un accesso mutuamente esclusivo a una risorsa, ma in Java ci sono anche i read/write lock che permettono un accesso concorrente a più processi (i “lettori”).

In Java, un **lock** è un oggetto su cui possono essere chiamati i metodi di `lock()` e `unlock()` per sincronizzare i processi nell'accesso a una risorsa condivisa.

I lock permettono una **sincronizzazione più estesa e flessibile** rispetto al “implicit monitor lock” fornito dallo statement `synchronized`, con diverse proprietà configurabili (anche non bloccanti) e anche *multiple variabili condizioni* associate allo stesso lock.

La programmazione concorrente in Java - 49

Le Concurrency Utilities: i Lock

Metodi `lock()` e `unlock()` per acquisire/rilasciare un lock. Attenzione che SOLO il thread che ha acquisito il lock può chiamare la corrispondente unlock (a differenza di un semaforo binario). Altrimenti si scatena un’eccezione.

```
Lock l = new ReentrantLock( ); // esempio di lock
l.lock();
try {
    // access the resource protected by this lock
} finally { // unlock() eseguita anche in caso di eccezione
    l.unlock();
}
```

Si confronti questo modo di procedere con il i blocchi `synchronized` che rilasciano automaticamente l’implicit monitor lock al termine delle istruzioni del blocco stesso.

La programmazione concorrente in Java - 50

Le Concurrency Utilities: i Lock

Molti altri metodi disponibili sui Lock.

Metodo `tryLock()` per tentare di acquisire lock in modo **non bloccante** (acquisisce il lock se libero, altrimenti ritorna subito il controllo).

Metodo `tryLock(long time, TimeUnit unit)` per tentare di acquisire lock con un **timeout** (acquisisce il lock se libero, altrimenti aspetta per il timeout).

Metodo `hasQueuedThread()` per verificare se ci sono threads sospesi sul lock.

Metodo `newCondition()` per creare un oggetto condizione associato con una specifica istanza di un lock. Si possono creare *diverse condizioni su uno stesso lock* (come si vedrà nel seguito).

La programmazione concorrente in Java - 51

La classe ReentrantLock

L’interfaccia Lock implementata in molte classi, per esempio la classe ReentrantLock.

Un lock reentrant permette l’accesso mutuamente esclusivo a una risorsa, e può essere acquisito più volte in modo rientrante da uno stesso thread.

Si può attivare un lock reentrant “fair” (specificando tale parametro al costruttore) in modo da garantire accesso FIFO tra i thread in attesa, ma a prezzo di perdita di prestazioni (“*often much slower*”).

Tale classe implementa tutti i metodi sopra descritti.

La programmazione concorrente in Java - 52

Differenze tra “lock” e “implicit monitor lock” (synchronized)

| | implicit monitor lock (synchronized) | lock |
|------------------------|--|---|
| Ordine di acquisizione | Acquisizione block-structured (e rilascio in ordine inverso). | acquisizione e rilascio in qualsiasi ordine (es. “chain locking”: acquisisco locks A e B, poi rilascio A e acquisisco C, poi rilascio B e acquisisco D...). Attenzione ai deadlock. |
| Modalità di chiamata | Chiamata metodo synchronized è bloccante. | Metodi per l’acquisizione: bloccanti, non bloccanti (tryLock), anche con timeout. Reentrant oppure non-reentrant |
| Queueing Fairness | Unfair (migliori prestazioni) | diverse semantiche di code, anche guaranteed ordering |
| Condition variable | wait() e notify(), come se ci fosse <i>una sola</i> variabile condition (intrinsic cond. var.) | Disponibili molte variabili condition associate a uno stesso lock. |
| Prestazioni | | Migliori prestazioni (ma attenzione...) |
| Facilità d’uso | Notazione facile e compatta | Semantica molto più sofisticata, ma richiede molta più attenzione (ai deadlock e a safety in caso di eccezioni) |

La programmazione concorrente in Java - 53

La classe ReentrantReadWriteLock

La classe ReentrantReadWriteLock ha la semantica della ReentrantLock ma aggiunge i lock specifici per risolvere problemi della tipologia lettori/scrittori.

Il metodo `writelock()` restituisce il lock di scrittura (su cui invocare `lock()`). Il metodo `readlock()` restituisce il lock di lettura (su cui invocare `lock()`).

I thread **lettori** chiamano il metodo `lock()` su un lock di lettura per chiedere l’accesso alla risorsa. Se la risorsa non è occupata da uno scrittore i lettori (anche più di uno) potranno entrare.

Un thread **scrittore** chiama `lock()` su un lock di scrittura per chiedere l’accesso **esclusivo** alla risorsa. Se la risorsa è libera un solo scrittore potrà entrare.

Attenzione: un oggetto ReentrantReadWriteLock può essere definito **fair** o **non fair** (non si fanno assunzioni sull’ordine con cui i thread acquisiranno la risorsa).

La programmazione concorrente in Java - 54

Esempio Lettori/Scrittori in Java con ReentrantReadWriteLock

```
...
private final ReentrantReadWriteLock rwlock = new ReentrantReadWriteLock();
private final Lock r = rwlock.readLock();
private final Lock w = rwlock.writeLock();
...
public void lettura() throws...{
    r.lock();           // oppure rwlock.readLock().lock()
    <accesso alla risorsa in lettura>
    r.unlock();        // usare try-finally (rischi safety e deadlock)
}
public void scrittura() throws...{
    w.lock();          ← Ci vuole synchronized?
    <accesso alla risorsa in scrittura>
    w.unlock();        // usare try-finally (rischi safety e deadlock)
}
...
}
```

La programmazione concorrente in Java - 55

I Read Write Lock

Ci sono varie implementazioni dei Read Write Lock che offrono semantiche molto diverse, per esempio:

Precedenze nei risvegli: uno scrittore che rilascia il lock può svegliare un altro scrittore oppure un lettore, oppure il primo dei processi in attesa arrivato.

Lettori. Si può anche decidere se dare la priorità ai lettori e quindi nel caso di un flusso continuo di lettori che acquisiscono il lock (soluzione efficiente) si rischia la starvation degli scrittori.

Read e Write Lock **rientranti o non rientranti**.

Possibile specificare **Downgrading** (un processo scrittore diventa lettore senza rilasciare il lock).

Possibile specificare **Upgrading** (un processo lettore diventa scrittore senza rilasciare il lock, ovviamente quando saranno usciti tutti gli altri lettori).

La programmazione concorrente in Java - 56

Quando usare le variabili condizione in Java

Consideriamo l'esempio di Bounded Buffer seguente (Goetz, et alii, Listing 14.5):

```
@ThreadSafe
public class SleepyBoundedBuffer<V> extends BaseBoundedBuffer<V> {
    public SleepyBoundedBuffer(int size) { super(size); }

    public void put(V v) throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isFull()) {
                    doPut(v);
                    return;
                }
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }

    public V take() throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isEmpty())
                    return doTake();
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }
}
```

Safe? Live? Efficiente?

La programmazione concorrente in Java - 57

Quando usare le variabili condizione in Java

Ovviamente il Bounded Buffer può essere realizzato in modo più efficiente ed elegante utilizzando wait e notify (lucido 27).

Ma guardate cosa scrive il Goetz:

BoundedBuffer in [Listing 14.6](#) implements a bounded buffer using `wait` and `notifyAll`. This is simpler than the sleeping version, and is both more efficient (waking up less frequently if the buffer state does not change) and more responsive (waking up promptly when an interesting state change happens). This is a big improvement, but note that the introduction of condition queues didn't change the semantics compared to the sleeping version. It is simply an optimization in several dimensions: CPU efficiency, context-switch overhead, and responsiveness. Condition queues don't let you do anything you can't do with sleeping and polling^[5], but they make it a lot easier and more efficient to express and manage state dependence.

No. Sono **certamente** da preferire le condition queue al polling, per mille motivi...

La programmazione concorrente in Java - 58

Le variabili condizione in Java

Ogni istanza di **lock** può avere associata una o più “variabili condizione”, chiamate istanze “condition” (mediante la chiamata del metodo `newCondition()` su uno specifico lock). In questo modo è possibile avere più wait-queue separate associate a uno stesso lock.

Metodi invocabili su una istanza `condition`:

- `await()` può essere invocato solo se si è acquisito il lock. In tale caso, **sospende** il thread sulla coda associata alla `condition` e **atomicamente rilascia** il lock.
- `signal()` risveglia un thread bloccato sulla coda `condition`
- `signalAll()` risveglia tutti i thread bloccati sulla coda `condition`

Attenzione:

- esiste la possibilità di “spurious wakeup” (“*as a concession to the underlying platform semantics*”) e quindi la guardia va sempre dentro un ciclo (ad es. `while`).

La programmazione concorrente in Java - 59

Le variabili condizione in Java

Le “variabili condizione” (**explicit condition queues**) sui lock sono molto simili alla **intrinsic condition queue** (associata a un oggetto `synchronized`)

Vantaggi delle *explicit* condition queues su quelle *intrinsic*:

- ogni lock può avere anche molte explicit condition queues
- accodamento threads fair oppure non fair
- sono previste 3 forme di condition waiting:
 - non-interrompibile (`awaitUninterruptibly()`, il thread sospeso viene svegliato solo da una signal)
 - interrompibile (`await()`, il thread sospeso viene svegliato da una signal oppure da un'interruzione, gestire le eccezioni in questo caso...)
 - con time out (`await(long time)`, come sopra, ma si aspetta al massimo un certo tempo, gestire eccezioni...)

La programmazione concorrente in Java - 60

Produttore/Consumatore con lock e condition in Java 1/2

```
class BoundedBuffer {  
    final Lock lock = new ReentrantLock();  
    final Condition notFull = lock.newCondition();  
    final Condition notEmpty = lock.newCondition();  
  
    final Object[] items = new Object[100];  
    int putptr, takeptr, count;  
  
    public void put(Object x) throws InterruptedException {  
        lock.lock();  
        try {  
            while (count == items.length) notFull.await();  
            items[putptr] = x;  
            if (++putptr == items.length) putptr = 0;  
            ++count;  
            notEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

La programmazione concorrente in Java - 61

Produttore/Consumatore con lock e condition in Java 2/2

```
public Object take() throws InterruptedException {  
    lock.lock();  
    try {  
        while (count == 0) notEmpty.await();  
        Object x = items[takeptr];  
        if (++takeptr == items.length) takeptr = 0;  
        --count;  
        notFull.signal();  
        return x;  
    } finally {  
        lock.unlock();  
    }  
}
```

La programmazione concorrente in Java - 62

Esercizio: Il canale di Suez in Java

Il canale di Suez mette in collegamento il mar Mediterraneo con il mar Rosso e può essere utilizzato da imbarcazioni per il trasporto di passeggeri, da navi per il trasporto di merci e da petroliere. Il canale è sufficientemente largo da permettere il transito contemporaneo di qualunque tipo di nave nei due versi di percorrenza, ma si supponga che per ragioni di sicurezza il transito delle petroliere sia ammesso solo quando non ci sono imbarcazioni di tipo diverso all'interno del canale.

Politica di gestione del canale:

- nell'accesso al canale, le imbarcazioni per il trasporto dei passeggeri hanno sempre la precedenza sulle altre navi;
- le petroliere hanno la precedenza sulle navi per il trasporto delle merci.

Attenzione a safety e liveness.

La programmazione concorrente in Java - 63

Esercizio: Canale di Suez con costrutti Java di basso livello

```
public void run() { // Thread navi  
  
    ...  
    CanSuez.richAccesso(TipoNave); // Nave richiede accesso  
    ...  
  
    // Nave percorre canale  
  
    ...  
    CanSuez.uscita(TipoNave); // Nave esce dal canale  
    ...
```

La programmazione concorrente in Java - 64

```

public synchronized void richAccesso(int tipo)
                                throws InterruptedException {
    while (Condizione(tipo)) {
        inAttesa[tipo]++;
        wait();
        inAttesa[tipo]--;
    }
    inTransito[tipo]++;
    show(); // un qualche metodo per stampare lo stato del canale
}

public synchronized void uscita(int tipo)
                                throws InterruptedException {
    inTransito[tipo]--;
    show();
    if(inTransito[tipo]==0) notifyAll();
}

```

Si noti la presenza della `notifyAll()`. Era meglio `notify()`?

La programmazione concorrente in Java - 65

```

private boolean Condizione (int tipo) {
    if ( (tipo == 0 && inTransito[1]!= 0) || //safety
        (tipo == 1 && inTransito[0]!= 0) ||
        (tipo == 1 && inTransito[2]!= 0) ||
        (tipo == 2 && inTransito[1]!= 0) ||
        (tipo == 1 && inAttesa[0]!=0)      || //precedenze
        (tipo == 2 && inAttesa[1]!=0) )
        return true;
    else
        return false;
}

private void show() {..} //stampa stato: navi in canale/attesa

```

La programmazione concorrente in Java - 66

Esercizio: Il canale di Suez con lock e condition Java

```
public Canale() {  
    ...  
    lock = new ReentrantLock(); // crea lock e condition variable  
    attesaNaveTipo = new Condition[3];  
    for (i=0; i<3; i++) attesaNaveTipo[i] = lock.newCondition();  
    ...  
}
```

La programmazione concorrente in Java - 67

```
public void richiestaAccesso(int tipo) throws InterruptedException {  
    lock.lock();           ←  
    try {  
        while (Condizione(tipo)) {  
            inAttesa[tipo]++; show();  
            attesaNaveTipo[tipo].await();  
            inAttesa[tipo]--;  
        }  
        inTransito[tipo]++;  
        show();  
        attesaNaveTipo[tipo].signal();  
    }  
    finally {  
        lock.unlock();  
    }  
}
```

Ci vuole synchronized?

Nota: Condizione(tipo) è quello di prima.

La programmazione concorrente in Java - 68

```

public void uscita(int tipo) throws InterruptedException {
    lock.lock(); ←
    try {
        inTransito[tipo]--;
        show();
        if (inTransito[tipo]==0) {
            switch(tipo) {
                case 0: // nave passeggeri
                    if (inTransito[2] == 0) attesaNaveTipo[1].signal();
                case 1: // petroliera
                    attesaNaveTipo[0].signal();
                    attesaNaveTipo[2].signal();
                case 2: // nave merci
                    if (inTransito[0] == 0) attesaNaveTipo[1].signal();
            }
        }
    } finally { lock.unlock(); }
}

private void show() {..} //stampa stato: navi in canale/attesa

```

Ci vuole synchronized?

Nota: Si confronti questo metodo di uscita con quello visto con gli strumenti Java di basso livello. Vantaggi e svantaggi?

La programmazione concorrente in Java - 69

Esercizio: una stretta strada di montagna in Java

Una strada di montagna è normalmente percorsa da **auto** e **camion**. La strada permette il passaggio contemporaneo di auto in entrambe le carreggiate. La strada è però stretta e se un camion accede in un verso, nessun altro veicolo può accedere in verso opposto. Ogni tanto, inoltre, la strada è percorsa da **un veicolo spazzaneve**: in questo caso, l'accesso alla strada deve essere impedito a ogni altro veicolo, in qualunque verso di percorrenza, finché lo spazzaneve non ha terminato la pulizia e liberato la strada. Lo spazzaneve può occupare la strada solo se questa è libera.

Utilizzando il costrutto monitor, si realizzi una politica di gestione della strada che tenga conto dei seguenti vincoli:

- lo spazzaneve ha la precedenza sugli autoveicoli (sia automobili sia camion);
- le automobili hanno sempre la precedenza sui camion.

La programmazione concorrente in Java - 70

Esercizio: una stretta strada di montagna (Java basics)

```
public synchronized void richiestaAccesso(int direzione, int tipo) throws InterruptedException {  
  
    while (Condizione(direzione, tipo)) {  
        inAttesa[direzione][tipo]++;  
        wait();  
        inAttesa[direzione][tipo]--;  
    }  
    inTransito[direzione][tipo]++;  
}  
  
public synchronized void uscita(int direzione, int tipo) throws InterruptedException {  
  
    inTransito[direzione][tipo]--;  
  
    if(inTransito[direzione][tipo]==0) {  
        notifyAll();  
    }  
}
```

La programmazione concorrente in Java - 71

```
private boolean Condizione(int dir, int tipo) {  
    ...  
    if (  
        (tipo == AUTOMOBILE && inTransito[dir][SPAZZANEVE] != 0) || // safety  
        (tipo == AUTOMOBILE && inTransito[otherdir][SPAZZANEVE] != 0) ||  
        (tipo == AUTOMOBILE && inTransito[otherdir][CAMION] != 0) ||  
  
        // lo spazzaneve deve essere l'unico a passare per la strada  
        (tipo == SPAZZANEVE && inTransito[dir][SPAZZANEVE] != 0) ||  
        (tipo == SPAZZANEVE && inTransito[dir][AUTOMOBILE] != 0) ||  
        (tipo == SPAZZANEVE && inTransito[dir][CAMION] != 0) ||  
        (tipo == SPAZZANEVE && inTransito[otherdir][SPAZZANEVE] != 0) ||  
        (tipo == SPAZZANEVE && inTransito[otherdir][AUTOMOBILE] != 0) ||  
        (tipo == SPAZZANEVE && inTransito[otherdir][CAMION] != 0) ||  
  
        // camion non possono incrociare nessuno nella direzione opposta  
        (tipo == CAMION && inTransito[dir][SPAZZANEVE] != 0) ||  
        (tipo == CAMION && inTransito[otherdir][SPAZZANEVE] != 0) ||  
        (tipo == CAMION && inTransito[otherdir][AUTOMOBILE] != 0) ||  
        (tipo == CAMION && inTransito[otherdir][CAMION] != 0) ||
```

La programmazione concorrente in Java - 72

```

// le automobili devono attendere lo spazzaneve
(tipo == AUTOMOBILE && inAttesa[dir][SPAZZANEVE] != 0) || // precedenze
(tipo == AUTOMOBILE && inAttesa[otherdir][SPAZZANEVE] != 0) ||

// camion minima priorità
(tipo == CAMION && inAttesa[dir][SPAZZANEVE] != 0) ||
(tipo == CAMION && inAttesa[otherdir][SPAZZANEVE] != 0) ||
(tipo == CAMION && inAttesa[otherdir][AUTOMOBILE] != 0)
)

return true o false a seconda della condizione.

```

La programmazione concorrente in Java - 73

Esercizio: una stretta strada di montagna (Java Conc. Util.)

```

public void richiestaAccesso(int dir, int tipo) throws
InterruptedException {

    lock.lock();
    try {
        while (Condizione(dir, tipo)) {
            inAttesa[dir][tipo]++;
            attesaVeicolo[dir][tipo].await();
            inAttesa[dir][tipo]--;
        }
        inTransito[dir][tipo]++;
        totaleInTransito++;

        attesaVeicolo[dir][tipo].signal();
    }
    finally {
        lock.unlock();
    }
}

```

Nota: Condizione (dir, tipo) è quello di prima.

La programmazione concorrente in Java - 74

```

public void uscita(int dir, int tipo) throws InterruptedException {
    ...
    lock.lock();
    try {    inTransito[dir][tipo]--; totaleInTransito--;
        if (totaleInTransito == 0) {
            if (inAttesa[otherdir][SPAZZANEVE] > 0)
                attesaVeicolo[otherdir][SPAZZANEVE].signal();
            else    if (inAttesa[dir][SPAZZANEVE] > 0)
                attesaVeicolo[dir][SPAZZANEVE].signal();
            else if (inAttesa[dir][AUTOMOBILE] > 0 &&
                     inAttesa[otherdir][AUTOMOBILE] > 0 ) {
                attesaVeicolo[dir][AUTOMOBILE].signal();
                attesaVeicolo[otherdir][AUTOMOBILE].signal(); }
            else if (inAttesa[dir][AUTOMOBILE] > 0) {
                attesaVeicolo[dir][AUTOMOBILE].signal();
                attesaVeicolo[dir][CAMION].signal(); }
            else if ( inAttesa[otherdir][AUTOMOBILE] > 0 ) {
                attesaVeicolo[otherdir][AUTOMOBILE].signal();
                attesaVeicolo[otherdir][CAMION].signal(); }
            else if (inAttesa[dir][CAMION] > 0) {
                attesaVeicolo[dir][CAMION].signal(); }
            else if (inAttesa[otherdir][CAMION] > 0)
                attesaVeicolo[otherdir][CAMION].signal();
        } // fine if
    }
}

```

La programmazione concorrente in Java - 75

```

// uscita ultimo camion (ma ponte NON vuoto)
else if (tipo == CAMION && inTransito[dir][tipo]==0) {
    // provo a svegliare auto nella direzione opposta
    if (    inAttesa[otherdir][AUTOMOBILE] > 0 &&
           !(inAttesa[otherdir][SPAZZANEVE] > 0 || 
             inAttesa[dir][SPAZZANEVE] > 0)) {
        attesaVeicolo[otherdir][AUTOMOBILE].signal(); }
}

// uscita ultima automobile (ma ponte NON vuoto)
else if (tipo == AUTOMOBILE && inTransito[dir][tipo]==0 &&
          inTransito[dir][CAMION]==0) {

    // provo a svegliare camion nella direzione opposta
    if ( inAttesa[otherdir][CAMION] > 0 &&
          !(inAttesa[otherdir][SPAZZANEVE]>0||inAttesa[dir][SPAZZANEVE]>0))
        { attesaVeicolo[otherdir][CAMION].signal(); }

}

} finally { lock.unlock();}}

```

Nota: Si confronti questo metodo di uscita con quello visto con gli strumenti Java di basso livello. Vantaggi e svantaggi?

La programmazione concorrente in Java - 76