



Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

Programmazione Concorrente e Distribuita Problemi tipici della programmazione concorrente e come evitarli

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it



Nondeterminismo

- Come mostrato negli esempi visti in precedenza, il numero di diversi possibili percorsi di esecuzione (non controllati dal programmatore) di un programma concorrente può essere molto grande anche per programmi molto semplici con un numero molto piccolo di thread.
- Questo aspetto prende il nome di nondeterminismo: il programmatore non può determinare l'ordine assoluto dell'esecuzione delle istruzioni.
- Il nondeterminismo implica che testare un programma concorrente è solitamente difficile.



Race Conditions

- Nei programmi concorrenti molti problemi sono causati dalle cosiddette “corse critiche” (o Race Conditions):

tutte quelle situazioni in cui thread diversi operano su una risorsa comune, ed in cui il risultato viene a dipendere dall'ordine in cui essi effettuano le loro operazioni.

Non aver pensato alle Race Condition può causare problemi seri...



Race Conditions: un semplice esempio

```
public class Counter {  
    long count = 0;  
    public void add (long value) {  
        long tmp = this.count;  
        tmp = tmp + value;  
        this.count = tmp ;  
    }  
}
```

- Questo codice (molto semplice) può produrre risultati sbagliati se eseguito da più thread contemporaneamente!



Race Conditions: un semplice esempio

```
public class CounterTmp extends Thread {
    static Counter counter = new Counter();
    public void run() {
        for(int i=0; i<10000; i++)
            counter.add(1);
    }
    public static void main(String args[]) throws InterruptedException {
        CounterTmp p1 = new CounterTmp();
        CounterTmp p2 = new CounterTmp();
        p1.start(); p2.start();
        p1.join(); p2.join();
        System.out.println("Counter = " + counter.count);
    }
}
```

- Ci aspettiamo che alla fine, avendo chiamato 20000 volte il metodo add(1), count valga 20000.
- Provate e vedrete ...

Luigi Lavazza - Programmazione Concorrente e Distribuita

- 5 -

Problemi tipici e come evitarli



Race condition: cosa va storto

```
public class Counter {
    long count = 0;
    public void add (long value) {
        long tmp = this.count;
        tmp = tmp + value;
        this.count = tmp ;
    }
}
```

• Task1: }
 tmp=this.count; //tmp=119

• Task2:
 tmp=this.count; //tmp=119
 tmp=tmp+value; //tmp=120
 this.count=tmp; //count=120

tmp=tmp+value; //tmp=120
 this.count=tmp; //count=120

Abbiamo eseguito 2 volte il metodo add(1), una volta in task 1 e una volta in task 2, ma count è passato da 119 a 120, anziché diventare 121!

Tempo ↓

Luigi Lavazza - Programmazione Concorrente e Distribuita

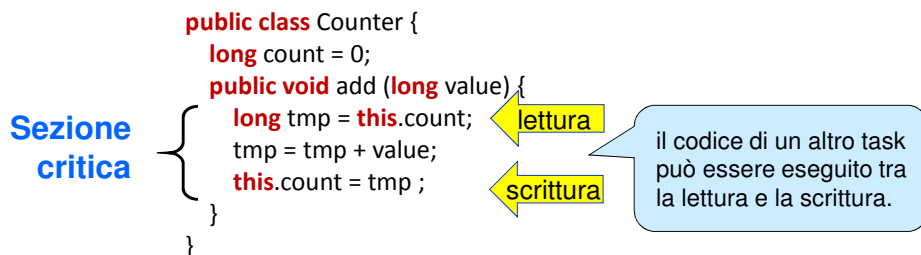
- 6 -

Problemi tipici e come evitarli



Race Conditions: quando può verificarsi?

- Sono necessarie due condizioni perché il programma possa fornire risultati diversi in diverse esecuzioni:
 1. Una risorsa deve essere condivisa tra due Thread (la variabile count nell'esempio precedente)
 2. Deve esistere almeno un percorso di esecuzione tra i Thread in cui una risorsa è condivisa in modo non sicuro (nell'esempio precedente il codice di add)



Race Conditions: come assicurare la correttezza del software?

- Non si può dimostrare la correttezza dei programmi concorrenti attraverso dei test fatti nel modo classico...
- Per dimostrare la correttezza di un programma concorrente esistono due modi:
 - ▶ Dimostrare per induzione che il programma soddisfa tutte le condizioni sufficienti perché possa essere considerato sicuro.
 - Ma dimostrare *a posteriori* che il software soddisfa le condizioni è spesso troppo costoso...
 - ▶ Sviluppare il programma utilizzando strategie note che permettano di **evitare situazioni di Race Conditions**.
 - Fare in modo che la risorsa condivisa sia in uno stato sicuro prima di consentire ad un altro thread di accedervi.
 - Bloccare l'accesso alle risorse condivise mentre sono in uno stato non sicuro è alla base delle strategie per ottenere programmi concorrenti sicuri



Come rendere sicuro il programma?

```

public class Counter {
    long count = 0;
    public void add (long value) {
        long tmp = this.count;
        tmp = tmp + value;
        this.count = tmp ;
    }
}

```

Qui bisogna bloccare

Qui si deve sbloccare

- L'oggetto counter (istanza di Counter) è condiviso.
 - ▶ C'è una «sezione critica» in cui avviene la lettura e scrittura (in tempi successivi) di una variabile condivisa.
- È necessario un meccanismo che **blocca** l'accesso alla sezione critica quando un thread entra, e lo **sblocca** quando il thread ne esce.
- In questo modo solo un thread alla volta può essere nella sezione critica.

Luigi Lavazza - Programmazione Concorrente e Distribuita

- 9 -

Problemi tipici e come evitarli



Come bloccare l'accesso agli oggetti

- Con un semaforo
 - ▶ Il primo thread che accede alla risorsa condivisa blocca l'accesso mediante il semaforo
 - ▶ Quando ha finito di usare la risorsa, sblocca l'accesso, consentendo ad un altro thread di accedere
 - ▶ Questo a sua volta bloccherà eventuali altri thread
 - ▶ Ecc.
- Il semaforo è binario (rosso o verde).
 - ▶ **acquire** acquisisce l'accesso bloccando altri task
 - eventualmente ponendo in attesa il task chiamante, se il semaforo è rosso.
 - ▶ **release** rilascia la risorsa
 - ed eventualmente riporta ready un task in attesa

Luigi Lavazza - Programmazione Concorrente e Distribuita

- 10 -

Problemi tipici e come evitarli



Tipi di semafori

- Contatori
 - Il semaforo viene inizializzato a un valore intero positivo
 - A ogni richiesta viene decrementato
 - Solo quando è zero il thread che fa una richiesta viene bloccato
 - A ogni uscita viene incrementato
 - E si sblocca un thread in attesa (se c'è)
- Binari (alias mutex)
 - Il semaforo può essere solo zero o uno (rosso o verde)
- Java fornisce un semaforo generico n-ario mediante la classe `java.util.concurrent.Semaphore`.
 - Un mutex è un semaforo istanziato con valore 1.



Il codice a prova di race condition

```
import java.util.concurrent.Semaphore;
public class CounterTmp extends Thread {
    private static final Semaphore sem = new Semaphore(1);
    static Counter counter = new Counter();
    public void run() {
        for(int i=0; i<10000; i++) {
            try {
                sem.acquire();
            } catch (InterruptedException e){}
            counter.add(1);
            sem.release();
        }
    }
    public static void main(String args[]) throws InterruptedException {
        CounterTmp p1 = new CounterTmp();
        CounterTmp p2 = new CounterTmp();
        p1.start(); p2.start();
        p1.join(); p2.join();
        System.out.println("Counter = " + counter.count);
    }
}
```

In questo modo non occorre modificare la classe Counter. Ci sarà sempre un solo thread che accede ad add.



Il codice a prova di race condition: alternativa

```
import java.util.concurrent.Semaphore;

public class Counter {
    private final Semaphore sem = new Semaphore(1);
    long count = 0;
    public void add (long value) {
        try {
            sem.acquire();
        } catch (InterruptedException e) { }
        long tmp = this.count;
        tmp = tmp + value;
        this.count = tmp ;
        sem.release();
    }
}
```

In questo modo non occorre modificare i thread. Ci sarà sempre un solo thread nella sezione critica del metodo add.

Una classe di questo genere si chiama «thread safe»



L'uso dei semafori può essere pericoloso

- I programmi non sono tutti semplici come quello che abbiamo visto.
- In un programma complicato può capitare di dimenticare di rilasciare un lock (o di rilasciarlo più di una volta) in qualche condizione.
- Questo porta facilmente a race conditions, deadlock, o altre situazioni scorrette.

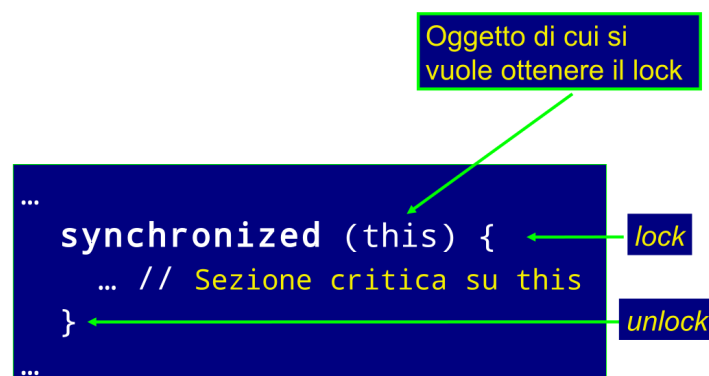


Il modificatore **synchronized**

- Associata **ad ogni istanza** della classe Object c'è un semaforo binario (che verrà indicato come un **lock**).
- Poiché ogni oggetto in Java estende la classe Object, ogni oggetto ha un suo lock (Java Language Specification 17.1)
- La parola chiave **synchronized** applicata ad un metodo o ad una qualunque sezione di codice implica che:
 - ▶ Quando si entra nel metodo/area synchronized si cerca di acquisire il lock associato all'oggetto
 - ▶ Quando si esce dal metodo/area synchronized si rilascia il lock associato all'oggetto, quindi un eventuale thread in attesa può acquisire il lock ed entrare nel metodo/area synchronized
- Quindi, la Race Condition dell'esempio precedente poteva essere facilmente risolta aggiungendo la parola chiave **synchronized** al metodo add. . . e l'oggetto **Semaphore** non era quindi necessario!



Il modificatore **synchronized**



- Se un oggetto ha più di un blocco sincronizzato, comunque uno solo di questi può essere attivo
 - ▶ Perché se un thread detiene il lock, tutti gli altri devono aspettare fuori.



Esempio che usa il modificatore **synchronized**

```
public class Counter {  
    long count = 0;  
    synchronized public void add (long value) {  
        long tmp = this.count;  
        tmp = tmp + value;  
        this.count = tmp ;  
    }  
}
```

In questo modo si controlla l'accesso al metodo add.



Altro esempio di uso del modificatore **synchronized**

```
public class Counter {  
    long count = 0;  
    public void add (long value) {  
        synchronized(this) {  
            long tmp = this.count;  
            tmp = tmp + value;  
            this.count = tmp ;  
        }  
    }  
}
```

In questo modo si controlla l'accesso alla sezione critica dentro al metodo add.



Altro esempio di Race Condition

```
public class AssegnaPostoErrato {
    private int totPostiDisp = 20;

    public boolean assegnaPosti(String cliente, int numPosti) {
        System.out.println("--Richiesta di " + numPosti + " da " + cliente);
        if (numPosti <= totPostiDisp) {
            System.out.println("--Assegna " + numPosti + " a " + cliente);
            totPostiDisp -= numPosti;
            return true;
        }
        return false;
    }

    int totalePosti() {
        return totPostiDisp;
    }
}
```

Sezione critica: si legge totPostiDisp e poi si aggiorna.

- Se più thread eseguono **assegnaPosti()** in parallelo, il numero di posti assegnato alla fine sarà maggiore di quello realmente disponibile! (Race condition)
- Non è detto che il problema si verifichi ad ogni esecuzione (non determinismo)

Luigi Lavazza - Programmazione Concorrente e Distribuita

- 19 -

Problemi tipici e come evitarli



Completiamo l'esempio...

```
public class Richiedente extends Thread {
    private int numPosti;
    private AssegnatorePosto assegnatore;

    public Richiedente(String nome, int numPosti, AssegnatorePosto assegnatore) {
        super(nome);
        this.numPosti = numPosti;
        this.assegnatore = assegnatore;
    }

    public void run() {
        System.out.println("--" + getName() + ": richiede " + numPosti + "...");
        if (assegnatore.assegnaPosti(getName(), numPosti))
            System.out.println("--" + getName() + ": ottenuti " + numPosti + "...");
        else
            System.out.println("--" + getName() + ": posti non disponibili");
    }

    public static void main(String args[]) throws InterruptedException {
        AssegnatorePosto assegnatore = new AssegnatorePosto();
        Richiedente client1 = new Richiedente("cliente1", 3, assegnatore);
        Richiedente client2 = new Richiedente("cliente2", 5, assegnatore);
        Richiedente client3 = new Richiedente("cliente3", 3, assegnatore);
        Richiedente client4 = new Richiedente("cliente4", 10, assegnatore);

        client1.start(); client2.start(); client3.start(); client4.start();
        client1.join(); client2.join(); client3.join(); client4.join();
        System.out.println("Numero di posti ancora disponibili: "
            + assegnatore.totalePosti());
    }
}
```

Luigi Lavazza - Programmazione Concorrente e Distribuita

- 20 -

Problemi tipici e come evitarli



un possibile output corretto

```
-cliente1: richiede 3...
-cliente3: richiede 3...
--Richiesta di 3 da cliente3
---Assegna 3 a cliente3
-cliente3: ottenuti 3...
-cliente2: richiede 5...
--Richiesta di 5 da cliente2
---Assegna 5 a cliente2
-cliente2: ottenuti 5...
--Richiesta di 3 da cliente1
---Assegna 3 a cliente1
-cliente4: richiede 10...
-cliente1: ottenuti 3...
--Richiesta di 10 da cliente4
-cliente4: posti non disponibili
Numero di posti ancora disponibili: 9
```

Luigi Lavazza - Programmazione Concorrente e Distribuita

- 21 -

Problemi tipici e come evitarli



un possibile output sbagliato

```
-cliente1: richiede 3...
-cliente3: richiede 3...
-cliente4: richiede 10...
--Richiesta di 10 da cliente4
---Assegna 10 a cliente4
-cliente4: ottenuti 10...
-cliente2: richiede 5...
--Richiesta di 5 da cliente2
--Richiesta di 3 da cliente3
---Assegna 3 a cliente3
--Richiesta di 3 da cliente1
-cliente3: ottenuti 3...
---Assegna 5 a cliente2
-cliente2: ottenuti 5...
---Assegna 3 a cliente1
-cliente1: ottenuti 3...
Numero di posti ancora disponibili: -1
```

Luigi Lavazza - Programmazione Concorrente e Distribuita

- 22 -

Problemi tipici e come evitarli



Come si corregge l'esempio appena visto?

- Bisogna fare in modo che un thread possa ottenere il lock dell'oggetto **AssegnatorePosto**, delimitando la parte critica con **synchronized**.

```
synchronized public boolean assegnaPosti(String cliente, int numPosti) {
    System.out.println("—Richiesta di " + numPosti + " da " + cliente);
    if (numPosti <= totPostiDisp) {
        System.out.println("—Assegna " + numPosti + " a " + cliente);
        totPostiDisp -= numPosti;
        return true;
    }
    return false; }
```



oppure...

```
public boolean assegnaPosti(String cliente, int numPosti) {
    System.out.println("—Richiesta di " + numPosti + " da " + cliente);
    synchronized(this) {
        if (numPosti <= totPostiDisp) {
            System.out.println("—Assegna " + numPosti + " a " + cliente);
            totPostiDisp -= numPosti;
            return true;
        }
    }
    return false; }
```

In questo caso si blocca solo la sezione critica.



Accesso sincronizzato

- Un metodo **synchronized** può essere eseguito da un solo thread alla volta.
- Non solo: mentre un thread sta eseguendo un metodo **synchronized**, nessun altro thread può eseguire alcun altro metodo **synchronized** dello stesso oggetto.
- I metodi **synchronized** hanno accesso **esclusivo** ai dati incapsulati nell'oggetto solo se a tali dati si accede esclusivamente con metodi **synchronized**
- I metodi non **synchronized** non sono esclusivi e quindi permettono **l'accesso concorrente** ai dati
- Variabili locali
 - ▶ Poiché ogni thread ha il proprio stack, se più thread stanno eseguendo lo stesso metodo, ognuno avrà la propria copia delle variabili locali, senza pericolo di "interferenza"



Come funziona synchronized

- Synchronization is built around an internal entity known as the intrinsic lock or monitor lock.
 - ▶ The API specification often refers to this entity simply as a "monitor."
- **Every object has an intrinsic lock** associated with it.
 - ▶ A thread that needs exclusive and consistent access to an object's fields has to **acquire the object's intrinsic lock** before accessing them, and then **release the intrinsic lock** when it's done with them.
 - ▶ A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock.
 - ▶ As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.
- When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns (even if the return was caused by an uncaught exception).



Come funziona synchronized

- ▶ You might wonder what happens when a **static synchronized method** is invoked, since a static method is associated with a class, not an object.
- ▶ In this case, the thread acquires the intrinsic lock for the Class object associated with the class.
- ▶ Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.



Synchronized e variabili statiche

- Metodi e blocchi **synchronized** non assicurano l'accesso mutuamente esclusivo ai dati "statici"
 - ▶ I dati statici sono condivisi da tutti gli oggetti della stessa classe
- In Java ad ogni classe è associato un oggetto di classe Class
- Per accedere in modo sincronizzato ai dati statici si deve ottenere il lock su questo oggetto di tipo Class:
 - ▶ si può dichiarare un **metodo statico** come **synchronized**
 - ▶ si può dichiarare un **blocco** come **synchronized sull'oggetto di tipo Class**
- Attenzione
 - ▶ Il lock a livello classe non si ottiene quando ci si sincronizza su un oggetto di tale classe e viceversa.



Synchronized e variabili statiche

```
class StaticSharedVariable {
    // due modi per ottenere un lock a livello di classe
    private static int shared;

    public int read() {
        synchronized(StaticSharedVariable.class) {
            return shared;
        }
    }

    public synchronized static void write(int i) {
        shared = i;
    }
}
```

Il parametro indica che si usa l'intrinsic lock della classe

Usa l'intrinsic lock della classe



Ereditarietà e **synchronized**

- La specifica **synchronized** non fa propriamente parte della segnatura di un metodo
- Quindi una classe derivata può ridefinire un metodo **synchronized** come non **synchronized** e viceversa
- Il fatto che **synchronized** non faccia parte della segnatura è molto comodo, perché ci consente di
 1. Definire classi adatte all'uso sequenziale senza preoccuparci dei problemi della concorrenza, e
 2. Poi modificare queste classi derivando sottoclassi che vengono rese adatte all'uso concorrente mediante **synchronized**.



Ereditarietà e **synchronized**

- Si può quindi ereditare da una classe “non sincronizzata” e ridefinire un metodo come **synchronized**, che richiama semplicemente l'implementazione della superclasse

```
public class AssegnatoreConcorrente extends AssegnatoreSequenziale {  
    synchronized public boolean assegnaPosti(String cliente, int numPosti) {  
        System.out.println("Assegnatore derivato");  
        return super.assegnaPosti(cliente,numPosti);  
    }  
}
```

- Questo assicura che gli accessi ai metodi nella sottoclasse avvengano in modo sincronizzato.



Cooperazione fra thread

- Tramite la sincronizzazione un thread può modificare in modo sicuro dei valori che potranno essere letti da un altro thread.
- Ma come fa l'altro thread a sapere che i valori sono stati modificati/aggiornati?
- Può un thread continuare a ciclare sul valore di un oggetto condiviso fino a quando questo non cambia valore?
... direi di no!
- Usare solo parti di codice **synchronized** non è sufficiente: servono dei meccanismi che mettano in attesa un thread e che lo risvegliano quando le condizioni sono cambiate



Il metodo **wait** (classe Object)

- Un Thread può chiamare il metodo **wait** all'interno di un metodo sincronizzato, cioè quando detiene un lock
 - ▶ chiamare **wait** in un contesto diverso genera un errore
- Quando un thread chiama **wait** va in attesa e rilascia il lock sull'oggetto



Luigi Lavazza - Programmazione Concorrente e Distribuita

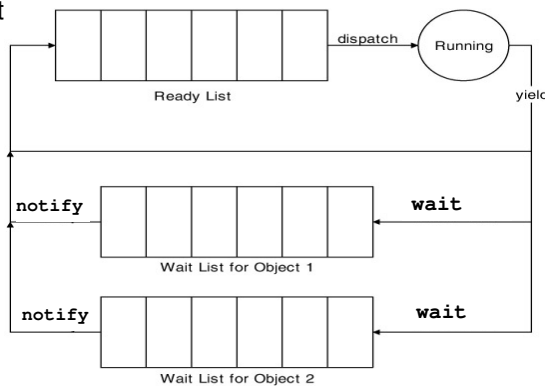
- 33 -

Problemi tipici e come evitarli



Il metodo **wait** (classe Object)

- La SVM mantiene un elenco di tutti i thread che sono pronti per essere eseguiti ("Ready List")
- Quando un thread va in attesa (**wait**), la SVM lo sposta in una "Wait List" e rilascia il lock sull'oggetto
- Quando un thread in attesa viene svegliato (mediante **notify**) la SVM lo sposta nella Ready List



Luigi Lavazza - Programmazione Concorrente e Distribuita

- 34 -

Problemi tipici e come evitarli



I metodi `notify` e `notifyAll`

- Una chiamata `notify` sposta un thread qualsiasi dal wait set di un oggetto al ready set.
 - ▶ La scelta del thread da spostare è non deterministica
- La chiamata a `notifyAll` muove tutti i thread dal wait set di un oggetto al ready set.
- Esempi:
 - ▶ Se un insieme di thread attendono la fine di un determinato compito, una volta che l'operazione è terminata si può usare `notifyAll()`
 - ▶ Se un insieme di thread attendono di entrare in un blocco esclusivo, solo uno dei thread in attesa può accedere dopo il suo risveglio: in tal caso, si preferisce la `notify()`.



Monitor

- Monitor
 - ▶ Una struttura dati con **tutti i metodi synchronized** in modo che, in un dato momento, un solo thread può essere eseguito in *qualsiasi* metodo
- In Java, se un thread `t1` entra in un metodo `synchronized ms1` di un determinato oggetto `o1`, nessun altro thread potrà entrare in **alcun** metodo sincronizzato dell'oggetto `o1`, sino a quando `t1` non avrà terminato l'esecuzione del metodo `ms1` (ovvero non avrà abbandonato il monitor dell'oggetto).



Semafori e Monitor in Java

- Il monitor è un costrutto di sincronizzazione di un linguaggio di alto livello e corrisponde ad un blocco di mutua esclusione per i thread (solo un thread può “entrare” in un monitor in un determinato istante)
- Semafori e monitor sono equivalenti.
 - ▶ Si possono definire semafori usando monitor
 - ▶ Si possono creare monitor usando semafori
- Java non fornisce i semafori come costrutto di base del linguaggio, anche se è dotato di una classe
`java.util.concurrent.Semaphore`



Implementazione di un monitor usando semafori

- Per trasformare una qualunque classe in un monitor basta
 - ▶ Includere un semaforo, in modo che ogni istanza abbia il suo semaforo
 - ▶ Mettere una **acquire** all'inizio di ogni metodo
 - ▶ Mettere una **release** alla fine di ogni metodo
- In Java dichiarare i metodi come **synchronized** fa esattamente ciò.
 - ▶ Si acquisisce e si rilascia il lock intrinseco dell'oggetto (che non a caso viene chiamato monitor)



Implementare un semaforo con un monitor

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

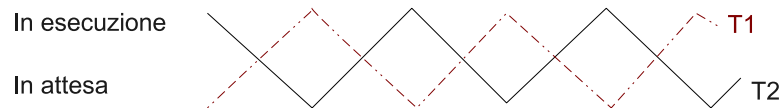


PROBLEMI VARI (COMPRESO DEADLOCK) E LORO PREVENZIONE



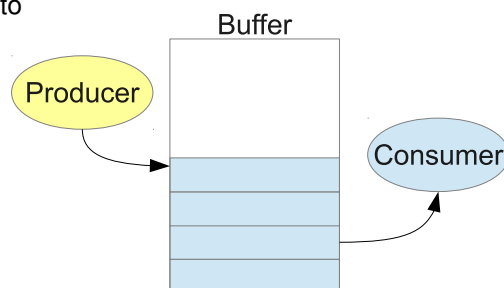
Il problema della coordinazione di più Thread

- Un problema che si verifica frequentemente in programmazione concorrente è l'esigenza di coordinare le attività dei Thread.
- Ad esempio:
 - ▶ un programma può richiedere a due o più thread un'esecuzione di un determinato compito a turno.



Problema del buffer limitato o problema del produttore-consumatore

- Un thread (il produttore deposita dati in una zona di memoria condivisa (buffer))
- Un altro thread preleva i dati dal buffer
- Il buffer è (ovviamente) limitato

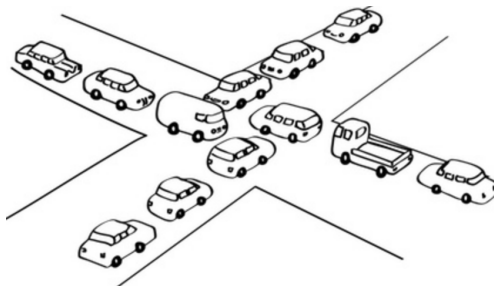


- il problema è assicurare che
 - ▶ il produttore non cerchi di inserire nuovi dati quando il buffer è pieno
 - ▶ il consumatore non cerchi di estrarre dati quando il buffer è vuoto.



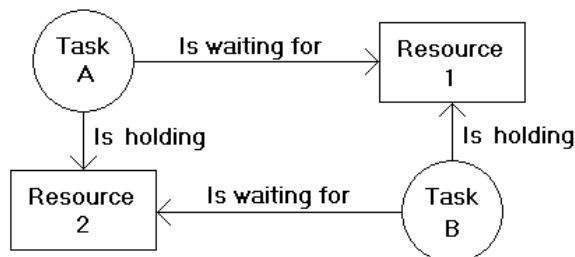
Deadlock (stallo)

- Quando ad un processo viene garantito l'accesso esclusivo (ad esempio tramite una mutua esclusione) ad una risorsa, possono crearsi ugualmente situazioni problematiche di stallo



Deadlock (stallo)

Hold & Wait Deadlock



- I due thread A e B sono in deadlock perché ognuno attende un evento che può avvenire soltanto tramite l'altro.
- Essendo tutti i thread in attesa, nessuno potrà mai creare l'evento di sblocco, quindi l'attesa si protrae all'infinito.



Condizioni necessarie perché si verifichi un Deadlock

- Ci sono quattro condizioni necessarie affinché un deadlock si verifichi. Queste sono generalmente espresse in termini di risorse assegnate a un thread.
 - ▶ **Mutual exclusion** - solo un'attività concorrente per volta può utilizzare una risorsa (cioè, la risorsa non è condivisibile simultaneamente).
 - ▶ **Hold and wait** - devono esistere attività concorrenti che sono in possesso di risorse mentre stanno aspettando altre risorse da acquisire.
 - ▶ **No preemption sulle risorse** - una risorsa può essere rilasciata solo volontariamente (non tolta con la forza) da un'attività concorrente.
 - ▶ **Circular wait** - deve esistere una catena circolare di attività concorrenti tale che ogni attività mantiene bloccate delle risorse che contemporaneamente vengono richieste dai Thread successivi.



Deadlock

- Un Deadlock può verificarsi in qualsiasi programma concorrente.
- La principale causa di deadlock è l'uso di circolarità nel lock degli oggetti che può portare ad una situazione in cui nessun thread può procedere, e il sistema è in stallo.
- Lo stallo circolare si verifica se esiste un gruppo di thread $\{t_0, t_1, \dots, t_n\}$ per cui t_0 è in attesa per una risorsa occupata da t_1 , t_1 per una risorsa di t_2 , ecc. t_n per una risorsa di t_0 .
- L'esempio che segue mostra un Deadlock circolare dove i due thread bloccano gli Objects A e B in ordine opposto.



Esempio di deadlock

```
import java.util.concurrent.ThreadLocalRandom;
class LockObjects implements Runnable{
    private Object a,b;
    public LockObjects(Object o1, Object o2){
        this.a=o1; this.b=o2;
    }
    public void run(){
        try{
            Thread.sleep(ThreadLocalRandom.current().nextInt(10, 100));
            System.out.println("In run");
            synchronized(a) {
                System.out.println("First item locked");
                Thread.sleep(ThreadLocalRandom.current().nextInt(10,100));
                synchronized(b) {
                    System.out.println("Lock worked"+a.toString()+
                        ", "+b.toString());
                }
            }
        } catch (InterruptedException e) { }
    }
}
```

Luigi Lavazza - Programmazione Concorrente e Distribuita

- 47 -

Problemi tipici e come evitarli



Esempio di deadlock

```
import java.util.concurrent.ThreadLocalRandom;

public class MakeDeadlock {
    public static void main(String arsg[]){
        Object a = new Object();
        Object b = new Object();
        Thread t1=new Thread(new LockObjects(a, b));
        Thread t2=new Thread(new LockObjects(b, a));
        t1.start(); t2.start();
    }
}
```

Prima lock su a,
poi su b

Prima lock su b,
poi su a

Luigi Lavazza - Programmazione Concorrente e Distribuita

- 48 -

Problemi tipici e come evitarli



Esempio di deadlock

- Possibili esecuzioni:

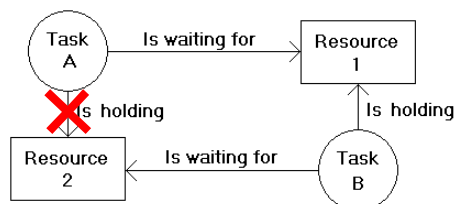
```
In run
First item locked
Lock worked java.lang.Object@5ef4b65d, java.lang.Object@13f0c45f
In run
First item locked
Lock worked java.lang.Object@13f0c45f, java.lang.Object@5ef4b65d
```

```
In run
First item locked
In run
First item locked
<deadlock>
```



Modi per evitare il Deadlock

- Esistono alcuni possibili approcci per affrontare le situazioni di Deadlock.
- Deadlock prevention** - il Deadlock può essere evitato se si fa in modo che almeno una delle quattro condizioni richieste per deadlock (Mutual exclusion, Hold and wait, No preemption e Circular wait) non si verifichi mai
- Deadlock removal** – non si previene il deadlock, ma lo si risolve quando ci si accorge che è avvenuto.
 - Ad es. rendendo le risorse preemptible





Deadlock prevention: esempi

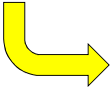
- No hold and wait:

- fare in modo che mai si possa detenere una risorsa mentre si è in attesa di un'altra risorsa

```

synchronized(a) {
    Thread.sleep(ThreadLocalRandom.current().nextInt(10,100));
    synchronized(b) {
        System.out.println(...);
    }
}

```



```

synchronized(a) {
    Thread.sleep(ThreadLocalRandom.current().nextInt(10,100));
}
synchronized(b) {
    System.out.println(...);
}

```

- Non è sempre possibile (ad es., se devo fare un'elaborazione non interrompibile che coinvolge sia a sia b)



Deadlock prevention: esempi

- No circolarità

- Si ordinano le risorse, e si richiede il lock seguendo l'ordine. Ad es., se A precede B, bisogna cercare di acquisire B solo se si detiene già A.
- Se tutti i thread si attengono a questa disciplina, non si possono creare deadlock circolari.
- Problema: posso scoprire che mi serve a (che precede b) solo dopo che ho acquisito b e fatto un po' di elaborazioni.