



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](mailto:luigi.lavazza@uninsubria.it)

---

# SINCRONIZZAZIONE FRA THREAD



# Sincronizzazione fra thread

---

- Spesso occorre che un Thread si fermi in attesa del verificarsi di una data condizione.
- Se il soddisfacimento della condizione dipende da un altro Thread, si ottiene una sincronizzazione tra thread.
- A questo scopo, servono dei **meccanismi che mettano in attesa un thread e che lo risvegliano quando le condizioni sono cambiate**



# Sincronizzazione fra thread

---

- Meccanismi:
  - ▶ Un thread si mette in attesa mediante la chiamata **wait**.
  - ▶ Una volta in wait, solo un altro thread lo può sbloccare.
  - ▶ Un thread può sbloccare un altro thread mediante la chiamata di **notify**.
- NB: sia **wait** sia **notify** sono metodi della classe **Object**.



## Il metodo `wait` (classe `Object`)

---

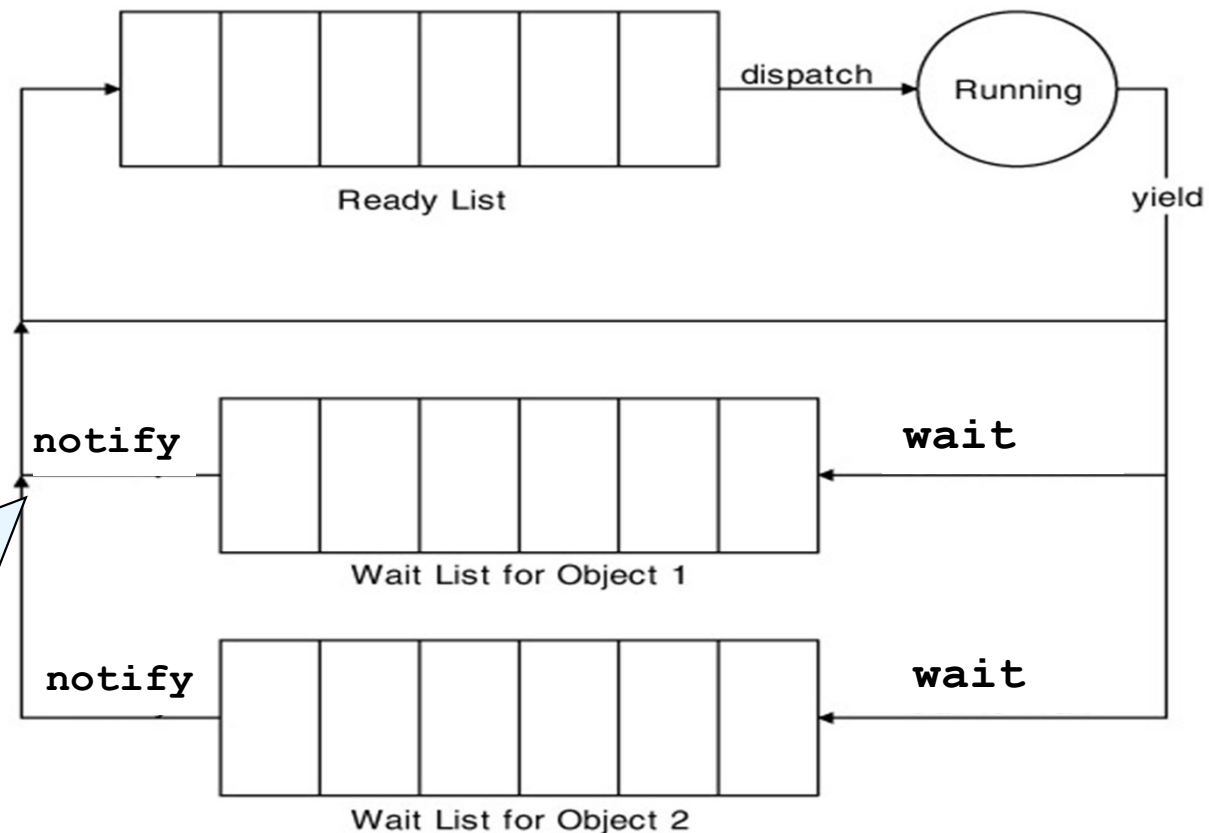
- Un Thread può chiamare il metodo `wait` all'interno di un metodo sincronizzato, cioè quando detiene un lock
  - ▶ chiamare `wait` in un contesto diverso genera un errore
- Quando un thread chiama `wait` va in attesa e rilascia il lock sull'oggetto

```
synchronized(this) {  
    // il thread detiene il lock sull'oggetto  
    try {  
        wait(); // il thread rilascia il lock sull'oggetto  
    } catch (InterruptedException e) {}  
    // il thread detiene il lock sull'oggetto  
    ...  
} // il thread rilascia il lock sull'oggetto
```

# 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

Un thread esce dallo stato waiting **solo** in conseguenza di una `notify` (o `notifyAll`), che può essere causata solo da un altro thread.





# 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.
- Come `wait`, anche `notify` deve essere eseguita in un blocco di codice `synchronized`.

# IMPORTANTE

---

- La **wait** sta dentro un blocco **synchronized** e rilascia il lock.
- Di conseguenza, rende possibile corse critiche.
- Per evitare problemi
  - ▶ Effettuare **wait** prima di fare qualunque modifica ai dati condivisi
  - ▶ Effettuare **notify** subito prima di uscire dalla zona critica
    - E comunque dopo aver fatto modifiche ai dati condivisi



# Monitor

---

- Se un thread  $t_1$  entra in un metodo **synchronized**  $ms_1$  di un determinato oggetto  $o_1$ , nessun altro thread potrà entrare in **alcun** metodo **sincronizzato** dell'oggetto  $o_1$ , sino a quando  $t_1$  non avrà terminato l'esecuzione del metodo  $ms_1$  (ovvero non avrà rilasciato il lock dell'oggetto).
- Ma ricordate che mentre il thread  $t_1$  sta eseguendo il metodo  $ms_1$  di  $o_1$ , un altro thread può eseguire un metodo **NON sincronizzato** di  $o_1$ .
- Monitor
  - ▶ Un oggetto con **tutti i metodi synchronized**, in modo che, in un dato momento, un solo thread può essere eseguito in qualsiasi metodo dell'oggetto



# Semafori e Monitor in Java

---

- Il monitor è un costrutto di sincronizzazione di molti linguaggi di alto livello
- Realizza la mutua esclusione per i thread
  - ▶ In qualunque momento solo un thread può eseguire in un monitor
- In Java per ottenere una classe monitor basta dichiarare **tutti i metodi synchronized**
  - ▶ perché sappiamo che in tal modo, in qualunque momento, un solo thread può essere eseguito in qualsiasi metodo dell'oggetto
- Semafori e monitor sono equivalenti.
  - ▶ Si possono definire semafori usando monitor
  - ▶ Si possono creare monitor usando semafori



# 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

---

- Un semaforo fa sostanzialmente due cose:
  - ▶ **acquire**: si acquisisce il lock o ci si blocca
  - ▶ **release**: si rilascia il lock, eventualmente svegliando un thread bloccato sul semaforo.
- **wait** e **notify** fanno esattamente lo stesso, ma sul lock intrinseco dell'oggetto, invece che su un semaforo qualunque.



# Implementare un semaforo con un monitor

---

```
public class MySemaphore {  
    private int value;  
    MySemaphore (int init) {  
        value = init;  
    }  
    synchronized public void release() {  
        value++;  
        notify();  
    }  
    synchronized public void acquire()  
        throws InterruptedException {  
        while (value == 0)    // NB: while, non if!  
            wait();  
        value--;  
    }  
}
```



# Proviamo il nostro semaforo

---

- Nell'esempio di race condition precedente



# RaceExample

---

```
public class RaceExample extends Thread {
    Counter myCounter;
    public RaceExample(Counter c) {
        myCounter=c;
    }
    public void run() {
        for(int i=0; i<100000; i++)
            myCounter.add(1);
    }
    public static void main(String args[])
        throws InterruptedException {
        Counter counter = new Counter();
        RaceExample p1 = new RaceExample(counter);
        RaceExample p2 = new RaceExample(counter);
        p1.start();  p2.start();
        p1.join();  p2.join();
        System.out.println("Counter = " + counter.count);
    }
}
```



# Counter

---

```
public class Counter {  
    long count = 0;  
    MySemaphore semaphore;  
    Counter() {  
        semaphore=new MySemaphore(1);  
    }  
    public void add (long value) {  
        try {  
            semaphore.acquire();  
        } catch (InterruptedException e) { }  
        this.count += value ;  
        semaphore.release();  
    }  
}
```





# Esecuzione

---

- Al termine dell'esecuzione otteniamo sempre  
**Counter = 200000**



# Ci vuole davvero while intorno a wait?

---

```
public class MySemaphore {  
    private int value;  
    MySemaphore (int init) {  
        value = init;  
    }  
    synchronized public void release() {  
        value++;  
        notify();  
    }  
    synchronized public void acquire()  
        throws InterruptedException {  
        if (value == 0)    // !!!  
            wait();  
        value--;  
    }  
}
```



# Esecuzione

---

- Al termine dell'esecuzione possiamo ottenere  
**Counter = 199998**
- Cosa è andato storto?

# Risveglio da **wait**

---

- Il thread **t** trova una condizione falsa e quindi fa una **wait** e va nello stato waiting (cioè blocked in attesa di evento **notify**)
- Quando qualcuno fa **notify**, **t** diventa ready, ma non va necessariamente in esecuzione.
- Mentre **t** è ready, qualche altro thread può fare modifiche che rendono la condizione nuovamente falsa.
- Quando **t** diventa running, deve ricontrollare nuovamente la condizione. Non può dare per scontato che sia ancora vera.



# Una possibile sequenza problematica

---

- p2 fa **wait** perché ha trovato **value** nullo
- p1 incrementa **value** e fa **notify**.
  - ▶ p2 diventa ready
- Lo scheduler lascia in esecuzione p1, che fa una **acquire**, rimettendo **value** a zero
- Lo scheduler manda in esecuzione p2
- p2 esegue **value--**, quando **value** è nullo.



## Stessa sequenza, con while

---

- p2 fa **wait** perché ha trovato **value** nullo
- p1 incrementa **value** e fa **notify**. p2 diventa ready, pronto a eseguire l'istruzione che segue **wait**
- Lo scheduler lascia in esecuzione p1, che fa una **acquire**, rimettendo **value** a zero
- Lo scheduler manda in esecuzione p2
- p2 torna a eseguire il test **value==0**, lo trova vero e quindi fa un'altra **wait**.

# Proprietà del ciclo `while(condizione){wait()}`

---

- L'istruzione che viene dopo il ciclo viene eseguita sempre solo dopo aver valutato la condizione e averla trovata falsa.
- Questo garantisce che la condizione è ancora falsa quando si esegue la prima istruzione dopo il ciclo
- Perché stiamo eseguendo un metodo **synchronized**, e se non facciamo **wait** (e non la facciamo, essendo la condizione falsa) nessuno può aver modificato l'oggetto.
- NB: la condizione deve riguardare unicamente lo stato dell'oggetto.

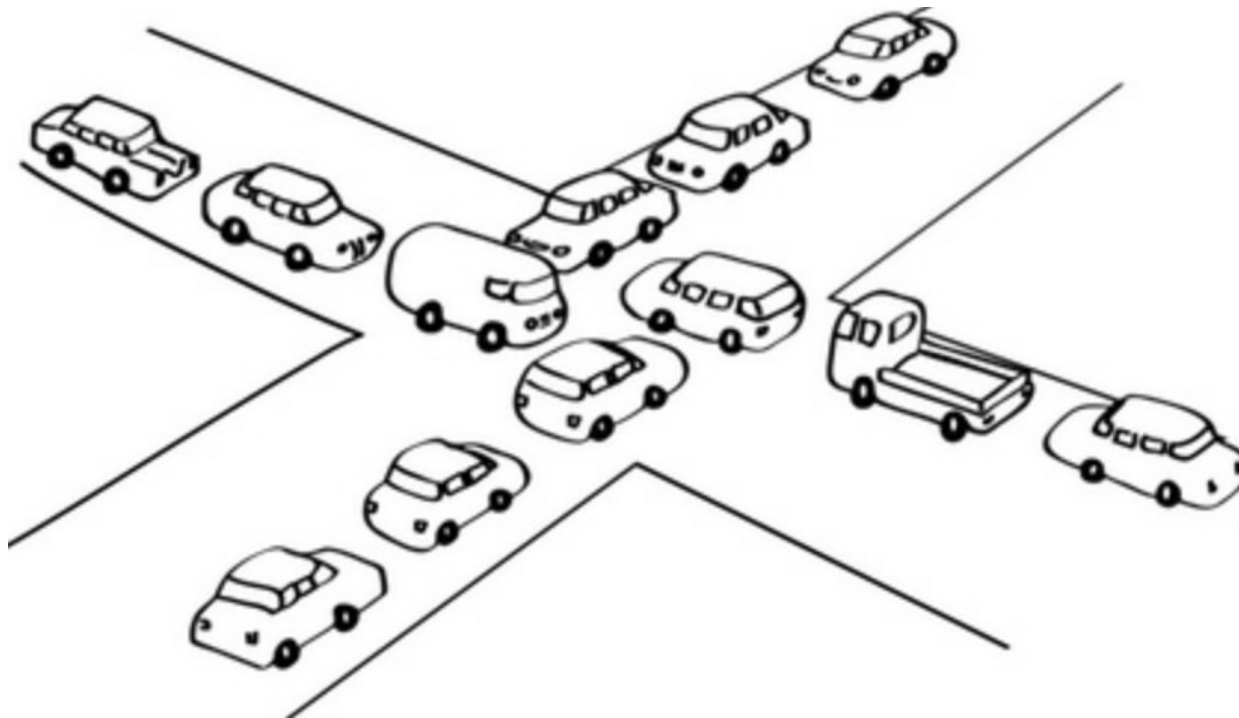
```
while (condizione)
    wait() ;
... ;
```

# **IL PROBLEMA DEL DEADLOCK E COME RISOLVERLO**



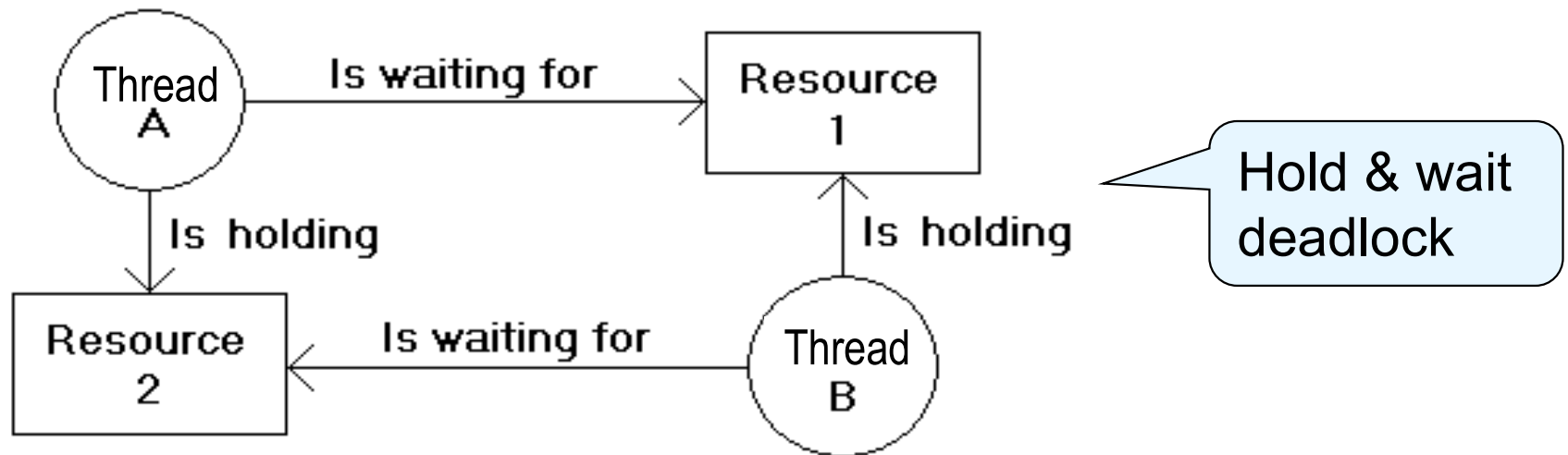
# Deadlock (stallo)

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



# Deadlock (stallo)

- Entrambi i thread A e B hanno bisogno di entrambe le risorse 1 e 2.
- Quando un thread ha ottenuto entrambe le risorse potrà fare il suo lavoro e infine rilasciarle.
- Thread A ha ottenuto la risorsa 2 e ha bisogno di acquisire la risorsa 1.
- Thread B ha ottenuto la risorsa 1 e ha bisogno di acquisire la risorsa 2.



- I due thread 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

---

- Il 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 primo, secondo;
    public LockObjects(Object o1, Object o2){
        this.primo=o1; this.secondo=o2;
    }
    public void run(){
        try{
            Thread.sleep(ThreadLocalRandom.current().nextInt(10,100));
            System.out.println("In run");
            synchronized(primo) {
                System.out.println("First item locked");
                Thread.sleep(ThreadLocalRandom.current().nextInt(10,100));
                synchronized(secondo) {
                    System.out.println("Lock worked "+primo.toString()+
                                         ", "+secondo.toString());
                }
            }
        } catch (InterruptedException e) { }
    }
}
```

NB: **synchronized**  
annidate!



# 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();
```

```
    }
```

```
}
```

t1 farà prima lock su a, poi su b

t2 farà prima lock su b, poi su a



# 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

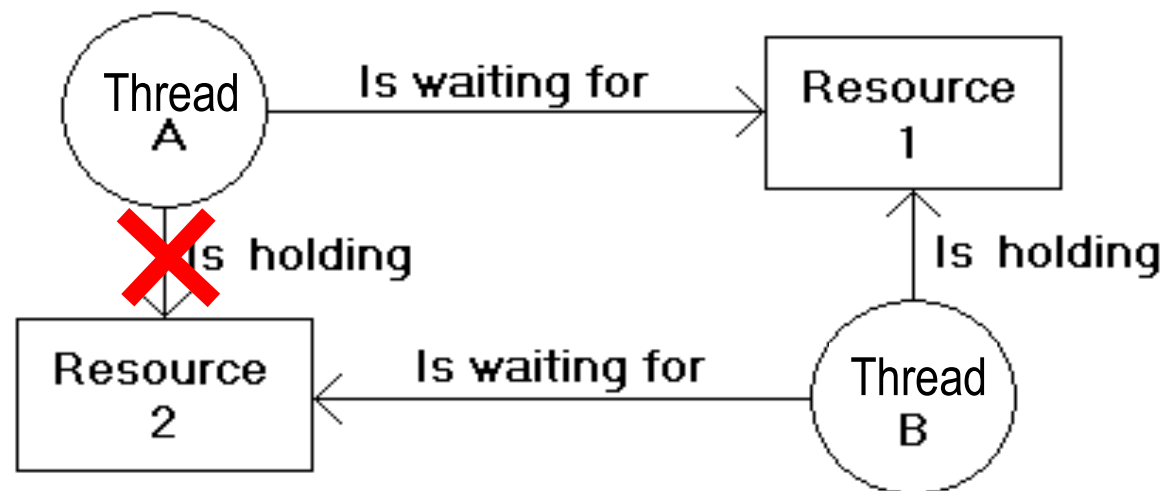
In run

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

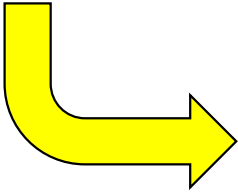




# Deadlock prevention: esempi

- 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 (o se non si ha bisogno di A, che quindi non richiederemo).
  - ▶ 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.
    - In questo caso dovrei ripristinare la situazione di partenza, rilasciare le risorse che ho acquisito (B) e ricominciare da capo dopo aver acquisito tutte le risorse che mi servono (A e B).