Programmazione Concorrente e multithread
Parte 2

Emanuele Ing. Benatti

Come “dialogano” i Thread: Condivisione di memoria

Condivisione di un oggetto fra due thread:

```
int[ ] x = new int[] { 1,2,3,4,5,6,7,8 }; // oggetto condiviso
MioThread t1 = new MioThread( x );
MioThread t2 = new MioThread( x );
t1.start();
t2.start();
// ora sia t1 che t2 possono "lavorare" con lo stesso oggetto x
```

x è fisicamente lo stesso oggetto e non viene “copiato” nei due Thread.

Come “dialogano” i Thread: Condivisione di memoria

Due o più thread dello stesso processo possono facilmente condividere una zona di memoria, tramite l'uso di variabili condivise

Per creare una variabile condivisa tra più thread in Java è sufficiente creare un oggetto (ad esempio un array) prima della creazione dei thread, e passare poi tale oggetto ai thread prima di avviarli (ad esempio nel loro costruttore)

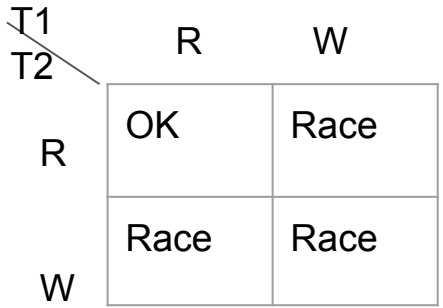
In questo modo ogni thread riceve un riferimento allo stesso oggetto (che quindi è presente una sola volta in memoria)

Accedendo all'oggetto, i thread potranno leggere/scrivere la **stessa area di memoria (--> memoria condivisa)**

Come “dialogano” i Thread: Condivisione di memoria

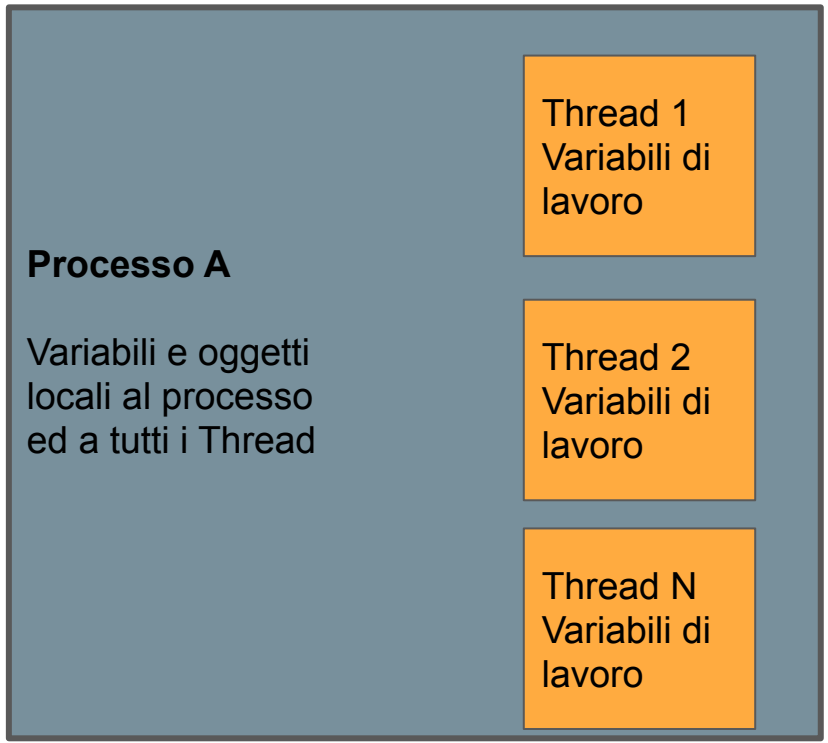
Se i thread effettuano solamente accessi in lettura sulla variabile condivise, non vi sono problemi

Se i thread effettuano anche accessi in scrittura (cioè modificano la variabile condivisa) occorre prestare particolare attenzione nella scrittura del codice, poiché potrebbe presentarsi il problema della "corsa critica" (race condition)

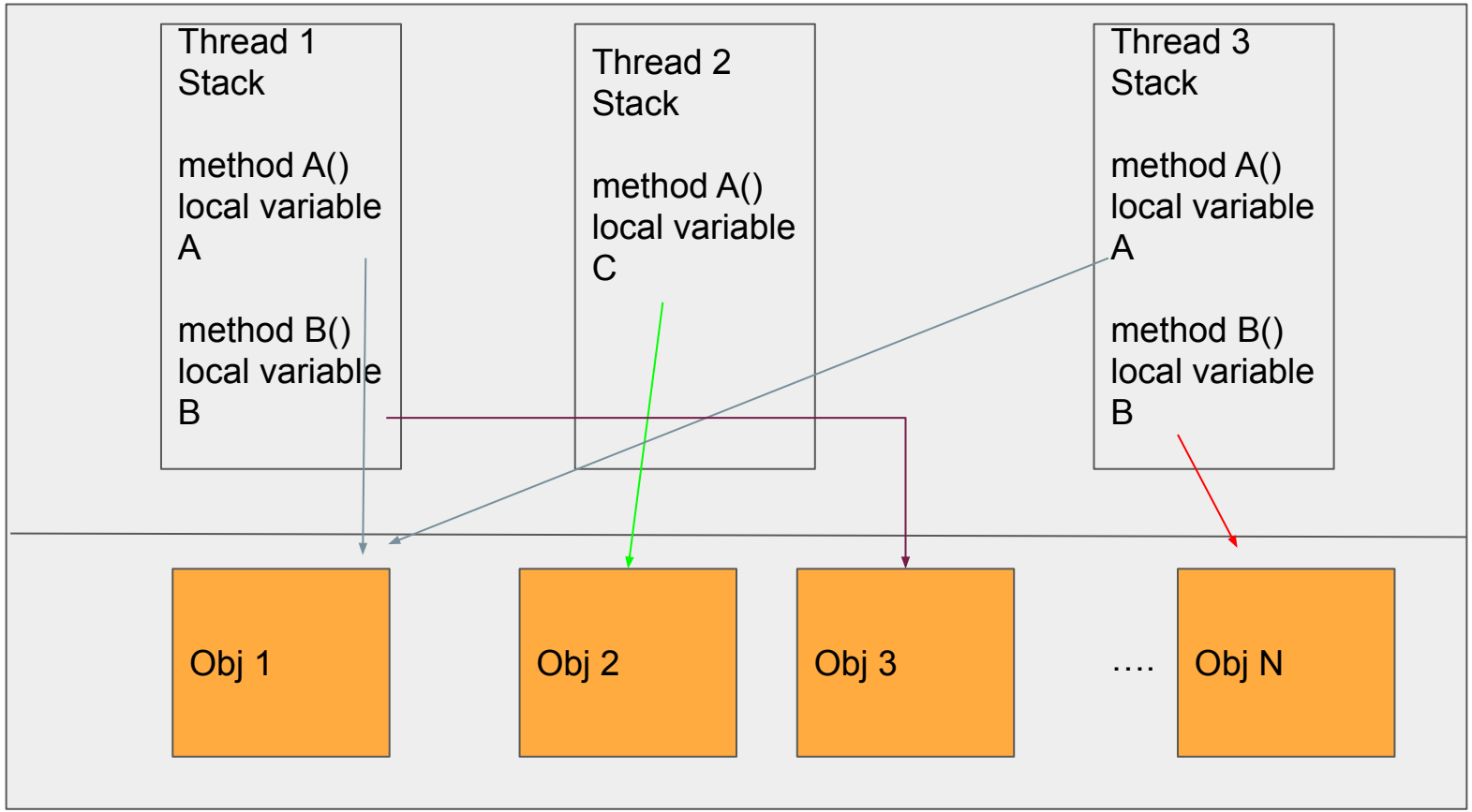


Una corsa critica succede quando due processi provano a modificare nello stesso momento la stessa area di memoria.

Come “dialogano” i Thread: Condivisione di memoria



Come “dialogano” i Thread: Condivisione di memoria



Corsa critica

Una corsa critica si sviluppa quando due (o più) thread tentano di modificare lo stesso oggetto nel momento in cui, mentre almeno un altro thread non ha terminato di eseguire le operazioni.

- Esempio: supponiamo che i thread T1 e T2 tentino di incrementare "contemporaneamente" una variabile condivisa n
- Se n=0 prima dei due incrementi, ci aspettiamo che n valga 2 dopo la modifica

Invece può accadere che n valga 1 al termine delle due istruzioni!

Esempio di Corsa critica

Abbiamo un conto corrente senza soldi. Vorrei che un Thread depositi 200 € un altro prelevi, se possibile 100€

Se eseguiamo "contemporaneamente" entrambi i Thread può capitare sia che il deposito funzioni (saldo 100€) che il saldo finale sia di 200€ in quanto il deposito non è andato a buon fine.

In definitiva: il programma ha una corsa critica e "non posso prevedere" il suo funzionamento

Corsa critica

Una corsa critica si sviluppa quando due (o più) thread tentano di modificare lo stesso oggetto nel momento in cui, mentre almeno un altro thread non ha terminato di eseguire le operazioni.

- Esempio: supponiamo che i thread T1 e T2 tentino di incrementare "contemporaneamente" una variabile condivisa n
- Se n=0 prima dei due incrementi, ci aspettiamo che n valga 2 dopo la modifica

Invece può accadere che n valga 1 al termine delle due istruzioni!

Esempio

N=0

Ho due Thread
T1 esegue N=N+2

T2 esegue N=N+3

Eseguo T1 e T2 in sequenza

Il risultato di questa esecuzione è 2,3 o 5 in modo quasi casuale, perchè?

Esempio

Perchè a livello di istruzioni macchina N=N+2 è composta da 3 istruzioni:

- load N,2
- execute add N,2
- store mov N

ed N=N+3

- load N,3
- execute add N;3
- store N=

Si genera una race condition quando l'esecuzione di una istruzione prevede almeno un'istruzione macchina su dati comuni (condivisi)

Esempio

Perchè a livello di istruzioni macchina N=N+2 è composta da 3 istruzioni:

- load N,2
- execute add N,2
- store mov N

ed N=N+3

- load N,3
- execute add N;3
- store N=

In alcune architetture potrebbero esserci anche più istruzioni, dipendenti da quante istruzioni macchina vengono realizzate per eseguire le singole operazioni elementari.

Race condition

Un programma contiene una race condition quando il risultato finale dell'elaborazione (output) è soggetto a variazioni casuali poiché dipende dalla particolare sequenza temporale con cui alcune istruzioni concorrenti sono eseguite.

Chi decide la sequenza dei Thread e dei processi?

Race condition

Si definisce (race) perchè senza alcun meccanismo di sincronia /thread più veloce accede prima ed esegue prima le istruzioni macchina sui dati condivisi.
Senza alcun controllo, le istruzioni portano ad una casualità nei programmi difficilmente gestibile che porta gravi malfunzionamenti potenziali.

Anche gli output e la gestione di dati condivisi nel modo sbagliato può portare a inconsistenze e anomalie (vedi proprietà acid informatica)

In altre parole, l'output dipende da quale thread vince la "corsa" verso l'istruzione critica. Poichè l'ordine esatto di esecuzione delle istruzioni macchina non può essere stabilito a priori, il programma produce, a parità di input, risultati casuali.

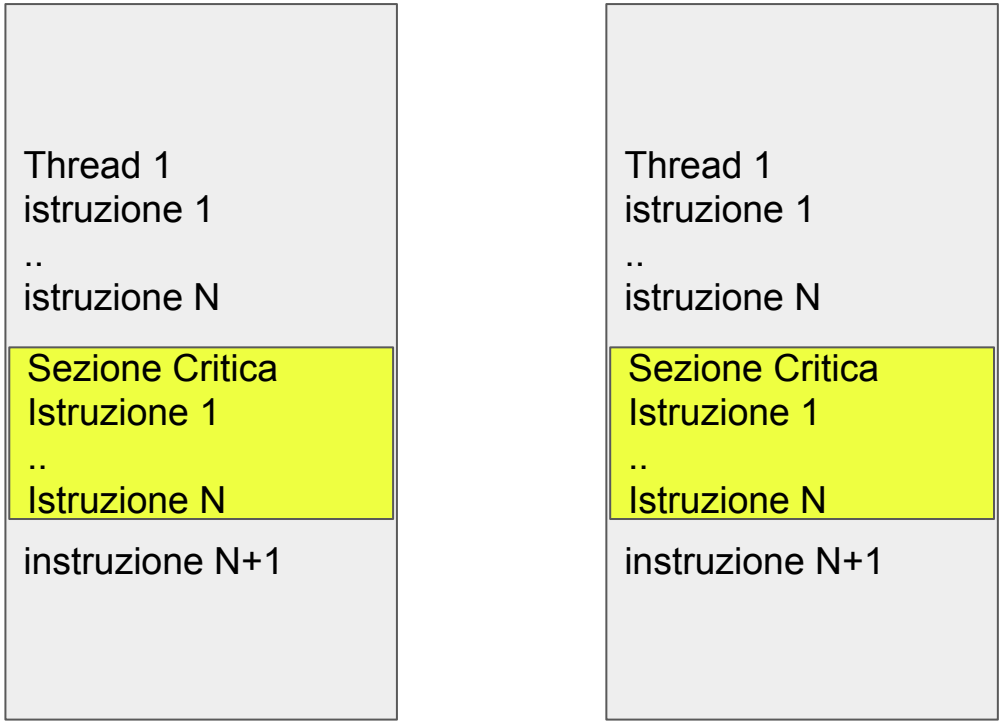
Soluzione: sezione critica

Una sezione critica è una porzione di codice del programma che non deve essere eseguita in modo concorrente da parte di più thread o processi.

Cioè

Una sezione critica è una parte di codice che può essere eseguita da un solo thread o un processo contemporaneamente.

Soluzione: sezione critica



Soluzione: sezione critica

Nella sezione critica avviene soltanto (ed esclusivamente) l’accesso alla memoria condivisa

MA

Non tutte il thread deve avere istruzioni devono essere nella sezione critica.

Esempio di Sezione critica

```
public class esempio{

public synchronized void metodo_1(){
}


public synchronized void metodo_2(){
}


public  void metodo_3(){
}
```

La parola chiave syncronized crea la sezione critica all'interno del metodo: tutto ciò che c'è all'interno viene visto come un'operazione atomica. In questo caso l'esecuzione di metodo 1 e metodo 2 è atomica mentre metodo 3 è libera.

Esempio di Sezione critica

```
public class esempio{

public synchronized void metodo_1(){
}


public synchronized void metodo_2(){
}


public  void metodo_3(){
}
```

Ogni metodo può essere reso atomico con synchronized n qualsiasi classe e/o ogni metodo non synchronized può chiamare oggetti synchronized.

Sezione critica sugli oggetti

Quanto detto su synchronized si applica a livello di oggetto, non di classe.
Nessun Thread può far partire un altro metodo synchronized sullo stesso oggetto. Se in un altro thread si crea un altro oggetto esso non influisce sugli altri oggetti della stessa classe.

Quindi dati due oggetti di esempio x=new esempio() e y.=new esempio();

x.metodo1() y.metodo2() //possono andare in parallelo

x.metodo1() x.metodo2() //sono in concorrenza!

Esempio corsa critica

Scrivi un programma che crei due thread condivisi della classe Counter.
Ogni thread incrementa 10000 volte il contatore.
Che cosa fa questo programma?

```
class Counter {
private int count = 0 ;
public void increment( ) {
count++;
  }
public int getCount() {
return count;
  }
}
```

Esempio corsa critica

Il programma alla fine stamperà un valore sempre diverso , ma certamente non 20000, questo è causato dal fatto che la funzione incrementa anche se ha una sola istruzione non è di fatto atomica.
In secondo luogo perchè la JVM potrebbe cambiare l'ordine di esecuzione.

Bisogna applicare synchronized al metodo e diventando atomico viene corretto.

Si dice in ultima analisi che l'accesso alla risorsa condivisa crea una race condition.

Classi Thread Safe

Nella teoria della programmazione concorrente si indica come thread safe per indicare la caratteristica di una porzione di codice che si comporta in modo corretto nel caso di esecuzioni multiple da parte di più thread.

I nostri programmi concorrenti che scriviamo devono essere Thread Safe.

Le classi di base (arrayList, HashSet, HashMap non sono thread safe, e di base possono avere problemi di corsa critica).

Il motivo è che la sincronia potrebbe peggiorare le prestazioni per programmi non paralleli (cioè tutti quelli che non usano le classi Thread, Semaphore e Syncronized), quindi si preferisce lasciare le classi di base “semplici”

ArrayList non è thread-safe

Nell’implementazione di ArrayList certamente il metodo add è una cosa simile a questa

```
class Object[] arr;
private int size=0;

public void add(Object x){
…
  arr[size]=x;
  size++;

…
}
```

add ha almeno una operazione all’interno e pertanto add non è atomica certamente. Eventuali altre istruzioni comunque non rendono atomica l’esecuzione se più thread modificano una lista. Per questo motivo non è thread safe.

ArrayList thread-safe

Si può rendere ThreadSafe ridefinendo la classe con Synchronized

```
class MiaLista {
private ArrayList arrList;
public synchronized void add(Object x) {
arrList.add(x);
  }
}
```

Oppure usare List list = Collections.syncronizedList (new ArrayList());

Che rende Thread safe una lista qualsiasi.

Semafori

Nella teoria della programmazione concorrente un semaforo è una variabile intera su cui è possibile fare due operazioni:

UP: incrementa di uno il valore della semaforo,

DOWN: decrementa di uno il valore del semaforo se il valore è maggiore di zero, altrimenti si mette in attesa che diventi maggiore di zero e lo decrementa

Semafori e risorse condivise

Nella teoria della programmazione concorrente un semaforo permette l'accesso di risorse condivise, tra thread o processi.

UP: incrementa di uno il valore della semaforo,
UP Significa rilascio della risorsa condivisa

DOWN: decrementa di uno il valore del semaforo se il valore è maggiore di zero, altrimenti si mette in attesa che diventi maggiore di zero e lo decrementa
DOWN permette l'acquisizione (acquire) della risorsa condivisa

Semafori binari (binary mutex)

I semafori più usati e più comuni sono i binary mutex cioè semafori che hanno solo zero e uno come valore della variabile che rappresenta la disponibilità della risorsa condivisa.

Il binary mutex rappresenta **mutua esclusione:** cioè solo un solo consumatore può usare la risorsa!

Semafori e risorse

1. Più in generale il semaforo = un contatore di quante risorse (“copie”) usabili sono disponibili.
2. La risorsa può essere un file, una stampante, una connessione di rete,
3. Un processo/Thread che vuole acquisire una risorsa effettua un **DOWN**, se non è disponibile attende.
   Analogamente può lasciare una risorsa effettuando un UP, che eventualmente notifica altri utilizzatori in **attesa**.
4. Se la risorsa è disponibile il DOWN avviene immediatamente

La classe Semaphore

	Semaphore(int permits) Creates a Semaphore with the given number of permits and nonfair fairness setting.	Il numero di risorse è incluso nel costruttore.
void	**acquire**() Acquires a permit from this semaphore, blocking until all are available, or the thread is **interrupted**.	Acquire=down , eventualmente bloccante per il thread se il valore del contatore è zero e il thread che ha invocato la acquire resta fermo fintantoche un altro thread non chiama la release()
void	**release**() Releases a permit, returning it to the semaphore.	La release può sbloccare più processi in attesa eventualmente sullo stesso semaforo.
int	**availablePermits**() Returns the current number of permits available in this semaphore.	

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html

Esercizio (e implementazione di un semaforo)

N persone desiderano entrare in una stanza. La stanza ha una capienza massima di C persone

Ogni persona, una volta entrata nella stanza, vi resta per T secondi, poi esce

Una persona che trova la stanza piena attende finchè non si libera un posto

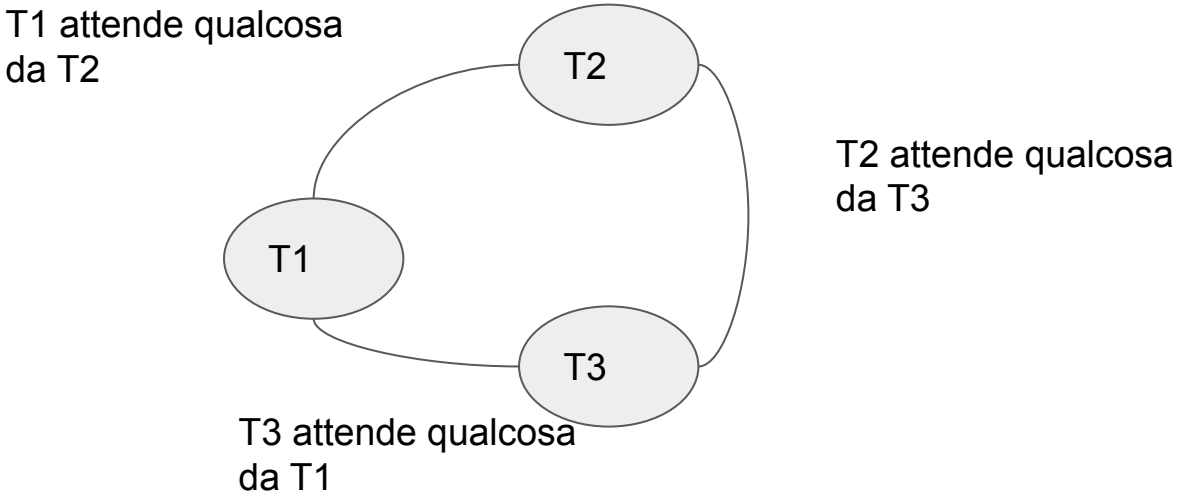
Scrivere un programma Java che simula la situazione appena descritta, modellando ciascuna persona come un Thread e utilizzando la classe Semaphore per regolare l'accesso

Esercizio (e implementazione di un semaforo)

```java
/**
 *
 * @author emanuele
 */
public class Semaphore {

    private int contatore;

    public Semaphore(int i)
    {
        this.contatore=i;
    }
    public Semaphore()
    {
        this.contatore=1;
    }

    public synchronized void release()
    {
        contatore++;
        if(contatore==1)
            this.notify();


    }
    public synchronized void acquire() throws InterruptedException
    {
        if(contatore==0)
            this.wait();
        contatore--;
    }

}
```

Deadlock

Situazione per cui i thread sono in attesa circolare di risorse occupate da un altro thread e per questo l'esecuzione si blocca

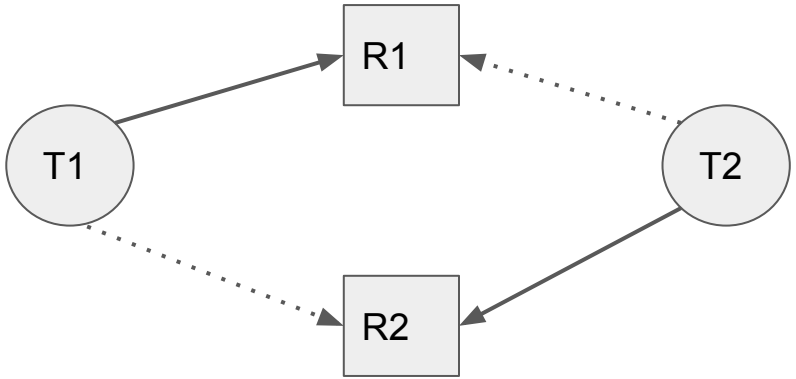


Esempio (2)

Due thread T1 e T2 e due risorse R1 e R2

T1 Richiede R1 , il semaforo su R1 =1 , T1 lo ottiene.
T2 Richiede R2 , il semaforo su R2 =1 , T2 lo ottiene.
T1, non rilasciando R1 richiede R2, semaforo R1=0, wait
T2, non rilasciando R2 richiede R1, semaforo R2=0, wait

Deadlock

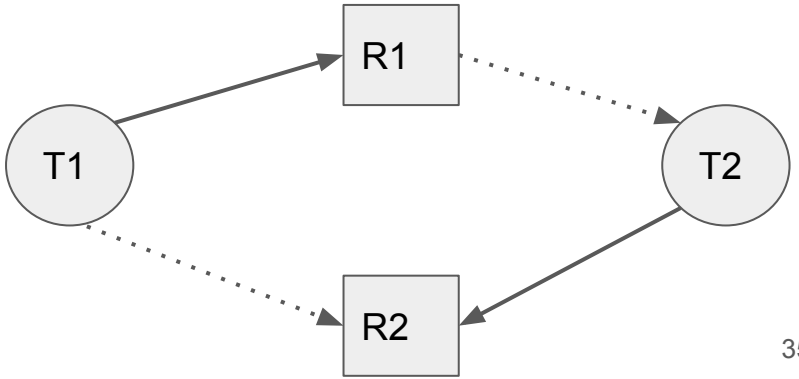


Esempio (2)

Due thread T1 e T2 e due risorse R1 e R2

Sequenza corretta
T1: Down(R1)
T1: Down(R2)
T1: istruzioni
T1: Up(R1)
T1: Up(R2)
T2: Down(R1)
T2: Down(R2)
T2: istruzioni
T2: Up(R1)
T2: Up(R2)

Deadlock Phase
T1: Down(R1)
T2: Down(R2)
T1: Down(R2)
T2: Down(R1)



Avoid Deadlock

- algoritmo dello struzzo

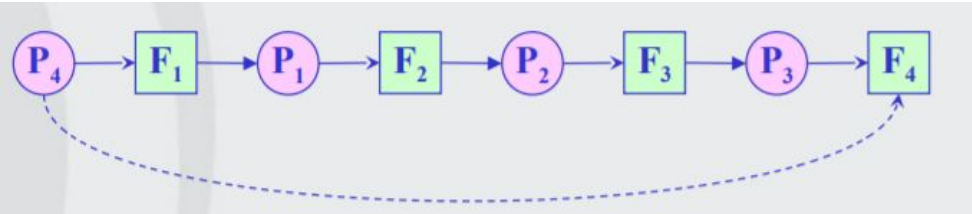
Non faccio niente...

Avoid Deadlock

- impedire l’attesa circolare

L'attesa circolare può essere eliminata in vari modi:
• Un processo può richiedere una sola risorsa per volta

• Le risorse sono numerate e possono essere richieste solo secondo l'ordine numerico



Avoid Deadlock

- impedire che logicamente possa esistere il deadlock

Se se si hanno a disposizione alcune informazioni si può – Il modello più semplice e più utile richiede che ciascun Thread dichiari il numero massimo di risorse necessarie
- L'algoritmo esamina dinamicamente lo stato di allocazione delle risorse per garantire che non accada mai una attesa circolare
- Si definisce stato è definito dal numero di risorse disponibili e allocate e dal numero massimo di risorse richieste

Avoid Deadlock

Stato sicuro

Uno stato si dice sicuro se esiste una sequenza di altri stati che porta tutti i Thread ad ottenere le risorse necessarie (e quindi terminare) altrimenti viene detto non sicuro.

Per evitare i deadlock è quindi sufficiente evitare gli stati non sicuri.

Avoid Deadlock

Stato sicuro