Sincronizzazione con Java

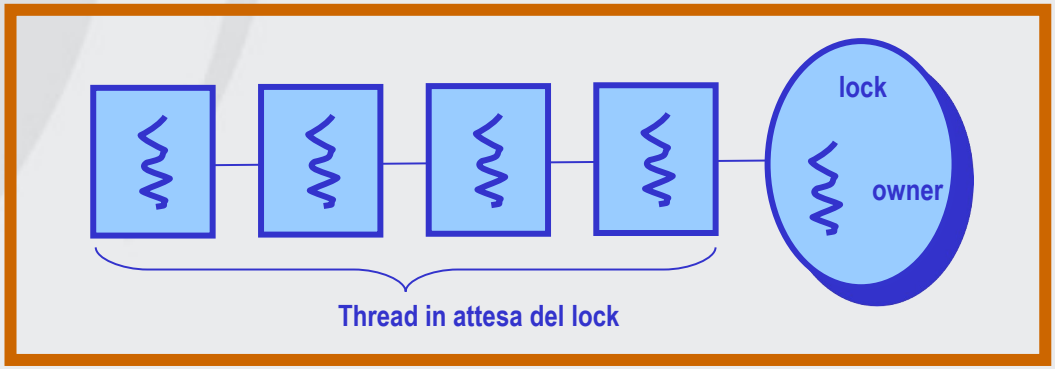
Corse critiche e sincronizzazione

- Java implementa un meccanismo simile al monitor per garantire la sincronizzazione fra *thread*
- Ogni oggetto ha un lock associato ad esso
  - Nelle classi possono essere definiti metodi `synchronized`

    ```
    synchronized int myMethod()
    ```
  - Un solo thread alla volta può eseguire un metodo synchronized
  - Se un oggetto ha più di un metodo sincronizzato, comunque uno solo può essere attivo
    - NB: non vi è nessun controllo su metodi non sincronizzati

Metodi synchronized

- La chiamata di un metodo sincronizzato richiede il “possesso” del lock
- Se un thread chiamante non possiede il lock (un altro thread ne è già in possesso), il thread viene sospeso in attesa del lock
- Il lock è rilasciato quando un thread esce da un metodo sincronizzato



Metodi synchronized

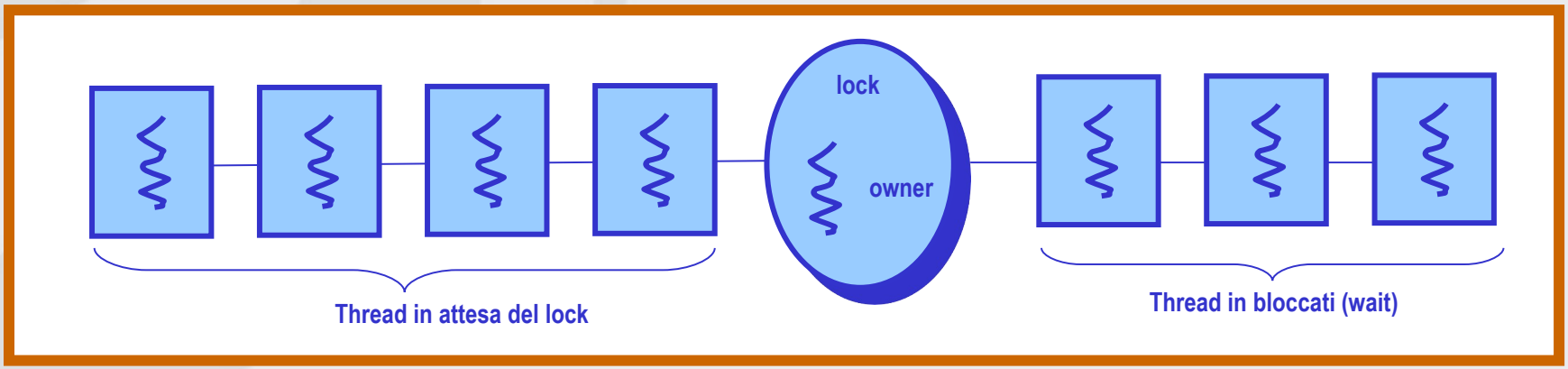
```
    synchronized void incrementaContatore() {
      int tmp = contatore + 1;
      if(tmp>maxContatore) maxContatore = tmp;
      aspetta(milliAspetta);
      contatore = tmp;
    }

    void synchronized decrementaContatore() {
      int tmp = contatore;
      aspetta(milliAspetta);
      contatore = tmp - 1;
    }
```

Sincronizzazione

- **Un Thread può decidere di non proseguire l'esecuzione**
  - **Può allora volontariamente chiamare il metodo `wait` all'interno di un metodo sincronizzato**
    - chiamare **`wait`** in un contesto diverso genera un errore

Il metodo wait()

- Quando un thread esegue **wait()**, si verificano i seguenti eventi:
  1. il thread rilascia il lock
  2. il thread viene bloccato
  3. il thread è posto in una coda di attesa
  4. gli altri thread possono ora competere per il lock



Il metodo notify()

- Nel momento in cui una condizione che ha provocato una attesa si è modificata si può riattivare un singolo thread in attesa tramite `notify`
- Quando un thread richiama notify():
  1) viene selezionato un thread arbitrario T fra i thread bloccati
  2) T torna fra i thread in attesa del lock
  3) T diventa Eseguibile

insert() con wait/notify

```
public synchronized void insert(Object item) {
   if(count==BUFFER_SIZE) {
      try {
        wait();
      } catch(InterruptedException e) {}
   }
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   if(count==1) notify();
}
```

Sincronizzazioni multiple

- notify() seleziona un thread arbitrario.
  Non sempre questo è il comportamento desiderato.
- Java non permette di specificare il thread che si vuole selezionare
- notifyAll() "risveglia" tutti i thread in attesa. In questo modo si permette ai thread stessi di decidere chi deve proseguire.
- notifyAll() è una strategia conservativa utile quando più thread sono in attesa

insert() con wait/notifyAll

```
public synchronized void insert(Object item {
  while(count==BUFFER_SIZE) {
    try {
      wait();
    } catch(InterruptedException e) {}
  }
  ++count;
  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  if(count==1) notifyAll();
}

private int count=0;
```

Confronto Monitor / Java

- Monitor
- Un solo processo può entrare nel monitor ed eseguire una sua procedura
- Variabili condizione
- wait(variabile_condizione)
- signal(var. condizione)

- (Qualunque) oggetto
- Un solo thread alla volta può eseguire uno dei metodi synchronized
- Una sola implicita (this)
- this.wait()
- this.notify() / notifyAll()

remove() con wait/notify

```
public synchronized Object remove() {
  while(count==0) {
    try {
      wait();
    } catch(InterruptedException e) {}
  }
  --count;
  Object item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  if(count==BUFFER_SIZE-1) notify();
  return item;
}
```

Metodo sleep

- `static void sleep( millisecondi )`
  - Il Thread si sospende (non richiede l'uso del processore) per un certo numero di millisecondi
  - Nel frattempo possono andare in esecuzione thread a bassa priorità
- NB
  - sleep non restituisce il possesso del lock
  - sleep è un metodo statico della classe Thread
  - wait è un metodo della classe Object

```
public void mysleep(long millis) {
    if(millis<0)
      millis = (long)(Math.random()*-millis);
    try {
      Thread.sleep(millis);
    } catch(Exception e) {}
}
```

Sincronizzazione di blocco

- Si definisce **scope** di un lock è il tempo fra la sua acquisizione ed il suo rilascio
- Tramite la parola chiave `synchronized` si possono indicare blocchi di codice piuttosto che un metodo intero
- Ne risulta uno scope generalmente più piccolo rispetto alla sincronizzazione dell'intero metodo

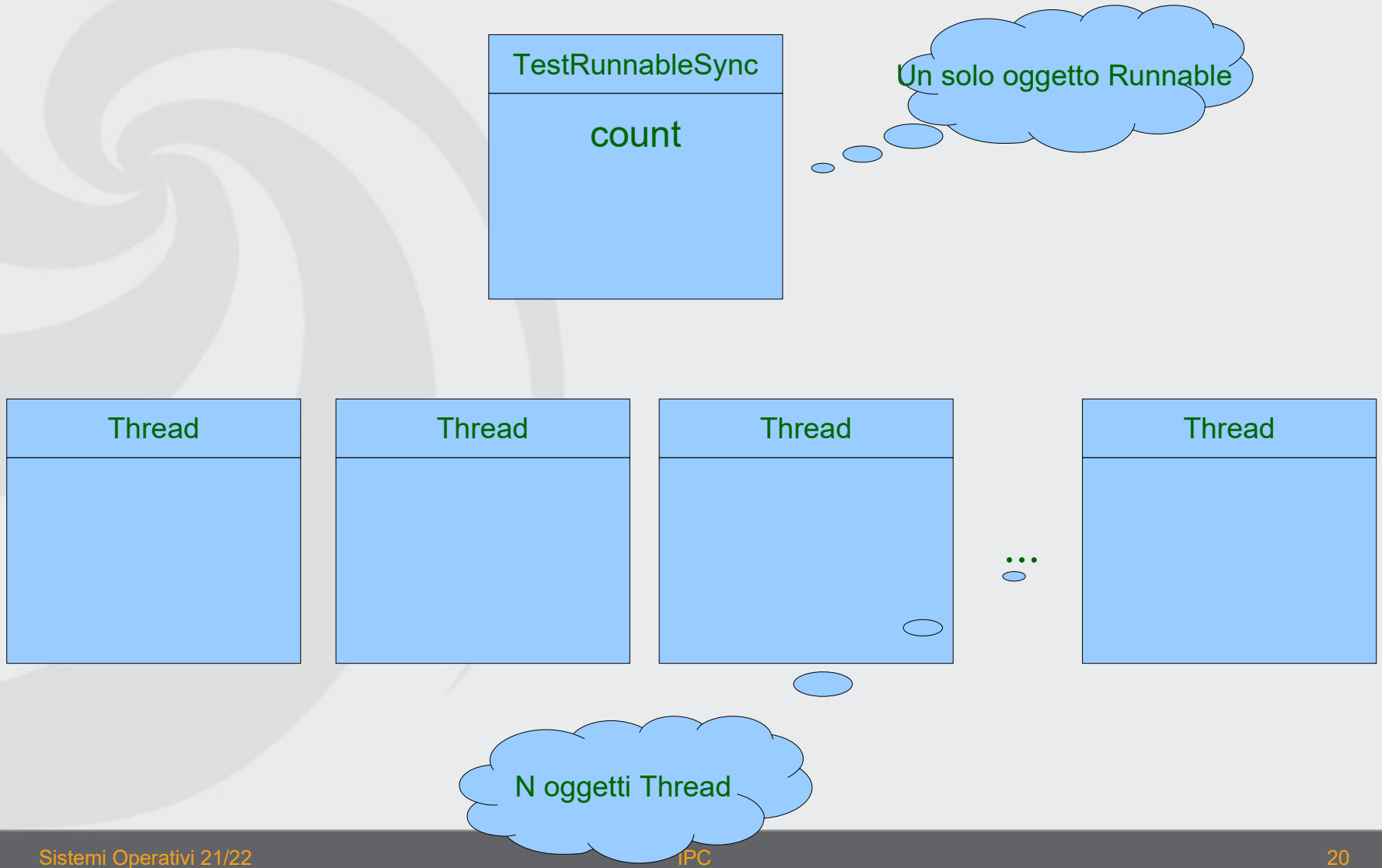
Sincronizzazione di blocco

```
Object mutexLock = new Object();

public void someMethod() {
  nonCriticalSection_1();
  synchronized(mutexLock) {
    criticalSection();
  }
  nonCriticalSection_2();
}
```

Sincronizzazione di blocco

```
public class TestRunnableSync implements Runnable {
   int count;
   public void run() {
      int tmp;
      synchronized (this) { tmp = count++; }
      String name = Thread.currentThread().getName();
      System.out.println(name + " " + tmp);
   }

   public static void main(String[] args) {
      int n=args.length>0?Integer.parseInt(args[0]):10;
      Runnable runnable = new TestRunnableSync();
      for(int i=0; i<n; i++) {
         Thread th = new Thread(runnable);
         th.start();
      }
   }
}
```

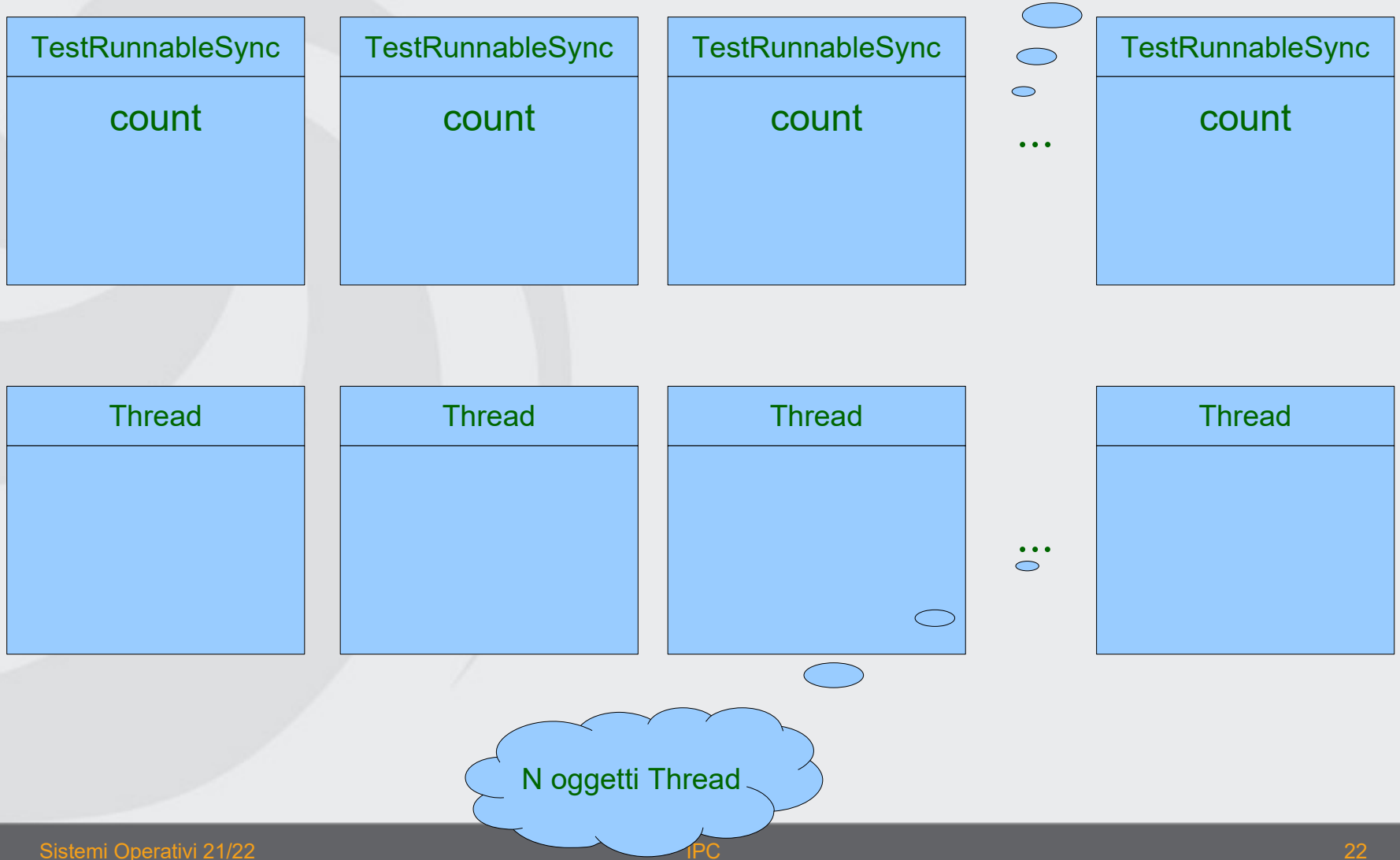
TestRunnableSync



Sincronizzazione di blocco

```
public class TestRunnableSync2 implements Runnable {
   int count;
   public void run() {
      int tmp;
      synchronized (this) { tmp = count++; }
      String name = Thread.currentThread().getName();
      System.out.println(name + " " + tmp);
   }

   public static void main(String[] args) {
      int n=args.length>0?Integer.parseInt(args[0]):10;
      for(int i=0; i<n; i++) {
         Runnable runnable = new TestRunnableSync();
         Thread th = new Thread(runnable);
         th.start();
      }
   }
}
```

TestRunnableSync2



La classe Timer

- La classe java.util.Timer permette la schedulazione di processi nel futuro (attenzione esiste anche la classe javax.swing.Timer dal comportamento parzialmente diverso)

  Costruttore: `Timer()`

  `void schedule(TimerTask task, long delay, long period)`

  - Permette di eseguire un task periodico dopo un ritardo specificato
  - Esistono altri costruttori e altri metodi (si veda **`scheduleAtFixedRate`**)

La classe TimerTask

- TimerTask è una classe astratta occorre estenderla e definire il metodo run (ciò che il task deve fare)

```
long periodo = 1000; // per esempio un secondo
java.util.Timer timer = new java.util.Timer();
timer.schedule(new MyTimerTask(), 0, periodo);

public class MyTimerClass extends
    java.util.TimerTask {

    public void run() {
        // ciò che devo fare
    }

}
```

dovrebbe richiedere un tempo inferiore a periodo

Un semplice orologio da riga di comando

```
import java.util.Date;

public class Clock extends Thread {

    public static void main(String[] args) {
            new Clock().start();
    }

    public void run() {
        while(true) {
            System.out.print(new Date() + "\r");
            mysleep(1000);
        }
    }

}
```

Un orologio per mezzo di TimerTask

```
import java.util.*;

public class ClockTimerTask extends TimerTask{

  public static void main(String[] args) {
    int period=Integer.getInteger("period",1000);
    int delay = Integer.getInteger("delay", 0);
    TimerTask task = new ClockTimerTask();
    new Timer().schedule(task, delay, period);
  }

  public void run() {
    System.out.print(new Date() + "\r");
  }

}
```

Nota: un orologio a livello shell unix

```
% while sleep 1 ; do date | tr '\n' '\r' ; done
```