Lezione 13

martedì 10 novembre 2020 16:14

notify() e notifyAll()

Più thread possono essere bloccati su una wait eseguita sullo stesso oggetto

Se i thread sono in attesa di condizioni diverse allora è necessario usare la notifyAll()

In tale situazione, se usassi una semplice notify(), potrebbe succedere che

- sveglio un solo thread che però non può andare avanti con la propria esecuzione (perché attendeva una condizione diversa da quella che è adesso verificata)
- il thread appena svegliato controlla la condizione e si ribloccia, mentre quello che potrebbe andare avanti non è stato risvegliato (notify risveglia un thread a caso)

Risultato finale: applicazione bloccata

Usare la notify() al posto di notifyAll() è un'ottimizzazione possibile quando:

- quando tutti i thread bloccati attendono

- quando tutti i thread bloccati attendono la stessa condizione
- solo uno dei thread può trarre beneficio dalla nuova situazione

Cosa succederebbe se nel codice dei produttori-consumatori sostituissimo la notifyAll() con delle notify()?

Supponiamo di avere 2 produttori e 1 consumatore e anche che buffer abbia dimensione 1

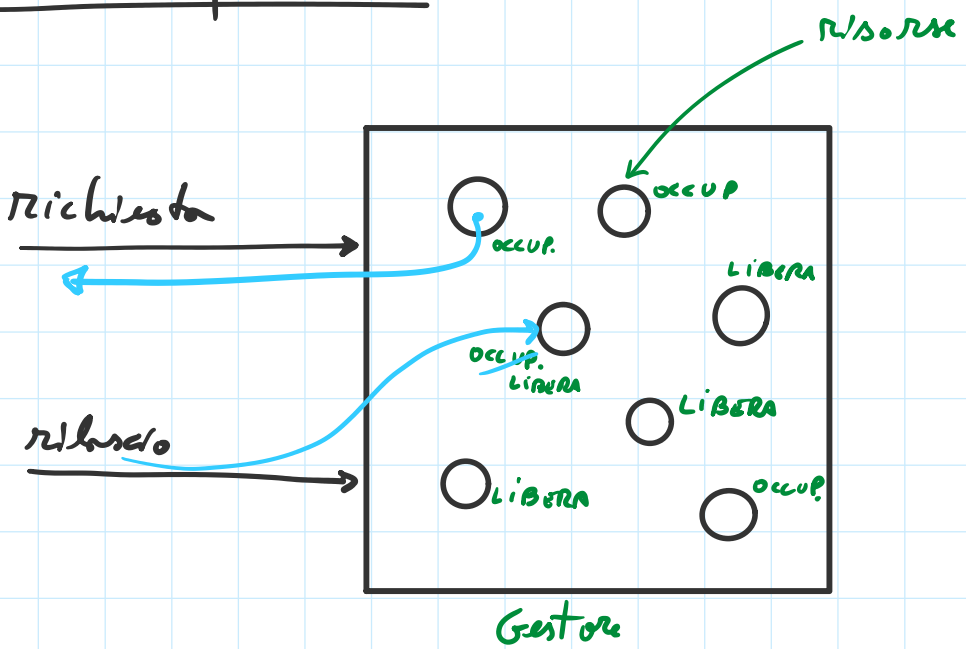
Inizio: buffer vuoto, P0 e P1 eseguibili, C0 eseguibile

Possibile sequenza di eventi:

P0 inserisce un valore
P1 prova a inserire e si blocca
P0 prova a inserire e si blocca
C0 estrae il valore e risveglia P0 (a caso)
→ C0 prova a estrarre un valore e si blocca
→ P0 inserisce un valore e risveglia P1 (a caso)
→ P1 prova a inserire e si blocca
→ P0 prova a inserire e si blocca

Deadlock!

Gestore di risorse equivalenti



int richiesta(): restituisce l'indice della risorsa che viene attribuita; può essere bloccante se non ci sono risorse libere

void rilascio(int k): rilascia la risorsa k-esima (acquisita in precedenza)

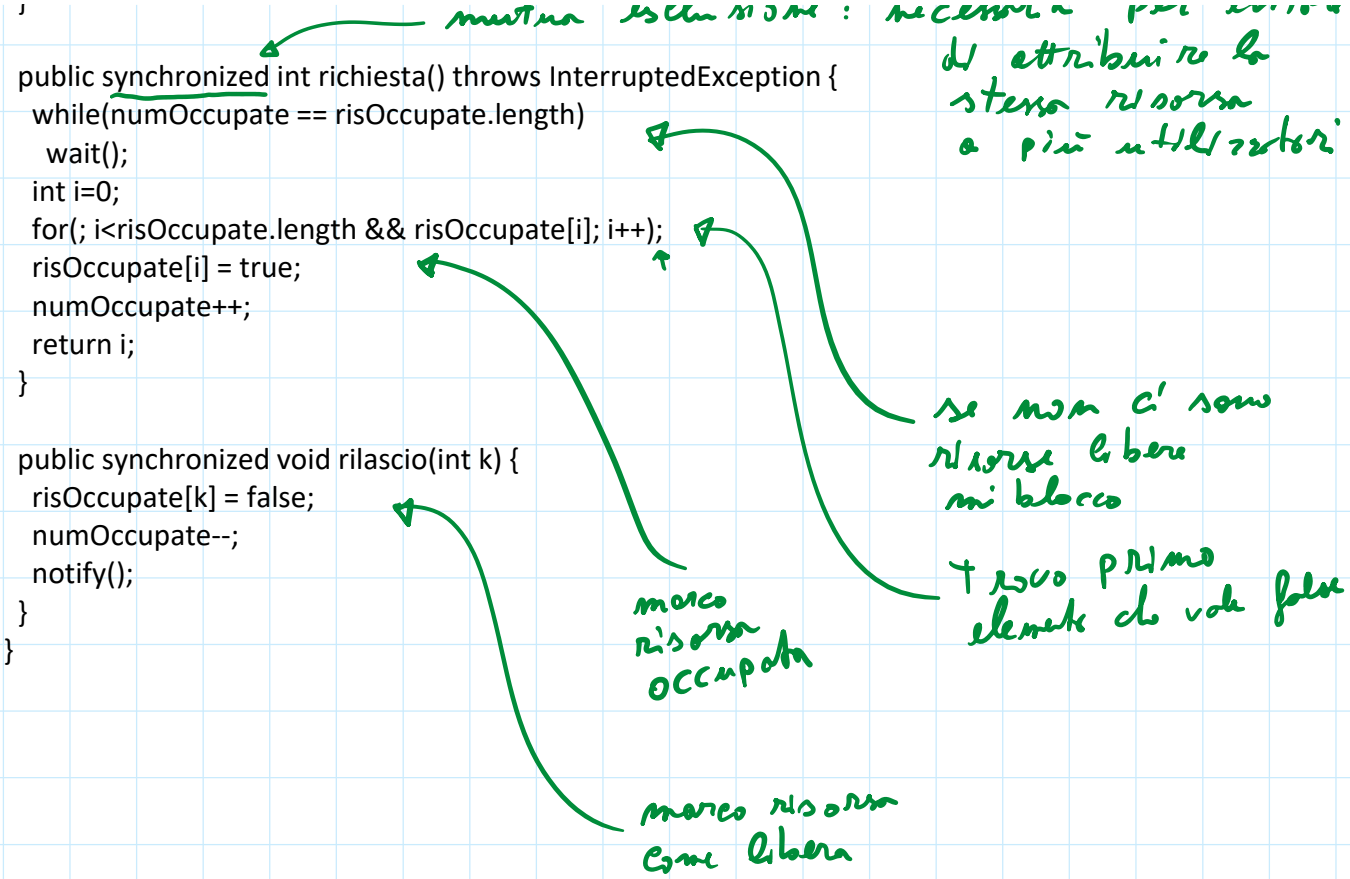
```
public class Gestore {

 private boolean[] risOccupate;
 private int numOccupate;

 public Gestore(int n) {
  risOccupate = new boolean[n];
 }

 public synchronized int richiesta() throws InterruptedException {
```

stato delle risorse (true = ris. occupata, false = ris. libera)

n: numero di risorse da gestire

valori default: false → libere

mutua esclusione: necessaria per evitare di attribuire la stessa risorsa



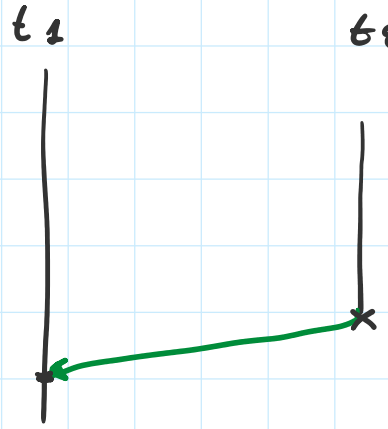

```
public class Utilizzatore extends Thread {

 private Gestore g;
 private int numRichieste;
 private static final int DEFAULT_NUM_RICHIESTE = 100;

 public Utilizzatore(String n, Gestore g, int r) {
  super(n);
  this.g = g;
  this.numRichieste = r;
 }

 public Utilizzatore(String n, Gestore g) {
  this(n, g, DEFAULT_NUM_RICHIESTE);
 }

 public void run(){
  try {
   for(int i=0; i<numRichieste; i++) {
    int k = g.richiesta();
    System.out.println(getName() + ": ho ottenuto la risorsa " + k);
    sleep((long)(Math.random()*1000));
    g.rilascio(k);
```

```
      System.out.println(getName() + ": ho rilasciato la risorsa " + k);
    }
   } catch(InterruptedException ie) {
    return;
   }
  }
}
```

```
import java.util.Scanner;
public class Prova {
 public static void main(String[] args){
  Scanner sc = new Scanner(System.in);
  System.out.println("Quanti utilizzatori?");
  int u = sc.nextInt();
  System.out.println("Quante risorse?");
  int r = sc.nextInt();
  Gestore g = new Gestore(r);
  for(int i=0; i<u; i++)
   new Utilizzatore("U"+i, g).start();
 }
}
```

Join

Supponiamo che un thread t_1 voglia attendere la terminazione di un thread t_2



Possiamo usare il metodo join() della

possiamo usare il metodo join() della classe Thread

t1 deve chiamare il metodo join() sull'oggetto thread tq

- t1 si blocca "dentro" il metodo join() e esce quando tq arriva in fondo al suo metodo run()

- se tq già terminato join() non è bloccante e non ha effetto

5 join() ne esistono tre versioni

public final void join(long millis) throws InterruptedExc.

blocca il chiamante fino a quando il thread termina o sono trascorsi "millis" millisecondi

public final void join(long millis, int nano)
throws InterruptedException

Come sopra ma tempo espresso anche con nanosecondi

public final void join() throws InterruptedException

attende che il thread su cui è stato chiamato

attende che il thread su cui è stato chiamato join finisca la propria esecuzione

(equivale a join(0))

↑ timeout infinito

Esempio: il main thread attiva un thread ausiliario che verifica se un numero è primo oppure no; il main thread attende che quello ausiliario abbia terminato per ottenere il risultato.

```
public class Lancia {
 public static void main(String[] args) {
  int n = Integer.parseInt(args[0]);
  Analizzatore a = new Analizzatore(n);
  a.start();
  try {
     a.join();
     boolean r = a.getRisultato();
     System.out.println(r ? "primo" : "non primo");
  } catch (InterruptedException ie) {
   //nothing
  }
 }
}
```

converte args[0] in un int usando il metodo statico parseInt() della classe Integer

si blocca fino a quando a non è arrivato in fondo al suo metodo run()

recupero il risultato (dopo che analizzatore ha finito)

```
public class Analizzatore extends Thread {

 private int n;
 private boolean risultato;

 public Analizzatore(int n) {
  this.n = n;
 }

 public void run(){
  risultato = primo(n);
 }

 public boolean getRisultato(){
  return risultato;
```

```
}

private boolean primo(int n){
  for(int i=2; i<n; i++) {
    if(n%i == 0)
      return false;
  }
  return true;
}
}
```

Interrompere un thread

L'interruzione delle operazioni di un thread deve essere cooperativa

- il thread deve controllare se gli viene chiesto di arrestare le proprie operazioni

È possibile inviare un interrupt a un thread

Per inviare un interrupt dobbiamo chiamare il metodo

```
public void interrupt()
```

sull'oggetto thread.

A ogni thread è associato un interrupt status (un flag)

Quando inviamo un interrupt a un thread possono succedere due cose

1) se il thread è bloccato su una sleep(), una wait(), una join() il thread esce con una InterruptedException e l'interrupt status non viene settato

2) se il thread esegue del codice "normale" viene settato l'interrupt status

Un thread può controllare il proprio interrupt status con

public static boolean interrupted()
restituisce true se il flag è settato, false altrimenti; in ogni caso resetta l'interrupt status

public boolean isInterrupted()
restituisce true se interrupt status è settato, false altrimenti
(non resetta!)

I metodi interrupt(), interrupted(), isInterrupted() sono della classe Thread.

Esempio: il main thread (comportamento definito dalla classe **Creatore**) ne attiva un altro ausiliario (comportamento definito dalla classe **Esecutore**); dopo due secondi il main thread interrompe quello ausiliario; quello ausiliario, quando interrotto, termina le proprie operazioni.

```
public class Creatore {

 public static void main(String[] args){
  Esecutore e = new Esecutore();
  e.start();
  try{
   Thread.sleep(2000);
  } catch(InterruptedException ie) {
   // do nothing
  }
  e.interrupt();
 }
}

public class Esecutore extends Thread {
 public void run(){
  while(!interrupted()) {
   fai();
  }
 }

 private void fai(){
  System.out.print(".");
 }
}
```