1 Lezione del 21-10-25

Continuiamo a parlare della sincronizzazione, approfondendo il funzionamento dei metodi relativi.

1.0.1 Metodo `wait()`

Il metodo `wait()` è definito come `public final void wait() throws InterruptedException`. Il thread che invoca `wait()`:

- Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Riacquisisce automaticamente il lock quando esece dalla wait.Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Riacquisisce automaticamente il lock quando esece dalla wait.

Esiste una versione con timeout di `wait()`. Questa è definita come `public final void wait(long timeout) throws InterruptedException`. In questo caso il thread che invoca `wait()`:

- Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Aspetta i millisecondi di timeout: se li supera esce forzatamente riacquisendo i lock;
- Riacquisisce automaticamente il lock quando esece dalla wait.Sospende la propria esecuzione;
- Rilascia il lock sull'oggetto;
- Riacquisisce automaticamente il lock quando esece dalla wait.

La `wait(timeout)` con `timeout` $= 0$ corrisponde alla `wait()` (timeout infinito).

1.0.2 Metodo `notify()`

Il metodo `notify()` ha firma `public final void notify()`. Quando viene invocato su un oggetto il primo fra i thread che erano in attesa su tale oggetto (cioè erano nella sua *wait list*, dove si entra invocando `wait()`) viene svegliato.

Il `notifyAll()` è una variante di `notify()`, con firma `public final void notifyAll()`, che sveglia tutti i thread in attesa nella *wait list*.

Notiamo che `wait()`, `notify()` e `notifyAll()` non devono essere per forza chiamati sulla classe madre, ma possono anche essere chiamati su altri oggetti, con la prerogativa che si possieda un lock su tali oggetti.

1.1 Produttori e consumatori

Vediamo come applicare questo schema ha uno degli esempi più noti della programmazione concorrente: quello del paradigma dei **produttori** e **consumatori**.

Ipotizziamo una situazione dove esistono n thread, P_0, P_1, ..., P_n detti *produttori*, ed m thread, C_0, C_1, ..., C_m detti *consumatori*. Fra produttori e consumatori c'è un certo *buffer*. Il buffer è *condiviso* tra tutti i produttori e i consumatori, ed è in grado di contenere un numero limitato di valori.

Ogni produttore ciclicamente:

- Produce un nuovo valore;
- Lo immette nel buffer.

Chiaramente, se il buffer è pieno, il produttore deve bloccarsi.

Ogni consumatore ciclicamente:

- Preleva un valore dal buffer;
- Lo consuma.

Come sopra, se il buffer è vuoto, il consumatore deve bloccarsi.

scrivi codice e riporta

1.2 Differenza fra `notify()` e `notifyAll()`

Approfondiamo la differenza fra i metodi per il risveglio di thread, cioè `notify()` e `notifyAll()`.

Su un oggetto possono essere in attesa più thread (alternativamente, in una *wait list* possono esserci più di un thread).

Se questi thread sono in attesa di condizioni diverse, è opportuno usare la `notifyAll()`. In tal situazione, se si facesse diversamente (usando una semplice `notify()`) si potrebbe infatti avere la sveglia di un solo thread che però non vede le proprie condizioni soddisfatte: nessuno viene effettivamente svegliato, si può incorrere in deadlock.

Se tutti i thread bloccati attendono la stessa condizione, o se solo uno (senza specificare quale) dei thread bloccati può trarre beneficio dalla situazione creatasi, conviene invece usare `notify()`.

Nell'esempio sopra, sul buffer si usa `notifyAll()` anziché `notify()` in quanto produttori e consumatori possono entrambi bloccarsi sui thread, ma aspettando condizioni diverse.