Laboratorio di "Sistemi Distribuiti"
A.A. 2024-2025

Concorrenza (parte 2)

Emanuele Petriglia

Slide aggiornate al 10 Aprile 2025

Indice dei contenuti

1. Concorrenza a “basso livello” in Java

Thread e Runnable

- `Thread`: è una classe che rappresenta un thread.
- `Runnable`: è un'interfaccia che rappresenta un'operazione che si può eseguire.
  - L'interfaccia deve essere implementata e l'istanza incapsulata in un thread per poter eseguire l'operazione in concorrenza.
- Esempio:

```
1  public class HelloWorld {
2      public static void main(String[] args) {
3          var thread = new Thread(new ExampleRunnable());
4          thread.start();
5          System.out.println("Hello World dal main!");
6      }
7  }
8  class ExampleRunnable implements Runnable {
9      public void run() {
10         System.out.println("Hello World dal thread!");
11     }
12 }
```

La parola chiave `synchronized`

- Ogni oggetto in Java è associato un monitor gestito dalla JVM.
- `synchronized` acquisisce il monitor di un oggetto.
  - Garantisce la mutua esclusione tra più thread **per lo stesso monitor**.
- Due modalità d'uso:

```java
public class Counter {
    private int count = 0;

    public synchronized void
    ↪ increment() {
        count++;
    }
}
```

```java
public class Counter {
    private int count = 0;

    public void increment() {
        synchronized (this) {
            count++;
        }
    }
}
```

Alcuni avvertimenti

1. Un'istanza `Thread` una volta avviata (con `Thread.start()`) e terminata non può essere riusata. Bisogna creare un nuovo thread.
2. Non si può sapere quando un thread verrà eseguito, ma si può sapere quando termina (con `Thread.join()`).
3. Mai usare `synchronized` con tipi primitivi (`int`, `double`...) e con oggetti immutabili (`String`).
4. La sincronizzazione ha un costo computazionale: ridurre al minimo essenziale il codice da proteggere.

Concorrenza a "basso livello"

Nei laboratori vediamo la concorrenza a "basso livello" che usa le primitive di Java. Esiste anche ad "alto livello" che sfrutta i meccanismi di base, è più facile da usare e più sicura a problemi di concorrenza.

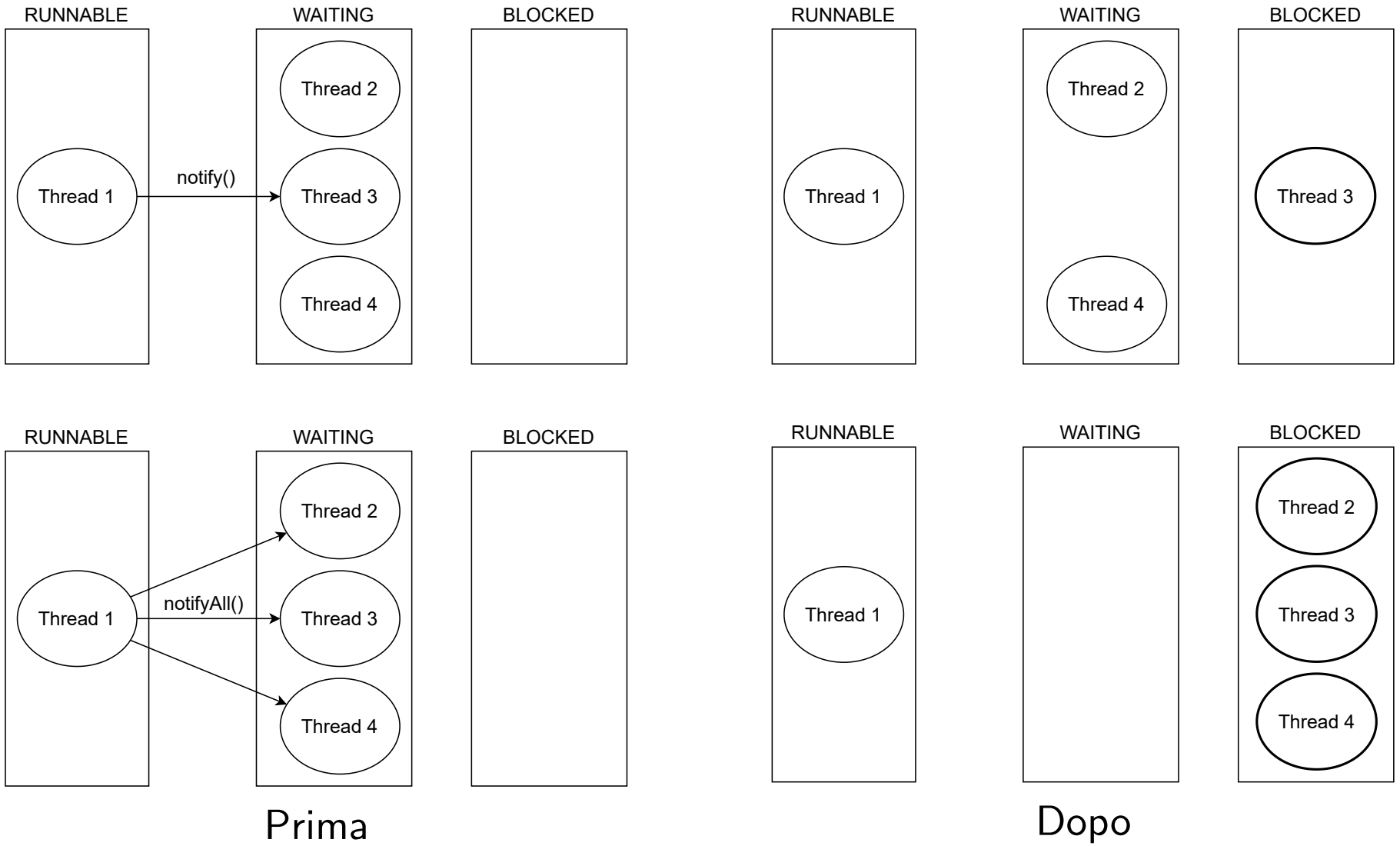
`wait()`, `notify()` e `notifyAll()`

- Metodi per la comunicazione tra processi:
  - `wait()`: interrompe il thread e **rilascia** il monitor **finché** un altro thread non invoca `notify()` o `notifyAll()` sullo stesso oggetto.
  - `notify()`: sveglia **un** thread a caso che è in attesa del monitor per lo stesso oggetto. Prosegue l'esecuzione **senza rilasciare il monitor**.
  - `notifyAll()`: sveglia **tutti** i thread in attesa del monitor per lo stesso oggetto. Prosegue l'esecuzione **senza rilasciare il monitor**.
- Differenza con `synchronized`: l'uso dei tre metodi è esplicito del programmatore (e quindi può sbagliare!).

Attenzione!

1. I tre metodi sono ereditati da qualsiasi classe Java!
2. I tre metodi vanno chiamati all'interno di `synchronized`!

Differenza tra `notify()` e `notifyAll()`



Esempio d'uso di `notify()`

```
1   class NumberGenerator {
2     private final Random rng = new Random();
3     private int number;
4     private boolean isEmpty = true;
5
6     void generate() {
7       while (true) {
8         try {
9           synchronized (this) {
10            while (!this.isEmpty) {
11              this.wait();
12            }
13            this.number = rng.nextInt();
14            this.isEmpty = false;
15            this.notify();
16            //this.notifyAll();
17          }
18        } catch (InterruptedException e) {
19          System.exit(-1);
20        }
21      }
22    }
23    void get() {
24      while (true) {
25        try {
26          synchronized (this) {
27            while (this.isEmpty) {
28              this.wait();
29            }
30            this.isEmpty = true;
31            // Consuma...
32            this.notify();
33            //this.notifyAll();
34          }
35        } catch (InterruptedException e) {
36          System.exit(-1);
37        }
38      }
39    }
40  }
41
42  public class HelloWorldNotify {
43    public static void main(String[] args) {
44      var generator = new NumberGenerator();
45      new Thread(generator::get,
        ↪  "C1").start();
46      new Thread(generator::generate,
        ↪  "P1").start();
47    }
48  }
```

Problema con `notify()`

- `notify()` risveglia un thread scelto a caso.
- Se ci sono più thread in attesa che eseguono task diversi (es. più produttori e consumatori), un produttore (P1) potrebbe risvegliare un altro produttore (P2).
  - P2 torna in wait e nessuno invoca `notify()` $\rightarrow$ deadlock!
  - Un consumatore (es. C1) potrebbe non venire mai chiamato.
  - Provare l'esempio precedente con 5 produttori.
- Soluzione: usare `notifyAll()`!
  - Risveglia tutti i thread in attesa e quindi tutti i thread prima o poi verranno eseguiti.

Promemoria

- Usare `notify()` solo se tutti i thread in attesa usano la stessa condizione.
- Altrimenti usare `notifyAll()`, anche se ha una penalità in più di prestazioni.

Usare sempre `while` con `wait()`

- Il seguente frammento di codice è sbagliato:

```
1  synchronized (object) {
2      if (/* Condizione non soddisfatta. */) {
3          object.wait();
4      }
5  }
```

- Perché? Quando il thread riacquisice il monitor la condizione potrebbe non essere più vera.
  - Altri thread potrebbero essere stati risvegliati prima e aver cambiato la condizione (es. più consumatori o produttori).
  - Oppure la JVM potrebbe svegliare "a caso" il thread senza che ci sia stato un `notify()`.

- **Soluzione**: racchiudere `wait()` sempre in un ciclo while:

```
1  synchronized (object) {
2      while (/* Condizione non soddisfatta. */) {
3          object.wait();
4      }
5  }
```

2. Esercizio A

Consegna

- Implementare una classe `CustomExecutor` che esegue dei task in dei thread in modo trasparente al programmatore.
- `CustomExecutor` espone:
  - Un costruttore `CustomExecutor(int numThreads)` che inizializza il pool dei thread (worker).
  - Il metodo `execute(Runnable task)` che aggiunge il task in una coda di task. Quando un worker è libero prende il task e lo esegue.
- Usare `notify()`, `wait()` e `synchronized` per gestire l'accesso alla coda dei task!
- Partire dallo scheletro, contiene il codice `main()` e un esempio di `Runnable` che prova il `CustomExecutor`

`Executors` e `ExecutorService`

L'esercizio si ispira alla classe `Executors` e `Executor` che fanno parte del package `java.util.concurrent`.

Suggerimenti (pt. 1)

- La sincronizzazione tra i thread del pool (worker) e il thread principale avviene sulla coda dei task.
  - Si può usare `notify()` perché tutti i thread in attesa hanno come condizione che ci sia almeno un elemento nella coda.
- Ignorare come terminare correttamente il pool dei thread.
- Punto di vista del thread main quando riceve un task:
  1. Aggiunge il task alla coda.
  2. Segnala che è stato aggiunto il task con `notify()`.
- Punto di vista di un worker:
  1. Se disponibile prende un task dalla code e lo esegue.
  2. Se la coda è vuota, attende (`wait()`) finché non c'è un task.
  3. Ripete all'infinito i primi due punti.

Suggerimenti (pt. 2)

- Per la coda, usare un `ArrayList`:

```
1  var tasks = new ArrayList<Runnable>();
2  tasks.add(task); // Aggiunge alla fine.
3  var task = tasks.removeFirst(); // Rimuove il primo.
```

- Il thread worker esegue un ciclo strutturato:

```
1  public void run() {
2      while (true) {
3          // Prima prende il task dalla coda.
4
5          // Poi esegue il task, controllando con
6          // try/catch eventuali errori (li ignora).
7      }
8  }
```

3. Esercizio B

Consegna

- Scrivere una classe `Buffer` che implementa un **buffer di stringhe** condiviso tra più thread di capacità limitata.
- `Buffer` espone:
  - Un costruttore `Buffer(int capacity)`,
  - Il metodo `add(String item)` che aggiunge la stringa in coda al buffer,
  - Il metodo `String get()` che estrae la prima stringa dal buffer e la restituisce.
- Il buffer viene usato da più thread: usare i costrutti `synchronized`, `wait()` e `notifyAll()` per gestire l'accesso concorrente.

Osservazione

Si tratta in sostanza di rendere *thread-safe* la classe `LinkedList`. Java già fornisce una classe simile ad alto livello.

Suggerimenti

- Partire dallo scheletro: bisogna solo implementare la classe `Buffer`!
  - Il resto del codice (main, il produttore e il consumatore) usa il buffer.
- Il focus è sulla sincronizzazione: usare `LinkedList` e proteggere l'accesso alla lista.
  - Quali metodi? Basta l'aggiunta `add()` e la rimozione `remove()`.
- Due casi di attesa per un thread:
  1. Un thread vuole aggiungere una stringa ma il buffer è pieno $\rightarrow$ attende che si libera un posto.
  2. Un thread vuole leggere una stringa ma il buffer è vuoto $\rightarrow$ attende che venga aggiunta una stringa.

4. Deadlock e starvation

Deadlock

- **Problema**: due o più thread attendono di acquisire dei monitor acquisiti dagli altri thread in modo indefinito.
- In questo caso **nessun thread avanza**, l'intero sistema è bloccato.
- Vedere esempio `HelloWorldDeadlock.java`.
- **Soluzione**: stabilire un ordine per acquisire i monitor e ridurre al minimo le acquisizioni ove possibile.
  - Ciò non garantisce l'assenza di deadlock ma la minimimizza.

Starvation

- **Problema**: uno o più thread non vengono mai (o quasi) eseguiti perché "sopraffatti" da altri thread con priorità più alta.
- In questo caso solo alcuni **thread avanzano**, mentre altri rimangono bloccati o avanzano a rilento.
- Vedere esempio `HelloWorldStarvation.java`.
- **Soluzioni**:
  - Usare sempre la stessa priorità tra i thread,
  - Preferire `notifyAll()` rispetto a `notify()`.
  - Usare meccanismi più ad alto livello che implementano una coda di thread all'acquisizione di un monitor.

5. Esercizio C

Consegna

- Risolvere il problema di starvation dell'implementazione di un lock fornito su e-Learning.
- Concentrarsi solo sulla classe `ReadWriteLock`, che implementa un lock in cui:
  - Più lettori possono acquisire il lock simultaneamente,
  - Uno scrittore acquisisce il lock in modo esclusivo.
- **Problema di starvation**: i lettori acquisiscono il lock ma nessun scrittore ci riesce $\rightarrow$ i thread scrittori non avanzano.

Osservazione

Si tratta in sostanza di implementare la classe `ReadWriteLock`.

Esempio

Esecuzione attuale:

```
1  > java ReaderWriter.java
2  R1 legge (0 volte)
3  R2 legge (0 volte)
4  R3 legge (0 volte)
5  R4 legge (0 volte)
6  R4 legge (1 volte)
7  R1 legge (1 volte)
8  R2 legge (1 volte)
9  ...
```

Esecuzione “desiderata”:

```
1  > java ReaderWriter.java
2  R1 legge (0 volte)
3  R4 legge (0 volte)
4  R2 legge (0 volte)
5  R3 legge (0 volte)
6  W1 scrive (0 volte)
7  W2 scrive (0 volte)
8  W3 scrive (0 volte)
9  R2 legge (1 volte)
10 R1 legge (1 volte)
11 R4 legge (1 volte)
12 R3 legge (1 volte)
```

Suggerimenti (pt. 1)

- Bisogna modificare la classe `ReadWriteLock`. Come garantire un accesso equo a tutti i thread?
  - Con una coda `LinkedList`.
- L'idea di usare la coda è:
  - Se un thread lettore vuole acquisire il lock può farlo subito se prima di lui in coda ci sono **solo thread lettori**.
  - Se un thread scrittore vuole acquisire il lock può farlo solo se è il primo nella coda.
  - Altrimenti il thread (lettore o scrittore) aspetta il suo turno.
- Cosa salvare nella coda? Un identificativo del thread (es. il nome).
- Metodi utili di `LinkedList`:
  - `add(E)`: aggiunge un elemento in fondo alla lista,
  - `remove(E)` rimuove l'elemento dalla lista,
  - `peek()`: restituisce l'elemento in testa alla lista senza rimuoverlo.

Suggerimenti (pt. 2)

- Per aggiungere la coda bisogna:
  1. Aggiungere la gestione della coda prima e dopo il `wait()`,
  2. Espandere `canRead()` e `canWrite()` in modo che riceva il nome del thread e lo cerchi nella coda.

Promemoria

- Per ottenere il nome del thread attuale:

```
var name = Thread.currentThread().getName();
```

- Per creare la lista come attributo:

```
private LinkedList<TYPE> waiting = new LinkedList<>();
```