Thread Java e Reti di Petri

Modellazione Formale della Concorrenza

Salvatore Capolupo

---

0 Contesto: programmazione concorrente in Java

Questa dispensa utilizza **Java 21** (LTS), la versione a supporto a lungo termine più recente al momento della stesura. I concetti di concorrenza trattati sono validi a partire da Java 5 e rimasti sostanzialmente stabili fino a Java 21.

JDK e JVM

Il **JDK** (*Java Development Kit*) è un insieme di strumenti che comprende il compilatore `javac`, il debugger, le librerie standard e, al suo interno, la **JVM**.

La **JVM** (*Java Virtual Machine*) è il motore di esecuzione: riceve il *bytecode* prodotto dal compilatore (file `.class`) e lo esegue traducendolo in istruzioni macchina native. È la JVM a gestire la memoria, il garbage collector e — aspetto fondamentale per questa dispensa — i *thread*. In un certo senso, la JVM *simula* il comportamento di un processore multi-core, distribuendo i thread sulle risorse fisiche disponibili.

All'interno della JVM opera uno **scheduler**, un componente che decide quale thread ottiene l'uso del processore in un dato istante. La CPU viene suddivisa in piccole finestre di tempo (*time-sharing*): lo scheduler assegna ciascuna finestra a un thread diverso, così rapidamente che all'utente i processi sembrano procedere simultaneamente.

La conseguenza più importante per il programmatore è che **non è possibile fare ipotesi temporali** sull'ordine di esecuzione dei thread. Scrivere codice che funzioni correttamente solo se il thread A termina prima del thread B è un errore: lo scheduler può scegliere qualunque ordine, e tale ordine cambia da una esecuzione all'altra, da una macchina all'altra, con diversi carichi di sistema. Questa non-determinabilità non è un difetto: è la natura stessa della concorrenza, e le reti di Petri diventeranno uno strumento formale preciso per ragionarci.

Due anomalie tipiche dei sistemi concorrenti - programmi Java con più Thread - meritano una definizione precisa prima di procedere.

Starvation (inedia). Un thread si trova in *starvation* quando, pur essendo pronto per essere eseguito, non ottiene mai accesso al processore o alla risorsa di cui ha bisogno, perché altri thread vengono sistematicamente preferiti dallo scheduler. Il thread non è bloccato in senso stretto: è *ready*, ma di fatto non avanza mai. Esempio tipico: uno scheduler con priorità in cui thread ad alta priorità arrivano in continuazione, impedendo l'esecuzione di un thread a bassa priorità.

Deadlock (stallo). Due (o più) thread si trovano in *deadlock* quando ciascuno aspetta una risorsa che l'altro detiene, e nessuno dei due è disposto a rilasciare la propria. Il

sistema si blocca completamente: nessun thread avanza. Esempio classico: il thread A ha acquisito la risorsa R1 e aspetta R2; il thread B ha acquisito R2 e aspetta R1. Nessuno dei due può procedere.

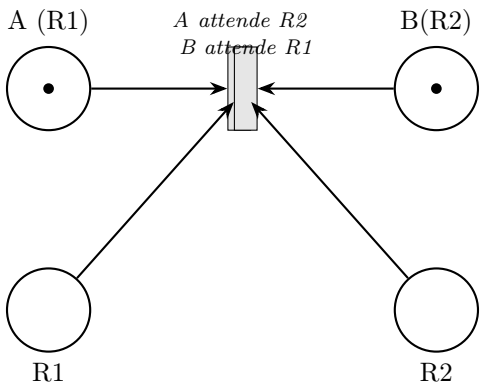

Figure 1: Deadlock tra due thread: nessuna transizione è abilitata. I posti R1 e R2 sono vuoti perché detenuti rispettivamente da A e B, ma le transizioni richiedono entrambe le risorse.

Nelle reti di Petri (il grafo qui sopra) il deadlock significa che nessuna transizione è abilitata e il sistema non può progredire.

1 Istanziazione Diretta (`extends Thread`)

Il thread coincide con l'oggetto creato, in questo caso. La rete modella l'intero ciclo di vita dell'entità: allocazione, esecuzione, terminazione. Quando si chiama `start()`, la JVM crea un nuovo flusso di esecuzione e invoca `run()` su di esso (prima o poi, come abbiam visto); il thread padre prosegue in parallelo, senza aspettare.

```
1 class MyThread extends Thread {
2     public void run() { /* corpo del thread */ }
3 }
4 new MyThread().start();
```

Il token (pallino dentro "new", qui sotto) rappresenta il thread: lo scatto di *start* lo porta da "new" a "running"; lo scatto di *end* lo porta a "terminated".

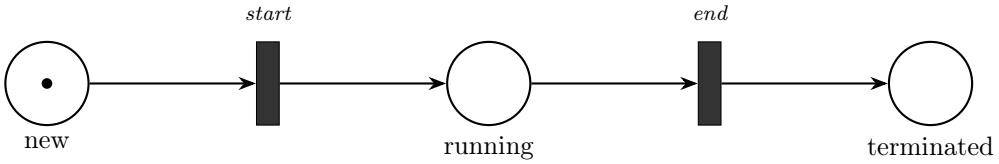

Figure 2: Ciclo di vita di un thread con `extends Thread`. Il token parte in *new*.

Domanda. Cosa succederebbe alla rete se si chiamasse `start()` due volte sullo stesso oggetto?

2 Disaccoppiamento Funzionale (`Runnable` e lambda)

Separazione tra l'esecutore (`Thread`) e il compito (`Runnable`). Il compito è una risorsa inerte finché un thread non lo "consuma". Questo approccio è preferito perché la classe che implementa `Runnable` può ancora estendere un'altra classe.

```
Runnable r = () -> { /* compito */ };
new Thread(r).start();
```

La transizione *bind* richiede la presenza simultanea di un token in *task* e di uno in *thread*. Solo in presenza di entrambi il thread diventa attivo.

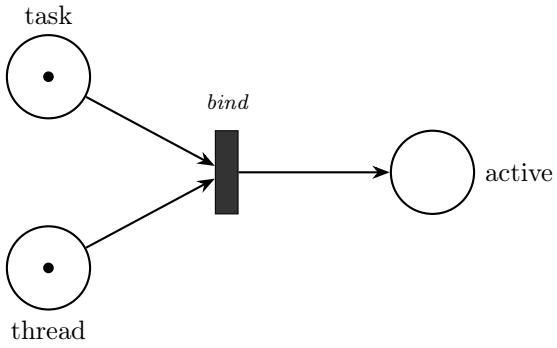

Figure 3: La transizione *bind* ha due posti di ingresso: entrambi devono contenere un token per abilitarla.

3 Accesso a Metodi Statici non Sincronizzati

Concorrenza pura: più thread accedono allo stesso metodo statico senza alcun vincolo di mutua esclusione. La transizione non limita il numero di token che possono scattare in parallelo. Poiché lo scheduler può interleave le esecuzioni in qualunque ordine, e poiché non possiamo fare ipotesi temporali, ogni situazione intermedia deve essere considerata possibile.

```
static class Utility { static void doWork() {} }
new Thread(() -> Utility.doWork()).start();
new Thread(() -> Utility.doWork()).start();
```

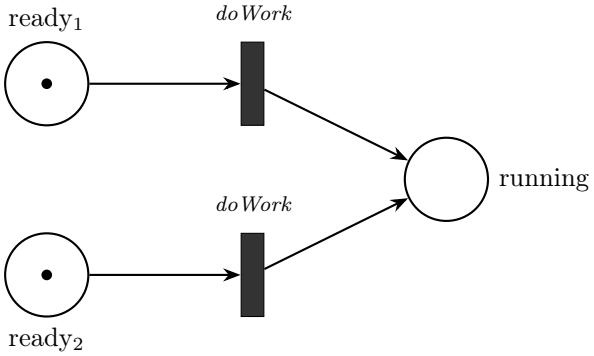

Figure 4: Nessun posto mutex condiviso: entrambe le transizioni possono scattare contemporaneamente.

Domanda. Esiste un limite massimo di thread allocabili / eseguibili in parallelo?

4 Sospensione Temporizzata (`Thread.sleep`)

Il thread transita in uno stato di attesa per una durata fissa τ millisecondi. Durante questo periodo il posto *sleeping* trattiene il token e la transizione di uscita non è abilitata. È importante notare: `sleep` **non** rilascia eventuali lock acquisiti; il thread li mantiene anche mentre dorme.

```
new Thread(() -> {
    try { Thread.sleep(500); }
    catch (InterruptedException e) {}
}).start();
```

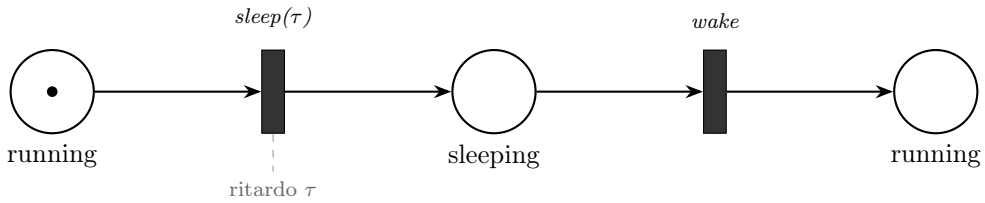

Figure 5: Il posto *sleeping* blocca il progresso del token per la durata τ (rete di Petri temporizzata).

Domanda. Come si modifica la rete se un altro thread può interrompere il sleep tramite `interrupt()`?

5 Race Condition

In assenza di protezione, due thread possono leggere lo stesso valore prima che uno dei due abbia completato la scrittura. Il risultato finale è non deterministico e potenzialmente errato. Questo è esattamente il motivo per cui non si possono fare ipotesi temporali: il problema si manifesta solo in certi scenari, quindi potrebbe (ad esempio) non emergere nei test e apparire in produzione. Per la gioia del cliente :-)

```
static int count = 0;
Runnable r = () -> count++;
```

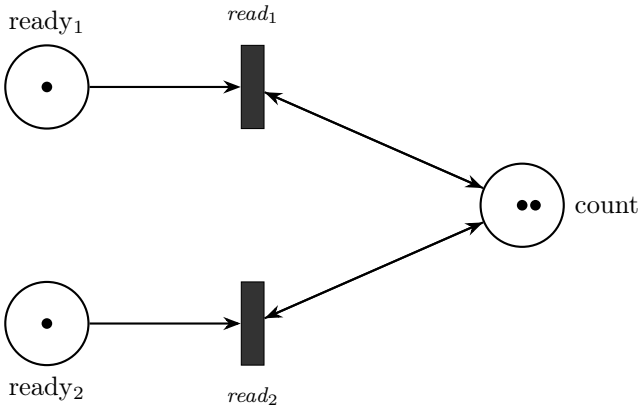

Figure 6: Entrambe le transizioni accedono contemporaneamente a *count*: la marcatura finale può essere incoerente.

6 Mutua Esclusione (blocco `synchronized`)

Il monitor viene modellato come un posto con un unico token. Finché quel token non è disponibile, nessun thread può entrare nella sezione critica. Il blocco `synchronized` in Java acquisisce il lock all'ingresso e lo rilascia automaticamente all'uscita, anche in caso di eccezione.

```
synchronized(lock) {
    /* Sezione Critica */
}
```

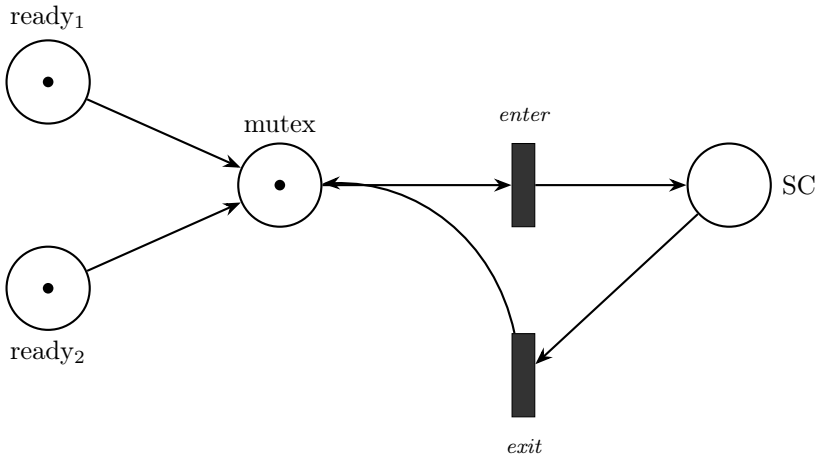

Figure 7: Il posto *mutex* con un solo token garantisce che al più un thread alla volta entri nella sezione critica.

Domanda. Ragiona sempre sulle reti mostrate, e immagine il token "muoversi" sulla stessa.

7 Coordinazione via `wait()` e `notify()`

Un thread che non trova la condizione soddisfatta rilascia il monitor e si sposta nel Wait Set, dove resta sospeso finché un altro thread non lo risveglia con `notify()`.

```
synchronized(lock) {
    while (!cond) lock.wait();
    /* ... usa la risorsa in mutex ... */
}
synchronized(lock) {
    cond = true;
    lock.notify();
}
```

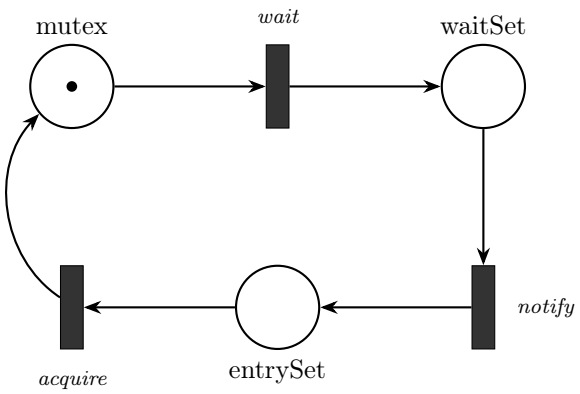

Figure 8: Il thread sospeso con `wait()` libera il mutex e si sposta nel Wait Set. `notify()` lo trasferisce all'Entry Set, da cui può riacquisire il monitor.

Domanda. Perché la condizione di risveglio va controllata con un `while` e non con un `if`? Cosa può accadere nella rete se si usa `if`?

8 Lock di Classe (Metodi Statici Sincronizzati)

Tutti i thread condividono un unico lock di classe, indipendentemente dall'istanza da cui provengono. L'intera classe diventa un collo di bottiglia strutturale: un solo thread alla volta può eseguire qualunque metodo statico sincronizzato di quella classe, in qualunque parte del programma venga invocato.

```
1 static synchronized void globalAction() { /* ... */ }
```

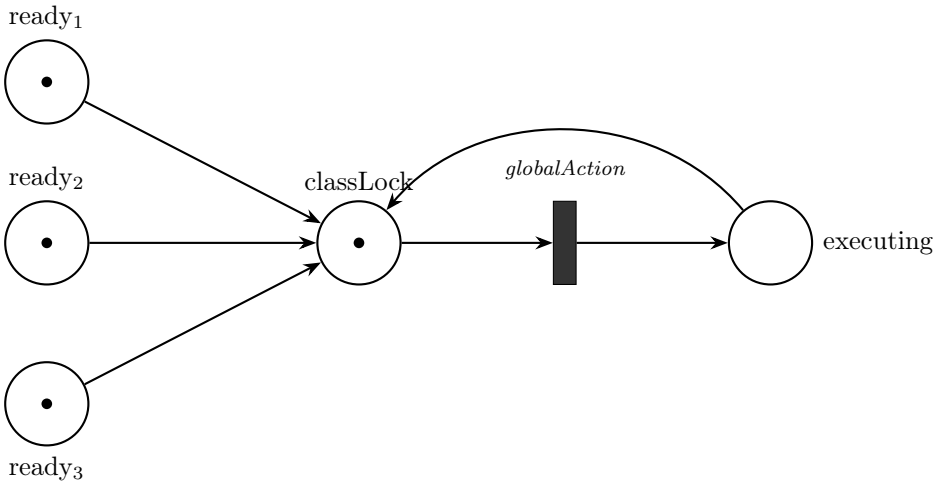

Figure 9: Tutti i thread convergono sullo stesso posto *classLock*: solo uno alla volta può scattare.

Domanda. Quali sono le implicazioni sul throughput di un sistema in cui molte operazioni usano metodi statici sincronizzati? Che alternativa architetturale si potrebbe adottare?

9 Segnalazione Multipla (`notifyAll`)

A differenza di `notify()`, che risveglia un solo thread arbitrario (scelto dallo scheduler, senza garanzie), `notifyAll()` svuota l'intero Wait Set trasferendo tutti i token nell'Entry Set contemporaneamente. È la scelta più sicura quando più thread attendono condizioni diverse sullo stesso monitor.

```
synchronized(lock) { lock.notifyAll(); }
```

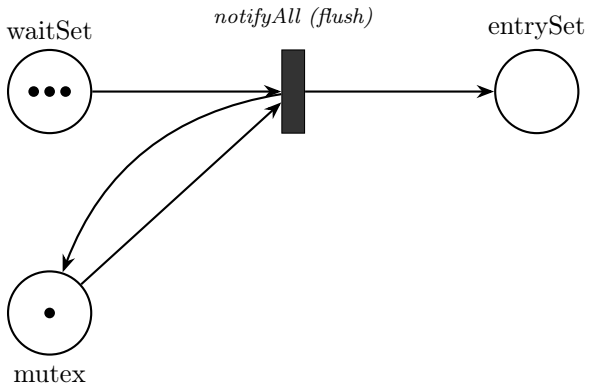

Figure 10: La transizione *flush* svuota *waitSet* in un unico scatto. I token in *entrySet* competono per riacquisire il mutex.

Domanda. In quali scenari `notifyAll()` è necessario rispetto a `notify()`? Costruire un esempio con due tipi di thread in attesa della stessa condizione.

10 Il Bounded Buffer (Produttore-Consumatore)

Il buffer di capacità N integra tutti i meccanismi precedenti: conteggio degli slot, mutua esclusione, attesa condizionata e risveglio multiplo. Il produttore aspetta se il buffer è pieno; il consumatore aspetta se è vuoto; entrambi notificano l'altro dopo ogni operazione.

```
public synchronized void put(Object x)
        throws InterruptedException {
    while (isFull()) wait();
    add(x);
    notifyAll();
}

public synchronized Object take()
        throws InterruptedException {
    while (isEmpty()) wait();
    Object x = remove();
    notifyAll();
    return x;
}
```

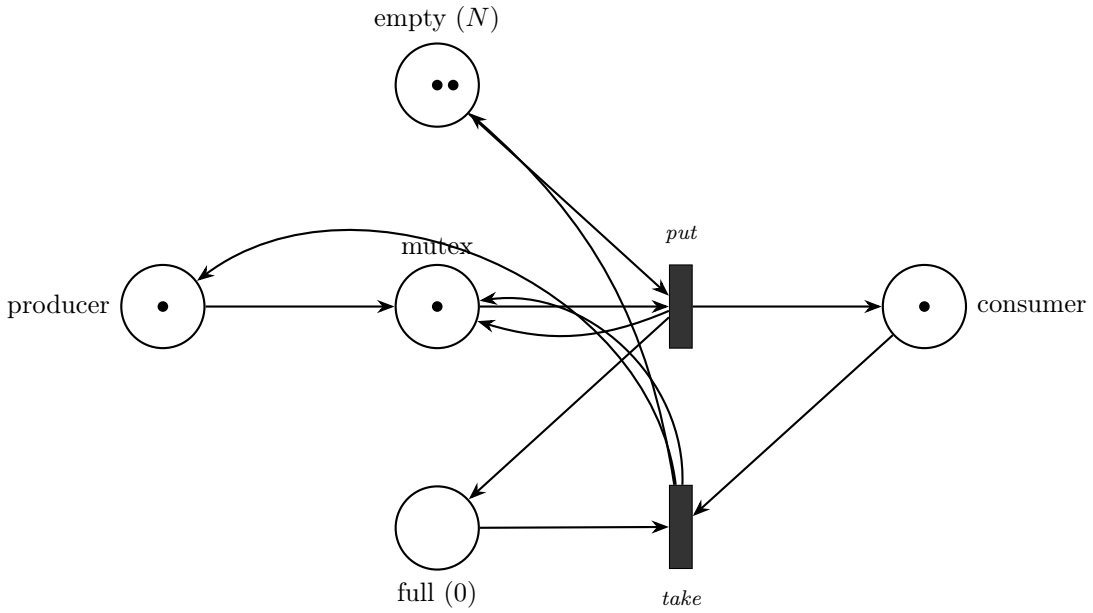

Figure 11: Rete completa del Bounded Buffer. *put* consuma uno slot vuoto e produce uno pieno; *take* fa il contrario. Il mutex serializza l'accesso.

Domanda. Come cambia la rete se si vogliono permettere più produttori e più consumatori in parallelo, mantenendo la correttezza del buffer? Quali posti o transizioni vanno replicati e quali no?

10 Esercizio campione: generatore di numeri primi con thread

Traccia

Si vogliono calcolare i primi N numeri primi usando due thread che lavorano in parallelo: un **thread generatore** che produce candidati interi da testare, e un **thread verificatore** che controlla se ciascun candidato è primo e, in caso affermativo, lo stampa. La comunicazione avviene tramite una singola variabile condivisa `candidato`: il generatore scrive il prossimo numero, il verificatore lo legge. Si garantisce che nessun numero venga letto due volte e nessuno venga saltato.

Livello: uso di `synchronized`, `wait()`, `notifyAll()`, e variabili di controllo booleane.

Auto-apprendimento / Verifica

Sostituire ogni segnaposto **[N=1,2,3,...]** con l'elemento corretto indicato nella lista a fondo pagina.

```
1 public class PrimiThread {
2
3     // Numero di primi da trovare
4     static final int QUANTI = 10;
5
6     // Variabile condivisa tra i due thread
7     static int candidato = 2;
8
9     // true = il generatore ha depositato un nuovo numero
```

```
10      // false = il verificatore ha gia' letto l'ultimo numero
11      static boolean pronto = false;
12
13      // true = il verificatore ha finito il suo lavoro
14      static boolean fine = false;
15
16      // Oggetto usato come monitor (lock condiviso)
17      static final Object lock = [1];
18
19      public static void main(String[] args) {
20          Thread generatore = new Thread(() -> {
21              int n = 2;
22              while ([2]) {
23                  [3](lock) {
24                      while (pronto) {
25                          try { lock.[4](); }
26                          catch (InterruptedException e) {}
27                      }
28                      candidato = n;
29                      pronto = [5];
30                      n++;
31                      lock.[6]();
32                  }
33              }
34          });
35
36          Thread verificatore = new Thread(() -> {
37              int trovati = 0;
38              while (trovati < QUANTI) {
39                  [3](lock) {
40                      while (![7]) {
41                          try { lock.[4](); }
42                          catch (InterruptedException e) {}
43                      }
44                      if (isPrimo(candidato)) {
45                          System.out.println("Primo trovato: "
46                              + candidato);
47                          trovati++;
48                      }
49                      pronto = [8];
50                      if (trovati == QUANTI) fine = [5];
51                      lock.[6]();
52                  }
53              }
54          });
55
56          generatore.[9]();
57          verificatore.[9]();
58
59          try {
60              verificatore.[10]();
61          } catch (InterruptedException e) {}
62
63          System.out.println("Fatto.");
64      }
65
66      static boolean isPrimo(int n) {
67          if (n < 2) return false;
```

```
68          for (int i = 2; [11]; i++) {
69              if (n % i == 0) return false;
70          }
71          return true;
72      }
73  }
```

Lista degli elementi

[1] `new Object()` — crea l'oggetto monitor sul quale sincronizzarsi

[2] `!fine` — il generatore continua finché il verificatore non segnala di aver finito

[3] `synchronized` — acquisisce il lock sull'oggetto `lock` prima di accedere alle variabili condivise

[4] `wait` — sospende il thread e rilascia il lock, in attesa di una `notify`

[5] `true` — imposta il flag a vero (il generatore ha depositato; oppure il lavoro è finito)

[6] `notifyAll` — risveglia tutti i thread in attesa sul monitor `lock`

[7] `pronto` — il verificatore attende finché non c'è un nuovo candidato disponibile

[8] `false` — segnala che il candidato è stato consumato e il generatore può scriverne uno nuovo

[9] `start` — avvia il thread creando un nuovo flusso di esecuzione nella JVM

[10] `join` — il main si sospende e aspetta che il thread verificatore termini

[11] `i * i <= n` — condizione di terminazione del ciclo di primalità: basta testare divisori fino a $\sqrt{n}$

Domanda. Perché `join()` viene chiamato solo sul *verificatore* e non anche sul *generatore*? In che condizione il generatore potrebbe rimanere bloccato per sempre dopo che il verificatore è terminato?