Filosofi a cena: esempio di programmazione concorrente

Luca Manini

2014

1 Introduzione

Classico problema di concorrenza, legato alla condivisione di risorse e all'accesso condiviso a qualche informazione "aggiornabile". Il problema è il seguente: ci sono cinque filosofi a cena, che passano la serata alternando momenti in cui pensano (*think*) a momenti in cui mangiano (*eat*). Ogni filosofo ha il suo piatto (spaghetti), ma ci sono solo cinque forchette (ai lati dei piatti) e ogni filosofo ha bisogno delle due forchette al lato dei suo piatto per mangiare! Morale: solo due filosofi, non "contigui" possono mangiare contemporaneamente.

La prima soluzione che viene in mente è: quando un filosofo vuole mangiare guarda se le due posate sono disponibili e se lo sono le prende e mangia per un po' (togliendo quindi ai suoi vicini la possibilità di mangiare) e poi posa le posate.

Il problema è che, essendo i cinque filosofi processi "paralleli", può succedere che tutti vedono le posate disponibili, tutti prendono quella alla loro sinistra e poi non possono però prendere quella a destra: sono tutti bloccati. Il nocciolo del problema è che le due azioni di controllare la disponibilità delle due forchette e di prenderle devono costituire un'unica azione "atomica" (proprio nel senso di indivisibile) che o ha successo globalmente o fallisce globalmente, senza mai permettere che la prima delle due abbia successo e che la seconda fallisca. La ragione fondamentale per cui la seconda può fallire e che tra l'esecuzione della prima e quella della seconda si "inserisca" un'azione di un altro filosofo; la soluzione sta quindi nel "sincronizzare" in modo intelligente le varie operazioni, limitando in un certo senso il parallelismo.

2 La soluzione classica

La soluzione classica è quella proposta da Dijstra nel 1956 e riportata in tutti i libri di testo, tra i quali il libro *Modern Operating Systems* di Andrew S. Tanenbaum (AST nel seguito) da cui ho preso la soluzione che illustro di seguito (un po' modificata).

Un esempio di "non soluzione", che non credo richieda spiegazioni, è il seguente:

```
#define N 5                     /* number of philosophers */
void philosopher (int i) {
```

```
  while (1) {

    think();
    take_fork(i);
    take_fork((i+1) % N);
    eat();
    put_fork(i);
    put_fork((i+1) % N);
  }
}
```

Un primo miglioramento potrebbe essere quello di proteggere tutto il corpo del `for`, esclusa la chiamata a `think` da un semaforo, ma così al massimo un solo filosofo potrebbe mangiare in ogni momento, mentre è ovvio che con cinque forchette si potrebbe arrivare a due!

La soluzione è avere:

1. un vettore che mantiene lo stato (*thinking', /hungry* e *eating*) di ciascun filosofo;
2. un singolo *mutex* per proteggere le *critical region* in cui si accede al vettore stesso;
3. un vettore di semafori su cui i filosofi possono "bloccarsi" in attesa della disponibilità di forchette.

3 L'implementazione

L'implementazione di AST è composta da cinque parti fondamentali:

1. le strutture dati;
2. la funzione `philosopher`;
3. la funzione `take_forks`;
4. la funzione `put_forks`;
5. la funzione `test`.

3.1 Le strutture dati

Ciò che serve sono i `define` di alcune costanti e le due macro `LEFT` e `RIGHT` per ottenere facilemente gli indici dei due filosofi "vicini". Tanenbaum usa gli interi come semafori, mentre io uso il tipo `sem_t` (predefinito in `thread.h`); per questo può inizializzare il *mutex* direttamente a 1, mentre io lo faccio nel `main` con la funzione `up` (*alias* di `sem_post`).

```
#define N 5
#define LEFT  (i-1) % N
#define RIGHT (i+1) % N
#define THINKING 0
#define HUNGRY   1
#define EATING   2
typedef int semaphore;
semaphore mutex = 1;
semaphore s[N];
int state[N];
```

3.2 La funzione philosopher

La funzione philosopher è banale, perché tutte le difficoltà sono nascoste in take_forks e put_forks. Nella mia implementazione, think e eat accettano un argomento (l'indice i) per permettere il *logging*.

```
void philosopher(int i) {

  while (1) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}
```

3.3 La funzione take_forks

Qui cominciano le difficoltà! Vediamo di capire cosa succede:

1. le due chiamate su mutex servono a proteggere la sezione critica, ossia l'accesso diretto o indiretto (via test) al vettore state;
2. il filosofo prima di cambia il suo stato in HUNGRY e poi chiama la funzione test che dopo controlla se è il caso e se è possibile prendere le forchette (ovvero se i due filosofi vicini non stanno mangiando) e se tutto va bene passa allo stato EATING e fa un up su s[i];
3. poi il filosofo esegue una down su s[i] che lo blocca se nella test non è stato eseguita la up corrispondente (verrà poi eventualmente sboccato dai vicini, vedremo come).

```
void take_forks(int i) {

  down(&mutex);
  state[i] = HUNGRY;
```

```
  test(i);
  up(&mutex);
  down(&s[i]);
}
```

3.4 La funzione `put_forks`

La funzione `put_forks` è relativamente semplice, le due chiamate a `test` servono a "sbloccare" (se ce ne fosse bisogno) i due vicini.

```
void put_forks(int i) {

  down(&mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  up(&mutex);
}
```

3.5 La funzione `test`

La funzione `test` è forse quella più "magica" e più difficile da capire (anche perché il nome non è forse scelto in modo ottimale). In pratica *offre* ad un filosofo la possibilità di mangiare a patto che sia nello stato `HUNGRY` e che i suoi vicini non stiamo mangiando. È importante notare che la funzione è chiamata da un filosofo in due "modi" e in due "occasioni" differenti:

1. su sé stesso nella `take_forks`,
2. sui vicini nella `put_forks`.

```
void test(int i) {

  if (state[i]     == HUNGRY &&
      state[LEFT]  != EATING &&
      state[RIGHT] != EATING) {

    state[i] = EATING;
    up(&s[i]);
  }
}
```

4 Una ricostruzione (in Python)

Ammesso che la soluzione di Dijstra non mi sarebbe mai venuta in mente, ma che però ho capito come funziona, adesso voglio vedere di trovare un "racconto" di una sua "ricostruzione" e già che ci sono usando Python!

Comincio con un parametro che indica il numero dei filosofi, una lista che mantenga lo stato di ciascuno di loro e un semaforo che protegga l'accesso alla lista (inizialmente disponibile).

```
import threading
import random
import time
import sys

PHIL_COUNT = 5
THINKING, HUNGRY, EATING = range(3)
STATE = [THINKING] * PHIL_COUNT
SEM_STATE = threading.Semaphore(1)
```

In molti punti dell'implementazione devo fare riferimento ad un dato filosofo e ai due che siedono alla sua sinistra e alla sua destra. Siccome se faccio un disegno della tavola numero i filosofi in senso orario, mi viene naturale usare l'indice successivo per il filosofo di sinistra e il precedente per quello di destra. Definisco delle funzioni che implementano questa scelta:

```
def LEFT(i):
    return (i + 1) % PHIL_COUNT

def RIGHT(i):
    return (i - 1) % PHIL_COUNT
```

Definisco alcune funzioni per la gestione dei semafori. Preferisco i nomi `wait` e `signal` quando uso i semafori per sincronizzare *thread* distinti, mentre trovo più espressivi `acquire` e `release` quando li uso come "lock" per proteggere delle *critical section*.

```
def wait(sem):
    sem.acquire()

def signal(sem):
    sem.release()

acquire = wait
release = signal
```

Poi posso scrivere le funzione `philosopher`, `think` e `eat` che sono abbastanza ovvie e le funzioni accessorie `get_think_time` e `get_eat_time` almeno in una loro versione banale.

```
def get_think_time(i):
  return random.random()

def get_eat_time(i):
```

```
  return random.random()

def show(s):
    sys.stderr.write(s + "\n")

def think(i):
  show("THINK %d" % i)
  t = get_think_time(i)
  time.sleep(t)

def eat(i):
  show("EAT %d" % i)
  t = get_eat_time(i)
  time.sleep(t)

RUN = True

def philosopher(i):

  while RUN:
      think(i)
      take_forks(i)
      eat(i)
      put_forks(i)
```

Adesso comincia la parte difficile: ricostruire `take_forks` e `put_forks`.

Forse è più facile cominciare da `put_forks`, per la quale la successione delle operazioni mi pare più intuitiva. Il filosofo si può mettere subito nello stato `THINKING` per poi offrire ai suoi due vicini la possibilità di mangiare.

```
def put_forks(i):
    acquire(SEM_STATE)
    STATE[i] = THINKING
    test_chance(LEFT(i))
    test_chance(RIGHT(i))
    release(SEM_STATE)
```

Ora il problema è cosa mettere in `test_chance` (che è la `test` della soluzione originale). Si può intuire che dovrà includere una `signal` su un semaforo su cui i vicini potrebbero essere in `wait`. Mi serve quindi una lista `TURN` di semafori.

```
TURN = [threading.Semaphore(0) for _ in range(PHIL_COUNT)]
```

D'altra parte un filosofo può sfruttare l'offerta solo se è nello stato `HUNGRY` e se i **suoi** vicini non stanno mangiando! Una prima approssimazione potrebbe quindi essere la seguente:

```
def test_chance(i):
```

```
    if (STATE[i] == HUNGRY and
        STATE[LEFT(i)] != EATING and
        STATE[RIGHT(i)] != EATING):

        signal(TURN[i])
```

Provo adesso a scrivere `take_forks`. La prima cosa da fare è "prenotarsi" per una possibile *chance* passando nello stato `HUNGRY`, poi provare la fortuna con `test_chance` e mettersi in `wait`. Il trucco qui è che mi metto in `wait` su un semaforo che, forse, è stato appena incrementato (nella `test_chance`).

```
def take_forks(i):
    acquire(SEM_STATE)
    STATE[i] = HUNGRY
    test_chance(i)
    release(SEM_STATE)
    wait(TURN[i])
```

Resta il problema di dove fare il passaggio a `EATING`! Ovviamente devo farlo all'interno di una *critical section* protetta da `SEM_STATE`, ma d'altra parte la `wait` ne deve rimanere fuori (altrimenti rischio di bloccare tutto). Potrei quindi pensare di passare a `EATING` all'interno di `test_chance`!

```
def test_chance(i):
    if (STATE[i] == HUNGRY and
        STATE[LEFT(i)] != EATING and
        STATE[RIGHT(i)] != EATING):

        STATE[i] = EATING
        signal(TURN[i])
        ii = [str(i) for i in range(PHIL_COUNT) if STATE[i] == EATING]
        s = ", ".join(ii)
        show("Eaters: %s" % s)
        if len(ii) > 2:
            raise Exception("Cazzarola")
```

Adesso serve un minimo di `main` per provare il tutto!

```
def main():

    tt = [threading.Thread(target=philosopher,
                           args=(i,))
                           for i in range(PHIL_COUNT)]
    for t in tt:
        t.start()

    for t in tt:
        t.join()
```

```
if __name__ == '__main__':
    main()
```