Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

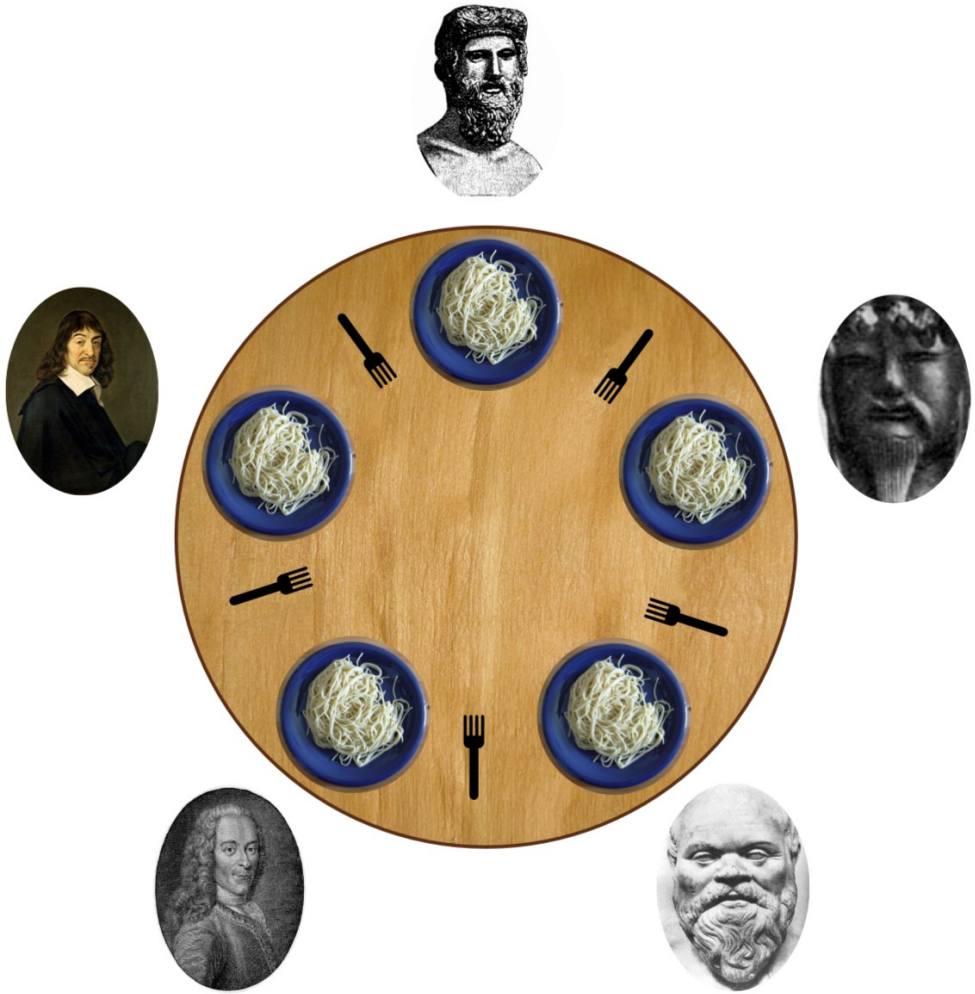
SUPSI

La cena dei filosofi

Amos Brocco, Ricercatore, DTI / ISIN

14 maggio 2012

Il problema dei filosofi a cena



Il problema dei filosofi a cena

Il problema: la pasta è scivolosa e quindi a ogni filosofo, per mangiare servono due forchette.

I filosofi concordano di condividere le forchette: ogni forchetta è condivisa fra i due filosofi vicini (soluzione poco igienica, ma funzionale).

Il problema per l'informatico

Simulare il comportamento dei filosofi con un programma:

- Ogni filosofo rappresenta un thread
  - Ogni filosofo vuole il massimo di autonomia di comportamento, alternando periodi ragionevoli durante i quali pensa o mangia.
  - Se pensa all'infinito non pone problemi agli altri, ma muore di fame
  - Se mangia all'infinito fa morire di fame i vicini.

- Le forchette rappresentano le risorse condivise

Filosofi a cena: sviluppo della soluzione

```
thread filosofo(int i) {

  while(TRUE) {
        pensa (per un certo tempo),
        afferra la forchetta di sinistra,
        afferra la forchetta destra,
        mangia (per un certo tempo),
        depone la forchetta sinistra
        depone la forchetta destra
  }
}
```

Nota: I tempi (pensa e mangia) possono essere scelti in modo casuale

Filosofi a cena: sviluppo della soluzione

Idea: se ogni forchetta è una risorsa condivisa, associamo ad ogni forchetta un mutex

```
thread filosofo(int i) {

 while(TRUE) {
    pensa (per un certo tempo)
    blocca mutex(i)         // afferra forchetta sinistra
    blocca mutex[(i+1)%N]   // afferra forchetta destra
    mangia (per un certo tempo)
    sblocca mutex(i)        //depone forchetta sinistra
    sblocca mutex[(i+1)%N] //depone forchetta destra
 }
}
```

...una soluzione apparentemente corretta... oppure no?

Filosofi a cena: sviluppo della soluzione

Idea: se ogni forchetta è una risorsa condivisa, associamo ad ogni forchetta un mutex

```
thread filosofo(int i) {

 while(TRUE) {
    pensa (per un certo tempo)
    blocca mutex(i)         // afferra forchetta sinistra
    blocca mutex[(i+1)%N]   // afferra forchetta destra
    mangia (per un certo tempo)
    sblocca mutex(i)        //depone forchetta sinistra
    sblocca mutex[(i+1)%N] //depone forchetta destra
 }
}
```

Se il filosofo *i* blocca il mutex *i* e non riesce a bloccare il mutex *(i+1)%N* poiché nel frattempo è stato bloccato dal filosofo *(i+1)%N* e questo si ripete per tutti i filosofi si ha un **deadlock**!

Situazioni critiche

- Se tutti i 5 filosofi afferrano nello stesso istante la forchetta alla loro sinistra (situazione poco probabile ma possibile su un sistema multiprocessore), si genera un deadlock.

- Se un filosofo afferra la forchetta di sinistra e deve aspettare all'infinito la forchetta di destra, fa morire di fame il collega a sinistra.

Filosofi a cena: sviluppo della soluzione

Come si comporta il filosofo se una forchetta è occupata ?

- **Aspetta**
  - Può produrre dei dead lock
- **Rinuncia e torna a pensare**
  - Può fare morire di fame un filosofo (starvation)

Filosofi a cena: sviluppo della soluzione

Cerchiamo una soluzione ottimale che garantisca la massima indipendenza fra i filosofi, e che eviti deadlock e starvation!

Osservazioni:

- Tutti possono pensare contemporaneamente
- Con 5 filosofi, e 5 forchette, fino a 2 filosofi possono mangiare contemporaneamente

Filosofi a cena: sviluppo della soluzione

```
thread filosofo(int i) {

  while(TRUE) {
        pensa (per un certo tempo),
        afferra le forchette,
        mangia (per un certo tempo),
        depone le forchette.
  }
}
```

Raggruppiamo l'acquisizione e il rilascio delle forchette in due funzioni

Filosofi a cena: sviluppo della soluzione

La procedura per afferrare le forchette deve essere migliorata:

- **può servire un semaforo (per ogni filosofo)?**

```
procedura afferra_forchette(i) { // i si riferisce al filosofo
    test(i) // verifica se  le due forchette sono libere
            // (senza  bloccarsi all'interno di una sezione
            //  critica)

    aspetta semaforo(i)  //  se è necessario aspettare, l'attesa avviene
                         //  fuori dalla sezione critica.
                         //  il semaforo sarà disponibile quando i
                         //  filosofi vicini avranno deposto le forchette
                         //  che interessano al filosofo i
}
```

Il semaforo dovrà essere incrementato solo se l'interessato sta aspettando, altrimenti c'è il rischio che il semaforo venga incrementato inutilmente!

Filosofi a cena: sviluppo della soluzione

Come testare se le forchette dei vicini sono libere ?

- Le forchette si possono prendere se i vicini non stanno mangiando
  - Quando un filosofo mangia deve mettersi nello **stato=MANGIA**

```
#define sinistro  (i+ N-1)%N
#define destro    (i+1)%N

procedura test(i) {
          Se  stato(sinistro) != MANGIA e  stato(destro)  != MANGIA) {
               stato(i) = MANGIA // se non mangiano i vicini,mangio io
               incrementa semaforo(i)
           }
}
```

Filosofi a cena: sviluppo della soluzione

Come faccio a capire se i vicini sono in attesa anche loro di qualche forchetta?

- Un filosofo vuole prendere le forchette se è affamato, e deve segnalarlo mettendosi nello **stato=AFFAMATO**

```
procedura afferra_forchette(i) {
    stato(i)=AFFAMATO          // Segnala che è affamato
    test(i)                    // verifica se le due forchette sono libere
    aspetta semaforo(i)        // aspetta se non ha le forchette
}
```

Filosofi a cena: sviluppo della soluzione

La stessa funzione di test potrà essere usata anche dai vicini quando deporranno le forchette per incrementare il semaforo.

- Siccome il semaforo *i* si può incrementare a condizione che il filosofo *i* sia **AFFAMATO**, è necessario completare il test come segue:

```
procedura test(i) {
          Se stato(i) = AFFAMATO
          e stato(sinistro) != MANGIA
          e  stato(destro)  != MANGIA {
              stato(i) = MANGIA
              incrementa semaforo(i) // Questo permetterà di non aspettare
                                     // in afferra_forchette
         }
}
```

Filosofi a cena: sviluppo della soluzione

Esistono momenti dove ogni filosofo è in uno stato diverso da "AFFAMATO" o "MANGIA" , cioè quando pensa: introduciamo perciò anche lo stato "PENSA"

- Questo stato viene assunto quando il filosofo depone le forchette
- La funzione depone_forchette deve preoccuparsi di eventualmente aprire i semafori dei vicini. Questo si realizza chiamando la stessa funzione **test().**

```
procedura depone_forchette(i) {
        stato(i) = PENSA
        test(sinistra) // Il vicino a sinistra mangia, se era affamato e ha le forchette
        test(destra) // Il vicino a destra mangia, se era affamato e ha le forchette
}
```

Filosofi a cena: sviluppo della soluzione

```
thread filosofo(int i) {

 while(TRUE) {
    pensa (per un certo tempo)
    afferra_forchette(i)
    mangia (per un certo tempo)
    depone_forchette(i)
 }
}
```

Abbiamo introdotto altre variabili condivise oltre alle forchette, come gli stati... non manca qualcosa?

Filosofi a cena: sviluppo della soluzione

```
procedura depone_forchette(i) {
      blocca mutex
      stato(i) = PENSA
      test(sinistra)    // i test non si
                        // bloccano, nessun
                        // pericolo di deadlock
      test(destra)
      sblocca mutex
}
```

Durante il test le variabili non devono essere modificate da altri

Filosofi a cena: sviluppo della soluzione

```
procedura afferra_forchette(i) {
     blocca mutex
     stato(i)=AFFAMATO
     test(i)
     sblocca mutex
     aspetta semaforo(i)  // l'attesa, se
                          // necessaria, avviene
                          // fuori dalla sezione
                          // critica, nessun
                          // pericolo di deadlock
}
```

Filosofi a cena: sviluppo della soluzione

- Le varabile **stato[N]** sono lette e scritte da una sola funzione per volta
- Le due funzioni non possono mai bloccarsi

La soluzione completa

```
#define N 5
#define SINISTRA (i+N-1)%N
#define DESTRA   (i+1)%N
#define PENSA    0
#define AFFAMATO 1
#define MANGIA   2

#define up(s) sem_post(s)
#define down(s) sem_wait(s)

int stato[N];
pthread_mutex_t mutex; // Il mutex deve essere sbloccato all'inizio
pthread_sem_t semaforo_filosofo[N];
```

La soluzione completa

```
void filosofo (int i)
{
    while (TRUE) {
        pensa();
        afferra_forchette(i);
        mangia();
        deponi_forchette(i);
    }
}
```

La soluzione completa

```
void afferra_forchette(int i)
{
    pthread_mutex_lock(&mutex);
    stato[i] = AFFAMATO;
    test(i);
    pthread_mutex_unlock(&mutex);
    down(&semaforo_filosofo[i]);
}

void deponi_forchette(i)
{
    pthread_mutex_lock(&mutex);
    stato[i] = PENSA;
    test(SINISTRA);
    test(DESTRA);
    pthread_mutex_unlock(&mutex);
}
```

La soluzione completa

```
void test(i)
{
  if (stato[i] == AFFAMATO
      && stato[SINISTRA] != MANGIA
      && stato[DESTRA] != MANGIA) {
    stato[i] = MANGIA;
    up(&semaforo_filosofo[i]);
  }
}
```

Il monitor

Concetto informatico introdotto C.A.R.Hoare (1974) e P.B.Hansen (1975)

- **synchronized** in Java

Il *monitor* è un insieme di

- Variabili
- Procedure
- Sequenza di inizializzazione (dei dati)

Le variabili sono accessibili solo tramite le procedure del monitor

Si dice che una *thread* (o un processo) **entra** nel monitor, quando invoca una sua procedura

Solo una *thread* (un processo) per volta può entrare nel *monitor*

Una procedura può invocare altre procedure del monitor non direttamente accessibile alle *thread*

Cinque filosofi e un monitor...

- Nel nostro esempio, appartengono al monitor
  - Le variabili stato(i)
  - Le funzioni afferra_forchette(i) e deponi_forchette(i)
- Ma non la funzione test(i), visto che non è invocata direttamente dalle thread, ma dalle funzioni del monitor!

```
while(TRUE) {
    pensa
    afferra_forchette(i) // funzione del monitor
    aspetta semaforo(i)
    mangia
    depone_forchette(i) // funzione del monitor
}
```