Azzolini Riccardo 2019-11-07

Problemi classici di sincronizzazione

1 Problema dei produttori e consumatori

In questo problema (detto anche problema del *bounded buffer*), si hanno:

- un **pool finito di buffer**, ciascuno dei quali può essere pieno (cioè contenere un'informazione), oppure vuoto (inizialmente, i buffer sono tutti vuoti);
- un insieme di **processi produttori** (producer), che scrivono informazioni nei buffer, riempiendoli;
- un insieme di **processi consumatori** (consumer), che consumano (leggono ed eliminano) le informazioni presenti nei buffer, svuotandoli.

Una soluzione di questo problema è valida se:

- l'accesso ai singoli buffer avviene in mutua esclusione (ma sono ammessi gli accessi concorrenti a buffer diversi);
- i produttori non possono riempire i buffer pieni, cioè sovrascrivere informazioni non ancora lette da alcun consumatore;
- i consumatori non possono svuotare i buffer vuoti (e, se due o più consumatori cercano di consumare in concorrenza la stessa informazione, deve riuscirci uno solo di essi).

Talvolta, è richiesta come condizione aggiuntiva una politica FIFO, cioè i buffer devono essere svuotati nello stesso ordine in cui sono stati riempiti.

Questo problema astratto ha numerose istanze concrete, come ad esempio la gestione delle stampe:

- il pool è una coda di stampa;
- i produttori sono i processi utente;
- il (singolo) consumatore è il daemon di stampa.

1.1 Soluzione con race condition

Una possibile soluzione è realizzare il pool con un array gestito in modo circolare (lo si riempie e lo si svuota nella stessa direzione, e, quando si raggiunge la fine, si riparte dall'inizio).

In questo caso, si suppone che i buffer siano 100, e che ciascuno di essi possa contenere un numero intero. Il pool è quindi un array di 100 interi.

Un'implementazione, soggetta però a race condition, di questa soluzione, è:

```
int buffer[100];
int count = 0;
int i = 0;
int j = 0;
```

- Programma del producer:

  ```
  while (true) {
      // ...
      int item = produce_item();
      while (count == 100) {}
      buffer[i] = item;
      count++;
      i = (i + 1) % 100;
      // ...
  }
  ```

- Programma del consumer:

  ```
  while (true) {
      // ...
      while (count == 0) {}
      int item = buffer[j];
      count--;
      j = (j + 1) % 100;
      // ...
  }
  ```

In quest'implementazione:

- la variabile `count` indica il numero di buffer pieni (quindi, assume valori $0 \leq$ `count` ≤ 100);
- `i` è l'indice del primo buffer vuoto, se esiste (cioè se `count` ≤ 99);
- `j` è l'indice del primo buffer pieno, se esiste (cioè se `count` ≥ 1);

- per gestire in modo circolare l'array, gli incrementi di `i` e `j` sono effettuati modulo 100;
- la variabile `item` è privata;
- `produce_item()` è una generica operazione di costruzione di un'informazione da inserire in un buffer (che è irrilevante per la soluzione del problema).

Per prevenire le race condition, sarebbe sufficiente che tutti gli accessi ai dati condivisi (`buffer`, `count`, `i`, `j`) avvengano all'interno di sezioni critiche, ma così:

- si avrebbe *busy wait*;
- un solo processo alla volta potrebbe accedere al pool di buffer, mentre una soluzione migliore potrebbe consentire accessi concorrenti a buffer diversi, dato che essi non causano problemi.

1.2 Soluzione con i semafori: singolo buffer

Nel caso più semplice, in cui il pool è costituito da un singolo buffer, una soluzione corretta (non soggetta alle race condition) si può realizzare con due semafori:

```
semaphore full = 0;
semaphore empty = 1;
```

- `full` assume valore 1 quando il buffer è pieno, e 0 quando è vuoto;
- `empty` vale 1 quando il buffer è vuoto, altrimenti vale 0.

Il codice eseguito dai due tipi di processi è:

- programma del producer:

```
while (true) {
    // ...
    int item = produce_item();
    wait(empty);
    buffer = item;
    signal(full);
    // ...
}
```

- programma del consumer:

```
while (true) {
    // ...
    wait(full);
    int item = buffer;
    signal(empty);
    // ...
}
```

Quindi:

- il producer aspetta che il buffer sia vuoto, inserisce un'informazione, e segnala che il buffer è pieno;
- il consumer aspetta che il buffer sia pieno, consuma l'informazione presente al suo interno, e segnala che il buffer è vuoto.

In generale, un processo controlla/aspetta di poter lavorare, lavora, e infine autorizza un processo dell'altra "specie" a lavorare.

Non è possibile realizzare una soluzione con un solo semaforo, perché alcuni processi (i produttori) devono bloccarsi quando il buffer è pieno e altri (i consumatori) quando il buffer è vuoto, ma, se ci fosse un unico semaforo, i processi potrebbero usarlo per bloccarsi in una sola di queste due situazioni.

1.3 Soluzione con i semafori: caso generale

Nel caso generale, con un array di `N` buffer gestito circolarmente, sono necessari 4 semafori:

```
semaphore full = 0;
semaphore empty = N;
semaphore sem_p = 1;
semaphore sem_c = 1;
```

- `full` indica il numero di buffer pieni;
- `empty` indica il numero di buffer vuoti;
- `sem_p` si usa per la mutua esclusione sull'indice `i` (l'indice del primo buffer vuoto);
- `sem_c` si usa per la mutua esclusione sull'indice `j` (l'indice del primo buffer pieno).

Un processo produttore:

```
while (true) {
    // ...
    int item = produce_item();
    wait(empty);
    wait(sem_p);
    buffer[i] = item;
    i = (i + 1) % N;
    signal(sem_p);
    signal(full);
    // ...
}
```

1. controlla (e, se necessario, aspetta) che ci sia almeno un buffer vuoto;
2. con mutua esclusione su `i`, inserisce un'informazione nel primo buffer vuoto e incrementa `i`;
3. segnala che c'è un nuovo buffer pieno (sbloccando un eventuale processo consumatore che era in attesa, oppure incrementando il valore di `full`).

Simmetricamente, un processo consumatore:

```
while (true) {
    // ...
    wait(full);
    wait(sem_c);
    int item = buffer[j];
    j = (j + 1) % N;
    signal(sem_c);
    signal(empty);
    // ...
}
```

1. controlla (e, se necessario, aspetta) che ci sia almeno un buffer pieno;
2. con mutua esclusione su `j`, consuma l'informazione contenuta nel primo buffer pieno e incrementa `j`;
3. segnala che c'è un nuovo buffer vuoto (sbloccando un eventuale processo produttore che era in attesa, oppure incrementando il valore di `empty`).

Con questa soluzione, possono avvenire in concorrenza una singola lettura e una singola scrittura ma non più letture o più scritture, a causa della mutua esclusione su `i` e `j`: ciò è necessario per evitare letture/scritture multiple dello stesso buffer.

Come nel caso con un singolo buffer, l'uso dei due semafori "ridondanti" `full` e `empty` è necessario per permettere ai processi di bloccarsi sia quando non ci sono buffer pieni

che quando non ci sono buffer vuoti: il valore di `full` si potrebbe ricavare da quello di `empty`, e viceversa, ma non è consentito effettuare calcoli sui semafori.

2 Problema dei lettori e scrittori

In questo problema si hanno:

- un **insieme di dati condivisi** (ad esempio una classe, un array, ecc.);
- un insieme di **processi scrittori** (writer), che modificano i dati;
- un insieme di **processi lettori** (reader), che accedono ai dati in sola lettura (a differenza dei processi consumatori del problema precedente, non consumano/cancellano ciò che leggono).

Una soluzione valida deve rispettare le seguenti regole:

- sono consentite le letture concorrenti;
- non è ammessa la concorrenza tra le scritture (altrimenti, potrebbero verificarsi delle race condition, cioè gli effetti di alcune scritture potrebbero "andare persi");
- non è ammessa la concorrenza tra scrittura e lettura (altrimenti, potrebbero essere letti dei dati inconsistenti).

Osservazione: Tutti i lettori hanno write_set vuoto, quindi sono processi tra loro indipendenti.

Un esempio di istanza concreta di questo problema astratto potrebbe essere:

- il dato è un conto bancario;
- gli scrittori sono i processi che gestiscono prelievi, depositi, ecc.;
- i lettori sono i processi che stampano i dati, leggono il saldo, prelevano dati statistici, ecc.

2.1 Idea della soluzione

- Prima di leggere, un reader deve aspettare che nessun writer stia scrivendo:

```
while (true) {
    ...
    while (someone is writing) {}
    set(I'm reading);
    {read}
    set(I'm not reading);
    ...
```

```
}
```

- Un writer deve aspettare che nessun altro processo stia lavorando (leggendo/scrivendo):

```
while (true) {
    ...
    while (someone is working) {}
    set(I'm writing);
    {write}
    set(I'm not writing);
    ...
}
```

2.2 Soluzione con i semafori

Si suppone di avere `R` lettori e `W` scrittori.

Servono 4 variabili condivise per tener traccia dello stato del sistema:

- `runR`: il numero di lettori che stanno leggendo;
- `runW`: il numero di scrittori che stanno scrivendo;
- `totR`: il numero di lettori che stanno leggendo (`runR`), o che vorrebbero leggere;
- `totW`: il numero di lettori che stanno scrivendo (`runW`), o che vorrebbero scrivere.

I valori di queste variabili rispettano le seguenti proprietà:

$$0 \leq \texttt{runR} \leq \texttt{totR} \leq \texttt{R}$$
$$0 \leq \texttt{runW} \leq \texttt{totW} \leq \texttt{W}$$
$$\texttt{runW} \leq 1$$
$$\texttt{runR} > 0 \rightarrow \texttt{runW} = 0$$
$$\texttt{runW} = 1 \rightarrow \texttt{runR} = 0$$

Si usano poi 3 semafori:

```
semaphore semR = 0;
semaphore semW = 0;
semaphore mutex = 1;
```

- `semR` controlla il lavoro dei lettori ("dando l'ok" per la lettura);
- `semW` controlla il lavoro degli scrittori ("dando l'ok" per la scrittura);
- `mutex` serve per la mutua esclusione sulle variabili condivise.

In ogni istante, i loro valori sono tali che:

$$0 \leq \texttt{semR} \leq \texttt{R}$$
$$0 \leq \texttt{semW} \leq 1$$
$$\texttt{semR} > 0 \rightarrow \texttt{semW} = 0$$
$$\texttt{semW} = 1 \rightarrow \texttt{semR} = 0$$
$$0 \leq \texttt{mutex} \leq 1$$

Il codice eseguito dai processi è:

- reader:

```
while (true) {
    wait(mutex);
    totR++;
    if (runW == 0) {
        runR++;
        signal(semR); // self scheduling
    }
    signal(mutex);

    wait(semR);
    {Read}

    wait(mutex);
    runR--;
    totR--;
    if (runR == 0 && runW < totW) {
//   equivalente: && totW > 0  (perché qui si ha sempre runW == 0)
        runW = 1;
        signal(semW);
    }
    signal(mutex);
}
```

- writer:

```
while (true) {
    wait(mutex);
    totW++;
    if (runW == 0 && runR == 0) {
        runW++; // equivalente: runW = 1
        signal(semW); // self scheduling
    }
    signal(mutex);
```

```
        wait(semW);
        {Write}

        wait(mutex);
        runW--; // equivalente: runW = 0
        totW--;
        while (runR < totR) {
            runR++;
            signal(semR);
        }
        if (runR == 0 && runW < totW) {
//   equivalente: && totW > 0  (perché qui si ha sempre runW == 0)
            runW = 1;
            signal(semW);
        }
        signal(mutex);
    }
```

La logica della soluzione è costituita da due fasi:

1. inizialmente, il processo cerca di capire se può lavorare: se non può, si mette in attesa sul semaforo `semR/semW`;
2. dopo aver lavorato, il processo cerca di capire se si sono create condizioni in cui è necessario sbloccare altri processi, e in tal caso esegue delle signal su `semR/semW`.

Per un processo lettore, la fase 1 è composta dalle seguenti operazioni:

1. acquisisce la mutua esclusione sulle variabili condivise;
2. dichiara di voler lavorare, incrementando `totR`;
3. determina se può lavorare, cioè se `runW == 0`: in tal caso, incrementa `runR` ed esegue una signal su `semR`;
4. rilascia la mutua esclusione;
5. fa una wait su `semR`: continua a eseguire solo se prima ha fatto la signal, altrimenti si blocca.

Siccome è il processo stesso a decidere se bloccarsi o meno su `semR`, si parla di *self scheduling*.

Un processo scrittore fa delle operazioni analoghe, ma può lavorare solo se nessuno sta scrivendo (`runW == 0`) e nessuno sta leggendo (`runR == 0`). Se uno scrittore determinasse che può lavorare, e poi, dopo aver rilasciato la mutua esclusione sulle variabili, ma prima di eseguire `wait(semW)`, perdesse la CPU, un eventuale altro lettore/scrittore

troverebbe le variabili tali da non poter fare anch'esso una signal: è quindi garantito che uno scrittore non acceda ai dati concorrentemente ad altri scrittori e/o lettori.

Nella fase 2, cioè dopo la lettura, un processo lettore esegue (con mutua esclusione) una serie di operazioni:

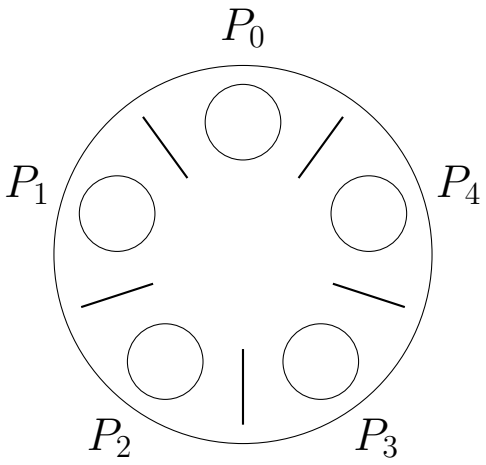
1. dichiara di aver finito di lavorare, decrementando `runR` e `totR`;
2. se non ci sono più lettori che stanno lavorando, ma c'è uno scrittore che vorrebbe lavorare, lo sveglia.

Non è mai necessario che un lettore svegli un altro lettore, perché, mentre avviene una lettura, è garantito che nessuno scrittore stia lavorando, quindi anche gli altri lettori non si possono bloccare.

Uno scrittore, invece, può dover svegliare un altro scrittore, oppure dei lettori. In particolare, quando sveglia i lettori, li sveglia tutti, eseguendo ripetutamente la signal su `semR`. Se ci sono sia lettori che scrittori in attesa, la definizione del problema non specifica quali svegliare per primi: in quest'implementazione, si dà la priorità ai lettori.

3 Filosofi a cena

5 filosofi sono seduti intorno a un tavolo, e alternano momenti in cui pensano a momenti in cui mangiano. Per mangiare, i filosofi hanno bisogno di due bacchette/forchette, ma sul tavolo ne sono disponibili solo 5: ciascun filosofo ne ha una condivisa con il filosofo alla sua destra, e una condivisa con il filosofo alla sua sinistra. Di conseguenza, mentre mangia, i filosofi alla sua destra e sinistra non possono mangiare.



Più precisamente, un filosofo attraversa ciclicamente tre stati:

1. pensa;
2. ha fame, quindi cerca di prendere le bacchette;
3. mangia.

Una soluzione valida deve fare sì che ogni filosofo affamato riesca prima o poi a mangiare, in modo che tutti i filosofi possano continuare all'infinito ad alternare momenti in cui pensano e momenti in cui mangiano.

Nelle istanze concrete di questo problema, i "filosofi" sono processi che competono con altri per delle risorse condivise, le quali possono essere usate solo in mutua esclusione, e ogni processo deve acquisire contemporaneamente più risorse per poter lavorare.

3.1 Esempi di soluzioni errate

Come primo tentativo di soluzione, si potrebbe stabilire che ciascun filosofo affamato:

1. prende la bacchetta alla sua sinistra appena diventa disponibile;
2. successivamente, aspetta che diventi disponibile la bacchetta destra, e infine prende anche quella.

Se, però, tutti i filosofi hanno fame contemporaneamente, si verifica un *deadlock*: ciascuno prende la bacchetta alla propria sinistra, ma così nessuno riesce a prendere quella a destra, e quindi si bloccano tutti.

Il deadlock si può evitare apportando una modifica alla soluzione precedente: se, dopo aver preso la bacchetta sinistra, un filosofo trova quella destra occupata, allora posa la sinistra. Questa strategia introduce invece la possibilità che si verifichi un *livelock*. Ad esempio, potrebbe succedere che

1. tutti i filosofi prendono la bacchetta sinistra
2. trovando quella a destra occupata, tutti i filosofi posano la bacchetta sinistra

e ciò si potrebbe ripetere all'infinito, impedendo comunque a tutti i filosofi di mangiare, pur senza che essi rimangano bloccati.

3.2 Soluzione con i semafori

Si usano le seguenti strutture dati:

- un array `state` che tiene traccia dello stato del sistema: per ogni filosofo P_i, `state[i]` può avere valore `THINKING`, `HUNGRY` o `EATING`;
- un semaforo `mutex`, per la mutua esclusione sull'array `state`;

- un semaforo `sem[i]` per ogni filosofo P_i, che serve per "dare l'ok" a P_i: se P_i non può mangiare (perché non ha le bacchette), allora si blocca su `sem[i]`.

```
#define N 5
#define LEFT ((i + N - 1) % N)
#define RIGHT ((i + 1) % N)

enum { THINKING, HUNGRY, EATING } state[N];
semaphore mutex = 1;
semaphore sem[N]; // inizializzati a 0
```

Ciascun filosofo P_i esegue la procedura `philosopher(i)`:

```
void philosopher(int i) {
    while (true) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

- L'implementazione di `think` e `eat` non è rilevante per la soluzione del problema.
- La procedura `take_forks`:

  ```
  void take_forks(int i) {
      wait(mutex);
      state[i] = HUNGRY;
      test(i);
      signal(mutex);
      wait(sem[i]);
  }
  ```

  1. in mutua esclusione, dichiara che P_i ha fame, e chiama la procedura `test` per prendere le bacchette se *entrambe* sono libere:

     ```
     void test(int i) {
         if (state[i] == HUNGRY
                 && state[LEFT] != EATING
                 && state[RIGHT] != EATING) {
             state[i] = EATING;
             signal(sem[i]);
         }
     }
     ```

2. se non sono libere entrambe le bacchette, si blocca su `sem[i]` (dato che non è stata eseguita la signal all'interno di `test`).

- La procedura `put_forks` viene eseguita interamente in mutua esclusione.

  ```
  void put_forks(int i) {
      wait(mutex);
      state[i] = THINKING;
      test(LEFT);
      test(RIGHT);
      signal(mutex);
  }
  ```

  Essa:

  1. lascia le bacchette, cioè segnala che P_i ha finito di mangiare ed è tornato a pensare;
  2. prova a far mangiare il filosofo a sinistra, se è affamato ed ha a disposizione entrambe le bacchette;
  3. prova a far mangiare il filosofo a destra, se è affamato ed ha a disposizione entrambe le bacchette.

Questa soluzione evita sia i deadlock che i livelock (perché le due bacchette vengono prese in modo indivisibile), quindi almeno alcuni filosofi riusciranno sempre a mangiare. Non è garantito, però, che ci riescano tutti: potrebbe accadere che si alternano sempre gli stessi filosofi, mentre altri non mangiano mai, perché non hanno mai a disposizione due bacchette (in pratica, non è soddisfatta la bounded wait). Ciò si può risolvere, ma così facendo il programma si complica.