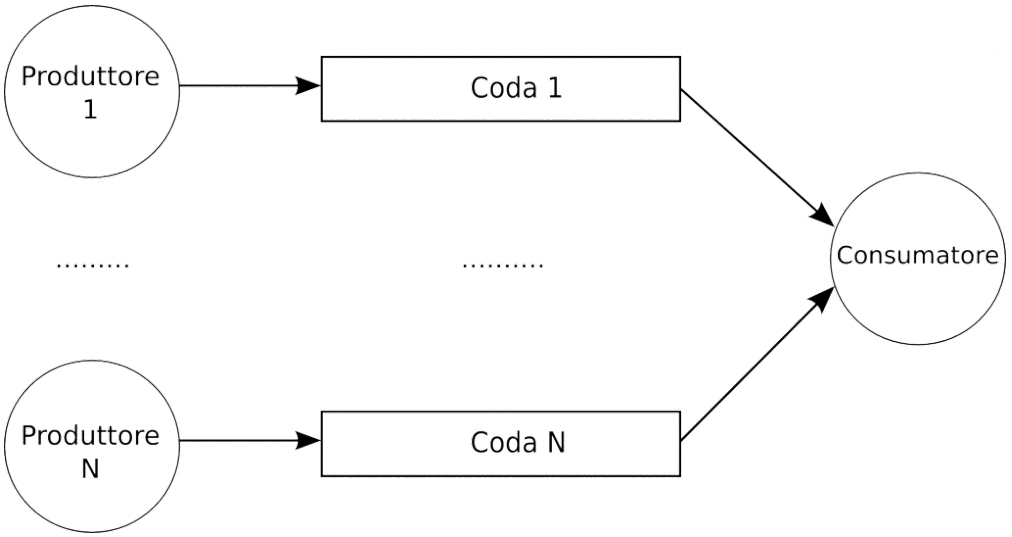
Programmazione con i semafori

La lezione scorsa abbiamo visto come si può realizzare la sincronizzazione produttore-consumatore tramite i semafori. Vediamo come risolvere altri problemi di sincronizzazione: *sincronizzazione molti a uno*, lettori-scrittori e *filosofi a cena*.

Sincronizzazione molti a uno

Consideriamo una variante del produttore-consumatore in cui ci sono `N` produttori che generano `N` dati che verranno usati tutti assieme dal consumatore. Ad esempio i produttori inviano un dato numerico e il consumatore computa una funzione `f` su tali dati.



La differenza rispetto al produttore-consumatore classico con tanti produttori è che in quel caso, anche se ci sono più produttori, il consumatore consuma un dato alla volta, indipendentemente da chi lo ha prodotto. Qui invece il consumatore deve attendere che tutti gli `N` produttori abbiano scritto il proprio dato per poi leggere tutti gli `N` dati in una volta. Utilizziamo quindi una coda diversa per ogni produttore:

```
queue dato_produttore[N];

Produttore(i) {
  while(1) {
    < produce d >

    dato_produttore[i].add(d);
  }
}

Consumatore {
  while(1) {
    for (i=0;i<N;i++)
      d[i] = dato_produttore[i].remove();

    < consuma d[0], ..., d[N-1] >
  }
}
```

Come possiamo sincronizzare i produttori e il consumatore usando i semafori? A differenza del problema classico il consumatore deve attendere `N` risorse diverse, ovvero gli `N` dati, uno per ogni produttore. Ci servono quindi `N` semafori `piene` e `N` semafori `vuote`, uno per ogni produttore. Proteggiamo poi le operazioni sulle code con un `mutex` per ogni coda e limitiamo il numero di elementi in coda a `MAX`:

```
queue dato_produttore[N];
semaphore vuote[N]={MAX, ..., MAX};
semaphore piene[N]={0, ..., 0};
semaphore mutex[N]={1, ..., 1};

Produttore(i) {
  while(1) {
    < produce d >

    P(vuote[i]);
    P(mutex[i]);
    dato_produttore[i].add(d);
    V(mutex[i]);
    V(piene[i]);
  }
}
```

```
Consumatore {
  while(1) {
    for (i=0;i<N;i++) {
      P(piene[i]);
      P(mutex[i]);
      d[i] = dato_produttore[i].remove();
      V(mutex[i]);
      V(vuote[i]);
    }

    < consuma d[0], ..., d[N-1] >
  }
}
```

In pratica quello che otteniamo sono `N` distinte sincronizzazioni produttore-consumatore verso un unico thread consumatore. Vediamo in particolare che il consumatore esegue un ciclo `for` dentro il quale si sincronizza con ogni singolo produttore.

Provare per esercizio a realizzare una sincronizzazione uno-a-molti.

Lettori e scrittori

Il problema dei lettori e scrittori è un classico problema di condivisione dell'informazione che richiede sincronizzazione. Abbiamo alcuni thread che accedono in sola lettura, i *lettori*, e altri che possono scrivere, quindi modificare che chiameremo *scrittori*.

Come sappiamo, la sola lettura di un dato non può creare di per sé incoerenze. Quindi non ci sono problemi se più lettori accedono ai dati contemporaneamente. Quando invece vogliamo modificare l'informazione, per evitare incoerenze, chiediamo che ci sia mutua esclusione: se uno scrittore sta modificando i dati nessun altro thread (lettore o scrittore) deve poter accedere ai dati.

L'accesso viene quindi regolato seguendo il principio: **un solo scrittore oppure tanti lettori**.

Come possiamo realizzare questa sincronizzazione utilizzando i semafori? L'idea è di

avere un semaforo di mutua esclusione che denominiamo `scrittura` e inizializziamo a `1`. Il codice dello scrittore è molto semplice in quando deve acquisire la mutua esclusione come si fa in una normale sezione critica:

```
semaphore scrittura=1;

Scrittore {
  while(1) {
    ...
    P(scrittura);

    < modifica i dati >

    V(scrittura);
    ...
  }
}
```

La parte complicata è far sì che i lettori possano accedere in tanti ma sempre in alternativa allo scrittore: se c'è uno scrittore nessun lettore deve accedere. Per realizzare questa sincronizzazione utilizziamo una variabile `num_lettori`, inizializzata a `0`, che tiene traccia del numero dei lettori attivi. L'idea è che il primo lettore acquisirà la mutua esclusione mentre i lettori successivi entreranno direttamente in sezione critica. In uscita, sarà l'ultimo lettore a rilasciare la mutua esclusione, permettendo a uno scrittore di accedere. Poiché `num_lettori` è una variabile condivisa tra tutti i lettori, dobbiamo proteggerne l'accesso tramite sezione critica. Utilizziamo allo scopo un semaforo `mutex` inizializzato a `1`.

```
int num_lettori=0;
semaphore mutex=1;

Lettore {
  while(1) {
    ...
    P(mutex);              // protegge num_lettori e accoda i lettori
    num_lettori++;
    if (num_lettori == 1)  // primo lettore
      P(scrittura);        // acquisisce la mutua esclusione in scrittura
```

```
      V(mutex);

      < legge i dati >

      P(mutex);              // protegge num_lettori
      num_lettori--;
      if (num_lettori == 0)  // ultimo lettore
        V(scrittura);        // rilascia la mutua esclusione in scrittura
      V(mutex);
      ...
    }
  }
```

Vediamo in dettaglio il funzionamento:

- Supponiamo che ci sia uno scrittore attivo. Il semaforo `scrittura` sarà rosso. Se un lettore cerca di accedere, esso acquisirà il `mutex` , incrementerà `num_lettori` e poiché è il primo ad entrare effettuerà `P(scrittura)` accodandosi sul semaforo dello scrittore. Cosa accade se arrivano altri lettori? Poiché abbiamo annidato due semafori di mutua esclusione, i lettori si accoderanno su `mutex` e attenderanno che il primo lettore possa accedere. Questo comportamento è corretto perché tali lettori entreranno solo quando il primo avrà accesso;
- Supponiamo che ci sia un lettore attivo con `num_lettori==1` . Il semaforo `scrittura` sarà rosso. Se arrivano altri lettori essi incrementeranno `num_lettori` e non eseguiranno la `P(scrittura)` accedendo direttamente in sezione critica. Se invece tenta di accedere uno scrittore esso si accoderà sul semaforo `scrittura` . Solo quando l'ultimo lettore lascia la sezione critica viene eseguita la `V(scrittura)` che libera un eventuale scrittore in attesa.

Starvation: Notiamo che questa soluzione può dare starvation agli scrittori in quanti i lettori continueranno ad accedere alla sezione critica nel momento in cui almeno un lettore è attivo. Uno scrittore potrebbe quindi attendere un tempo indefinito. Esistono soluzioni alternative che danno priorità agli scrittori o equità di accesso (assenza di starvation) ma la complessità aumenta. Quella presentata è la soluzione più semplice.

Filosofi a cena

Un altro problema classico di sincronizzazione è quello dei *filosofi a cena*. Ci sono 5 filosofi cinesi seduti a una tavola rotonda. Ognuno ha davanti a sé un piatto e tra ogni piatto c'è una bacchetta. Per mangiare un filosofo usa due bacchette, quella alla sua sinistra e quella alla sua destra che però sono condivise con i filosofi vicini. Di conseguenza due filosofi vicini non possono mangiare contemporaneamente.

I filosofi oltre a mangiare, naturalmente, pensano ma questa attività avviene in modo indipendente: l'unico momento in cui si devono sincronizzare è quindi quando mangiano. Schematizziamo il filosofo `i`-esimo nel modo seguente:

```
Filosofo(i) {
  while(1) {
    < pensa >

    < raccoglie le bacchette sx e dx >
    < mangia >
    < deposita le bacchette sx e dx >
  }
}
```

Come possiamo risolvere questo problema con i semafori? Intuitivamente ogni bacchetta è una risorsa sulla quale 2 filosofi possono competere. Sappiamo che i semafori possono essere usati per regolare l'acquisizione e il rilascio di generiche risorse, quindi consideriamo un array di 5 semafori `semaphore bacchette[5]`, uno per ogni bacchetta. Poiché ogni bacchetta è un'istanza di una risorsa, i semafori saranno inizializzati tutti a `1`. La raccolta della bacchetta sarà una `P`, mentre il rilascio una `V`. Otteniamo il seguente codice:

```
semaphore bacchette[5]={1,1,1,1,1};

Filosofo(i) {
  while(1) {
    < pensa >

    P(bacchette[i]);
    P(bacchette[(i+1)%5]);
    < mangia >
```

```
    V(bacchette[i]);
    V(bacchette[(i+1)%5]);
  }
}
```

Nel codice sopra abbiamo assunto che la bacchetta di sinistra abbia lo stesso indice `i` del filosofo mentre quella di destra abbia indice `(i+1)%5`. L'operazione di `%5` è utile per modellare il fatto che la tavola è rotonda: il filosofo con indice `4`, infatti, avrà la bacchetta `4` alla sua sinistra e la bacchetta `(4+1)%5 = 0` alla sua destra.

È facile convincersi che la soluzione proposta impedisce a due filosofi vicini di mangiare assieme, come richiesto. Se, ad esempio, il filosofo 3 sta mangiando i semafori `bacchette[3]` e `bacchette[4]` saranno rossi. Se il filosofo 4 cerca di mangiare, esso troverà `bacchette[4]` rosso e si metterà in attesa sulla coda di tale semaforo.

Stallo: Cosa potrebbe accadere se i filosofi decidono di mangiare tutti assieme? Supponiamo che tutti raccolgano la bacchetta alla loro sinistra. I semafori diventano tutti rossi e tutti i filosofi rimangono in attesa della bacchetta alla loro destra. Si forma una situazione di attesa circolare irrisolvibile detta anche *stallo* o *deadlock*. Vedremo più avanti come prevenire o risolvere le situazioni di stallo con tecniche generali. Vediamo qui, nel caso specifico dei filosofi a cena, come possiamo modificare il codice in modo che non si formi mai una situazione di attesa circolare.

Soluzione 1: quattro filosofi a tavola

Una prima possibilità per prevenire lo stallo è limitare a 4 in numero di filosofi che possono mangiare. Se ci sono al più quattro filosofi non sarà mai possibile che tutte le bacchette di sinistra siano allocate. Questa soluzione si può realizzare aggiungendo un semaforo `posti` inizializzato a `4`:

```
semaphore bacchette[5]={1,1,1,1,1};
semaphore posti=4;

Filosofo(i) {
  while(1) {
```

```
    < pensa >

    P(posti);
    P(bacchette[i]);
    P(bacchette[(i+1)%5]);
    < mangia >
    V(bacchette[i]);
    V(bacchette[(i+1)%5]);
    V(posti);
  }
}
```

Se tutti e cinque i filosofi cercano di prendere la bacchetta di sinistra solo i primi 4 ci riusciranno mentre il quinto attenderà sulla coda del semaforo `posti` lasciano la sua bacchetta sulla tavola.

Soluzione 2: raccolta atomica

Una seconda possibilità per evitare lo stallo è far sì che i filosofi raccolgano le bacchette in modo atomico. Attenzione che mutua esclusione non è sinonimo di atomicità. Ad esempio potremmo pensare di realizzare questa soluzione come segue:

```
semaphore bacchette[5]={1,1,1,1,1};
semaphore mutex=1;

Filosofo(i) {
  while(1) {
    < pensa >

    P(mutex);
    P(bacchette[i]);
    P(bacchette[(i+1)%5]);
    V(mutex);
    < mangia >
    V(bacchette[i]);
    V(bacchette[(i+1)%5]);
  }
}
```

In effetti si evita lo stallo perché i filosofi si accodano innanzitutto sul `mutex`. Consideriamo però il caso in cui un filosofo prende la bacchetta di sinistra e attende per quella di destra, che è già in uso. Il filosofo si blocca in sezione critica e quindi altri filosofi che potrebbero mangiare sono costretti ad attendere sul `mutex`. In sostanza questa soluzione sincronizza troppo e riduce inutilmente il parallelismo.

Purtroppo non è semplice realizzare atomicità con i semafori perché sarebbe necessario rilasciare la mutua esclusione ogniqualvolta si deve attendere. Vedremo come questa soluzione sia immediata da realizzare tramite i Monitor.

Soluzione 3: Il filosofo mancino

Possiamo, infine, forzare uno dei filosofi a raccogliere prima la bacchetta di destra e poi quella di sinistra. In questo modo si rompe la simmetria e di conseguenza si previene la situazione di attesa circolare. Non è più possibile, infatti, che tutti e 5 i filosofi raccolgano la bacchetta di sinistra mettendosi in attesa su quella di destra, perché almeno uno cercherà di raccogliere prima la bacchetta di destra. Per implementare questa soluzione selezioniamo come filosofo mancino il quinto (indice 4):

```
Filosofo(i) {
  while(1) {
    < pensa >

    if (i == 4) {
      P(bacchette[0]);       // destra
      P(bacchette[4]);       // sinistra
    } else {
      P(bacchette[i]);       // sinistra
      P(bacchette[(i+1)%5]); // destra
    }

    < mangia >

    // rilascia le bacchette, non importa l'ordine in quanto la V non è bloccante
    V(bacchette[i]);
    V(bacchette[(i+1)%5]);
  }
}
```