1 Lezione del 28-10-25

1.1 Lettori e scrittori

Sempre sull'argomento della sincronizzazione fra processi, vediamo l'esempio di più **scrittori** che voglioni scrivere su una risorsa che è letta da più **lettori**.

Iniziamo a vedere quali politiche vogliamo assicurare:

1. Chiaramente, fra gli scrittori c'è una stretta politica di mutua esclusione: non si può scrivere in 2 o più contemporaneamente.
2. Anche fra lettori e scrittori deve esserci mutua esclusione (non possiamo leggere ciò che è inconsistente perché ci si sta scrivendo);
3. In tutti gli altri contesti, vorremmo permettere di avere più lettori contemporanei.

Per risolvere il problema della mutua esclusione fra scrittori (1) prevediamo un semaforo di mutex `sem wrt = 1`, che viene prelevato in fase di scrittura:

```
proc writer {
  wait(wrt); // mutex

  // scrivi

  signal(wrt); // mutex
}
```

Il semaforo `wrt`, inizialmente pensato per la mutua esclusione fra scrittori (1), può essere usato anche dai lettori per risolvere il problema (2). Il problema in questo caso sarà che non assicureremo la politica (3) di letture contemporanee: facendo la `wait()` su `wrt` sblocchiamo infatti un lettore per volta.

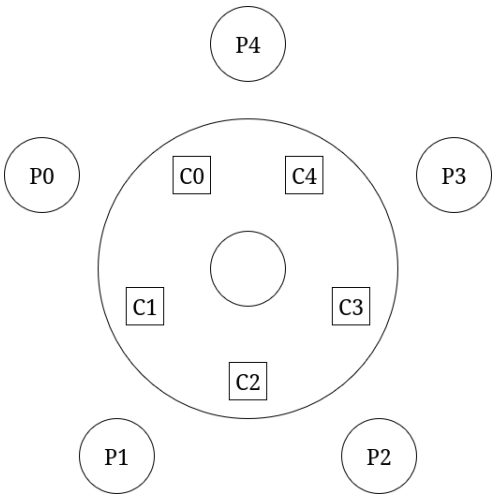
Dotiamoci quindi di un contatore `int readCount = 0`, che tiene conto dei processi lettori che attualemente stanno leggendo la risorsa. Proteggiamo quindi il contatore con un nuovo semaforo di mutex `sem mutex = 1`.

```
proc reader {
  wait(mutex);
  {
    readCount++;
    if(readCount == 1) wait(wrt); // solo il primo aspetta
  }
  signal(mutex);

  // lettura

  wait(mutex);
  {
    readCount--;
    if(readCount == 0) signal(wrt); // solo l'ultimo rilascia
  }
  signal(mutex);
}
```

In questo caso saranno solo rispettivamente il primo e l'ultimo lettore a prendersi la briga di fare la `wait()` e quindi la successiva `signal()` sul semaforo condiviso con gli scrittori.

1.2 Problema dei 5 filosofi

Veniamo quindi ad un esempio celebre di programmazione concorrente.



Ipotizziamo una situazione dove 5 filosofi $p_0, p_1, ...$ sono seduti ad una tavola circolare, al centro della quale è posta una scodella di riso. Sulla tavola, una alla destra di ogni filosofo, ci sono esattamente 5 bacchette $c_0, c_1, ...$ Ogni filosofo per mangiare ha bisogno di due bacchette. Il problemaè: come possono i filosofi coordinarsi per mangiare tutti, e quindi ottenere tutti ciclicamente le 2 bacchette?

Contemporaneamente, possono mangiare al massimo 2 filosofi: ad esempio, se sta mangiando p_0, possono mangiare contemporaneamente solo p_2 o p_3.

Vediamo il comportamento del singolo filosofo. Questo potrà trovarsi in uno di 3 stati:

```
enum State {
  THINKING, // non ha fame
  HUNGRY,   // sta cercando di ottenere 2 bacchette
  EATING    // sta mangiando
}
```

1. Un primo approccio può essere quello di dotarsi di un semaforo di mutex per bacchetta, cioè avere `sem chopstick[5] = 1`. In questo caso lo pseudocodice del filosofo sarà:

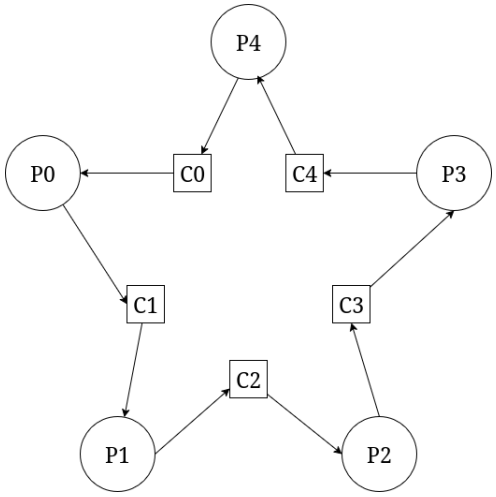
   ```
   proc philosopher {
     // pensa

     // aspetta bacchette
     wait(chopstick[i]); // sx
     wait(chopstick[(i + 1) % 5]); // dx

     // mangia

     // rilascia bacchette
     signal(chopstick[(i + 1) % 5]); // dx
     signal(chopstick[i]); // sx
   }
   ```

   Questo approccio può però portare a *deadlock* nel caso in cui tutti i processi riescano ad effettuare la prima `wait()`: in questo caso si troveranno con la bacchetta a sinistra

presa da loro, e quella a destra presa dal vicino, col risultato che nessuno può procedere e i filosofi muoiono di fame. Peccato!

Possiamo modellizzare questa situazione con un grafo, dove nodi circolari rappresentano i **filosofi** (processi), e nodi quadrati rappresentano le **bacchette** (*risorse*). Useremo le frecce da processi e risorse per rappresentare l'**attesa** di una risorsa (chiamata di `wait()`), e le frecce da risorse a processi per rappresentare il **possesso** di una risorsa (`wait()` terminata).

Vediamo che la situazione di deadlock appena descritto in questo caso è rappresentata da un **ciclo** nel grafo processi-risorse:



Per risolvere questo problema, vorremo prima approfondire il concetto di deadlock, e quindi implementare appropiate tecniche di deadlock **detection** (*rilevamento* di deadlock) e deadlock **avoidance** (*risoluzione* o *prevenzione* di deadlock).

1.3 Monitor

I **monitor** rappresentano un'astrazione di alto livello che permettono la sincronizzazione di processi. Sostanzialmente, sono strutture dati contenenti funzioni (*operazioni*) che vengono eseguite in mutua esclusione (cioè in maniera *atomica*) all'interno di un certo contesto (dove si condivide codice di *inizializzazione* e *dati*).

All'interno di un monitor prevediamo variabili di **condizione**, su cui sono permesse operazioni di `wait()` e `signal()`. Chiaramente queste variabili di condizioni saranno visibili solo all'*interno* del monitor, cioè del codice delle operazioni definite dal monitor.

- La `wait()` mette in attesa un processo finché un altro non esegue una `signal()`;
- La `signal()` sveglia i processi che erano in wait sulla variabile di condizione. La particolarità della `signal()` è che non ha effetti se nessuno è in stato di `wait()`.

1.3.1 Gestione delle variabili di condizione

Assumiamo che un processo P invochi `x.signal()` sulla variabile di condizione `x`, mentre un altro processo Q si trova nello stato `x.wait()`. Cosa dovrebbe succedere a questo punto?

Ci sono due opzioni:

- **Signal and wait**: P aspetta che Q esca dal monitor o si metta in attesa di un'altra condizione: è il caso *preemptive;*
- **Signal and continue**: Q aspetta che P esca dal monitor o si metta in attesa di un'altra condizione.

Noi adotteremo la soluzione *signal and wait*.

1.3.2 Monitor per problema dei 5 filosofi

Vediamo quindi come un monitor può essere usato per risolvere il problema dei 5 filosofi.

```
monitor Philosophers {
  enum State {
    THINKING, // non ha fame
    HUNGRY,   // sta cercando di ottenere 2 bacchette
    EATING    // sta mangiando
  }
  State state[5];

  // variabili di condizione
  condition self[5];

  void pickup(int i) {
    state[i] = HUNGRY; // hai fame
    test(i); // puoi mangiare?
    if(state[i] != EATING) self[i].wait(); // se puoi mangia
  }

  void putdown(int i) {
    state[i] = THINKING; // non stai piu' mangiando
    test((i - 1) % 5); // sx puo' mangiare?
    test((i + 1) % 5); // dx puo' mangiare?
  }

  void test(int i) {
    if(
      (state[(i - 1) % 5] != EATING) && // sx non sta mangiando?
      (state[i] != HUNGRY) &&           // hai fame?
      (state[(i + 1) % 5] != EATING)    // dx non sta mangiando?
    ) {
      state[i] = EATING; // mangia
      self[i].signal();
    }
  }

  initialization_code() {
    for(int i = 0; i < 5; i++) {
      state[i] = THINKING;
    }
  }
}
```

A questo punto il processo filosofo sarà molto semplice:

```
process philospher {
  Philosophers.pickup();

  // mangia

```

```
6   Philosophers.putdown();
7 }
```

Abbiamo che questo approccio è privo di deadlock. Infatti, i processo filosofi non provano ad ottenere le bacchette finché non hanno la sicurezza di poter prendere *entrambe* le bacchette.