Corso di Informatica

Modulo T5

C2–Produttore-consumatore

Prerequisiti

- Concetto di semaforo
- Risorsa permanente o consumabile
- Sincronizzazione
- Significato e uso di **wait**() e **signal**()

Introduzione

Ci interessiamo ora di un altro modo di comunicare fra *thread*, nel caso in cui ci sia cooperazione che, nel caso più generale, corrisponde allo scambio di messaggi.

In tal caso la sincronizzazione è ancora indispensabile per far in modo che i messaggi non vadano persi.

Cooperazione

Partiamo anche qua con un esempio.

Supponiamo di avere due processi P1 e P2 che abbiano bisogno dei dati prodotti da un terzo processo P3.
Il sistema è multiprogrammato, quindi non ci è dato di conoscere in che ordine avviene l'elaborazione dei processi.

Fra P1, P2 e P3 c'è una forma di **cooperazione**.

Problema: garantire che P1 e P2 restino inattivi fino a quando P3, avanzando, non fornisca loro i dati.

Modello produttore-consumatore

Il problema della **cooperazione** tra processi prevede di gestire un meccanismo di **produzione-consumo** e può essere posto come segue.

Dati due processi **P** e **C**, occorre garantire che:

- il processo P, detto **processo produttore**, invii continuamente **messaggi** al processo C;
- il processo C, detto **processo consumatore**, elabori i **messaggi** ricevuti, nello stesso ordine con cui li ha ricevuti.

Per messaggio intendiamo una risorsa consumabile

Esempi produttore-consumatore

- Un processo **P** scrive dati in un file e **C** li deve leggere.
- **P** acquisisce un evento da un input dalla tastiera e lo inserisce in una coda di eventi e **C** legge gli eventi dalla stessa coda.
- Un processo che acquisisce i dati dalla tastiera
- Più processi debbano condividere informazioni
- Un processo che svolga un calcolo parallelo
- L'esecuzione parallela di moduli della stessa applicazione per sfruttare la modularità del software;
- La gestione tramite lo *spooling* delle richieste di stampa

Sincronizzazione e buffer

Risulta ovvio che, come nel caso della mutua esclusione, anche in questi casi, affinchè i due *thread* possano comunicare, è necessario operare una **sincronizzazione**

Inoltre, per poter funzionare, questo meccanismo richiede la presenza di un **buffer** in cui il processo produttore **P** deposita il dato; il processo consumatore **C** ha il compito di prelevare il dato e "consumarlo".

Considerando l'esempio precedente della tastiera, la presenza del buffer è resa evidente dal classico "*beep*" emesso qualora il buffer di tastiera fosse pieno.

Sincronizzazione e buffer

L'accesso al buffer per prelevare o depositare un messaggio è una tecnica di comunicazione tra processi, tipicamente implementata con dei **semafori**.

Supponiamo che:

- il buffer possa contenere una sola informazione;
- Il buffer sia una risorsa seriale e non prerilasciabile;
- i *thread* in cooperazione avanzino con velocità simile (se così non fosse, sarebbe necessario un buffer è necessario in grado di memorizzare un numero di messaggi maggiore di uno).

Si presentano allora i problemi seguenti:

Problemi tipici e…

Come trattare la situazionel “buffer pieno”?
Come trattare la situazione “buffer vuoto”?

E poi ...

Come informare il produttore che il buffer è vuoto e che pertanto può depositarvi un’informazione?
Come informare il consumatore che nel buffer è presente un nuovo messaggio?

…soluzioni intuitive

Detta in modo molto semplificato, la soluzione a questi problemi potrebbe espressa nel seguente modo:

Quando il buffer è pieno, si deve addormentare il produttore.
Quando il buffer è vuoto, si deve addormentare il consumatore.
Quando il buffer è vuoto, occorre risvegliare il produttore.
Quando il buffer è pieno, occorre risvegliare il consumatore.

Una soluzione errata potrebbe dar luogo ad uno **stallo** (*deadlock*), in cui entrambi i processi aspettano indefinitamente di essere risvegliati.

Situazioni da evitare

In altre parole, occorre gestire la situazione in modo che:

- il produttore, prima di inserire il dato nel buffer, deve assicurarsi che questo sia vuoto, altrimenti c'è il rischio che vada a sovrascrivere un dato precedentemente inserito nel buffer e non ancora utilizzato.
- il consumatore deve essere sicuro che il produttore abbia inserito un nuovo dato nel buffer cosi da prelevarlo. Tale controllo serve ad evitare il rischio che lo stesso dato venga letto più volte.
- il produttore e il consumatore non possano accedere contemporaneamente al buffer.

Soluzione pratica

Si può risolvere il problema usando tre semafori:

- *Mutex:* inizializzato a 1, garantisce la mutua esclusione nell'accesso al buffer
- *Prelevato*: inizializzato a 0, memorizza il numero di record pronti nel buffer, cioè scritti ma non ancora letti, e blocca chi tentasse di leggere da un buffer vuoto
- *Depositato*: inizializzato a N, memorizza il numero di posti liberi nel buffer, cioè di spazi in cui si possono scrivere record senza sovrascrivere record non ancora letti, e blocca chi tentasse di scrivere in un buffer pieno.

Soluzione pratica

I due processi *Produttore*() e *Consumatore*() avranno la seguente struttura (*Depositato* inizializzato con 0 e *Prelevato* inizializzato con 1)

```
void Consumatore()
Stringa dato;
Inizio
  Ripeti
    wait(Depositato):
    wait(Mutex);
    dato = buffer;
    signal(Mutex);
    signal(Prelevato);
    Consuma(dato);
  Fintantochè (ci sono dati);
Fine;
```

```
void Produttore()
Stringa dato;
Inizio
  Ripeti
    Produci(dato);
    wait(Prelevato):
    wait(Mutex);
    buffer = dato;
    signal(Mutex);
    signal(Depositato);
Fintantochè (ci sono dati);
Fine;
```


Argomenti

- Cooperazione
- Modello produttore-consumatore
- Esempi produttore-consumatore
- Sincronizzazione e buffer
- Problemi tipici e…
- … soluzioni intuitive
- Situazioni da evitare
- Soluzione pratica

Altre fonti di informazione

- P.Camagni, R.Nicolassy – Java: interfacce grafiche e programmazione concorrente, ed. Hoepli Informatica
- **http://java**.sun.com/docs/books/tutorial/essential/threads/index.html
- Google: stallo in Java c'era parecchia roba
- Linguaggi\Java\Altro materiale\Threads.htm