di Andrea de Prisco

Comunicazioni inter process

In questa puntata di Multitasking parleremo dei meccanismi di comunicazione tra processi. In particolare parleremo della cosiddetta comunicazione ad ambiente globale in cui strutture dati (o più in generale risorse) condivise tra più utilizzatori vengono utilizzate da un utente per volta onde evitare errori da collisione. Vere e proprie catastrofi che lascerebbero il sistema in uno stato inconsistente e quindi soggetto a fornire risultati e comportamenti errati fino al completo crash del sistema stesso.

Passaggio a livello incustodito

Se volete pensate pure ad un passaggio a livello senza né sbarre né segnalazioni luminose in caso di transito del treno. E magari pure automobilisti assolutamente incuranti del pericolo, che oltrepassano i binari senza nemmeno rallentare. Altrettanto supponiamo che facciano i macchinisti dei vari treni. Quanto pensate possa durare la tranquillità in quel posto? No, no, rimettiamo le sbarre e cominciamo a ragionare.

La sezione di strada attraversata dai binari (o se volete la porzione di binari attraversata dalla strada) rappresenta in pratica un esempio «reale» di risorsa condivisa. Due diversi processi (le auto e i treni che passano) utilizzano gli stessi metri quadri di spazio per passare. Dato che, come è noto, non è possibile far passare nello stesso istante sia un treno che una macchina è necessario in qualche modo arbitrarne l'accesso in modo tale che non si verifica mai l'inconveniente del fatidico «botto».

Giocoforza le ferrovie di tutto il mondo hanno in pratica stabilito di avere assoluta priorità sui passaggi a livello quindi la condivisione di risorsa è semplicemente risolta impedendo al flusso di auto, con due belle sbarre bianche e rosse, di attraversare i binari quando c'è un treno in transito.

Converrete con noi che nulla vieterebbe di fare l'esatto contrario: mettere delle sbarre al treno ogni volta che passa un'auto (così s'impara!). Il problema sarebbe altrettanto risolto: anche così non avremmo di certo collisioni sull'incrocio, ma è chiaro che in questo modo i treni di tutt'Italia sarebbero bloccati sui vari passaggi a livello in attesa della notte fonda...

Quindi non basta risolvere la collisione «punto e basta», ma è necessario trovare sempre la soluzione meno dolorosa per tutti gli utenti del sistema.

Sempre in merito ai trasporti va comunque segnalato (per dovere di cronaca... informatica) che, per quanto rudimentale, il passaggio a livello, nonostante la sua intrinseca parzialità, è comunque ben più intelligente (e sicuro) degli incroci stradali normalmente arbitrati dai semafori: e i risultati sono vi-

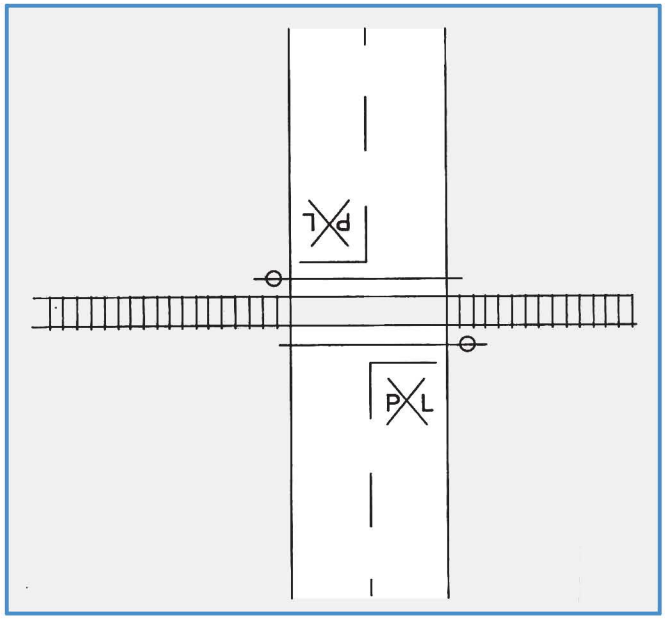

Figura 1 - Un passaggio a livello rappresenta un esempio reale di struttura condivisa da due processi: le auto e i treni in transito.

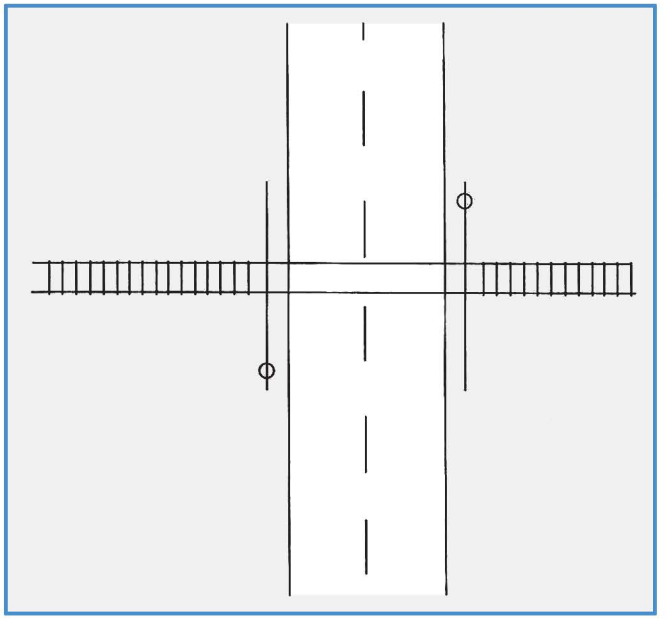

Figura 2 - Ponendo le sbarre sulla ferrovia avremmo comunque un arbitraggio della struttura condivisa, ma con risultati finali per nulla soddisfacenti.

sibili a tutti nelle grandi come nelle piccole città. Comunque, in questa puntata parleremo molto di semafori. Non di quelli stradali, ovviamente, ma solo di semafori informatici dal comportamento ben più produttivo (e anche qui i risultati si vedono...). Già: come evitare che due o più processi si scontrino nell'accesso ad una risorsa condivisa? Semplice, basta installare un semaforo...

L'esempio tipico

Prima di mostrarvi il funzionamento dei semafori «informatici» è bene dare uno sguardo ai possibili incidenti che potrebbero accadere in un sistema in cui la condivisione delle risorse non è gestita affatto.

L'esempio tipico è quello della modifica, ad esempio l'incremento, di una cella di memoria condivisa tra due o più processi. Mettiamoci nell'ipotesi in cui la nostra macchina non abbia una singola istruzione di incremento memoria ma l'operazione richiede più istruzioni essendo necessario eseguire l'incremento in un registro interno del processore. Immaginiamo che la cella di memoria interessata stia all'indirizzo 1000. Una possibile soluzione potrebbe essere:

```
MOVE  $1000,R1
ADD   #1, R1
MOVE  R1,$1000
```

La prima istruzione copia il contenuto della cella 1000 nel registro interno R1, la seconda somma 1 al contenuto di tale registro, la terza ricopia il valore così ottenuto nella cella di partenza. Ma come dicevamo prima, abbiamo fatto l'ipotesi che la cella 1000 potesse essere utilizzata anche da altri processi: cosa succede se un altro processo la utilizza contemporaneamente al primo?

Dipende da quello che ci fa e dall'attimo esatto in cui ciò avviene. In un sistema multitask, infatti, se non si prendono le opportune precauzioni, il quanto di tempo che assegnato ad ogni processo può scadere anche nel momento (questione di sfortuna) meno propizio. È vero altrettanto che comunque dopo altri «n» quanti di tempo il controllo torna al processo di partenza (che può così continuare quello che stava facendo) ma quando ci sono di mezzo strutture dati condivise (nell'esempio di sopra la cella 1000) e modificabili i risultati ottenuti possono non essere più tanto giusti. Immaginiamo quindi che un secondo processo decida anch'esso di incrementare la cella 1000. Per maggiore chiarezza supponiamo che esegua la medesima sequenza di prima utilizzando il registro interno R2. Naturalmente potrebbe anche usare R1 in quanto i registri sono proprio la prima cosa che viene salvata (e ripristinata) quando si esegue un cambio di contesto a seguito di una sospensione per I/O o per «quanto di tempo scaduto».

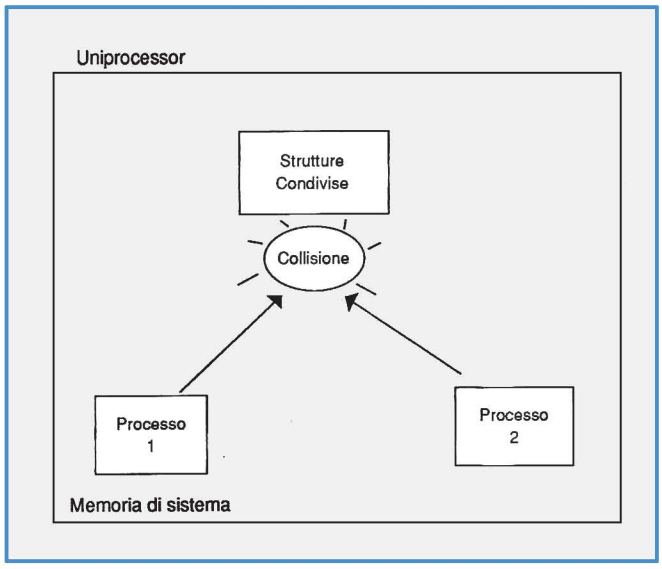

Figura 3 - In un sistema uniprocessor la collisione può essere evitata disabilitando il multitasking durante le sezioni critiche.

```
MOVE  $1000,R2
ADD   #1,R2
MOVE  R2,$1000
```

È la sequenza di istruzioni del secondo processo che incrementa la cella condivisa. Se i due processi eseguono entrambi la porzione di codice sopra mostrata, se tutto va bene alla fine la cella 1000 sarà stata incrementata di due.

Del resto è chiaro: agli occhi del processore che è l'unico in grado di eseguire istruzioni è come se avesse eseguito la sequenza di istruzioni:

```
MOVE  $1000,R1      ;
ADD   #1, R1        ;  PROCESSO 1
MOVE  R1,$1000      ;
.
.
.
MOVE  $1000,R2      ;
ADD   #1, R2        ;  PROCESSO 2
MOVE  R2,$1000      ;
.
.
```

dove il primo gruppo appartiene al primo processo, il secondo gruppo al secondo processo. L'ipotesi vista riguarda il caso in cui il processo 1 sia sospeso solo dopo aver terminato la modifica alla cella 1000 quindi prima del momento in cui anche il processo 2 metta mano alla stessa cella.

Ma nell'ipotesi, tutt'altro che remota, in cui il processo 1 venga interrotto qualche istruzione prima, il processore potrebbe eseguire «quelle» sei istruzioni in questa sequenza:

```
MOVE  $1000,R1      ;  PROCESSO 1
.
.
.
MOVE  $1000,R2      ;
ADD   #1, R2        ;  PROCESSO 2
MOVE  R2,$1000      ;
.
.
ADD   #1, R1        ;  PROCESSO 1
MOVE  R1,$1000      ;
.
.
```

I più attenti avranno già «sgamato» che in questo secondo caso la cella 1000, al termine delle medesime sei istruzioni, non è stata incrementata di due ma, erroneamente, di uno. Infatti in

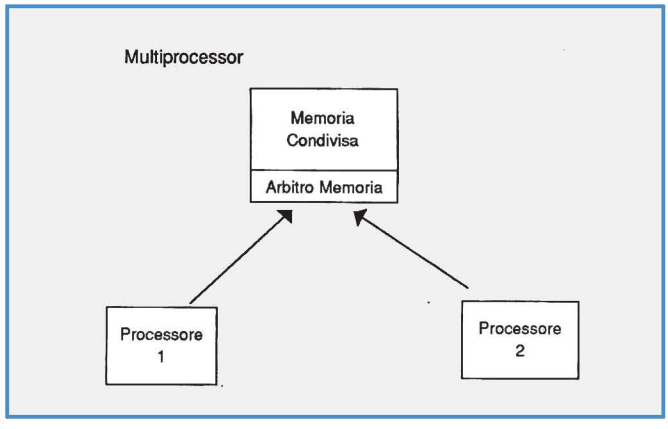

Figura 4 - In un sistema multiprocessor è necessario bloccare anche l'arbitro di memoria per garantire accessi esclusivi alle strutture dati condivise.

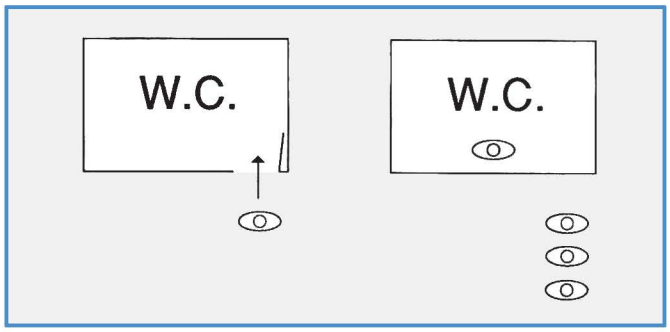

Figura 5 - Il meccanismo Lock-UnLock può essere equiparato al comportamento di un individuo che deve entrare da solo in una stanza richiudendo la porta dietro di sé. Un secondo individuo, prima di entrare, dovrà aspettare la liberazione del locale.

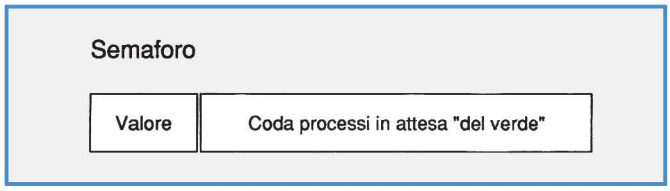

Figura 6 - Semaforo è una struttura dati di sistema formata da un campo valore (rosso-/verde) e da una lista di processi in attesa sul semaforo.

R1 viene salvato il vecchio valore della cella 1000 e quindi al suo rientro in gioco l'incremento effettuato dal processo 2 risulta annullato dal fatto che nella cella condivisa ci finisce il vecchio valore incrementato di uno e non il nuovo valore già incrementato dal processo 2.

Questo è ovviamente solo un esempio semplice di come sia possibile produrre errori quando sono in gioco sezioni critiche di codice (che modificano strutture condivise) e non si prendono, conseguentemente, gli opportuni provvedimenti.

E abbiamo solo incrementato una cella di memoria! Immaginate cosa può succedere se la nostra struttura condivisa era una lista o un buffer circolare con operazioni di inserimento, estrazione, inizializzazione, ecc. ecc.

Gli opportuni provvedimenti

Come nell'esempio del passaggio a livello, esistono vari modi di risolvere il problema più o meno efficientemente.

In un sistema multitask uniprocessor il sistema più semplice è quello di disabilitare in toto il multitasking quando il processo entra in una sezione critica. In questo modo si può essere certi che nessun altro processo prenda il nostro posto essendo in quel momento il sistema semplicemente monotask. Utilizzando ad esempio la coppia di istruzioni per disabilitare e poi riabilitare gli interrupt avremmo effettivamente (e con dispendio minimo) l'effetto voluto:

```
DIS                    :
MOVE  $1000,R1         :
ADD   #1, R1           :   PROCESSO 1
MOVE  R1,$1000         :
ENA   .                :
```

Analogamente, per i sistemi multitask multiprocessor, il blocco sarà esteso anche all'arbitro della memoria che così permetterà solo al processo in questione di agire correttamente.

E questo metodo funziona ugualmente per strutture dati e operazioni più complesse: è esattamente come tagliare la testa al toro.

Ma quando le sezioni critiche sono molto lunghe, si corre il rischio di sospendere il multitasking per intervalli troppo lunghi, con a volte problemi per quei sistemi in cui le temporizzazioni software sono di importanza cruciale per il corretto funzionamento di tutta la macchina. E poi è ingiusto fermare tutto e tutti (nel caso multiprocessor anche gli altri processori) solo perché un processo sta facendo una cosa «delicata». Una soluzione certamente migliore è di riuscire a fermare solo i processi che intendono modificare (o utilizzare) le stesse strutture dati nello stesso momento «critico».

Chiavi, Semafori, Monitor

Un primo (e mal funzionante...) approccio per proteggere le sezioni critiche senza fermare il multitasking è quello di definire un certo numero di chiavi, che indicheremo con K, sulle quali eseguire operazioni di Lock e UnLock. In pratica un processo prima di entrare in una sezione critica (ripetiamo: inizia una sequenza di istruzioni indivisibile) esegue una Lock su una determinata chiave K (relativa alla struttura dati interessata) e con il ritorno da questa è implicitamente autorizzato ad utilizzare la struttura dati condivisa. Appena terminate le operazioni della sezione critica esegue in corrispondenza alla Lock precedente una UnLock con la quale libera l'accesso per gli altri utenti.

La Lock è molto semplice e si basa in pratica sulla lettura e modifica simultanea del valore associato alla chiave K:

```
        MOVE #0,R1
loop:   SWAP R1,K
        BEQ  loop
        RTS
```

La UnLock è molto più semplice:

```
MOVE #1,K
```

In pratica quando la chiave vale 1 vuol dire che la sezione critica può essere eseguita dopo aver posto la chiave a 0.

Grazie all'operazione di SWAP possiamo con una sola istruzione (quindi *intrinsecamente* indivisibile) modificare una cella di memoria (all'indirizzo K) e averne una copia del suo contenuto precedente in un registro interno. Poniamo a 0 la chiave e contemporaneamente possiamo verificare che precedentemente era posta ad 1. Nel caso contrario (stiamo ponendo a 0 qualcosa che già è 0) vuol dire che la chiave non era disponibile e dobbiamo ritentare di nuovo.

È chiaro che un meccanismo di questo tipo funziona, ma ad un costo troppo elevato: la prima cosa da NON fare in un sistema multitask è proprio l'attesa attiva: in gergo, loopare attendendo un determinato evento.

Sarebbe giustificato l'uso della coppia Lock-UnLock solo nel caso in cui le sezioni critiche sono poche e molto brevi, e quindi la probabilità di eseguire attesa attiva è abbastanza bassa. Un sistema migliore per implementare la mutua esclusione è quello dei semafori, dal comportamento esteriore molto simile a quello delle Lock.

Anche per i semafori abbiamo due funzioni una da utilizzare all'inizio di una sezione critica per avere l'uso esclusivo della struttura dati condivisa e una da invocare non appena si esce dalla sezione critica per concedere l'uso ad altri utenti che ne faranno richiesta.

Le due funzioni si chiamano P e V e hanno un solo parametro: il semaforo sul quale effettua l'interrogazione/selezione, come per le Lock, relativo ad ogni struttura sulla quale eseguire la mutua esclusione.

Non pensate però, per favore, ai dementi semafori stradali: qui l'analogia è semmai con i semafori ferroviari che indicano verde quando la tratta successiva è libera (quindi il treno può accedervi), rosso nel caso contrario in cui un treno è già presente sulla tratta successiva e occorre attenderne la liberazione.

Ma questa volta l'attesa non sarà attiva: associato ad un semaforo infatti avremo oltre al valore del semaforo stesso (verde/rosso) anche una lista di processi che, avendo «trovato rosso», attendono in stato di attesa (scusate il rigiro di parole) che il semaforo diventi verde.

L'uso è molto semplice: facciamo un esempio. Immaginiamo di dover aggiungere un elemento in coda ad una lista condivisa da più processi. Diciamo che

la lista si chiama LISTA, ogni elemento ha due campi CORPO e SUCC, l'elemento da inserire si chiama ELEMENTO e che il semaforo ad essa associato si chiami SemLISTA. Il codice è il seguente:

```
P(SemLISTA)
DUMMY := LISTA;
WHILE DUMMY.SUCC != NULL
      DUMMY := DUMMY.SUCC;
DUMMY.SUCC := ELEMENTO;
ELEMENTO.SUCC := NULL;
V(SemLISTA);
```

commentiamolo brevemente. Con la chiamata alla funzione P sul semaforo SemLISTA il processo chiamante, se il semaforo è verde, può continuare la sua elaborazione ponendolo a rosso altrimenti viene sospeso fino a quando non diventa nuovamente verde. L'elemento DUMMY è usato per scorrere la lista fino all'ultimo elemento, ovvero fino all'elemento che ha la costante NULL nel campo SUCC. Trovato l'ultimo elemento si può inserire il nuovo arrivato in coda eseguendo un aggiornamento dei relativi puntatori. Terminata la breve operazione è necessario eseguire una V sullo stesso semaforo per farlo tornare verde per i futuri accessi.

Adesso vediamo una possibile implementazione delle funzioni P e V.

```
P(Semaforo)::

if Semaforo.valore = VERDE
   then Semaforo.valore := ROSSO
   else
      begin
        Accoda(Semaforo.lista, RUNNING)
        Prerilascio(RUNNIGN)
      end
```

La costante RUNNING indica il processo in esecuzione in quel momento quindi lo stesso processo che esegue quelle istruzioni. La funzione Prerilascio permette di sospendere il processo in esecuzione ponendolo in stato di attesa. Per il resto il codice si commenta da sé: se il semaforo è verde lo pone a rosso altrimenti pone il processo chiamante in lista sulla coda relativa a quel semaforo e lo sospende. La funzione V è complementare alla P:

```
V(Semaforo)::
if Semaforo.lista = VUOTA
   then Semaforo.valore := VERDE
   else PoniPronto(Preleva(Semaforo.lista))
```

Qui troviamo un'altra ipotetica funzione di sistema che pone in stato di pronto (lo inserisce nella coda dei processi pronti) il processo passato come parametro che è quello restituito dalla funzione Preleva che agisce sulla coda del semaforo. In pratica la V funziona in questo modo: se la coda associata al semaforo è vuota, il semaforo può ritornare verde essendo ormai libera la risorsa condivisa. Nel caso contrario viene lasciato rosso, ma un processo che precedentemente si era accodato sul semaforo (e quindi veniva posto in stato di attesa) adesso passa in stato di pronto e quindi eseguito quanto prima.

```
Monitor LISTA::

TYPE coda: {
           VAR Corpo: string;
           VAR Succ : pointer;
           }

VAR lista: coda;

Procedure ENTRY Inserisci(elemento: coda)
begin
  VAR dummy: coda;

  if lista = NULL
    then
      begin
        lista = elemento;
        lista.succ = NULL;
      end
  else
    begin
      dummy := lista;
      while dummy.succ != NULL do
            dummy := dummy.succ;
      dummy.succ := elemento;
      elemento.succ := NULL;
    end
end

Procedure ENTRY Preleva(elemento: coda)
begin
  if lista != NULL then
    begin
      elemento := lista;
      lista := lista.succ;
    end
  else elemento := NULL;
end

INIT:
begin
  lista := NULL
end
```

Figura 7

Le P e le V funzionano tutto sommato abbastanza bene: logicamente a loro volta sono anch'esse operazioni di sistema indivisibili quindi non c'è rischio che due o più processi leggano contemporaneamente lo stato del semaforo trovandolo tutti verde. Il sistema operativo infatti farà in modo da non effettuare permutazioni di contesto mentre è in corso l'esecuzione di una P o di una V.

Tra gli svantaggi più evidenti della cooperazione a mezzo semafori troviamo il fatto che è lasciata all'utente la responsabilità di eseguire una V dopo ogni P così come la raccomandazione di non annidare chiamate di P (ovvero di trattare sezioni critiche all'interno di sezioni critiche più grosse) poiché è alto il rischio di incorrere in uno stallo. Per questi ed altri motivi dello stesso genere è meglio non dare visione diretta all'utente della sincronizzazione tramite semafori ma fornire alcuni meccanismi più di alto livello per implementare in maniera diversa la concorrenza.

Un sistema abbastanza diffuso è quello dei cosiddetti monitor che permettono all'utente programmatore di inglobare in un unico modulo tanto le strutture dati condivise quanto le funzioni che agiscono sulle stesse. Sarà compito del sistema garantire la mutua esclusione nelle chiamate a queste funzioni. In pratica se il processo 1 sta eseguendo la funzione Inserisci di un monitor che ingloba una lista, qualsiasi altro processo che tenderà di accedere alla stessa o ad altre funzioni del medesimo monitor verrà posto in stato di attesa fino a quando il processo 1 non avrà terminato.

La definizione di un monitor è abbastanza semplice.

In figura 7 troviamo ancora una volta l'esempio della lista condivisa.

All'inizio ci sono le dichiarazioni degli oggetti condivisi: nel nostro caso la lista in questione. Seguono le definizioni di due procedure ENTRY ovvero richiamabili dall'esterno in modo esclusivo. Termina la definizione del monitor l'inizializzazione delle strutture dati condivise, nel nostro caso la lista inizializzata a «lista vuota». Da notare che gli utilizzatori non hanno assoluta visione della lista in questione ma, in pratica, di un «black box» accessibile solo attraverso due funzioni per inserire e estrarre elementi secondo una politica FIFO (First In First Out).

Conclusioni

I meccanismi di sincronizzazione brevemente mostrati in quest'articolo riguardano, come detto nell'introduzione, le comunicazioni inter process «ad ambiente globale». Riassumendo, in un sistema siffatto, esiste una certa quantità di strutture dati condivise, in pratica accessibili da più processi concorrenti attraverso meccanismi di mutua esclusione.

Un metodo alternativo di comunicazione interprocess è quello cosiddetto «ad ambiente locale». Con questo sistema, che vedremo più approfonditamente nei prossimi numeri, non esistono zone di memoria condivise dai processi (al livello di quest'ultimi, s'intende!) ma ogni comunicazione o sincronizzazione avviene attraverso scambio di messaggi. Non esisterà più il monitor come struttura «morta» atta solo a proteggere l'accesso contemporaneo ad una determinata struttura dati condivisa, ma quest'ultime saranno sempre inglobate (e quindi assolutamente private) in altri processi, quindi «vivi», concorrenti. L'esempio visto prima del monitor che gestisce una lista condivisa, nel modello di cooperazione ad ambiente locale si traduce nella creazione di un vero e proprio processo gestore della risorsa (la lista) che su una porta d'ingresso riceve comandi ed eventualmente il dato da inserire (se si tratta appunto di un inserimento) e dalla sua porta d'uscita restituisce il risultato dell'operazione che nel caso di prelevamento corrisponderà all'elemento letto.

Appuntamento dunque ai prossimi numeri dove, tra l'altro, mostreremo anche qualche esempio di programmazione multitask in vari linguaggi di programmazione concorrente.

MC