TECNICHE DI PROGRAMMAZIONE STRUTTURATA ESTESE AD UN AMBITO DI PROCESSI CONCORRENTI

P. Ancilotti (*) - M. Boari (**) - N. Lijtmaer (*)

(*) *Istituto di Elaborazione della Informazione del Consiglio Nazionale delle Ricerche. Pisa - Italia*
(**) *Istituto di Automatica Facoltà di Ingegneria Università di Bologna. Bologna - Italia*

1. INTRODUZIONE

L'introduzione del concetto di processo ha fornito uno strumento adatto sia alla descrizione del comportamento dinamico dei grossi sistemi, sia alla organizzazione del loro progetto [14, 15].

Studi fatti sulle proprietà dei processi hanno permesso di definire la *Multiprogrammazione* come l'insieme di tecniche usate per il controllo di processi concorrenti su un sistema costituito da uno o più elaboratori [4].

Il campo di applicazione dei processi paralleli e dunque della multiprogrammazione riguarda vari casi [13].

- La necessità di sfruttare le caratteristiche intrinseche di un « multi-processor ».
- La sovrapposizione di lunghe attività di ingresso-uscita con il tempo di elaborazione.
- Progetto di sistemi operativi.
- Applicazioni in tempo-reale.
- Studi di simulazione.
- Programmazione Euristica.

Un impiego non controllato della multiprogrammazione, pur introducendo notevoli vantaggi per quanto riguarda l'efficienza, crea una problematica particolare:

- Incremento nella complessità del sistema.
- La necessità di prendere in considerazione la velocità relativa dei processi.
- La non riproducibilità dei risultati.
- Difficoltà nella formulazione delle asserzioni logiche che permettono di verificare la correttezza del sistema.

La multiprogrammazione strutturata ha come scopi essenziali: rendere la complessità proporzionale alle dimensioni del sistema, eliminare errori dipendenti dal tempo, e facilitare la formulazione delle asserzioni per una eventuale prova di correttezza. Inoltre le tecniche proprie della programmazione strutturata permettono di eliminare la maggior parte degli

errori in tempo di compilazione e rendere più semplice la fase di « debugging » e di « maintenance ».

2. COMBINAZIONE DI PROCESSI

Concetti intuitivi, come informazione e unità di elaborazione (processor), sono alla base della nostra esperienza di programmazione. Una trattazione formale di: « processor », processi, variabili di stato, insieme di variabili di stato, spazio degli stati, « computation », si trova nel lavoro di Horning e Randell [14].

Data un'unità di elaborazione P, un insieme di informazioni $Y = \{ y_0, y_1, \ldots, y_n \ldots \}$, un insieme di stati $S = \{s_0, s_1, \ldots, s_n \ldots \}$; se P opera su y_0 per fornire y_1 come risultato, passando dallo stato s_0 allo stato s_1:

$$(y_1, s_1) \text{ è il successore di } (y_0, s_0)$$

$$y_0, s_0 \xrightarrow{P} y_1, s_1$$

In certi casi però un solo valore y_0 può avere più di un successore

$$y_0, s_0 \xrightarrow{Q} y_1, s_1$$

$$y_0, s_0 \xrightarrow{Q} y_2, s_2$$

$$y_0, s_0 \xrightarrow{Q} y_k, s_k$$

Il « processor » Q, in questo caso si dice *non-deterministico.*

Formalmente un processo può essere definito come una terna (Y, P, V) dove

Y è l'insieme di informazione

P l'unità di elaborazione o « processor »

V è l'insieme dei valori iniziali delle coppie (y, s)

La definizione di processo non fa riferimento al tempo, ma ad una sequenza ordinata di stati. Nel modello si assume che le operazioni dei « processors » richiedono un *tempo nullo* e che il tempo venga aggiornato in forma discreta alla fine di ogni operazione [11].

Si ottiene una *combinazione seriale* di due processi se tutti gli stati finali di un processo sono gli stati iniziali dell'altro.

Schematicamente la combinazione seriale può essere rappresentata dal grafo di precedenza di fig. 2.1.

Si ottiene una *combinazione non seriale* se ogni processo agisce sulle proprie variabili di stato a istanti che sono indipendenti dagli altri processi. Alcune delle variabili di stato possono essere comuni.

Schematicamente questo tipo di combinazione è rappresentata dal grafo di precedenza in Fig. 2.2.

Dal punto di vista formale il processo R, risultante d'una combinazione è il prodotto dei processi componenti: $R = P_1 \times P_2 \times \ldots \times P_n$.

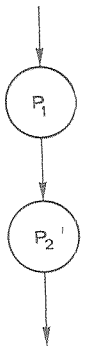


Fig. 2.1.

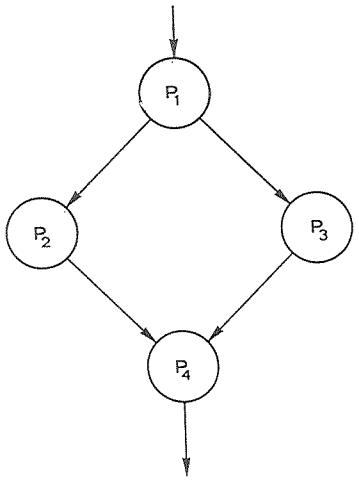


Fig. 2.2.

Un sistema complesso può essere visto come un processo. Questo processo può essere descritto come combinazione di altri processi, ognuno dei quali, a sua volta, può essere visto come combinazione di processi più semplici [10].

Mentre è scopo della programmazione sequenziale lo studio delle proprietà della combinazione seriale è scopo della programmazione parallela — multiprogrammazione — lo studio delle proprietà della combinazione non seriale. Nella letteratura, processi che vengono combinati secondo le regole delle combinazioni non seriali, compaiono con il nome di *processi concorrenti* [3].

A seconda delle restrizioni imposte alla definizione di processi concorrenti, si ottengono due sottoclassi: *processi disgiunti e processi cooperanti.* Sono disgiunti se non hanno tra di loro variabili di stato comuni e cooperanti in caso contrario.

In generale la combinazione tra processi dà luogo a interazioni di tre tipi: *cooperazione*, riguardante l'interazione prevedibile e desiderata; *interferenza*, relativa a interazioni non prevedibili e non desiderate; e *competizione*, relativa a quelle interazioni che anche se prevedibili e accettabili non sono desiderate.

Per esempio, prendiamo in considerazione due processi concorrenti, P_A e P_B che prevedono di scambiare informazioni per mezzo di un «buffer» (variabile di stato comune), Fig. 2.3).

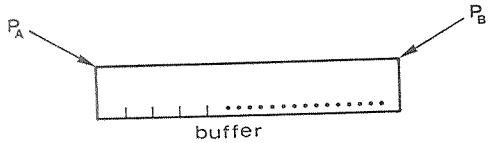


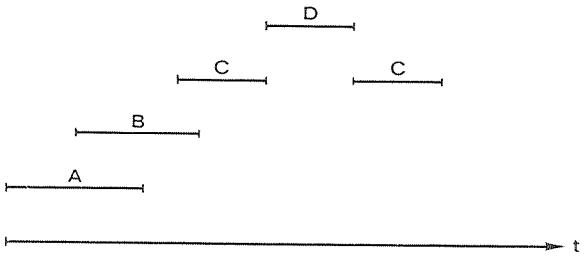
Fig. 2.3.

Un primo tipo di interazione si ottiene in quanto P_A e P_B competono per lo sfruttamento della risorsa « buffer »: *competizione*. Essendo P_A il produttore dell'informazione è immediato che P_B non potrà prelevare questa informazione prima che P_A l'abbia prodotta, altrimenti si genererebbe una *interferenza* non desiderata tra i processi. È necessario, dunque, imporre dei vincoli a P_A e P_B tali che permettano la *cooperazione* tra di loro.

È scopo della *multiprogrammazione strutturata*: garantire la cooperazione, evitare l'interferenza e controllare la competizione. Inoltre la possibilità di analizzare un processo complesso come una combinazione di processi sequenziali permette di definire certi vincoli sulla struttura delle combinazioni di processi, tali che la complessità del processo risultante sia proporzionale alla sua dimensione. Infine asserzioni relative ai processi sequenziali componenti possono essere gradualmente sostituite da asserzioni relative al processo risultante.

3. PROCESSI CONCORRENTI

L'evoluzione nel tempo di processi concorrenti permette di rilevare due casi: questi processi possono essere completamente sovrapposti nel tempo (overlap) o intercalati (interleaving)



$A \,//\, B \,//\, C$ sono sovrapposti
$C \,//\, D$ sono intercalati.

Fig. 3.1.

Le proprietà logiche dei processi concorrenti sono le stesse indipendentemente dal tipo di evoluzione nel tempo. In particolare non sarà fatta nessuna assunzione sulle velocità relative dei singoli processi (purchè siano positive). Questo vincolo è necessario in quanto la combinazione dei processi è determinata dinamicamente durante l'esecuzione di un sistema da parte dell'algoritmo di « scheduling » e non può essere prevista a priori. Inoltre questo vincolo introduce una diminuzione nella complessità del sistema perchè ne permette l'analisi o il progetto senza considerare la casistica che deriverebbe tenendo conto di tutte le possibili velocità relative. Si noti, inoltre, che nella definizione di processo si prescinde dalla nozione di tempo.

Molti studi sono stati fatti in passato [1], per esprimere la concorrenza all'interno di un

programma. Senza riportare tutti i costrutti linguistici introdotti da vari autori, del resto spesso equivalenti, prenderemo in considerazione due soluzioni di particolare interesse e generalità dovute rispettivamente a Conway [5] e a Dijkstra [8].

La prima soluzione, ripresa anche da Dennis e Van Horn [6], introduce tre primitive(*): FORK, QUIT, JOIN che hanno come scopo essenziale l'attivazione e la terminazione di processi concorrenti.

FORK PROC

Questa primitiva attiva un nuovo processo la cui esecuzione inizia con l'istruzione che ha come etichetta PROC.

QUIT

Dopo l'esecuzione di questa primitiva il processo cessa di esistere.

JOIN C,A

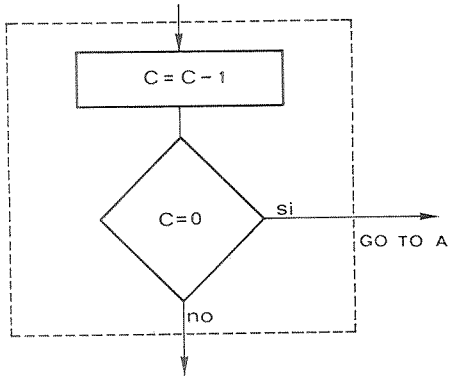


Fig. 3.2.

Dove C è il nome di un contatore e A è una etichetta. L'esecuzione di *JOIN*, schematizzata nella Fig. 3.2, implica il decremento del valore del contatore C. Se il suo valore è zero il processo termina e viene attivato un nuovo processo in A. Altrimenti continua in sequenza.

La soluzione proposta da Dijkstra nel 1965 per esprimere la concorrenza fra processi consiste nella seguente notazione linguistica

$$cobegin \quad P_1; P_2; \ldots; P_n \quad coend$$

che indica l'esecuzione parallela dei processi $P_1; P_2; \ldots P_n$.

(*) S'intende per primitiva una singola istruzione macchina, o un loro insieme, non interrompibile.

Per esempio il processo: *begin*

P_0;
cobegin $P_1; P_2; \ldots P_n$ *coend*
P_{n+1};
end

dà luogo al grafo di precedenza mostrato in Fig. 3.3.

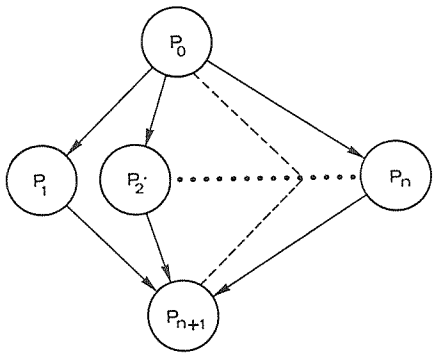


Fig. 3.3.

3.1. Un esempio di programmazione parallela: sistema di gestione di m terminali attivi contemporaneamente

Le specifiche del problema sono: dati m terminali che richiedono servizio contemporaneamente è necessario memorizzare, dopo una breve analisi di validità, tutta l'informazione su disco. (Sistema di « spooling »). È immediato che una prima soluzione è quella di sequenzializzare tutti i processi. Si cercheranno invece, soluzioni che tengano conto del parallelismo intrinseco del sistema. Una prima soluzione viene presentata sfruttando le primitive di CONWAY, mettendo in rilievo come la loro generalità di uso possa condurre ad algoritmi non strutturati. La soluzione che utilizza la proposta di Dijkstra permette in modo immediato di ottenere un algoritmo strutturato. Il paragrafo finirà con un confronto tra le due soluzioni. La notazione linguistica utilizzata non è propria di nessun linguaggio, è però, di immediata comprensione.

Le seguenti ipotesi sono comuni alle due soluzioni:

a) Ogni richiesta da un terminale dà luogo ad un processo.

b) Ognuno di questi processi adopera un sistema di tre « buffers » indipendenti: PRECEDENTE, ATTUALE, SUCCESSIVO.

c) m è il numero di terminali attivi; i è l'indice del generico terminale.

d) MAX è il numero di « records » che ogni terminale trasmetterà (MAX $\geqslant$ 2).

3.1.1. Algoritmo 1

```
begin   i : = 1;
        while i ⩽ m do
        fork PROCESSO;
        i : = i + 1;
        quit;
end

PROCESSO:  begin
                    LETTURA (MAX);
                    LETTURA (PRECEDENTE);
                    ELABORAZIONE (PRECEDENTE);
                    t : = 2;
                    fork DISCO;
                    LETTURA (ATTUALE);
                    while < not EOF >  do
                    u : = 2;
                    fork INPUT;
                    ELABORAZIONE (ATTUALE);
                    join (t, CONTINUA 1);
                    quit;
CONTINUA 1:         PRECEDENTE: = ATTUALE;
                    t : = 2;
                    fork DISCO;
                    join (u, CONTINUA 2);
                    quit;
CONTINUA 2:         ATTUALE: = SUCCESSIVO;
                    end
                    join (t, FINE);
                    quit;
FINE:               ELABORAZIONE (ATTUALE);
                    REGISTRA (ATTUALE);
                    quit;
               end
DISCO:         begin
                    REGISTRA (PRECEDENTE);
                    MAX: = MAX — 1;
                    if MAX = 1 then join (t, FINE);
                               else join (t, CONTINUA 1);
                    quit;
                    end
```

```
INPUT: begin
                LETTURA (SUCCESSIVO);
                join (u, CONTINUA 2);
                quit;
       end
```

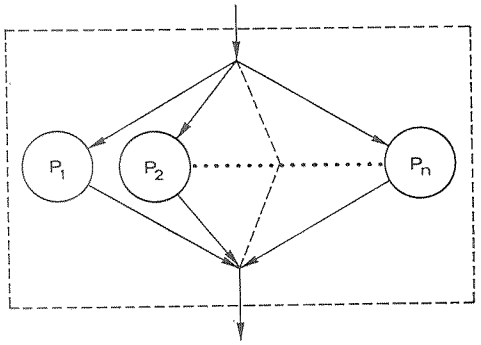
Questo algoritmo non è sicuramente la migliore soluzione sfruttando le primitive di Conway. È possibile dare una soluzione ben strutturata, però non è immediata. Inoltre la primitiva *JOIN* ha gli stessi inconvenienti del *GO TO*. È necessario sottolineare la difficoltà di lettura di questo algoritmo. Particolarmente difficile diventa la determinazione dei punti d'inizio e fine di ogni processo e delle variabili su cui agisce.

3.1.2. Algoritmo 2

```
cobegin PROCESSO 1; ... PROCESSO i; ... PROCESSO m; coend
PROCESSO i: begin
                LETTURA (MAX); LETTURA (ATTUALE);
                cobegin LETTURA (SUCCESSIVO);
                        ELABORAZIONE (ATTUALE);
                coend
                MAX: = MAX — 2;
                while MAX > 0 do
                   PRECEDENTE: = ATTUALE; ATTUALE: = SUCCESSIVO;
                   cobegin      LETTURA (SUCCESSIVO);
                                ELABORAZIONE (ATTUALE);
                                REGISTRA (PRECEDENTE);
                   coend
                   MAX: = MAX — 1;
                end
                cobegin REGISTRA (ATTUALE);
                        ELABORAZIONE (SUCCESSIVO);
                coend
                REGISTRA (SUCCESSIVO);
            end
```

3.1.3. *Confronto tra le due soluzioni proposte*

L'algoritmo 2 è una soluzione del problema che tiene conto del parallelismo dei processi. Inoltre ha tutte le caratteristiche di un algoritmo sequenziale ben strutturato in quanto adopera le sole strutture di controllo: *concatenazione ripetizione*. Utilizza inoltre una nuova struttura di controllo per la concorrenza che ha le stesse caratteristiche delle strutture di controllo precedenti, cioè un solo punto di inizio e un solo punto di fine (fig. 3.4.)



cobegin P_1; P_2; ...; P_n *coend*

Fig. 3.4.

È interessante notare che il comportamento dinamico dell'algoritmo può essere dedotto dal testo stesso, e che il processo risultante dalla combinazione di P_1, P_2 ... P_n è *non deterministico* pur garantendo *una riproducibilità nei risultati finali.*

Da un confronto fra le due soluzioni (Algoritmo 1 e 2) appare chiaro che la semplicità della seconda soluzione si è ottenuta a scapito della maggiore flessibilità delle primitive di controllo utilizzate.

4. PROCESSI DISGIUNTI

Nel paragrafo 2, sono stati definiti come sottoclasse dei processi concorrenti, i *processi disgiunti.* La caratteristica essenziale dei processi disgiunti è che ognuno di loro agisce, modificandole, su variabili di stato diverse. Possono esserci variabili di stato comuni purchè non siano alterate da nessuno dei processi. È immediato che in questo caso l'interferenze tra i processi sono nulle a meno di errori nella stesura dei programmi. Una soluzione, che permette la rilevazione di questi errori in tempo di compilazione, consiste nella introduzione di una particolare dichiarazione mediante la quale si definisce privata ogni variabile che può essere modificata dal processo. Per esempio nell'algoritmo 2, all'interno del *while,* sono stati definiti tre processi: LETTURA, ELABORAZIONE, REGISTRA che sono disgiunti perchè agiscono, modificandole, su variabili (« buffers ») diverse: SUCCESSIVO, ATTUALE, PRECEDENTE. Se all'interno della struttura di controllo *cobegin ... coend* si introducono le dichiarazioni:

P_1: *PRIVATE* SUCCESSIVO ....
P_2: *PRIVATE* ATTUALE .....
P_3: *PRIVATE* PRECEDENTE .....

il compilatore è in grado di controllare che nessun processo faccia riferimento a variabili private di altri processi. Le variabili dichiarate private possono essere locali al blocco *cobegin ... coend,* o globali.

Si può provare [13, 2] che se A_1, A_2, ... A_i sono asserzioni vere prima dell'esecuzione dei processi P_1, P_2, ..., P_i e se R_1, R_2, ... R_i, sono asserzioni vere sui risultati dei singoli processi, allora l'esecuzione *cobegin* P_1; P_2; ...; P_i *coend* assicura che $R_1 \wedge R_2 \wedge \ldots \wedge R_i$ sarà vera se è vera l'asserzione iniziale $A_1 \wedge A_2 \wedge \ldots \wedge A_i$.

5. PROCESSI COOPERANTI

In contrapposizione ai processi disgiunti sono stati definiti nel paragrafo 2, i processi cooperanti, ossia quelli che interagiscono per mezzo di variabili comuni. Se le variabili comuni sono risorse fisiche o logiche i processi competono per la loro utilizzazione. È necessario dunque la sincronizzazione di tutti i processi sulla gestione di queste risorse mediante un meccanismo di *mutua esclusione*. Questo metodo di interazione è di tipo indiretto, perchè ogni processo ignora l'identità degli altri. Un metodo di interazione diretta si ha, invece, quando i processi devono cooperare per uno scopo comune, scambiandosi informazione.

La forma più semplice di cooperazione fra due processi si ha quando la sola informazione scambiata consiste in un *segnale temporale* che indica il verificarsi di un dato evento. Questa situazione si presenta ad esempio, quando un processo deve essere attivato ad intervalli fissati di tempo, oppure nel caso di completamento di un'operazione di ingresso-uscita. L'informazione scambiata può in altri casi consistere in *messaggi* (insieme di dati) che vengono generati da un processo e utilizzati da un altro. Un tipico esempio è rappresentato da due processi preposti al riempimento e svuotamento di un « buffer ».

Sorge la necessità di introdurre meccanismi di sincronizzazione che garantiscano la corretta cooperazione tra i processi. Questi problemi di sincronizzazione possono essere risolti disponendo, tra le istruzioni del linguaggio in cui sono scritti i programmi per i processi concorrenti, di opportune operazioni primitive. Il loro uso deve garantire che un processo che non può continuare l'esecuzione rimanga in attesa fino all'arrivo di un segnale da parte di un altro processo in grado di riattivarlo.

Ad esempio Dijkstra [8] ha introdotto le primitive *wait*; *signal*, originariamente chiamate P e V, che operano su variabili di tipo semaforo. In particolare è stato dimostrato da Habermann [12] che ogni problema di sincronizzazione può essere risolto mediante queste primitive.

Tuttavia, come si vedrà nel seguito, per alcuni problemi l'impiego di diversi meccanismi di sincronizzazione rende più agevole la strutturazione dei programmi e facilita l'eliminazione di possibili errori in fase di compilazione.

5.1. Definizione di semaforo e delle primitive di sincronizzazione wait e signal

Un semaforo è una variabile s comune a più processi il cui stato è definito da una costante non negativa C(s) assegnata quando la variabile è inizializzata e dalle quantità inizialmente nulle:

$nw(s)$ = numero di volte che la primitiva *wait*(s) è stata eseguita.
$ns(s)$ = numero di volte che la primitiva *signal*(s) è stata eseguita.
$np(s)$ = numero di volte che un processo è stato abilitato a proseguire in seguito ad una *wait*(s).

Al semaforo è associata una coda a cui appartengono i processi la cui esecuzione deve essere ritardata in attesa di un segnale da parte di altri processi.

L'esecuzione di una *wait*(s) ha il seguente effetto sullo stato del semaforo: $nw(s) \leftarrow nw(s) + 1$; se $nw(s) \leqslant C(s) + ns(s)$ il processo può proseguire l'esecuzione e quindi $np(s) \leftarrow np(s) + 1$; diversamente il processo sospende l'esecuzione e viene messo nella coda. In altri termini, un processo che esegue una *wait*(s) viene ritardato se il numero complessivo delle *wait*(s) eseguite sul semaforo eccede il numero delle *signal*(s) aumentato di $C(s)$.

L'esecuzione di una *signal*(s) ha il seguente effetto sullo stato del semaforo: se $nw(s) > C(s) + ns(s)$ viene reso attivo un processo presente nella coda cioè $np(s) \leftarrow np(s) + 1$. Il contatore $ns(s)$ viene incrementato di uno, cioè $ns(s) \leftarrow ns(s) + 1$. Dopo l'esecuzione di una *signal*(s) il processo continua la sua attività.

Signal(s) e *wait*(s) sono le sole operazioni che possono agire sulla variabile s. Dalla definizione di *wait*(s) e *signal*(s) si può dimostrare che la relazione:

$$np(s) = \min [nw(s),\ C(s) + ns(s)] \tag{1}$$

è invariante rispetto all'esecuzione di *wait*(s) e *signal*(s) [12]. La (1) afferma che il numero di volte che un processo è stato abilitato a proseguire non è maggiore del numero di volte che è stata eseguita la *wait*(s), nè è maggiore del numero di volte che è stata eseguita la *signal*(s) aumentata di $C(s)$.

Come vedremo, la proprietà che la relazione (1) sia invariante rispetto all'esecuzione di *wait*(s) e *signal*(s) può essere utilizzata per dimostrare che l'interazione tra processi concorrenti avviene correttamente.

5.2. Scambio di segnali temporali

Come si è detto precedentemente la forma più semplice di interazione tra due processi si ha quando la sola informazione scambiata consiste in un segnale di tempo che indica il verificarsi di un dato evento. Si considerino ad esempio, due processi P_1 e P_2 di cui il primo necessita per la sua attivazione di un segnale proveniente dal secondo. Il numero di attivazioni di P_1 deve essere in ogni istante indipendente dalla velocità relativa dei due processi e uguale al numero di segnali inviati da P_2 oppure uguale al numero di richieste di attivazione da parte di P_1. Cioè indicando con:

n_1 = il numero di richieste di attivazione di P_1
n_2 = il numero di segnali di attivazione inviati da P_2
n_3 = il numero di volte in cui P_1 è stato riattivato

deve essere ad ogni istante:

$$\text{se} \quad n_2 \geqslant n_1 \quad \text{allora} \quad n_3 = n_1 \tag{2}$$

$$\text{se} \quad n_2 < n_1 \quad \text{allora} \quad n_3 = n_2 \tag{3}$$

Una soluzione al problema può essere facilmente ottenuta mediante l'introduzione di una variabile s di tipo semaforo, con $C(s) = 0$, e delle due primitive *wait*(s) e *signal*(s).

Si ha infatti:

```
cobegin
    P1: begin
            while < COND. TRUE > do
              .
              .
              .
            wait(s);
              .
              .
              .
            end
        end
    P2: begin
            while < COND. TRUE > do
              .
              .
              .
            signal(s)
              .
              .
              .
            end
        end
coend
```

Il processo P_1 usa una *wait*(s) per effettuare una richiesta di attivazione e il processo P_2 una *signal*(s) per inviare a P_1 un segnale di attivazione. Usando la simbologia propria del semaforo le condizioni (2) e (3) si scrivono:

$$\text{se} \quad \text{ns}(s) \geqslant \text{nw}(s) \quad \text{allora} \quad \text{np}(s) = \text{nw}(s)$$
$$\text{se} \quad \text{ns}(s) < \text{nw}(s) \quad \text{allora} \quad \text{np}(s) = \text{ns}(s)$$

Poichè la relazione (1) diventa in questo caso ($C(s) = 0$)

$$\text{np}(s) = \min\,[\text{nw}(s),\ \text{ns}(s)]$$

le condizioni (2) e (3) sono sicuramente soddisfatte. Si noti che l'impiego delle *wait*(*s*) e *signal*(*s*) forniscono una soluzione al problema di interazione indipendentemente dalla velocità relativa dei due processi. Ciò semplifica il compito del programmatore in misura tanto maggiore quanto più grandi sono le dimensioni del problema e rende più facile la comprensione del programma.

5.3. Mutua esclusione

Si considerino due processi P_1 e P_2 che abbiano accesso ad una tabella di dati; il primo processo introduce nella tabella una stringa di caratteri, il secondo li legge e provvede alla loro stampa. Si supponga che l'informazione contenuta nella tabella sia significativa solo al completamento dell'aggiornamento da parte del processo P_1.

Chiamando con *sezione critica* l'insieme di istruzioni durante le quali i due processi rispettivamente aggiornano e leggono la tabella, è necessario che le due sezioni critiche si escludano mutuamente nel tempo.

Per essere corretta la soluzione deve soddisfare alle seguenti condizioni:

a) un solo processo alla volta può essere nella sua sezione critica;
b) un processo deve completare la sua sezione critica in un tempo finito;
c) l'accesso di un processo ad una sezione critica non deve essere ritardato indefinitamente.

Utilizzando un semaforo s con $C(s) = 1$ e le due primitive *wait*(*s*) e *signal*(*s*) si può scrivere:

```
cobegin
    P1: begin
            while ⟨ COND. TRUE ⟩ do
              .
              .
              .
            wait(s);
            sezione critica;
            signal(s);
            end
        end
    P2: begin
            while ⟨ COND. TRUE ⟩ do
              .
              .
              .
            wait(s);
```

```
            sezione critica;
            signal(s);
            end
        end
coend
```

Si può ora facilmente verificare utilizzando la relazione (1) che la soluzione presentata soddisfa alla prima delle tre condizioni. Infatti il numero di processi n_1 entro la sezione critica è espresso ad ogni istante da:

$$n_1 = \mathrm{np}(s) - \mathrm{ns}(s) \tag{4}$$

La relazione (1) si scrive in questo caso ($\mathrm{C}(s) = 1$)

$$\mathrm{np}(s) = \min\,[\mathrm{nw}(s),\ \mathrm{ns}(s) + 1]$$

cioè

$$\mathrm{np}(s) \leqslant \mathrm{ns}(s) + 1 \tag{5}$$

Dalla (4) e (5) si ha che

$$n_1 = \mathrm{np}(s) - \mathrm{ns}(s) \leqslant 1$$

Poichè d'altra parte *wait*(*s*) precede sempre *signal*(*s*) si ha:

$$\mathrm{np}(s) - \mathrm{ns}(s) \geqslant 0$$

quindi

$$0 \leqslant n_1 \leqslant 1$$

Si può anche facilmente verificare che se ad un dato istante non c'è nessun processo presente nella sezione critica cioè

$$\mathrm{np}(s) = \mathrm{ns}(s) \tag{6}$$

nessun processo può essere in attesa di entrare nella sezione critica. Infatti dalla (1) e dalla (6) si ha che:

$$\mathrm{np}(s) < \mathrm{ns}(s) + 1 \text{ e quindi } \mathrm{np}(s) = \mathrm{nw}(s)$$

La condizione *b*) è sempre verificata a meno che non si presenti un caso di « deadlock » [3]. La condizione *c*) è soddisfatta adottando una regola di priorità per selezionare i processi della coda associata al semaforo in modo che nessuno sia ritardato indefinitamente; tale regola potrebbe essere ad esempio la « first in-first out » che seleziona i processi a seconda dell'ordine di arrivo.

L'uso delle primitive *wait*(*s*) e *signal*(*s*) per realizzare la mutua esclusione pone tuttavia una serie di problemi. Si noti innanzitutto che la soluzione presentata è corretta solo nell'ipotesi che le sezioni critiche di ogni processo siano racchiuse tra le primitive *wait*(*s*) e *signal*(*s*) nell'ordine indicato. Uno scambio dell'ordine delle due primitive cioè:

signal(*s*);
sezione critica;
wait(*s*)

può provocare un errore il cui verificarsi dipende dalla velocità relativa dei due processi. Si supponga infatti che i processi P_1, P_2, P_3 debbano mutuamente escludersi durante l'uso di una stessa risorsa, e che tale mutua esclusione sia programmata per errore nel modo seguente:

```
cobegin
    P1: begin
        signal(s);
        sezione critica S1;
        wait(s);
        end
    P2: begin
        wait(s);
        sezione critica S2;
        signal(s);
        end
    P3: begin
        wait(s);
        sezione critica S3;
        signal(s);
        end
coend
```

Se durante l'esecuzione di S_1, P_1 viene interrotto da P_2, P_2 ha accesso a S_2. Durante l'esecuzione di S_2, P_2 può a sua volta venire interrotto da P_3 che può accedere a S_3. La soluzione programmata consente quindi erroneamente ai tre processi di essere contemporaneamente entro le rispettive sezioni critiche. Si noti che l'errore si manifesta *solo* nell'ipotesi che il processo P_1 sia interrotto durante l'esecuzione della sua sezione critica e che il processo interrompente sia o P_2 o P_3 e che P_2 o P_3 eseguano la loro sezione critica.

Un altro tipo di errore si ha qualora il programmatore scriva:

```
cobegin
    P1: begin
        wait(s);
        sezione critica S1;
        wait(s);
        end
```

```
        P2:  begin
             wait(s);
             sezione critica S2;
             signal(s);
             end
coend
```

Il processo P_1 una volta entrato nella sezione critica non è più in grado di uscire, esso rimane infatti in attesa di un segnale da parte di P_2. Ma poichè P_2 non può accedere alla sezione critica entrambi i processi si bloccano e si verifica un caso di « deadlock ».

Una caratteristica comune a tutti i tipi di errori esaminati è di essere dipendenti dalla velocità relativa dei processi e quindi praticamente impossibili da riprodurre. Poichè d'altra parte le primitive *wait*(*s*) e *signal*(*s*) e le variabili di tipo semaforo possono essere impiegate per risolvere qualunque tipo di problema di sincronizzazione, non si può richiedere che sia il compilatore a rilevare tali errori. In particolare un compilatore non può accorgersi se le primitive sono scambiate, se una o più di esse sono state omesse per errore, se una variabile semaforo è inizializzata in modo non corretto, ecc. Inoltre il compilatore non è in grado di stabilire una corrispondenza tra un semaforo usato per avere accesso esclusivo ad una risorsa e le sezioni critiche che fanno riferimento a tale risorsa. È quindi possibile per errore che una sezione critica faccia riferimento ad una risorsa il cui uso è protetto da un altro semaforo. Per superare questi inconvenienti è stata suggerita [3] la seguente notazione linguistica:

var V: *shared* type T
region V *do* sezione critica *end*

dove V indica l'insieme delle variabili globali a cui fanno riferimento le sezioni critiche dei processi.

Tale notazione offre i seguenti vantaggi:

a) consente di dichiarare V modificabile da uno o più processi;

b) associa esplicitamente una sezione critica su V.

c) consente al compilatore di generare istruzioni che realizzano correttamente la mutua esclusione delle sezioni critiche eliminando quindi gli errori che possono sorgere da un uso scorretto dei semafori.

L'esempio precedente diventa in questo caso:

```
begin
        var V: shared type T;
cobegin
        P1: begin
                while ⟨ COND. TRUE ⟩ do
```

```
            region V do sezione critica end;
            end
        end

    P2 : begin

            while < COND. TRUE > do
            region V do sezione critica end;
            end
        end
coend
end
```

5.4. Scambio di messaggi

Si considerano due processi A e B che si scambiano messaggi mediante un « buffer »; in particolare A, chiamato processo *produttore*, genera messaggi e li deposita nel « buffer » e B, chiamato processo *consumatore*, li preleva dal « buffer » e li utilizza.

La comunicazione deve essere soggetta a due vincoli:

a) il produttore non può inserire un messaggio nel « buffer » se questo è pieno;

b) il consumatore non può prelevare un messaggio dal « buffer » se questo è vuoto;

Nel primo caso il produttore deve essere ritardato fino a che il consumatore non abbia prelevato un messaggio e nel secondo il consumatore deve essere ritardato fino a che il produttore non abbia depositato un messaggio.

È inoltre necessario, affinchè la comunicazione avvenga correttamente, che:

c) produttore e consumatore non interferiscano cioè non abbiano accesso contemporaneamente alla stessa posizione del « buffer »;

d) non si verifichi una situazione di « deadlock » cioè il produttore in attesa che il consumatore prelevi un messaggio e il consumatore in attesa che il produttore depositi un messaggio.

Si supponga il « buffer » suddiviso in N porzioni identiche ciascuna delle quali contenente un messaggio e gestito come una lista circolare mediante le due variabili P_1 e P_2 che individuano rispettivamente la prima porzione vuota e piena del « buffer »; inizialmente sia $P_1 = P_2$. Dopo aver preparato un messaggio il produttore lo deposita nel « buffer » nel modo seguente:
buffer (P_1) : = messaggio prodotto

$P_1 : = P_1 + 1 \pmod N$

Il prelievo di un messaggio da parte di B avviene nel modo seguente:
messaggio prelevato: = buffer (P_2)

$P_2 : = P_2 + 1 \pmod N$

La soluzione al problema di sincronizzazione può essere ottenuta mediante l'impiego di due semafori *pieno*, *vuoto* tale che $C(pieno) = 0$ e $C(vuoto) = N$ e delle due primitive *wait*(s) e *signal*(s). Si ha infatti:

cobegin

Produttore A: *begin*
.
.
.
wait(*vuoto*)
buffer (P_1) : = messaggio prodotto
$P_1 : = P_1 + 1 \pmod N$
signal(*pieno*)
.
.
.
end

Consumatore B: *begin*
.
.
.
wait(*pieno*)
messaggio prelevato : = buffer (P_2)
$P_2 : = P_2 + 1 \pmod N$
signal(*vuoto*)
end
.
.

coend

Fig. 5.4.1.

Le condizioni *a*) e *b*) sono sicuramente soddisfatte in quanto si può dimostrare che:

$$0 \leqslant \mathrm{np}(vuoto) - \mathrm{np}(pieno) \leqslant N \tag{7}$$

Si ha infatti:

$$\mathrm{ns}(pieno) \leqslant \mathrm{np}(vuoto) \leqslant N + \mathrm{ns}(vuoto)$$
$$\mathrm{ns}(vuoto) \leqslant \mathrm{np}(pieno) \leqslant \mathrm{ns}(pieno)$$

da cui con semplici passaggi si ottiene la (7).

Per quanto riguarda la condizione *c*) si noti che indicando con p_1 e p_2 rispettivamente il numero di volte in cui P_1 e P_2 sono stati incrementati, le operazioni di deposito e prelievo agiscono sulla stessa porzione del « buffer » se:

$$p_1 = p_2 \pmod N \tag{8}$$

Durante l'operazione di deposito (buffer (P_1) : = messaggio prodotto) si ha:

$$p_1 = \mathrm{ns}(pieno)$$
$$\mathrm{ns}(pieno) = \mathrm{np}(vuoto) - 1 \leqslant N + \mathrm{ns}(vuoto) - 1$$

Durante l'operazione di prelievo (messaggio prelevato : = buffer (P_2)) si ha:

$$p_2 = \mathrm{np}(pieno) - 1$$
$$\mathrm{np}(pieno) = \mathrm{ns}(vuoto) + 1 \leqslant \mathrm{ns}(pieno)$$

Si ottiene pertanto:

$$\mathrm{np}(pieno) \leqslant \mathrm{ns}(pieno) \leqslant N + \mathrm{np}(pieno) - 2$$

da cui:

$$0 \leqslant \mathrm{ns}(pieno) - \mathrm{np}(pieno) \leqslant N - 2$$

e quindi

$$0 \leqslant p_1 - p_2 - 1 \leqslant N - 2 \tag{9}$$

Poichè la (9) che vale quando i due processi stanno contemporaneamente lavorando sul « buffer » è in contraddizione con la (8), la condizione *c*) è soddisfatta.

Un caso di « deadlock » infine si manifesta se si verificano le seguenti condizioni:

nessun produttore attivo $(\mathrm{ns}(pieno) = \mathrm{np}(vuoto))$ e produttore in attesa $(\mathrm{nw}(vuoto) > \mathrm{np}(pieno))$, cioè

$$\mathrm{ns}(pieno) = \mathrm{np}(vuoto) = N + \mathrm{ns}(vuoto) \tag{10}$$

nessun consumatore attivo $(\mathrm{ns}(vuoto) = \mathrm{np}(pieno))$ e consumatore in attesa $(\mathrm{nw}(pieno) > \mathrm{np}(pieno))$ cioè

$$\mathrm{ns}(vuoto) = \mathrm{np}(pieno) = \mathrm{ns}(pieno) \tag{11}$$

Poichè (10) e (11) sono in contraddizione tra di loro la condizione *d*) è soddisfatta.

Si noti che, affinchè la soluzione proposta sia valida è necessario che le procedure di deposito e di prelievo siano le sole procedure che operano sul « buffer ».

La soluzione proposta in Fig. 5.4.1 è corretta, però qualunque errore sia nell'ordine in cui si scrivono le primitive *wait*(*s*) e *signal*(*s*), sia nella specifica delle variabili semaforo, non è rilevabile in fase di compilazione ed è dipendente dal tempo.

Per ovviare questi inconvenienti sono state introdotte [3] due primitive *send* (M, B) e *receive* (M, B), dove

M è una variabile che identifica il messaggio scambiato tra i due processi

B è il « buffer » attraverso cui avviene lo scambio dell'informazione.

L'algoritmo precedente (Fig. 5.4.1) diventa dunque:

```
begin
        Var B: shared type Buffer;
    cobegin
        Produttore:    begin
                         .
                         .
                         .
                       Var M1: private type Messaggio;
                         .
                         .
                         .
                       send (M1, B);
                         .
                         .
                         .
                       end
        Consumatore:   begin
                       Var M2: private type Messaggio;
                         .
                         .
                         .
                       receive (M2, B);
                         .
                         .
                         .
                       end
    coend
end
```

Fig. 5.4.2.

Le primitive *send* (M, B) e *receive* (M, B) soddisfano le condizioni *a*), *b*), *c*) e *d*) elencate all'inizio del paragrafo. Inoltre si dimostra [7] che un insieme di processi collegati tra di loro per mezzo di meccanismi di trasmissione di messaggi, che soddisfano certe condizioni, da luogo ad un *sistema funzionale*. Il sistema è funzionale se i risultati sono riproducibili indipendentemente della velocità relativa dei processi.

Le condizioni che deve soddisfare il meccanismo di trasmissione di messaggi sono:

a) Ad ogni « buffer » sia associato un unico consumatore ed un unico produttore.
b) Il consumatore deve prelevare i messaggi nello stesso ordine con cui sono stati depositati dal produttore, (gestione FIFO), da un « buffer » che può essere anche infinito.

Si noti però che questa importante proprietà sugli insiemi dei processi è applicabile solo se i processi comunicano tra di loro tramite le primitive *send* (M, B) e *receive* (M, B).

Inoltre nessuno dei processi può agire sui « buffers », se non per mezzo di queste primitive. È dunque chiaro che un compilatore è in grado di rilevare qualunque errore sulla trasmissione.

Send (M, B) e *receive* (M, B) costituiscono il meccanismo di sincronizzazione più idoneo per lo scambio di messaggi.

5.5. Altri meccanismi

Esistono altri meccanismi, per esempio le *sezioni critiche condizionali* introdotte da Hoare [13]. Le sezioni critiche condizionali sono adatte a risolvere la sincronizzazione tra processi che richiedono il verificarsi di una certa condizione sui componenti di una struttura di dati comune. Questo meccanismo è sicuramente utile a descrivere un sistema multiprogrammato, resta però la problematica relativa alla sua realizzazione pratica.

6. CONCLUSIONI

Alle strutture di controllo tipiche della programmazione strutturata: *concatenazione, selezione e ripetizione* è stata aggiunta un'altra struttura *cobegin ... coend* che gode delle stesse proprietà delle precedenti e permette di esprimere la concorrenza tra processi. Lo schema proposto, permette inoltre l'analisi dei processi concorrenti, in forma immediata, dal testo del programma senza considerare la variabile tempo e dunque prescindendo dalle velocità relative dei processi. Eliminando gli errori dipendenti dal tempo viene garantita la riproducibilità dei risultati, senza eliminare però l'indeterminismo proprio di un sistema parallelo.

Sono stati proposti inoltre, un insieme di meccanismi atti a risolvere il problema della sincronizzazione: *semafori* per lo scambio di segnali temporali, *sezioni critiche* per la mutua escluzione e *message buffers* per lo scambio di dati. Si è mostrato come, nonostante i semafori siano sufficientemente generali da risolvere qualunque problematica di sincronizzazione, l'uso di meccanismi ad hoc garantisce che i programmi siano ben strutturati.

Resta aperto il problema di ricavare asserzioni formali appropriate alle descrizioni delle proprietà dei processi cooperanti. Resta inoltre il problema di estendere sistemi attuali (PEARL, [16]) per l'assistenza nella scrittura e test di programmi strutturati sequenziali ai processi cooperanti, non deterministici. Il problema è tutt'altro che banale, però la sua soluzione potrebbe dar luogo a una notevole riduzione nei costi di progetto e programmazione dei sistemi complessi, aumentando l'affidabilità e migliorando la loro comprensione.

7. BIBLIOGRAFIA

[1] J. L. Baer, *A Survey of Some Theoretical Aspects of Multiprocessing*. ACM Comp. Surveys Vol. 5, n. 1, 1973.
[2] P. Brinch Hansen, *Structured Multiprogramming*. Comm. ACM Vol. 15, n. 7, 1972.
[3] P. Brinch Hansen, *Operating System Principles*. Prentice Hall, 1973.
[4] P. Brinch Hansen, *Concurrent Programming Concepts Lecture Notes*, « International Summer School on Structured Programming and Programmed Structures » Munich 1973, Germany.
[5] M. Conway, *A multiprocessor System Design*. Proceeding AFIPS 1963 Fall Joint Comp. Conf. Spartan-Books.
[6] J. B. Dennis and E. C. Van Horn, *Programming Semantics for Multiprogrammed Computations*. Comm. ACM 9, 3, 1966.
[7] J. B. Dennis, *Concurrency in Software Systems*. Comp. Group Memo 65-1, Project MAC-MIT, June 1972.
[8] E. W. Dijkstra, *Cooperating Sequential Processes. Programming Languages*, F. Gennys, ed. Academic Press, New York, 1968.
[9] E. W. Dijkstra, *Hierarchical Ordering of Sequential Processes*. Acta Informatica 1, 2, 1971.
[10] E. W. Dijkstra, *Notes on Structured Programming. Structured Programming*. Academic Press, 1972.
[11] A. N. Habermann, *On the Harmonious Cooperation of Abstract Machines*, Ph. D. Thesis, Technical University of Eindhoven, Netherlands, 1967.
[12] A. N. Habermann, *Synchronization of Communicating Processes*. Comm. ACM Vol. 15, n. 3, 1972.
[13] C. A. R. Hoare, *Towards a Theory of Parallel Programming* in Operating Systems Techniques, C. A. R. Hoare and R. H. Perrott, editors, Academic Press, 1972.
[14] J. J. Horning and B. Randell, *Process Structuring*. ACM Computing Surveys, Vol. 5, n. 1, 1973.
[15] U. Montanari, *Processi Cooperanti*, Seminario su Programmazione Strutturata. Milano, 12-13 Febbraio 1974.
[16] R. A. Snowdon, *System for the Preparation and Validation of Structured Programs*, in Program Test Methods, W. C. Hetzel, Editor. Prentice-Hall 1973.