

Appunti sulla sincronizzazione (corso di Sistemi Operativi a.a. 2001–'02)

Eugenio G. Omodeo

Università degli studi di L'Aquila

Dipartimento di Informatica

67010 – Loc. Coppito (AQ).

e-mail: *omodeo@univaq.it*

Maggio 23, 2002

Indice

1	Condivisione di risorse e corse critiche	2
1.1	Cooperazione: perché ?	2
1.2	Mutua esclusione, e corse / sezioni critiche	2
1.3	Il problema della sezione critica	4
1.4	Scambio di messaggi fra processi	4
1.5	Problema del buffer a capacità limitata	4
1.6	Problemi di stallo	6
2	Sincronizzazione di processi con memoria comune	7
2.1	Formulazione del problema della sezione critica	8
2.2	Disabilitazione delle interruzioni	9
2.3	Alternanza stretta	10
2.4	La mutua esclusione non basta	11
2.5	Algoritmo di Peterson per la sincronizzazione di 2 processi	11
2.5.1	Correttezza dell'algoritmo di Peterson	12
2.6	Algoritmo della panetteria per la sincronizzazione di N+2 processi	12
2.7	HW di sincronizzazione	13
2.7.1	Attesa limitata con TSL	14
2.8	Inconvenienti di un'attesa laboriosa, e un'alternativa	15
2.8.1	Paradosso dell'inversione delle priorità	16
2.9	Buffer limitato: approccio senza attesa attiva al problema	16
2.10	Semafori generali	17
2.10.1	Versione 'busy' del semaforo	18
2.10.2	Mutua esclusione tramite semaforo	19
2.10.3	Un problema semplice di sincronizzazione	20
2.11	Buffer limitato: soluzione con 3 semafori	20
2.12	Allocaz. dinamica di un insieme di risorse equivalenti	21
2.13	Semafori con coda	23

2.13.1	Semafori binari	24
2.13.2	Come esprimere un semaforo generale per mezzo di tre semafori binari	25
2.14	Sincronizzazione a piú alto livello: i monitor	26
2.15	Altre strutture di sincronizzazione: contatori di eventi e scambio di messaggi .	30

1 Condivisione di risorse e corse critiche

1.1 Cooperazione: perché ?

[SG], p.109, enuncia alcune buone ragioni per le quali la cooperatività fra processi deve essere assicurata:

- Condivisione dell'informazione
(ad es. piú utenti possono aver necessità di condividere un file ,
o lo stesso editor —perché duplicarlo?—,
o due processi lo stesso compilatore, o un pipe ...),
- perfino un utente singolo può voler operare in parallelo editing, compilazione, stampa di uno stesso file
- Aumento di efficienza grazie alla suddivisione di un compito in sotto-compiti che possono venire svolti in parallelo
- Modularità nel disegno, cominciando dallo stesso S.O.

1.2 Mutua esclusione, e corse / sezioni critiche

sincronizzazione:

Benché le risorse naturali e logiche possano essere condivise, generalmente possono venir utilizzate da un solo processo alla volta. Una risorsa che ammette un solo utente per volta viene chiamata una “risorsa critica”. Se diversi processi desiderano condividere l'uso di una risorsa critica, debbono sincronizzare le loro operazioni in modo che al piú uno di loro abbia, in ciascun momento, il controllo della risorsa.

[TB], pag.27

L'esigenza di sincronizzazione si può cogliere attraverso riflessioni ed esempi. Quando una memoria (quale che ne sia la natura) è condivisa fra processi, possono —in assenza di sincronizzazione— verificarsi situazioni anomale dette

corse critiche:

*situazioni ... nelle quali due o piú processi stanno leggendo o scrivendo un qualche dato condiviso ed il risultato finale dipende dall'ordine in cui vengono schedulati i processi, vengono dette **race conditions**, ovvero corse critiche.*

[T], pag.38

Questo brano si accompagna in [T] con un esempio di ‘malagestione’ di spooler directory, delineato anche in Figura 1 di queste note; dopodiché [T] conclude con le parole:

*due processi accodano
file nella spooler directory...*

processo A	processo B
1. next_free_slot := IN + 1	2. next_free_slot := IN + 1
5. QUEUE[next_free_slot] := 'pippo' (soprascrivendo, cosí, 'pluto')	3. QUEUE[next_free_slot] := 'pluto'
6. IN := next_free_slot (le cose appaiono in ordine)	4. IN := next_free_slot

*... ma, per una casualità, il demone della stampante
non manderà 'pluto' alla stampa*

Figura 1: ESEMPIO DI CORSA CRITICA

*La difficoltà descritta sorgeva perché il processo B cominciava ad usare le variabili condivise prima che il processo A avesse a sua volta finito di usarle ... abbiamo bisogno di **mutua esclusione**, un qualche modo di assicurarsi che se un processo sta usando una var. o un file condiviso, gli altri proc. saranno impossibilitati a fare altrettanto.*

[T], pag.38

(Osserviamo, di sfuggita, che problemi del genere si pongono già ai livelli piú bassi di un S.E.:

mutua esclusione: *[relativa all'HW]*

La mutua esclusione viene implementata in HW rendendo indivisibili le operazioni di immagazzinamento. Vale a dire, se ciascuno di due processi cerca di immettere un valore nella stessa locazione, l'HW arbitra; un accesso viene consentito —l'altro processo deve attendere finché il primo non ha terminato. Questo arbitrato, chiamato memory interlock, basta ...

[TB], pag.31)

Prima di poter descrivere le tecniche per ottenere mutua esclusione al livello della gestione SW dei processi, conviene introdurre il concetto di

sezione critica:

Entro ciascun processo, le regioni che accedono a risorse critiche possono essere isolate. Queste regioni, chiamate sezioni critiche, devono avere la proprietà di essere mutuamente esclusive. Vale a dire, al piú un processo potrà star eseguendo una sezione critica rispetto ad una risorsa.

[TB], pag.28

1.3 Il problema della sezione critica

Quattro obiettivi di una cooperazione affidabile ed efficiente fra processi stanno dietro alle tecniche di sincronizzazione che verranno discusse piú avanti:

Mutua esclusione. Rispetto ad una risorsa comune, *non piú di un processo alla volta* può star eseguendo la propria sezione critica.

Progresso. Se uno o piú processi che condividono una risorsa richiedono l'accesso alle rispettive sezioni critiche, *prima o poi dev'essere consentito l'accesso* ad uno di loro.

Attesa limitata. Se piú processi che condividono una risorsa richiedono l'accesso alle rispettive sezioni critiche, *nessuno dev'essere prevaricato dagli altri a tempo indefinito*.

Temporizzazione arbitraria.

Supponiamo che ogni processo venga eseguito a velocità non-nulla.

Non facciamo alcuna ipotesi sulla velocità relativa dei processi in concorrenza fra loro. . . . né sul numero di CPU disponibili.

Ciononostante, *il risultato finale dev'essere sempre lo stesso*.

1.4 Scambio di messaggi fra processi

Dopo che avremo discusso le tecniche fondamentali di sincronizzazione, la seguente osservaz. sarà piú chiara:

“Un problema con i monitor, e con i semafori, è che furono progettati per risolvere il problema della mutua esclusione su una o piú CPU dotate di memoria comune . . . Passando ad un sistema distribuito formato da CPU multiple, ciascuna con la sua memoria privata, collegate attraverso una rete locale, queste primitive diventano inapplicabili. La conclusione è che i semafori sono di troppo basso livello ed i monitor non sono utilizzabili tranne che per pochi linguaggi di programmazione; inoltre nessuna di queste primitive fornisce scambio di informazione tra macchine. È necessario qualcos'altro.”

[T] p.52

1.5 Problema del buffer a capacità limitata

Dall'es. già visto, potrebbe sembrare che la patologia delle corse critiche sorgesse semplicemente dall'impiego di var. d'appoggio private, a causa del quale risulterebbe indebitam. frammentata la modificaz. delle var. condivise. Il guaio è che il ricorso a zone di lavoro “private” —potrebbe perfino trattarsi di registri, messi in salvo e ripristinati dal meccanismo di context-switching— non è evitabile. Questo si coglie bene dall'es. di *buffer a capacità limitata* discusso alle pp.163–165 di [SG], che qui di seguito riprendiamo:

L'area condivisa fra un processo “produttore” ed uno “consumatore” è un array gestito come coda circolare:

```

const M = ... ;
type  item = ... ;
      index = 0 ... M - 1 ;
var   BUFFER : array[ index ] of item ;
      IN , OUT : index ;           /* inizialm. 0 */
      COUNTER : 0 ... M ;         /* inizialm. 0 */

```

Il produttore fa questo:

```

repeat
  ...
  ...
  produci un item in nextp;
  ...
  ...
  while COUNTER = M do ;; /* busy waiting */
  BUFFER[IN] := nextp ;
  IN := (IN+1) mod M ;
  COUNTER += 1 ;
until false ;             /* cioè "for ever" */

```

Il consumatore fa questo:

```

repeat
  while COUNTER = 0 do ;; /* busy waiting */
  nextc := BUFFER[OUT] ;
  OUT := (OUT+1) mod M ;
  COUNTER -= 1 ;
  ...
  ...
  consuma l'item in nextc;
  ...
  ...
until false ;           /* cioè "for ever" */

```

La patologia può manifestarsi semplicem. per le istruz. di incr-/decr-emento di COUNTER, che in linguaggio macchina diverranno:

```

reg1 := COUNTER ;
reg1 += 1 ;
COUNTER := reg1 ;

```

e, rispettivam.,

```

reg2 := COUNTER ;
reg2 -= 1 ;
COUNTER := reg2 ;

```

Non è escluso che $reg_1 \equiv reg_2$; ma di sicuro i valori dei registri non interferiranno, grazie alle protezioni previste nel context-switching.

Per definitezza, supponiamo che $M = 99$ e che un produttore relativam. veloce sia riuscito a mettere 5 item nel buffer, prima ancora il consumatore abbia completato lo smaltim. del primo. Dunque COUNTER = 5 (mentre IN=5, OUT=0).

Un 6° item è in corso di inserimento, e produttore e consumatore si vengono a trovare pressoché

insieme a modificare COUNTER, allorché la presenza di altri processi li rallenta moltissimo entrambi, causando la seguente corsa critica :

```
val.
...
...
reg1 := COUNTER 5
reg1 += 1        6
[ context switch ]
reg2 := COUNTER 5
reg2 -= 1        4
[ context switch ]
COUNTER := reg1 6
[ context switch ]
COUNTER := reg2 4
```

Senza il secondo (o seconda serie che sia, di) context-switch, COUNTER risulterebbe 6, anche questo —a buon senso— sbagliato.

Il corretto val., 5, si ottiene con la temporizzaz. seguente :

```
val.
...
...
reg1 := COUNTER 5
reg1 += 1        6
COUNTER := reg1 6
[ context switch ]
reg2 := COUNTER 6
reg2 -= 1        5
COUNTER := reg2 5
```

1.6 Problemi di stallo

deadlock:

Un insieme di processi si trova in una situazione di stallo se ogni processo dell'insieme aspetta un evento che solo un altro processo dell'insieme può provocare.

[T], pag.249

Un processo richiede delle risorse; se le risorse non sono disponibili al momento entra in uno stato di attesa. Può accadere che qualche processo in attesa non cambi stato mai più, perché le risorse da lui richieste sono in mano ad altri processi ugualmente in attesa? Questa situazione viene chiamata un deadlock.

...

“ ‘Quando due treni si appressano uno all'altro ad un incrocio, debbono entrambi fermarsi del tutto, e né l'uno né l'altro ripartirà sin quando l'altro non si sarà tolto di mezzo.’ ”

[SG], pag.217

Nei confronti del deadlock, o “stallo”, si possono adottare tre diverse politiche :

- prevenzione
(Il sistema non deve essere in nessun modo in grado di raggiungere uno stato ‘insicuro’, da cui potrebbe originare uno stallo —vedi fig. di p.229 in [SG])
- riconoscimento automatico
(uno stallo, dopo che si è verificato, deve essere rilevato da un programma che si preoccupa di far ritirare una o più delle richieste accampate)
- riconoscimento da parte dell’operatore
(poiché i deadlock sono, tutto sommato, infrequenti, lasciare che si verifichino piuttosto che incorrere in gran costi di politica preventiva; sarà poi l’operatore ad intervenire)

Il rischio,

- con la prima politica, è che le risorse rimangano di frequente sotto-utilizzate
- con la terza, è che i tempi morti nell’uso del calcolatore vengano a costare più di una politica di gestione automatica

2 Sincronizzazione di processi con memoria comune

Non è facile trovare una definizione di *sincronizzazione* abbastanza generale da poter racchiudere tutta la gamma di problemi che vengono catalogati sotto questa voce.

Come prima riflessione su questi problemi, possiamo richiamare le

forme di interazione tra processi:

- *L’interferenza, costituita da errori durante l’accesso a risorse comuni (accessi non necessari o erronei).*
- *La competizione, per l’accesso esclusivo a risorse comuni.*
- *La cooperazione, cioè lo scambio di informazioni, tramite risorse comuni.*

... uno dei problemi fondamentali nella programmazione concorrente consiste nel prevenire e rilevare il maggior numero possibile di interferenze tra processi. Infatti la caratteristica peculiare di questi errori è la loro dipendenza dal tempo che li rende particolarmente insidiosi e difficili da rilevare in fase di collaudo del programma.

[AB], pag.68

Poco oltre lo stesso testo dice:

“Per quanto riguarda la cooperaz. tra processi, sono necessari costrutti che consentano la sincronizzaz. degli accessi ad una risorsa comune tramite la quale i processi si scambiano informazioni

...

Per quanto riguarda la competizione ... è normalm. necessario assicurare la mutua esclus. negli accessi alla risorsa” [AB], p.71

La difficoltà di separare nettamente le problematiche di cooperaz. da quelle di competiz. è già emersa dall’esempio della sez.1.5; perciò parleremo *tout court* di sincronizzaz. riferendoci alle une e alle altre

Riassumendo queste premesse, possiamo dire che le tecniche di sincronizzaz. servono a prevenire interferenze. Servono, inoltre —anzi, prioritariamente—, ad assicurare una gestione equa delle risorse condivise fra più processi ed una ragionevole efficienza globale nell’impiego delle stesse.

Per circoscrivere l’ambito di questo capitolo, è bene sottolineare che stiamo parlando di risorse diverse dal tempo di CPU: per risorsa, almeno per ora, intendiamo una struttura-dati (talvolta comprensiva delle procedure associate) allocata nella memoria comune della macchina concorrente (cfr. [AB] p.68)

Quanto allo scheduling della CPU fra i processi, la sua stessa esistenza costituirà, nello studio della sincronizzaz., un notevole intralcio. Questo perché puntiamo a soluzioni la cui correttezza prescindano interamente dal comportamento dello scheduler. Per poter prescindere da esso, dovremo però regolarci come se esso fosse del tutto imprevedibile e quindi, in certo senso, ‘ostile’

Emblematico della classe di problemi di cui stiamo intraprendendo lo studio sarà il solito problema della sez. critica, ma ne tratteremo anche altri, a scopo illustrativo: vedi ad es. la sez.2.12 di queste note, ed i problemi dei filosofi a cena, del barbiere che dorme e dei lettori e scrittori, in [T] pp.60–65

2.1 Formulazione del problema della sezione critica

Consideriamo processi concorrenti, in numero superiore ad 1:

$$\varpi_0, \varpi_1, \dots, \varpi_{N+1}$$

Il testo di ciascun ϖ_i comprende una *sezione critica* in cui ϖ_i può modificare variabili o file condivisi con gli altri.

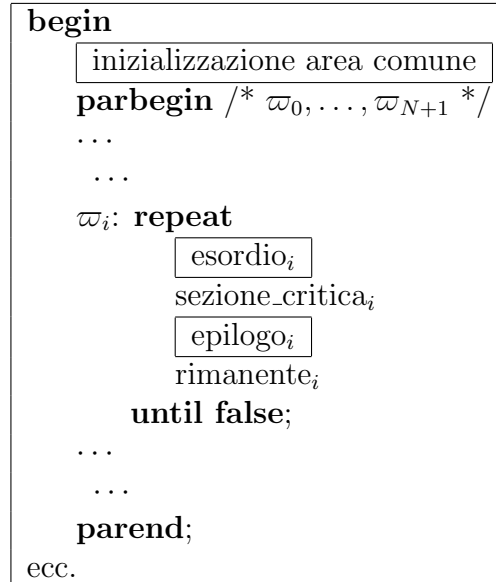
Per poter progettare più facilmente un ‘protocollo’ di cooperazione che realizzi la mutua esclusione, prevediamo che ciascun ϖ abbia:

- un *esordio* (o ‘*entry section*’), attraverso cui deve inevitabilm. passare per poter accedere alla propria sez. critica; ed

- un *epilogo* (o *'exit section'*) attraverso cui ne fuoriesce.

Per studiare un caso delicato del problema della sincronizzazione, consideriamo $N + 2$ processi potenzialmente perpetui.

D'altro lato, per semplicità, la memoria condivisa verrà gestita come una singola risorsa.



(Il costrutto **parbegin** $\sigma_0, \dots, \sigma_{N+1}$ **parend** significa che le istruz. σ_i possono venire eseguite in concorrenza)

Come vanno progettati inizializzazione, esordi ed epiloghi?
 (Serviranno var. condivise ausiliarie)

Gli obiettivi sono quelli descritti nella sez.1.3. All'ipotesi di temporizzazione arbitraria aggiungiamo quella che le istruzioni del ling. macchina siano *'atomiche'*, cioè indivisibili; per cui, se ne eseguiamo due in concorrenza, il risultato sarà equivalente alla loro esecuzione in un ordine imprecisato. Se ad es. una **load** e una **store** vengono eseguite in concorrenza, la **load** otterrà il nuovo o il vecchio valore: non una miscela dei due.

Un obiettivo importante sarà anche di poter prevedere quante volte, al massimo, ϖ_i dovrà *'cedere il turno'* per l'accesso alla sez. critica ad altri processi: una soluzione *equa* sarà quella in cui ciò non avverrà più di $N + 1$ volte.

2.2 Disabilitazione delle interruzioni

Può sembrare la soluzione più ovvia, per la mutua esclusione, che un processo disabiliti le interruz. subito prima di entrare nella propria sez. critica e le riabiliti non appena ne esce. Senza interrupt, come potrebbe essergli tolto il controllo?

Ammesso che vi sia un'istruzione che la consenta, la disabilitaz. degli interrupt ha due inconvenienti:

- troppo potere per il singolo processo:
sebbene la sez. critica consista generalm. di “straight code”, che viene eseguito rapidamente e non può portare a cicli perpetui, basterebbe dimenticare la riabilitaz. degli interrupt per inibire il S.O. (e quindi tutti gli altri processi) a tempo indefinito
- disabilitare gli interrupt, quando piú processori condividano la memoria, può comportare scambio di messaggi fra le CPU; di conseguenza, parecchio tempo speso per evitare che sia solo una la CPU preoccupata della corretta gestione della risorsa critica

“disabilitare le interrutz. è una tecnica che in qualche caso può risultare utile nel kernel, ma non è appropriata come meccanismo di mutua esclusione fra processi”
[T] p.39

2.3 Alternanza stretta

Due processi, ϖ_0, ϖ_1 , vengono sincronizzati così
(con la condivisione `var TURN: 0..1`):

<pre>repeat while TURN ≠ 0 do;; sez. critica₀ TURN := 1; rimanente₀ forever</pre>	<pre>repeat while TURN ≠ 1 do;; sez. critica₁ TURN := 0; rimanente₁ forever</pre>
---	---

Supponiamo che ϖ_0 passi per primo attraverso la sua sez. critica

Poi, molto lentamente, ϖ_0 esegue rimanente₀

Nel frattempo ϖ_1 potrebbe percorrere la propria sez. critica diverse volte; invece non può farlo piú di una, e consuma inutilm. tempo nel proprio esordio

Come caso limite, se rimanente₀ contenesse un ciclo perpetuo, ϖ_1 risulterebbe inibito per sempre. Caso limite analogo, se mancasse la **repeat** esterna di ϖ_0

Conclusione: la mutua esclusione è soddisfatta, il requisito di progresso no (o forse male)
[Vedi es. directory di spool a p.41 di [T]]

Osserviamo che conviene così rafforzare il requisito di progresso:

progresso:

Quando nessun proc. è impegnato nella propria sez. critica, ma vi sono proc. che desiderano entrarvi, nel decidere a quale spetti per primo l'accesso, si dovrà tener conto di quei soli processi che non sono nella loro sez. rimanente. Inoltre questa decisione non potrà essere rinviata senza termine.

[SG], pag.166

2.4 La mutua esclusione non basta

Per risolvere gli inconvenienti della ‘soluzione’ appena vista, sembra opportuno tener conto, nella sincronizzaz., dello stato di ogni processo. Al posto di TURN, adoperiamo

var FLAG: **array** [0..1] of boolean

inizializzato

FLAG[0] := FLAG[1] := false

in quanto né ϖ_0 né ϖ_1 ha ancora richiesto accesso alla sez. critica (né vi si trova impegnato).

<p>repeat</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> FLAG[0] := true; while FLAG[1] do;; </div> <p>sez. critica₀</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> FLAG[0] := false; </div> <p>rimanente₀</p> <p>forever</p>	<p>repeat</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> FLAG[1] := true; while FLAG[0] do;; </div> <p>sez. critica₁</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> FLAG[1] := false; </div> <p>rimanente₁</p> <p>forever</p>
--	--

Tuttavia il requisito di progresso è ancora violato, dato che potrebbero susseguirsi

FLAG[0] := true ; FLAG[1] := true

dopodiché ϖ_0 e ϖ_1 non potrebbero proseguire (Situaz. di spinlock)

Invertendo l'ordine delle istruz. in entrambi gli esordi si aggirerebbe solo la difficoltà a progredire : si violerebbe invece la mutua esclusione

2.5 Algoritmo di Peterson per la sincronizzazione di 2 processi

La prima soluz. soddisfacente al problema della sincronizzaz. di due processi su una risorsa critica, dovuta al matematico olandese T. Dekker, venne pubblicata da Dijkstra nel 1965. Essa fonde le idee delle due ‘soluzioni’ discusse sopra: fa uso sia di TURN che di FLAG, condivise e dichiarate come prima (anche l'inizializzaz. di FLAG è la stessa). È riportata in [SG] ma non in [T].

Sorprendentemente, una soluz. piú semplice (realizzata con gli stessi ingredienti) non apparve fino al 1981. Vediamola :

<p>repeat</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> FLAG[0] := true; TURN := 1; while FLAG[1] and (TURN = 1) do;; </div> <p>sez. critica₀</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> FLAG[0] := false; </div> <p>rimanente₀</p> <p>forever</p>	<p>repeat</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> FLAG[1] := true; TURN := 0; while FLAG[0] and (TURN = 0) do;; </div> <p>sez. critica₁</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> FLAG[1] := false; </div> <p>rimanente₁</p> <p>forever</p>
---	---

2.5.1 Correttezza dell'algoritmo di Peterson

Assumiamo che

- le sez. critiche non modifichino FLAG né TURN
- lo scheduler sia abbastanza equo da permettere l'avvicendamento —sia pure saltuario— fra ϖ_0 e ϖ_1

Mutua esclusione: Ipotizziamo per assurdo che ϖ_0 e ϖ_1 vengano entrambi a trovarsi in sez. critica

Avremo allora che

- $\text{FLAG}[0] = \text{FLAG}[1] = \text{true}$
- TURN ha il val. che gli è stato assegnato dal processo piú “tardivo”, che con ciò stesso si è preclusa l'entrata nella sez. critica —contraddiz. \square

Progresso:

- Se ϖ_0 e ϖ_1 hanno espresso entrambi interesse per la risorsa critica (ponendo i rispettivi FLAG a true), e la risorsa non è stata ancora assegnata, essa verrà rilasciata al meno tardivo dei due
- Se ϖ_i ha richiesto accesso, e ϖ_{1-i} è ancora nel rimanente, ϖ_i può entrare in sez. critica \square

Attesa limitata:

- L'avvicendam. (non l'alternanza!) in zona critica è garantito dal fatto che quando ϖ_{1-i} arriva dopo che ϖ_i ha già mostrato il proprio interesse, allora gli cede il passo \square

2.6 Algoritmo della panetteria per la sincronizzazione di N+2 processi

Sono condivise :

```
var FLAG: array [0 .. N + 1] of boolean;  
    NUM: array [0 .. N + 1] of integer
```

inizializzate rispettivam a false e a 0

È privata di ciascun ϖ_i :

```
var j: 0 .. N + 1
```

Testo di ϖ_i :

repeat

```
FLAG[i] := true; /* cartellino... */
NUM[i] := 1 + maxj=0N+1 NUM[j];
FLAG[i] := false; /* ...preso */
for j := 0 to N+1 do begin
  while FLAG[j] do;;
  while NUM[j]≠0 and
    ⟨ NUM[j], j ⟩ < ⟨ NUM[i], i ⟩ do;;
end;
```

sez. critica_i

```
NUM[i] := 0;
```

rimanente_i

forever

(Richiamiamo che l'ordinamento usuale tra numeri induce un ordinamento lessicografico fra le liste di numeri: in partic. due coppie possono essere confrontate in base alla loro prima componente; in sottordine —vale a dire: se le coppie hanno prima componente uguale—, allora in base alla seconda componente)

2.7 HW di sincronizzazione

La disabilitaz. degli interrupt è stata già scartata

Tuttavia sono comuni delle speciali istruz. HW che consentono di effettuare come 'atomiche' delle operaz. relativam. laboriose

(Anche queste sono problematiche in architetture multiprocessore)

La piú comune —chiamata TSL— è concettualm. equivalente a

```
function Test_and_Set(var lock: boolean):
                               boolean;
begin
  Test_and_Set := lock;
  lock := true
end
```

In alternativa, può essere utile una istruz. equivalente a

```
procedure Swap(var x, y: boolean);
begin
  [ x, y ] := [ y, x ]
end
```

Per mezzo della TSL, la mutua esclus. di $N + 2$ processi può essere implementata con una sola variabile condivisa, `Lock`, inizialm. falsa :

```

 $\varpi_i$ :  repeat
           while  Test_and_Set( Lock )  do ;;
           sezione_critica_i
           Lock := false;
           rimanente_i
           forever;

```

Naturalm, tutti i ϖ_i devono attivare e disattivare il `Lock` secondo questo stesso protocollo, altrimenti la mutua esclusione salterà

Espressi in un assembler fittizio, epilogo ed esordio figurano così nella fig. di p.43 in [T] :

```

esordio:  tsl register, Lock
           cmp register, #0
           jnz esordio
           ret

epilogo:  mov Lock, #0
           ret

```

Una soluz. al problema della mutua esclusione basata su `Swap` invece che su TSL viene discussa in [SG] pp.174–175¹, dove compare anche il seguente algoritmo che rispetta, oltre alla mutua esclus., il requisito di attesa limitata:

2.7.1 Attesa limitata con TSL

La tecnica sopra-descritta di sincronizzaz. tramite TSL assicura mutua esclusione e progresso, ma non l'attesa limitata, come possiamo vedere dal seguente

scenario con due processi:

1. ϖ_0 chiama TSL, pone `Lock` a 1
2. prima che ϖ_0 sia uscito dalla sua sez. critica. . .
3. . . subentra ϖ_1 , che chiama la sua TSL ed entra / rimane in attesa laboriosa, finché
4. subentra ϖ_0 , che velocissimamente esce dalla sez. critica, fa il rimanente e torna a fare come al punto 1

¹esordio: `key := true; repeat Swap(Lock, key) until not key`

... e tutto si ripete! ...

Ecco una nuova soddisfacente soluzione al problema della sez. critica con $N + 2$ processi.

Sono condivise :

```
var FLAG: array [0...N + 1] of boolean;  
    LOCK: boolean
```

inizialmente false

Sono private di ciascun ϖ_i :

```
var key: boolean;  
    j: 0...N + 1
```

Testo di ϖ_i :

```
repeat
```

```
    FLAG[i] := true; /* in attesa... */  
    key := true;  
    while FLAG[i] and key do  
        key := Test_and_Set(LOCK);  
    FLAG[ i ] := false; /* ... ora non piú */
```

sez. critica_i

```
    j := (i + 1) mod (N + 2);  
    while j ≠ i and not FLAG[j] do  
        j := (j + 1) mod (N + 2);  
    if j=i then LOCK := false  
        else FLAG[j] := false;
```

rimanente_i

```
forever
```

2.8 Inconvenienti di un'attesa laboriosa, e un'alternativa

busy waiting:

*Il testare continuamente una variabile, in attesa che assuma un certo valore viene detto **attesa attiva** [o "laboriosa"]. Di norma ciò dovrebbe essere evitato, dal momento che porta ad uno spreco del tempo di CPU; l'attesa attiva viene usata solo in quei casi in cui ci si aspetta che il tempo di attesa sia breve.*

[T], pag.40

Un es. letterario di attesa laboriosa:

... se la vita è sempre varia e impreveduta e cangiante, l'illusione è monotona, batte e ribatte sempre sullo stesso chiodo.

*Tutti cercando il van, tutti gli danno
colpa di furto alcun che lor fatt'abbia:
del destrier che gli ha tolto, altri è in affanno;
ch'abbia perduta altri la donna, arrabbia;*

*altri d'altro l'accusa: e così stanno
che non si san partir di quella gabbia;
e vi son molti, a questo inganno presi,
stati le settimane intiere e i mesi.*

(XII, 12)

(Da: Italo Calvino racconta l'Orlando Furioso, Einaudi, 1988)

spinlock:

quando un processo vuole entrare in una sezione critica, esegue un controllo per vedere se l'ingresso alla sezione è possibile. Se non lo è, il processo si blocca nell'esecuzione di un ciclo molto stretto, fino a che l'ingresso non è ammesso.

[T], pag.43

Gli spinlock sono utili in sistemi multi-processore . . . Il vantaggio di uno spinlock è che non serve context switch quando un processo deve attendere su un lock, ed un context switch può richiedere tempo considerevole. Così, quando ci si aspetta che i lock vengano tenuti impegnati per breve tempo, gli spinlock sono utili.

[SG], pag.177

2.8.1 Paradosso dell'inversione delle priorità

Con l'attesa attiva, neanche l'algoritmo di Peterson garantisce l'attesa limitata se non come conseguenza di una certa equità a livello di scheduling

Potremmo infatti —con uno scheduler a priorità immutabili— incorrere nel paradosso che

un processo B a bassa priorità preclude a tempo indefinito una risorsa critica ad un processo A ad alta priorità

La cosa si verifica così:

- dopo che B è entrato in sez. critica, A diventa *ready*, per cui lo scheduler lo presceglie
- B diventa *ready*, e dunque è impossibilitato a rilasciare la risorsa, il che causa lo spinlock senza fine di A

In alternativa all'attesa laboriosa (che è sempre 'vigile'), si possono utilizzare primitive SLEEP e WAKEUP(j), tramite le quali ϖ_i può, rispettivam., auto-sospendersi e “rimettere in moto” (cioè rendere *ready*) ϖ_j

2.9 Buffer limitato: approccio senza attesa attiva al problema

Riprendiamo l'es. della sez.1.5 (buffer a capacità limitata).

Chiamiamo ϖ_0 il produttore e ϖ_1 il consumatore.

Invece di busy waiting i due fanno, rispettivam., questo:

```
if COUNTER = M then sleep()
```

e questo:

```
if COUNTER = 0 then sleep()
```

dove `sleep` è una funzione di libreria che serve ad auto-sospendere il processo chiamante

Ciascun ϖ_i risveglia l'altro (ϖ_{1-i}), uno cosí:

```
[ dopo COUNTER += 1 ]
```

```
if COUNTER = 1 then wake_up( $\varpi_1$ )
```

e l'altro cosí:

```
[ dopo COUNTER -= 1 ]
```

```
if COUNTER = M-1 then wake_up( $\varpi_0$ )
```

(quando lo 'svegliato' viene trovato già sveglio, le cose restano com'erano)

Anche qui viene affrontato solo il problema di sincronizzaz. (intesa come cooperaz.), e non quello di mutua esclusione

In effetti, quando `COUNTER` è discosto da `M` e da `0`, ϖ_0 e ϖ_1 rischiano di finire insieme in sez. critica ... Ma c'è di peggio ...

Può verificarsi la seguente corsa critica:

ϖ_1 ha appena caricato `COUNTER` in un registro, per confrontarlo con `0`; allorché

ϖ_0 subentra a ϖ_1 e constata (dopo averlo incrementato) che il `COUNTER` in effetti valeva `0`, per cui "sveglia" ϖ_1 ; allorché

ϖ_1 subentra a ϖ_0 , e constatato che `COUNTER` è (sarebbe meglio dire: è stato?) a `0`, si addormenta.

La sveglia è arrivata troppo presto!

(Le conseguenze sono descritte a p.45 di [T])

Potremmo ricorrere ad un

bit di attesa della sveglia:

Quando viene spedita una sveglia ad un processo che è già sveglio, questo bit viene messo a 1. Più tardi, quando il processo cerca di sospendersi, se il bit d'attesa della sveglia è a 1, verrà messo a 0, ma il processo rimarrà sveglio.

[T], pag.45

...ma davvero basta un solo bit? E se avessimo più produttori e consumatori?

2.10 Semafori generali

"Dijkstra suggerí (nel 1965) di usare una variabile intera per contare il numero di sveglie salvate per uso futuro ..."

Un semaforo può valere 0, il che indica che non è stata ancora salvata alcuna sveglia, o qualche valore positivo, se una o più sveglie sono pendenti ..."

DOWN e UP (generalizzaz. di SLEEP e WAKEUP, rispettivam.) ...

L'operaz. UP incrementa il val. del semaforo su cui viene invocata. Se uno o piú processi erano sospesi su quel semaforo, incapaci di completare una precedente operaz. DOWN, uno di essi viene scelto dal sistema (... a caso [?]) per permettergli di completare la propria DOWN

... sono operazioni fatte in un'unica, indivisibile azione atomica" [T] p.45

2.10.1 Versione 'busy' del semaforo

semaforo:

Un semaforo S è una variabile intera alla quale si accede —inizializzazione a parte— tramite due sole operazioni, entrambe atomiche:

```
DOWN  [ oppure P, talvolta wait ]
UP    [ oppure V, talvolta signal ]
```

Caratterizzaz. classiche di DOWN e UP sono

```
down(S):  repeat until  S > 0;
          S -= 1
```

```
up(S):    S += 1
```

[SG], pagg.175–176²

L'indivisibilità, nel caso della down, che possiamo riscrivere

```
1:  if  S > 0  then  S -= 1  else  goto  1
```

significa che l'esecuz. non può essere interrotta durante il test $S > 0$ né, se questo dà esito affermativo, prima che l'operaz. di decremento sia ultimata.

(Un semaforo B che ammetta due soli valori, 0 / true ed 1 / false, potrebbe venir implementato così:

```
down(B):
  1:  if  Test_and_Set(B)  then  goto  1;

up(B):   B := false; -- indivisibilm.  )
```

²In realtà abbiamo modificato varie piccole cose rispetto al [SG]; dato che, per es., quello usa gli id. wait e signal, che in [T] hanno un altro significato.

2.10.2 Mutua esclusione tramite semaforo

```

begin
  MUTEX := 1;
  parbegin /*  $\varpi_0, \dots, \varpi_{N+1}$  */
  ...
  ...
   $\varpi_i$ : repeat
    down(MUTEX);
    sezione_critica_i
    up(MUTEX);
    rimanente_i
  forever;
  ...
  ...
  parend;
ecc.

```

In che modo MUTEX assicura la mutua esclusione?

- Assumiamo che il valore iniziale, s_0 , di un semaforo S sia ≥ 0
- Evidentem. avremo $val(S) \geq 0$ anche in ogni successivo istante
- Inoltre questo val., dopo l'esecuz. completa di n_Up operaz. UP ed n_Down operaz. DOWN, varrà

$$s_0 + n_Up(S) - n_Down(S)$$

- Passando al caso partic. di MUTEX, in cui $s_0 = 1$, abbiamo dunque che

$$n_Down(MUTEX) - n_Up(MUTEX) \leq 1$$

- Un altro invariante è che

$$0 \leq n_Down(MUTEX) - n_Up(MUTEX),$$

dato che ogni chiamata UP fa seguito ad una chiamata DOWN da parte del medesimo ϖ_i

- D'altronde l'espressione che figura nelle due disequaz. rappresenta il numero massimo di processi che potranno trovarsi simultaneam. in sez. critica

Marginalm. osserviamo che

MUTEX è un semaforo binario!³

(potremmo quindi semplificarne la caratterizzaz. come visto in fine sez.2.10.1)

³Segue infatti da $0 \leq n_Down(MUTEX) - n_Up(MUTEX) \leq 1$ che $val(MUTEX) = 1 - (n_Down(MUTEX) - n_Up(MUTEX))$ è compreso fra 0 ed 1

Domande: sono soddisfatti i requisiti di progresso ed attesa limitata?

Risposte: Per quanto riguarda il progresso la risposta è sí, poiché si dimostra la tesi che

Ad ogni passo, se $val(MUTEX) = 0$, allora c'è un proc. in sez. critica.

(In caso contrario, può entrare in sez. critica il primo proc. che ne faccia richiesta)

Procediamo per induz. sul numero d'ordine, i , dell'ultimo passo effettuato

$i = 0$: Essendo falso l'antecedente della tesi, essa è vera

$i = j + 1$:

- a) se il passo in questione ha fatto una UP, l'antecedente della tesi è falso
- b) se il passo ha completato una DOWN, è vero il conseguente della tesi
- c') se il passo ha solam. intrapreso una DOWN, per ip. induttiva c'era un proc. in sez. critica, che dunque vi si trova ancora
- c'') piú in generale, un'azione diversa da a) e da b) non cambia lo stato di cose preesistente, giacché non altera il val. di MUTEX e non fa fuoriuscire processi dalla sez. critica

Per quanto riguarda l'attesa limitata, cosa ci assicura che in sez. critica non si venga a trovare sempre il medesimo processo? Le cose stanno come nel caso della mutua esclusione realizzata tramite TSL (vedi inizio della sez.2.7.1)

2.10.3 Un problema semplice di sincronizzazione

Per essere certi che una certa istruz. di un processo avvenga dopo una certa istruz. di un concorrente...

...basta inizialm. porre a 0 un apposito semaforo [binario?] SY

L'istruz. da effettuare per seconda sarà preceduta da
down(SY)

Quella da effettuare per prima sarà seguita da
up(SY)

Cfr. [SG], p.176

2.11 Buffer limitato: soluzione con 3 semafori

Per la terza volta eccoci a considerare il problema del buffer a capacità limitata gestito in concorrenza da un produttore ed un consumatore

Ci serviamo di un semaforo binario per la mutua esclusione e di due semafori che contano le componenti libere e occupate del buffer

```

const M           = ... ;    /* positiva */
type  item        = ... ;
        index       = 0 ... M - 1 ;
var   MUTEX := 1   : 0 ... 1 ;
        LIB := M    : 0 ... M ;
        OCC := 0    : 0 ... M ;
        BUFFER   : array[ index ] of item ;
        IN := 0 ,
        OUT := 0   : index

```

Produttore :

```

 $\varpi_0$  : repeat
    ...
    ...
    produci un item in nextp;
    ...
    ...
    down( LIB );
    down( MUTEX );
    inserisci nextp in BUFFER
    up( MUTEX );
    up( OCC )
forever

```

Consumatore :

```

 $\varpi_1$  : repeat
    down( OCC );
    down( MUTEX );
    estrai nextc da BUFFER
    up( MUTEX );
    up( LIB )
    ...
    ...
    consuma l'item in nextc;
    ...
    ...
forever

```

Questa soluz. è riportata sia nella fig. di p.47 in [T] che nelle fig. di p.182 in [SG]

2.12 Allocaz. dinamica di un insieme di risorse equivalenti

Il problema ripreso qui sotto è estesamente trattato in [AB], pp.76-79.

Struttura-dati condivisa tra $\varpi_0, \dots, \varpi_{N+1}$:

```

const M           = ... ;
type  item        = ... ;
       index       = 0 ... M - 1 ;
var   RISORSA    : array[ index ] of item ;
       MUTEX      : 0 ... 1 ;
       N_LIBERE   : 0 ... M ;
       LIBERA     : array[ index ] of boolean

```

Ciascun ϖ_i dispone di una var. privata x

Ciascuna RISORSA[_] dev'essere concessa ad un processo solo alla volta, ma

le risorse sono "intercambiabili", per cui un processo non dovrà attenderne una specifica, occupata al momento, quando ve ne sono altre libere

```

begin
  MUTEX := 1;
  N_LIBERE := M;
  for i := 1 to M do
    LIBERA[i] := true;
  parbegin /*  $\varpi_0, \dots, \varpi_{N+1}$  */
  ...
  ...
   $\varpi_i$ : repeat
    richiesta(x);
    uso della risorsa il cui indice è x
    rilascio(x);
    rimanentei
  forever;
  ...
  ...
  parend;
ecc.

```

```

procedure richiesta( var x : 1...M );
var i: 0...M;
begin
  down(N.LIBERE);
  down(MUTEX);
  i := 0;
  repeat
    i += 1
  until LIBERA[i];
  LIBERA[i] := false;
  x := i;
  up(MUTEX)
end;

procedure rilascio( x : 1...M );
begin
  down(MUTEX);
  LIBERA[x] := true;
  up(MUTEX);
  up(N.LIBERE)
end;

```

Per quanto riguarda la procedura rilascio, permane il dubbio che la `down(MUTEX)` e la corrispondente istruz. `up(MUTEX)` non servano veramente

Una serie di illustrazioni d'uso dei semafori si trova nella sez.2.3 di [T]

2.13 Semafori con coda

L'implementaz. dei semafori proposta nella sez.6.4.2 di [SG] dà un senso a val. negativi per un semaforo S : tali valori possono esprimere il numero dei processi in attesa della risorsa custodita da S

I val. positivi, come al solito, danno il numero di sveglie inutilizzate

L'aspetto nuovo è che —come suggerito da [T]— invece dell'attesa attiva adesso si effettua la sospensione

A ciascun semaforo è associata la coda dei processi in attesa:

```

type semaforo =
  record
    sveglie : integer;
    coda   : list of processo
  end

```

Le primitive diventano:

```

down(S) : S.sveglié -= 1 ;
          if S.sveglié < 0 then
            begin
              aggiungi il qui presente a S.coda
              sleep()
            end

up(S) :   S.sveglié += 1 ;
          if S.sveglié ≤ 0 then
            begin
              rimuovi un proc.  $\varpi$  da S.coda
              wake_up(  $\varpi$  )
            end

```

Permane l'obbligo di implementare UP e DOWN come operaz. indivisibili

Quest'ultimo problema può essere risolto con una delle soluz. già discusse al problema della sez. critica, per es.

- disabilitaz. delle interruz. oppure uso di TSL (su monoprocesso)
- attesa attiva (per fortuna la risorsa semaforo è quasi sempre libera —grazie alla rapidità di esecuz. di UP e DOWN— per cui il tempo sprecato incide poco)

Inoltre, i semafori generali sono realizzabili tramite semafori binari (vedi sez. 6.4.4 di [SG] e sez.2.13.2 di queste note); dunque soprattutto questi ultimi necessitano di un'implementaz. sofisticata

Si noti come la gestione delle code associate ai semafori non sia demandata allo scheduler, bensì gestita da quella stessa parte del kernel che implementa i semafori

2.13.1 Semafori binari

Un semaforo il cui massimo valore possibile è 1, viene detto binario. Esempio tipico, è il MUTEX visto nella sez.6.10.2

- Se un sem. binario viene implementato in modo “spinning”, non può prendere che i val. 0 ed 1
- se viene implementato tramite coda a fini di mutua esclus. fra due processi, potrà assumere valori
 - 1:** nessuno dei due proc. in sez. critica
 - 0:** un proc. in sez. critica, l'altro *ready* (o *running*)
 - 1:** un proc. in sez. critica, l'altro ‘addormentato’ nella coda del semaforo

- se viene implementato —sempre tramite coda— per la mutua esclusione fra $N+2$ processi, potrà prendere i valori

$$1, 0, -1, \dots, -N - 1$$

- Per l'implementaz. 'spinning' di un semaforo binario B , bastano le seguenti:

$down_2$: **repeat until** B ; $B := false$

up_2 : $B := true$

- Per l'implementaz. tramite coda di un sem. binario B , bastano le seguenti (ove $B.tace =_{Def} (B.svegli \leq 0)$):

$down_2$: **if** $B.tace$ **then begin**
 aggiungi il qui presente a $B.coda$
 sleep() **end**
else $B.tace := true$

up_2 : **if** $B.coda \neq \emptyset$ **then begin**
 rimuovi un proc. ϖ da $B.coda$
 wake_up(ϖ) **end**
else $B.tace := false$

2.13.2 Come esprimere un semaforo generale per mezzo di tre semafori binari

Un sem. generale può essere reso per mezzo di una var. intera C (mai inferiore a -1) e di tre sem. binari: B , D ed U .

C andrà inizializzata secondo le esigenze del caso —comunque ≥ 0 —;

essa coinciderà con il “numero di sveglie” del semaforo che stiamo emulando —naturalm., finché questo numero è ≥ -1

Inizializzaz. di B , D ed U : le rispettive code sono vuote;

D ed U ‘suonano’, viceversa B ‘tace’ (vedi fine della sez.2.13.1)

La coda del sem. generale verrà ripartita fra B e D
 —nella coda di B , non vi sarà mai più di un elemento

Generalm. la coda di D sarà utilizzata solo dopo che quella di B si sia saturata

(Però, l'el. della coda di B verrà rimosso prima di quelli della coda di D ;
 l'instabilità risultante da ciò verrà eliminata il prima possibile)

Significato di D : serve ad impedire una sovrapposiz. (“interferenza”) fra due **down**

Significato di U : serve ad impedire una sovrapposiz. (“interferenza”)
 fra una **up** ed una **up/down**

La down del sem. generale si realizza così:

```
down2(D);
down2(U);
C -= 1;
if C < 0 then
  begin
    up2(U);
    down2(B)
  end
else
  up2(U);
up2(D)
```

La up del sem. generale, così:

```
down2(U);
C += 1;
if C ≤ 0 then up2(B);
up2(U)
```

2.14 Sincronizzazione a più alto livello: i monitor

“Come è noto dalla teoria sui S.O., il meccanismo semaforico è molto potente e può essere utilizzato per qualunque problema di sincronizzaz. e di comunicaz. fra processi ... Successivam. saranno presentati meccanismi linguistici di più alto livello, meno potenti ed in certi casi meno efficienti, ma certam. di più semplice uso e più sicuri per quanto riguarda il problema delle interferenze”

[AB] p.72

(Ma è proprio vero che i semafori risolvono tutti i problemi di sincronizzazione? Cfr. in merito [SG] p.214. Un es. delle difficoltà di uso dei semafori si trova all’inizio della sez.2.2.7 di [T])

Ci siamo venuti gradualmente distaccando dall’idea che una risorsa condivisa sia un’area-dati comune, per avvicinarci all’idea che possa essere una procedura. In effetti, dal concetto di risorsa condivisa, siamo passati dapprima al concetto di sezioni critiche (\approx porzioni di codice che non devono essere percorse in contemporanea da diversi processi). In seguito abbiamo imparato ad evitare possibili interferenze evitando l’ingresso simultaneo nella stessa procedura da parte di più processi —basti guardare a come è stato utilizzato il semaforo MUTEX per l’allocaz. dinamica di risorse equivalenti (sez.2.12 di queste note), per la soluz. del problema dei filosofi a cena (sez.2.3.1 di [T]), ed infine (ridenominato D od U) per l’emulaz. di un semaforo generale tramite sem. binari.

Citiamo, a questo riguardo:

“Le procedure sono un meccanismo d’interfaccia molto più sicuro che le strutture-dati comuni”

[Bri] p.21

(Osserviamo che, per rappresentare risorse in modo astratto, le procedure certo non sono meno accettabili delle aree-dati. Che siano superiori lo possiamo arguire dalla lenta ma costante affermazione nelle metodologie di produzione del SW del concetto di *tipo astratto di dati*)

Da questo genere di riflessioni al concetto di *monitor* il passo è breve⁴. L'idea nasce da un articolo di Dijkstra del 1971, ma la prima notazione linguistica per i monitor è di Brinch Hansen (1973), che successivamente la incorporerà in *Concurrent Pascal*. La semantica dei monitor proposta da Hoare (1974) è un po' più generale di quella di Brinch Hansen, come accenneremo sotto. Il costrutto monitor si trova in vari linguaggi di programmazione concorrente.

monitor:

Un monitor è una collezione di procedure e strutture-dati che, pur essendo condivisa fra più processi, può essere adoperata da uno solo alla volta. Il concetto è analogo a quello di una camera per cui c'è una sola chiave. Se un processo si accinge a usare la camera e la chiave è appesa alla porta, esso può aprire la camera, entrare, servirsi delle procedure del monitor. Se non c'è la chiave appesa, il processo deve attendere che l'attuale utilizzatore abbia lasciato la camera —rimettendo a posto la chiave. Non è lecito a nessuno stare nella camera per sempre.

...

Un monitor contiene una o più procedure rientranti, con strutture-dati globali statiche. La prima volta che viene invocato, il monitor inizializza le proprie variabili.

[TB], pag.42

Va sottolineato che un monitor è un oggetto passivo, come una camera; non è un processo. Il monitor diventa vivo solo quando un processo sceglie di utilizzare i suoi servizi.

[TB], pag.43

... quando un processo chiama una procedura di monitor, le prime istruzioni [invisibili] della procedura controlleranno se un qualche processo è attivo dentro il monitor; se è così, il processo chiamante sarà sospeso finché l'altro processo non avrà lasciato il monitor.

[T], pag.50

Questa è la sintassi di un monitor in *Concurrent Pascal*:

⁴La scelta del vocabolo 'monitor' non è stata delle più felici. A volte si parla, intendendo lo stesso, di 'capsula'.

```

type monitor_id = monitor
    dich_variabili

procedure entry P0(...);
begin ... end;
procedure entry P1(...);
begin ... end;
    .....
    .....
procedure entry Pk(...);
begin ... end;

begin
    codice d'inizializzaz.
end

```

Un esempio di monitor è il buffer di riga (adattato qui da [Bri] p.52 —per un es. piú generale, vedi fig.2.14 di p. 51 in [T]):

```

type lineBuffer = monitor

var contenuto : linea;
    pieno : boolean;
    emittente, ricevente : condition;

procedure entry ricevi(var testo: linea);
begin
    if not pieno then ricevente.wait;
    testo := contenuto; pieno := false;
    emittente.signal;
end;

procedure entry trasmetti(testo: linea);
begin
    if pieno then emittente.wait;
    contenuto := testo; pieno := true;
    ricevente.signal;
end;

begin pieno := false end

```

Qui occorre spiegare il senso delle variabili di tipo **condition**.

Si tratta, in sostanza, di code:

*“Le sole operazioni che possono essere invocate su una variabile-condizione x sono **wait** e **signal**. La*

x .wait

significa che il processo che invoca questa operazione è sospeso fino a che un altro processo non invochi

`x.signal ;`

la `x.signal` dà il riavvio ad esattamente un processo sospeso [associato alla condizione `x`]. Se nessun processo è sospeso, la `signal` non ha alcun effetto; cioè, lo stato di `x` è come se quell'operazione non fosse mai stata effettuata. Si contrasti questa operazione con la `UP` associata ad un semaforo, che altera sempre lo stato di un semaforo.” [SG] pp.191-192

Non si pensi per questo che il monitor sia un costrutto piú debole di quello semaforico. Un facile esempio di monitor è, per l'appunto, il semaforo (adattato qui da [TB] p.43):

```
mutex :  monitor

var  s : integer;
    pos : condition;

procedure entry  down();
begin
    if  s < 1  then  pos.wait;
    s := s - 1;
end;

procedure entry  up();
begin
    s := s + 1;
    if  s=1  then  pos.signal;
end;

begin  s := 1  end
```

All'inverso, in linguaggi (come C) che non prevedano il costrutto monitor, il concetto di monitor può tradursi in disciplina di programmazione. Come rendere il monitor per mezzo di semafori, è indicato dalla fig.2.16 di p.57 in [T].

In merito ai monitor, si veda la sez.2.2.7 di [T]; un'illustrazione che riassume molto vividamente il concetto è la fig.6.18 a p.192 del [SG]

Quando P libera Q con una `signal`, a chi tocca proseguire?

- forse a P , che già occupava il monitor
- forse meglio a Q (questa la posizione di Hoare) —fin quando non lasci il monitor o si blocchi su una nuova condizione.

(Proseguendo, P potrebbe falsificare di nuovo la condizione di cui Q è stato in attesa finora)

Soluzione di compromesso (in *Concurrent Pascal*):

- signal deve sempre essere l'operazione conclusiva

2.15 Altre strutture di sincronizzazione: contatori di eventi e scambio di messaggi

[Vedi sezioni 2.2.6 e 2.2.8 in [T]]

Bibliografia

- [AB] Paolo Ancilotti, Maurelio Boari. *Principi e Tecniche di Programmazione Concorrente*, 359 pp. Utet libreria, Collana di Informatica, 1987.
- [Bri] Per Brinch Hansen. *The Architecture of Concurrent Programs*, 317 pp. Prentice-Hall Series in Automatic Computation, 1977.
- [C91] Stefano Ceri. *Architettura dei sistemi informatici*, 153 pp. (20000lit) Edizioni clup di Città Studi (MI), 1991 (la IV ristampa è del '94).
- [G] Graham Glass. *UNIX for Programmers and Users - A complete guide*, Prentice-Hall, 1993.
- [Dei90] H.M. Deitel. *Operating Systems*, 2nd edition. Reading, MA, Addison-Wesley, 1990.
- [LL93] John R. Levine, Margaret Levine Young. *Usare UNIX senza fatica*. 322 pp. (32500lit) McGraw-Hill Libri Italia srl, 1993.
- [M94] Piero Maestrini. *Sistemi Operativi*, 604 pp. (54000 lit). McGraw-Hill Italia, Serie di Informatica, 1994.
- [R90] Gianfranco Rossi. Il nucleo di un sistema operativo a processi. Dispense per il corso di Sistemi per l'Elaborazione dell'Informazione II, Università di Udine, a.a. 1989/90.
- [S92] W. R. Stevens. *Advanced programming in the UNIX environment*. Addison-Wesley, Reading, MA, 1992.
- [SG] Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*, 4th edition, 780 pp. Addison-Wesley, World Student Series, 1994.
- [SPG91] Abraham Silberschatz, James L. Peterson, Peter B. Galvin. *Sistemi Operativi*, traduz. della terza edizione in inglese, 614 pp. (72000lit). Addison-Wesley Italia Editoriale srl, 1991.
- [TB] Dionysios C. Tsichritzis, Ph. A Bernstein. *Operating Systems*, 298 pp. Academic Press, A series of monographs and textbooks, 1974.
- [T81] Andrew S. Tanenbaum. *Computer Networks*, 517 pp. Prentice-Hall Software Series, 1981.
- [T] Andrew S. Tanenbaum. *I Moderni Sistemi Operativi*, 741 pp. (72000lit). Prentice Hall Int., Jackson libri, 1994. (Traduz. dell'edizione in inglese del 1992)